Network Traffic Analysis using Traffic Dispersion Graphs (TDGs): Techniques and Hardware Implementation

UCR - Technical Report

Marios Iliofotou (UCR), Prashanth Pappu (Cisco), Michalis Faloutsos (UCR),
Michael Mitzenmacher (Harvard University), Sumeet Singh (Cisco), George Varghese (UCSD)

May, 2007

Abstract

Monitoring network traffic and detecting unwanted applications has become a challenging problem, since many applications obfuscate their traffic using arbitrary port numbers or payload encryption. Apart from some notable exceptions, most traffic monitoring tools follow two types of approaches: (a) keeping traffic statistics such as packet sizes and inter-arrivals, flow counts, byte volumes, etc., or (b) analyzing packet content. In this work, we propose the use of Traffic Dispersion Graphs (TDGs) as a powerful way to monitor, analyze, and visualize network traffic. TDGs model the *social behavior* of hosts ("who talks to whom"), while the edges can be defined to represent different interactions (e.g. the exchange of a certain number or type of packets). With the introduction of TDGs, we are able to harness the wealth of tools and graph modeling techniques from a diverse set of disciplines. First, we fully explore the abilities of TDGs as an intuitive and visually powerful tool. Second, we demonstrate their usefulness in application classification and intrusion detection solutions. Finally, we provide a hardware-aware design and implementation for TDG-based techniques. We conclude that TDGs are powerful, useful, and can be implemented efficiently in hardware. They constitute a promising new chapter for network monitoring techniques.

Keywords: *Network Monitoring, Network Introspection, Traffic Measurements, Host Social Behavior, Graph Mining.*

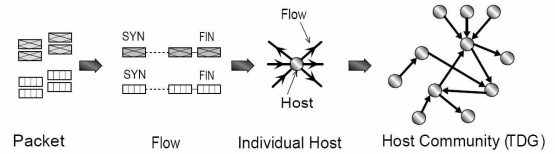
1 Introduction

The fundamental problem that motivates this paper is the need for better tools to monitor networks, that allow the detection and control of undesired applications in a network. Worms and viruses continue to be an expensive concern given that networked services and disruptions are estimated to have cost $17.5 billion in 2005 [1]. At the same time P2P applications have fundamentally affected the music industry forcing the music industry to go online. P2P has also revolutionized telephone service via Skype. One consequence of this is a proliferation of companies that help ISPs detect and potentially delay or block Skype and other P2P traffic. However, P2P authors have responded by obfuscating their traffic. In both the case of malcode and P2P, the standard approaches using *content signatures* seems destined to fail in the face of encryption (e.g., for P2P traffic) and polymorphism (e.g., for worms).

These forces suggest the need for a more fundamental *behavioral* approach to characterize traffic in the face of obfuscation. At the same time, the field *Social Network Analysis* has provided an important set of metrics and analytical tools for researchers in fields ranging from anthropology to psychology. Colloquially, social networks study "who knows who and through who". If we replace individuals in social network theory with say individual IP addresses, it is natural to ask the question: *can we use social network measures to more finely characterize applications?*

Comparison with current monitoring tools: Current monitoring and application classification methods can be classified by their level of observation: (a) packet level, such as signature-based worm detection, [2, 3] (b) flow level, based on NetFlow records such as statistical approaches, [4, 5], and (c) host level, such as host-profiling approaches [6, 7].

Using a network wide social network graph is the natural next step in the progression of packet, flow, and host level monitoring as shown in the following figure. This is because a flow **aggregates** a set of packets, a host aggregates a set of flows originating and terminating at the host, and a graph aggregates a group of hosts. In other words, we can analyze the "social" interaction of the network as a whole, which leads to a graph where each node is an IP address, and each directional edge represents an interaction between two nodes.



We use the term **Traffic Dispersion Graph** or **TDG** to refer to this graph. We argue that there is a wealth of information embedded in a TDG. For example, a popular website will have a large in-degree, while DNS servers will form a hierarchy. Despite what may appear at first, the definition of TDGs is non-trivial, as it hinges on how we define an edge. An edge can represent the exchange of at least one packet, but it can also be the exchange of at least of one TCP SYN packet, or more than, say, five packets of any type. In other

words, a TDG can represent a particular type of interaction, which gives them significant descriptive power, as we discuss later in detail.

Apart from some notable exceptions, TDGs have been mostly ignored. We attribute this to two concerns, namely, (a) whether TDGs can really capture interesting information that we don't already know, and (b) whether they can be useful in practice since they seem to require the maintenance of network-wide information. A few research efforts, most of them recent, hint at the capabilities of TDGs. The first work using graphs for intrusion detection appears in 1999 [8], without any recent follow up work. Recently, a few efforts use graph-based techniques to detect worm outbreaks and pinpoint the origin of the infection [9, 10]. Note that these efforts focus on worms, and they do not seem to be concerned with hardware implementation issues. In addition, some companies like Mazu [11] and Arbor [12] Networks, seem to use *proprietary* graph-based techniques. Here, we build on the limited current work, but we argue that it has not come close to exploring the full potential of TDGs.

In this paper, our main goal is to propose TDGs as a radically different way of modeling traffic behavior, and show that they: (a) provide powerful new capabilities, and (b) can be implemented efficiently in hardware. TDGs describe the traffic along a new "dimension", the network-wide social behavior, which complements the traffic characterization at the packet, flow and host level. We provide preliminary evidence that we can implement TDGs in hardware, potentially at Gbps links, which can increase their impact in practice tremendously. In fact, deployability is a running theme in our work, and we consider it when we explore graph metrics for TDGs.

Our contributions focus on the following questions:

- *What properties do TDGs have, and do these properties lend themselves naturally to visualization or automatic tools?* We first show the promise of TDGs as a visualization tool. In a qualitative way, we explore the potential of TDGs and, at the same time, derive some guidelines which we use later in our work. We also show that TDGs are not just another scale-free graph, as they have different properties (e.g. rich club connectivity) compared to scale-free graphs. In fact, TDGs are not one family of graphs with common properties. We find that different applications have significantly different TDGs, which is exactly what gives them descriptive power.

- *For what applications do TDGs appear immediately useful?* We show that TDGs are useful by using them for (a) application classification and (b) intrusion detection. We identify metrics that can easily pinpoint applications effectively, and we develop compact visualizations based on pairs of carefully chosen metrics.

- *Can TDGs be implemented at high speeds?* Having established that TDGs are interesting and useful, we examine their practical limitations: we show that they can be implemented at high speeds with low memory requirements. Towards this goal, we develop techniques for sampling a TDG graph efficiently. We show how we can implement several fundamental measures (e.g. degree distributions) in an online fashion and with reasonable hardware resources.

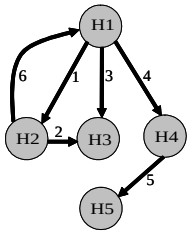


Figure 1: Example TDG with five nodes and six directional links.

We envision our graph-based techniques and their efficient implementation as the beginning of a toolset for dynamically composable hardware functions. The different definitions of an edge and the graph metrics can be thought of as filters, which can be combined. For example, we can decide to identify the top-10 most active destination port numbers, and for each one, monitor edges with TCP SYN packets, and estimate the resulting degree distribution of the resulting TDG graphs. Clearly, the TDG functionality could be synergistic with packet-based and flow-based approaches. For example, given a target packet signature, we can generate a TDG graph with edges that have exchanged this packet at least once.

The outline of the rest of this paper is as follows. In Section 2 we introduce the formal definition of a TDG, associated with a key, and the concept of edge filtering. In Section 3, we visualize and provide a quantitative description of TDGs. In Section 4 we show they are useful in identifying concealed applications and malcode. We discuss implementation issues in Section 5. Related work is discussed in Section 6 and, finally, we conclude the paper in Section 7.

2 Traffic Dispersion Graphs

Definition: A traffic dispersion graph is a graphical representation of the various interactions ("who talks to whom") of a group of nodes. In IP networks, a node of the TDG corresponds to an entity with a distinct IP address and the graph[1] captures the exchange of packets between various sender and destination nodes. For example, Fig. 1 depicts a TDG with five nodes $\{H_1, ..., H_5\}$. In the general case, the directed graph is not simple since we could have an edge (H_1, H_2) and an edge (H_2, H_1) as shown in Fig. 1. Moreover, note that a TDG, by definition, is a graph that evolves

[1]We want to make the distinction that the physical topology of the network does not have an effect on the TDG that the nodes form. However, given that our work will be based on measured data, our observation point(s) will most likely not be able to capture all the interactions, but only the packets that pass through the point of observation. We discuss this issue further in Section 3.

Name	*Date/Time*	*Type*	*Duration*	*Unique IPs*	*Unique Dst. Port Numbers*	*5-tuple Flows*	*Avg. Utilization Mbps (Kpkt/s)*
WIDE	2006-03-03/13:00	Backbone	2 h	1,041,622	TCP=62,463 UDP=64,727	4,670,259	31.0 (9.7)
AUCK	2003-12-05/12:05	Access Link	1 h	97,982	TCP=50,251 UDP=11,370	854,417	13.0 (3.0)
OC48	2003-01-15/10:00	Backbone	1'02 h	2,945,800	TCP=65,536 UDP=55,264	22,109,681	589.0 (127.8)
UCSD	Controlled Honeypot	LAN	5 m	466	TCP=157 UDP=6	1470	0.389 (0.041)

Table 1: The set of publicly available trace from the WIDE Backbone, CAIDA and the University of Auckland (source: NLANR).

both in time and space as various nodes interact with each other. Hence, the edges in a TDG have an implicit temporal relation which is shown in Fig. 1 by labeling the edges in the order in which the corresponding node-interactions were observed in the network. This also means that a given *static* TDG has an associated time interval over which it evolved. We discuss the effect of the time interval on the observed TDG properties in later sections.

Edge Filters: One of the fundamental questions in using TDGs is the definition of an edge. This basic question can be answered in many different ways depending on the goal of the study. We start with the observation that the edges in a TDG are directed because there is a clear definition of sender and receiver in every data packet. In general, the directed edges in a TDG can be used to identify the initiator of the interaction between a pair of nodes. As we will later see, directed edges in a TDG are very useful in identifying various node behaviors and also in establishing their causal relationship[2].

Besides direction, it is important to define what kind or level of interaction between a pair of nodes should be translated into an edge in the TDG. We call this process *Edge Filtering*. One simple edge filter is to add an edge (u, v) between nodes u and v when the first packet is sent from u to v in the interval of observation. Once an edge is added, the filter ignores any further packets sent from u to v. We call this edge filter as the Edge on First Packet (EFP), and is mainly used for translating UDP flows between nodes into TDG edges. Note that the EFP filter considers the sender of the first packet as the initiator of the interactions between the two nodes. Hence, this filter may sometimes inaccurately determine the initiator of node interactions, particularly when the interaction starts before the beginning of the time interval of observation.

Unlike UDP flows, TCP flows have an explicit definition of initiation of interaction between two nodes. For TCP flows, we can choose to add a directed edge (u, v) between two nodes u and v when the first SYN packet is sent from u to v. We call this filter as the Edge on First Syn Packet (EFSP) filter. While, the EFSP filter is applicable only to TCP flows, it can accurately determine the initiator of the interactions between a pair of nodes.

Various Types of Filters: In addition to this basic edge filtering, we can enrich the definition of what constitutes an edge by imposing "stricter" rules that capture different aspects of the interaction. For example, we can have filters for "allowing" an edge between a pair of nodes based on: **(a)** the number of packets/bytes exchanged, **(b)** the type and sequence of packets (e.g., TCP three-way handshake), **(c)** the transport protocol used (TCP, UDP, ICMP etc.), **(d)** the application based on port number or port number range (e.g., Port Number 80, or Port Range 6880-6889), and finally **(e)** looking at properties of the content, such as payload size or by using deep packet inspection (e.g., generate a TDG using all packets that match a suspicious content signature [3]).

[2]However, we could choose to consider undirected edges, which will enable us to use the more extensively studied graph metrics for undirected graphs, as discussed in later sections.

3 Understanding TDGs

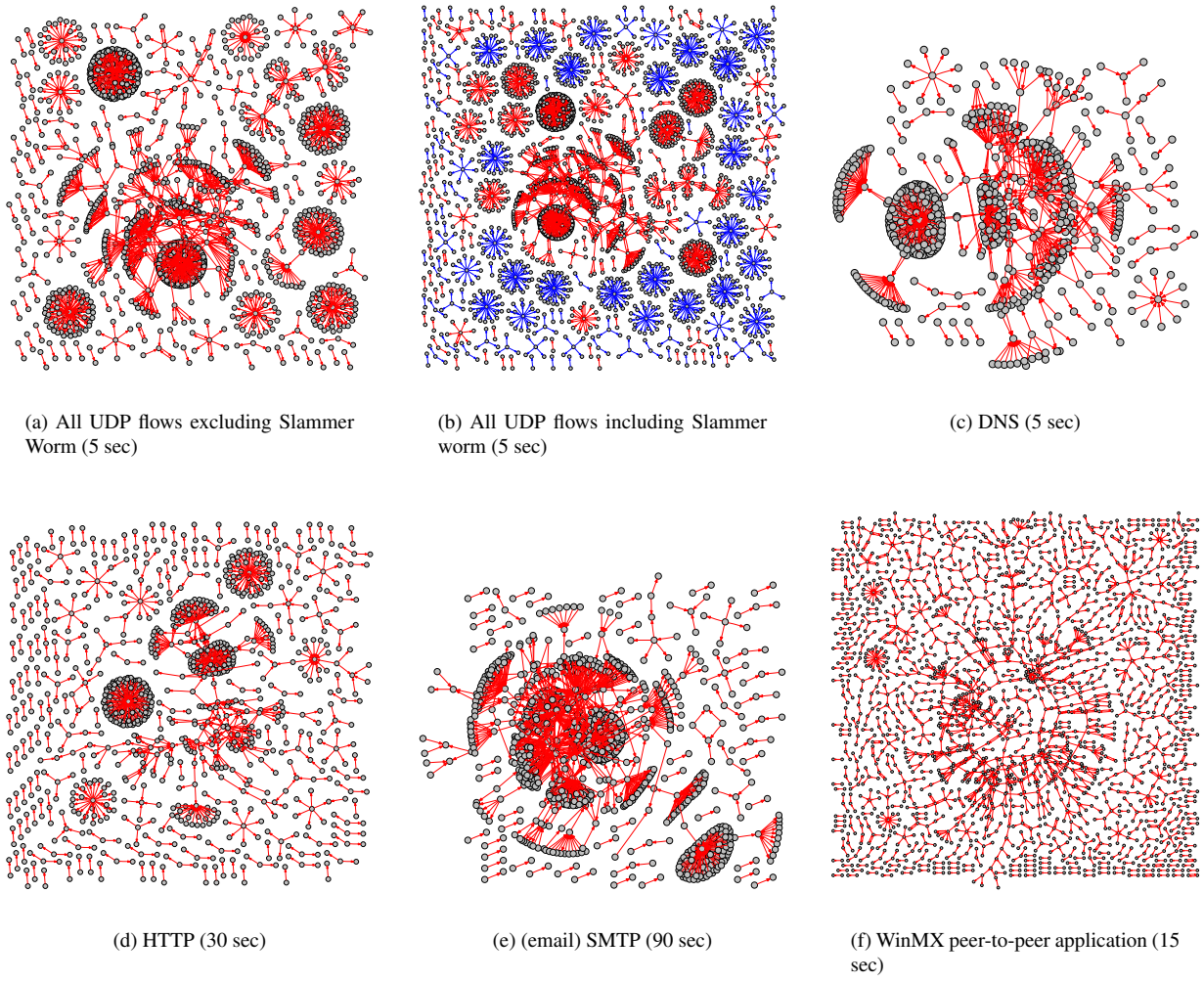
In this section, we illustrate the ability of TDGs to capture interesting traffic phenomena and properties. We begin with an intuitive understanding of their capabilities and then we study and quantify some distinguishing features.

Network Traffic Traces: To study and analyze TDGs, we use a variety of publicly available real-traffic traces as well as a LAN trace generated in a controlled honeypot environment (Table 1). For processing and generating the statistics we use the CoralReef [13] suite developed at CAIDA [14]. As can be seen from Table 1, the traces cover both an access link to a large enterprise network (i.e., University of Auckland) as well as backbone traffic (CAIDA [15], WIDE [16]). We verified our finding with many other traces provided by the same online sources [16, 14, 17]. However, we chose not to include those results for brevity. All of our traces, except UCSD, are capture on a single bidirectional link and hence reveal only the node interactions that cross the monitoring point. Note that if TDGs are implemented on a firewall or router, they will most likely see exactly that. Therefore, analyzing TDGs derived from one point of observation is closer to a practical deployment. Clearly, the TDGs formed by such traces are inherently bipartite. Observing all the network interactions, say within an enterprise network, would provide a more complete view of the network. We discuss this issue later in our paper.

3.1 TDG Visualization

We argue that TDGs provide excellent visual insight into the network traffic. So far, visualization of traffic in monitoring tools has largely been limited to measures of traffic volumes on a per flow basis.

TDGs lend themselves to simple yet insightful graphical visualizations. For example, the graphs in Fig. 2 show a simple set of TDGs, where we filter the edges for distinct Port Numbers. Throughout the paper and unless stated otherwise, when the legacy application for a port uses TCP, we use the EFSP edge filter on the corresponding destination port (e.g., Port 25 for SMTP, Port 80 for HTTP etc). When we examine UDP interactions, we use the EFP edge filter on the desti-



(a) All UDP flows excluding Slammer Worm (5 sec)

(b) All UDP flows including Slammer worm (5 sec)

(c) DNS (5 sec)

(d) HTTP (30 sec)

(e) (email) SMTP (90 sec)

(f) WinMX peer-to-peer application (15 sec)

Figure 2: Visualization of TDGs that correspond to various communities of hosts from backbone packet traces. The observation intervals were chosen so as to capture good visual details for each TDG.

nation port of interest (e.g., Port 53 for DNS). Since we use edge filtering by port number, the TDGs capture aspects of any application that uses these port numbers. For ease of presentation, we will refer to each such graph with the name of the dominant application at that port. We are fully aware that port-based classification is not necessarily accurate [6]. However, the port-based filtering is in line with the use of our approach as a monitoring and intrusion detection tool. For example, if at some point, traffic at TCP Port 80 appears significantly different, it could be: (a) a new benign or malicious application tunneling its traffic under that port, (b) a change in the behavior of the traditional application (i.e., Web in our example).

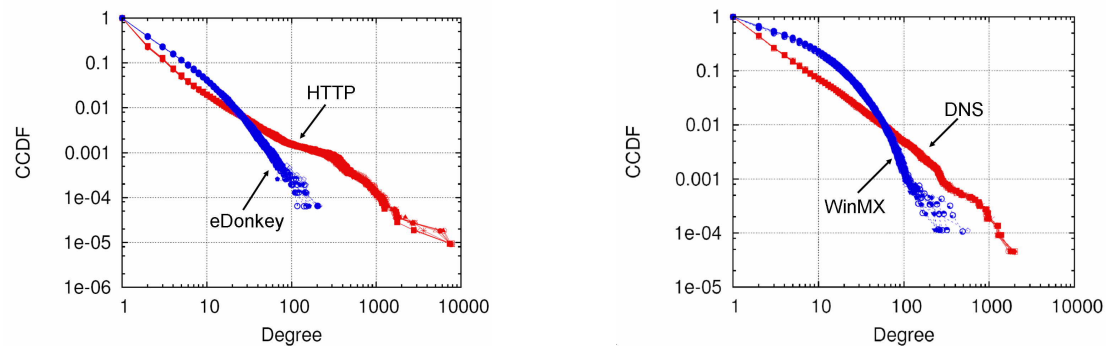
The graphs of Fig. 2 were drawn with the use of GraphViz [18]; a graph visualization tool which optimizes the layout by placing the bigger components towards the center of the graph. Studying these TDGs (Fig. 2), we can quickly reach the following conclusions, which are corroborated with a plethora of other similar visualizations:

(i) TDGs are not a single family of graphs. We can see that the TDGs present significant visual differences, which we quantify with graph metrics in Table 2, and we discuss later in this section.

(ii) TDGs capture many interesting patterns of node interactions. We can identify several distinctive structures and patterns in TDGs, which are indicative of the behavior of different applications.

Node degrees - The degrees of various nodes and their connectivity in a TDG helps us in visually determining the type of relationship between the nodes. In general, the TDGs corresponding to protocols with a prevalence of client-server interactions, such as DNS (Fig. 2(c)) and SMTP (Fig. 2(e)), are dominated by a few high degree nodes whereas the TDG of the popular peer-to-peer application WinMX has many similar degree nodes (Fig. 2(f)).

Node roles - In many TDGs, the role of a node can be inferred from the direction of its edges (not easily distinguishable at this visualization scale). For example, Fig. 2(d)



(a) HTTP Vs eDonkey

(b) DNS Vs WinMX

Figure 3: The empirical Complementary Cumulative Distribution Functions (CCDF) of the degrees of: (a) two EFSP generated TDGs, the heavy-tailed HTTP and the TDG corresponding to the peer-to-peer protocol 'eDonkey' (TCP Port 4662). (b) EFP generated TDGs of DNS and the peer-to-peer protocol 'WinMX' (UDP Port 6257). Multiple functions are derived from consecutive disjoint 300 sec time intervals of the trace. Stability of measured distributions, across disjoint intervals, is shown by the multiple overlapping curves. All four TDGs are from the OC48 trace.

presents an HTTP TDG. This TDG uses the EFSP edge filter and hence the directed edges accurately determine the initiator of the interactions between a pair of hosts. This makes it easy to spot Web servers for example, since they are "pointed to" by edges (non zero in-degree). Also, note that most hosts have either zero in-degree or zero out-degree indicating that they act either solely as a server or solely as a client. Interestingly, however, there are a few nodes with both non-zero in-degree and out-degree, which can correspond to HTTP proxies or Web caching systems.

Node hierarchies - Some TDGs exhibit a multi-level hierarchy while other graphs have mostly two-level clusters (disconnected star-shaped subgraphs). By hierarchy, we mean that there are few dominant nodes, which have the highest degrees, and then we have many nodes with smaller degrees that are connected with each other and/or to the dominant nodes. For example, the big component at the center of the UDP TDG in Fig. 2(a) (this graph is generated by using all the UDP packets, irrespective of their destination port number) exhibits a distinctive hierarchy and corresponds to DNS. Fig. 2(c) isolates and presents a magnification of this DNS TDG. A similar hierarchy is also visible in the SMTP TDG shown in Fig. 2(e).

Node chains - Long chains of nodes are very common in TDGs of peer-to-peer applications like WinMX shown in Fig. 2(f) and are mostly non-existent in the TDGs of other applications.

Number of connected components and their sizes - We note that there are many disjoint components in each of the TDGs suggesting that there are many smaller communities within a TDG. This comes in contrast to many other types of graphs such as the Internet topology or the web-page graph [19]. Also, the number of connected components and the size of the largest component varies a lot across different TDGs. Note the differences in the distribution of the sizes of various components in the HTTP TDG which has many components and the DNS and SMTP TDGs which are dominated by a single component.

(iii) TDGs can be used to detect specific traffic anomalies. Even in very broad communities, anomalous host behaviors very easily distort the expected distribution of components and patterns in a TDG. For example, the UDP flows in the TDG of Fig. 2(b) includes flows due to a worm scanner (Slammer worm) and Fig. 2(a) shows the same graph without the worm scanner activity. The difference is easy to spot - the rapid scanning done by Slammer infected nodes creates an anomalously high number of many star-shaped (a single node with high out-degree) components of similar size in the UDP TDG in Fig. 2(b).

Although our goal here is to visually examine the various properties of TDGs, good visualization methods have their own value. In fact, effective visualization and human monitoring can often be a more viable alternative to complicated automated methods for anomaly detection. Many times complete automation of anomaly detection is not possible[3]. We conclude that appropriately created TDGs can very quickly provide a wealth of useful information. Motivated by this observation, we attempt to quantify TDG properties with various graph metrics.

3.2 Quantitative Description of TDG Properties

There is a plethora of sophisticated graph metrics from diverse disciplines that can be used in the analysis of graphs [20, 19]. However, in this work we want to identify metrics that not only capture TDG properties but also lend themselves to an efficient real-time hardware implementa-

[3]Anecdotal information has it that Paypal, the electronic money transfer service, addressed its initial problem of fraud with a set of visualization techniques focusing on various transactions. These visuals were monitored by humans, who could more easily identify anomalous visual patterns and further investigate potentially fraudulent transactions. This is considered as a significant factor why Paypal gained a distinct advantage over its competitors.

	Nodes	*Edges*	*Avg. Degree*	*In-&-Out(%)*	*MAX Degree*	*MAX Degree Ratio*	*Size of GCC(%)*	*Assortativity Coef.*	*Avg Depth*	*MAX Depth*
AUCK-HTTP	2,738(283)	3,524(273)	2.582(0.095)	0.23(0.07)	485(73)	0.177(0.013)	77.01(7.16)	-0.217(0.029)	1.004(0.005)	1.583(0.515)
AUCK-SMTP	1,397(54)	1,514(67)	2.168(0.021)	1.42(0.31)	353(11)	0.253(0.010)	98.05(0.40)	-0.717(0.025)	1.103(0.121)	1.750(0.453)
AUCK-DNS	3,244(189)	5,010(324)	3.087(0.039)	10.02(0.63)	2,134(200)	0.657(0.028)	94.99(0.82)	-0.456(0.010)	1.262(0.336)	2.333(0.492)
AUCK-NetBIOS	2,961(667)	3,020(733)	2.032(0.056)	0.00 (0.00)	256(0)	0.091(0.020)	21.26(13.37)	-0.778(0.055)	1(0)	1(0)
OC48-HTTP	109,090(1,432)	138,301(874)	2.536(0.023)	0.09(0.01)	7,618(217)	0.070(0.002)	61.24(2.19)	-0.110(0.009)	1.002(0.001)	2(0)
OC48-SMTP	6,913(76)	9,799(146)	2.835(0.027)	3.41(0.21)	350(11)	0.051(0.002)	79.92(0.96)	-0.157(0.007)	1.028(0.009)	2.9(0.57)
OC48-DNS	22,025(384)	52,126(1109)	4.733(0.039)	11.00(0.21)	1,869(117)	0.085(0.004)	97.99(0.23)	-0.175(0.005)	1.180(0.012)	3.5(0.53)
OC48-WinMX	8,890(225)	33,593(599)	7.560(0.171)	28.68(0.56)	315(115)	0.035(0.012)	98.99(0.16)	-0.138(0.055)	1.273(0.030)	3.9(0.57)
WIDE-HTTP	10,922(10,512)	12,590(10,675)	2.389(0.102)	0.22(0.08)	3,837(10479)	0.155(0.230)	57.46(13.88)	-0.224(0.142)	1.012(0.009)	2(0)
WIDE-SMTP	2,242(61)	3,061(203)	2.732(0.148)	4.52(0.49)	340(80)	0.152(0.0363)	91.04(1.71)	-0.331(0.035)	1.132(0.062)	2.417(0.515)
WIDE-DNS	9,830(321)	18,799(613)	3.825(0.0263)	6.99(0.24)	4,242(137)	0.432(0.0058)	98.85(0.20)	-0.227(0.005)	1.243(0.049)	2.333(0.652)
WIDE-NetBIOS	3,486(887)	3,475(892)	1.993(0.0282)	0.02(0.04)	267(30)	0.081(0.0231)	9.74(2.76)	-0.745(0.052)	1(0)	1.083(0.288)

Table 2: Measured features for TDGs generated within a 300 sec time window. Values in parenthesis provide the standard deviation for the measured quantity after generating each TDG twelve times; each for every five-minute-long disjoint interval of the one hour long traces (12x300sec=1hour).Small standard deviations suggest that TDG properties are stable over the duration of observation.

tion (Section 5). In this section we present a series of fundamental graph metrics computed over real-traffic TDGs. A set of experimental results is summarized in Table 2. Graphs are calculated over 12 consecutive, disjoint, 300 second long observation intervals, which corresponds to an hour of monitored traffic for each trace. Note that the values in parenthesis provide the standard deviation of each metric over the twelve intervals, and which is typically small. This suggests that (a) the TDG properties seem very stable over the duration of observation, and (b) that 300 seconds is a reasonable interval of observing the formation of TDGs. Someone could argue that the observed stability is because we have the same set of hosts interacting with each other over the duration of the trace. Fig. 11 shows that this is not the case since the number of newly observed nodes is near linearly increasing over time. We discuss here the different metrics of Table 2 and their importance.

3.2.1 Scalar Graph Metrics

Average Degree: Is calculated by counting both in-coming and out-going edges, hence is the degree of a host if we ignore the directivity. Average degree indicates the popularity of a host, i.e., how many distinct IP addresses a host interacted over the observation interval. The average degree is a first approach into quantifying the coarseness of a graph and graphs with high average degrees tend to be tightly connected [20].

Max Degree Ratio: MAX Degree is the maximum degree in the graph and MAX Degree Ratio (MDR) is the MAX Degree normalized by the number of nodes in the graph minus one (i.e. the maximum *possible* degree of a node in the graph). *Discussion:* It is interesting to note in Table 2 that in the OC48 trace, the average degree of WinMX (P2P), 7.56, is higher than that of DNS, 4.73, while, by contrast, the maximum degree of WinMX, 315, is much lower than that of DNS, 1896. This suggests that these two metrics can potentially be used to distinguish the two TDGs. For Table 2 we can see that high average degree and high MDR is distinctive characteristics of DNS since: (a) the presence of a dominating high degree DNS server leads to high MDR and (b) the hierarchy (Fig. 2(c)) is the reason we have high average degree.

In-and-Out degree (InO) property: The InO measures the percentage of nodes that have non-zero in-degree and non-zero out-degree. Such nodes are both initiators and "receptors" of initial communications, as we mentioned earlier and thus act both as *clients and servers*. For example, in the OC48 trace, we see that HTTP has practically zero percent of such nodes (pure client server application), while WinMX has 28% of nodes with "dual" role. As we will later show in Section 4, this property seems like an excellent metric for distinguishing client-server from peer-to-peer communities.

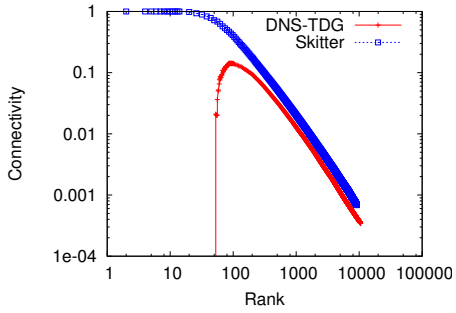


Figure 4: The Rich Club Connectivity metric of a DNS TDG. Size of clique as fraction of the network size Vs the number of nodes (in order of non-increasing degree) for the WIDE trace. By contrast, the rich club connectivity of the AS Internet topology graph from Skitter(CAIDA).

Size of Giant Connected Component (GCC): With the term "component" we refer to maximally connected subgraphs. If we consider again the graph as undirected, we can calculate the size of its largest connected component; a metric which is often used in graph analysis. We report this quantity as percentage of the total number of nodes in the TDG. We demonstrate the usefulness of this metric in the next section. Intuitively we expect densely connected TDG communities, commonly found in P2P protocols and network-gaming overlays, to have a large connected component that concentrates the majority of participating hosts.

Assortativity Coefficient: The *assortativity coefficient* measures the tendency of high degree vertices to connect with other high degree vertices. Instead of a formal definition [20], we focus more on its meaning. The assortativity coefficient r is the Pearson correlation coefficient of the degrees at either end of an edge and lies in the range $[-1, 1]$. If $r = 0$, the graph appears to have random degree correlations, and

there is no linear relationship between the degrees of adjacent nodes. If $r > 0$ then the graph is assortative, and high degree vertices are likely to connected to other high degree vertices. Conversely, if $r < 0$ then the graph is disassortative, and high degree vertices are likely to connect to low degree vertices.

Depth: For this metric, we consider the dynamic (temporal) nature of the graph, and we consider edges and nodes in the order of first appearance in an online fashion. We attempt to capture the "spread" of the communication. Thus, the first time we see a directed edge (H_1, H_2) , if we have not seen node H_1 before we give it a depth of zero. If H_2 is a new node to the graph then $depth(H_2) = depth(H_1) + 1$. From the TDG example of Fig. 1, node H_1 has zero depth, nodes H_2, H_4 have depth of one and nodes H_3, H_5 have depth of two. Note that we only set the depth of a node the first time we encounter it. In the next section, we show how by using depth we can detect the spreading of traffic in malware applications such as worms [9].

3.2.2 Non-Scalar Metrics

Non-scalar metrics reveal more detail structural characteristics of TDGs. For example, they show that *TDGs are not yet-another family of Scale-Free(SF) graphs.* In general, TDGs are not SF graph although they can have some common properties such as heavy-tailed degree distributions. Scale-free graphs appear in many real network topologies both in biological, social, and complex systems, such as citation networks, the hyperlinks of the pages on the Web [19].

Node Degree Distribution. The degree distribution of any TDG can be represented by its corresponding marginal in-degree and out-degree distributions. In this work, we use the degree distribution of the undirected graph in the same way as we discussed for average degree. The degree distribution measurements of Fig. 3 indicate that: (a) many TDGs (e.g., HTTP, DNS) exhibit skewed degree distributions. High variability typically denotes that most nodes have low degree, whereas few nodes have two to three orders of magnitude more edges than the average node degree, and (b) our measured quantities are stable over the intervals of observation since the empirical distributions from various disjoint 300 sec intervals of the trace are very close together. These plots are in double logarithmic axis to proper visualize all data, such logarithmic plots are going to be used extensively throughout the paper due to the high variability of measured data.

High-Variability: In Fig. 3, the corresponding average degrees and their standard deviations[4] are: (a) for HTTP, 2.48, with $\sigma = 32.96$, and (b) for DNS, 4.83, with $\sigma = 27.87$. For typical exponentially distributed data the Coefficient of Variation[5](CV) is close to 1, for large enough samples. However, the CVs for the HTTP and DNS TDGs are 13.3 and 5.8 respectively, indicating a significant level of variation from the average degree. Even though this is not the case for all our traces, the distributions of HTTP and DNS (Fig. 3) can be closely described by a power law relationship of the form $P(X > x) = 1 - P(X \leq x) \cong c \cdot x^{-\alpha}$, with exponents $\alpha = 1.10$ for HTTP and $\alpha = 1.27$ for DNS and goodness-of-fit (R^2) 0.98, 0.99 respectively. For such heavy-tailed probability distributions we know that empirical mean and variances are unreliable metrics to use (especially when these values are derived from sampled data). A better approach is to focus on the entire empirical distribution [22], as we do in the following sections. For completeness, we report that the average degree and standard deviations for WinMX and eDonkey are 7.6 with $\sigma = 12.2$ ($CV_{WinMX} = 1.6$) and 2.6 , $\sigma = 4.6$ ($CV_{eDonkey} = 1.8$) respectively. Clearly, even though not exponentially distributed, the degrees of these two protocols exhibit much smaller variability.

Joint Degree Distribution (JDD): JDD goes one step further than the degree distribution and reveals the probability $P(k_1, k_2)$ that a randomly selected edge will connect nodes of degree k_1 and k_2. Therefore, the joint degree distribution provides information about the directly adjacent neighbors of nodes in the graph. An extensive discussion and definitions regarding JDD can be found in [20] and [23]. The JDD for four TDGs derived from the first 300 sec interval of the OCS48 trace are shown in Fig. 10. Note that the contour plots have logarithmic x- and y-axis, as well as probabilities $P(k_1, k_2)$. *Example:* In Fig.10, the intersection of $x = 1.5$ and $y = 0.5$ give the probability of a randomly selected edge to connect nodes of degree $10^{\mathbf{1.5}} \approx 32$ and $10^{\mathbf{0.5}} \approx 3$. For example, in Fig. 10(d), this region falls in the 2^{nd} most dark colored area and by using the corresponding colormap we find that the probabilities are in the range: $|log_{10}P(32, 3)| \approx (2\ to\ 1.5) \Rightarrow P(32, 3) \approx 10^{-2}\ to\ 10^{-1.5} \cong 1\%\ to\ 3.2\%$.

Fig.10(c) graphically illustrates the JDD of a traditional client-server application such as HTTP. As expected, the region with the higher concentration of edges (darker region) is for low k_2 with high k_1 (and vise versa due to symmetry). The concentration of edges is gradually decreasing as we jointly increase k_1 and k_2. The white colored region at the top right corner indicates the zero probability of high degree nodes to be directly connected with each other. On the other hand, Fig. 10(d) quantifies what we originally observed in the P2P (WinMX) visualization of Fig. 2(f), where average degree nodes are connected with each other. This is shown with the darker colors of the contour plot placed in the middle of the graph, illustrating the prevalence of edges connecting "medium" degree nodes.

Rich Club Connectivity(RCC) metric: . The rich club connectivity is typically analyzed with the following procedure [24]. We sort nodes in the order of decreasing degree (x-axis in the plot) which we call the **degree rank** of the node.

[4]The Standard Deviation(σ) calculated here captures the variability of the degrees found in a single TDG and is different than the σ of the average degree across multiple TDGs as given in Table 2.

[5]The *Coefficient of Variation (CV)* of a random variable X is given by the formula $CV(X) = \sqrt{Var(X)}/E[X]$ and is a standard empirical-based measure for quantifying the variability of a random variable [21].

Then, we consider the group of nodes form node 1 to node k, and we find the number of edges that exist between these nodes, over the maximum possible number of edges between these nodes. In Fig.4, we plot the rich club connectivity for a TDG graph at port 53 (which corresponds to DNS) from the WIDE trace.

Interestingly in TDGs we have no clustering of high degree nodes. The peak at connectivity is where we have so many top nodes in the a graph that is closer to being a perfect clique. Always in TDGs the peak is for average degree nodes, showing that those hosts act as the connectivity hubs of the network. In Fig. 4 we observe a sharp spike at around degree rank 54. This means that the top 54 high degree nodes are not connected with each other at all. Then, when we consider a few nodes of degree rank from 55 to approximately 95, connectivity increases rapidly. By contrast, the rich club connectivity of most scale-free networks such as the Internet AS topology exhibits a pronounced rich club phenomenon as shown in Fig. 4 since the highest degree nodes form a clique. We observed the same phenomena for several other TDGs.

Before we provide an intuitive explanation, we have to remember that the data is observed at a single link. The high degree nodes, DNS servers, do not *appear* to exchange traffic with each other partly because they may be using caches, and partly because our point of observation may not be on the path between them. On the other hand, DNS clients can communicate with many servers, so the moment we start including them, the connectivity of the rich club starts to increase.

4 Using TDGs to Identify Concealed Applications and Malcode

We have seen that the TDGs of various applications have distinctive patterns and properties, and these patterns can be captured (Table 2) by various graph measures such as InO degree property, Max Depth, and degree distribution. In this section, we explore the next logical question: whether TDGs can be used to identify concealed applications and malcode. This is important because standard approaches based on identifying payload signatures [25] are not viable in the face of obfuscation (e.g., encrypted P2P communication and polymorphic worms) and are also processing intensive (to do string search at wire speeds). By contrast, application classification using TDGs requires only the parsing of packet headers, allowing implementation simplicity as well as the ability to deal with obfuscated payloads.

We first describe in Section 4.1 the ability of TDG graph measures to discriminate between applications. Next, in Section 4.2 we show the ability of similar TDG graph measures to detect outbreaks as well as concealed applications.

4.1 Application Identification

Given a set of TCP or UDP flows (filtered by say port numbers) passing through a monitoring device, the problem of application identification requires us to determine the type of application or protocol being used by the nodes in the community. Application identification techniques are useful, for example, in detecting concealed P2P applications which often use random ports. We address the problem of application identification by looking for distinguishing properties in the TDG of the set of flows using component distributions, degree distribution, and the InO property.

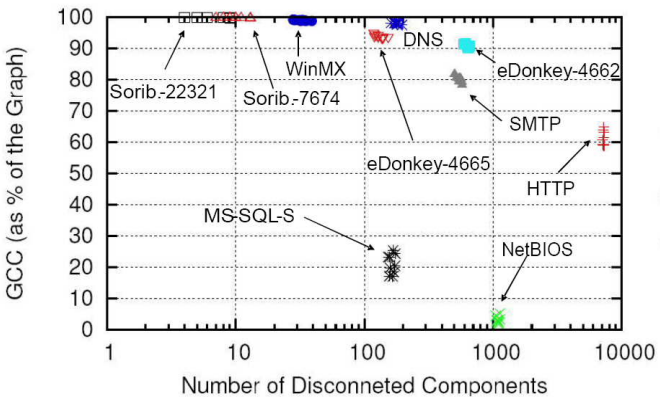


Figure 5: Scatter plot: Size of largest connected component versus the number of connected components per destination-port number for multiple intervals of the OC48 backbone trace. We show the top 10 most popular destination-ports, and speculate as to their application of origin. The simple combination of graph metrics separates fairly well the behavior of applications.

Characterization using Component Distributions: One distinctive characteristic of TDGs is the formation (or not) over time of a large connected component that concentrates the majority of participating IP hosts. Recall that in DNS we have a gigantic connected component that contains over 97% of the graph's nodes. On the other hand, the HTTP TDG has a large number of small disconnected components containing typically two or three hosts. This suggests using component measures to discriminate between applications.

In Fig. 5 we illustrate the discrimination afforded by component measures using a two dimensional scatter plot. The x-axis shows the number of disconnected component in log-scale, while the y-axis shows the percentage of nodes concentrated in the largest connected component GCC of the

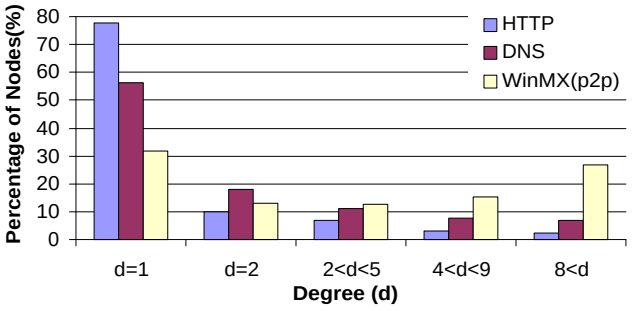


Figure 6: Groups of nodes based on their degree for the TDGs of HTTP, DNS and WinMX(P2P) in quickly increasing ranges (bins).

TDG. On the plot, we report the TDGs for the top 10 most active (in terms of number of generated 4-tuple flows) destination ports form our OC48 backbone trace.

In order to test our results over different time periods, we sliced the trace into twelve disjoint consecutive 300 second intervals. Therefore, for each of the ten TDGs we generated twelve points, one each for a 300 second disjoint time interval of the trace. It is reassuring to see that there is not much variation among the results for the twelve 300 second intervals because the twelve points for each port are clustered near each other. Once again this provides confidence about our choice of 300 sec to compute a TDG.

More importantly, the plot shows that we can approximately determine the type of the application based on its position in the scatter plot in Fig. 5. For example, the points gathered at the top-left part of the plot correspond to ports used by well known peer-to-peer applications [26]. No other application falls in this region, suggesting that our application identification technique is particularly suited for detecting concealed P2P applications. Also, at the lower-right section of the figure we see applications with many disjoint components and no single large component. Applications like NetBIOS (UDP Dst. Port 137) and ms-sql-s (TCP Dst. Port 1433) fall in this region.

Degree Distributions: While discrimination based on component measures appears promising, we will see in Section 5 that component measures are hard to infer from sampled (and hence low memory) versions of TDGs but that approximate degree distributions are easier to estimate. Fig. 2 shows similarity among DNS and SMTP TDGs, where the majority of hosts have low degree and few nodes have very high degrees, compared to WinMX(P2P) in which there exists a prevalence of medium degree nodes. We quantify this observation in Fig.6 with a bin distribution of the undirected degrees for various TDGs, demonstrating that different applications tend to have different node popularities. For example, for WinMX the bin containing nodes with degree $d > 8$ contains 25% of the nodes, which is much higher than the percentages for other popular applications (less than 10%).

Thus this information can be used to identify concealed P2P applications. It is also interesting to observe that even though DNS seems very similar to P2P applications with respect to component size distributions in Fig. 5 we can easily distinguish DNS and P2P using their degree distributions. This is because in DNS the majority of the nodes are connected due to some highly connected hubs (e.g., a large DNS .com server) whereas in P2P applications, the degree distribution is less skewed, with most of the peers having average popularity (Fig. 3).

InO Property: While degree distributions are useful, the InO property referred to earlier is also valuable for more strongly characterizing P2P applications and outbreaks. As with degree distribution, InO can be easily estimated from a sampled TDG. Recall we defined the InO property as the percentage of nodes in a TDG having both non-zero in and out degree. Fig. 7(a) shows the InO property for various TDGs of some the most popular destination ports of the OC48 backbone trace. As expected, P2P applications show a larger number of InO nodes (over 28% and 32% for WinMX and Soribada respectively). while a very low number of InO nodes is found in purely client-server applications such as HTTP[6].

Fig. 7(b) shows a scatter plot of the in-degree compared with out-degree of the nodes in eDonkey TDG and HTTP TDG respectively. Both TDGs were formed using the EFSP filter. The prevalence of InO in eDonkey clearly distinguishes it from HTTP where the few nodes with the InO property can either be Web caches or Web proxies.

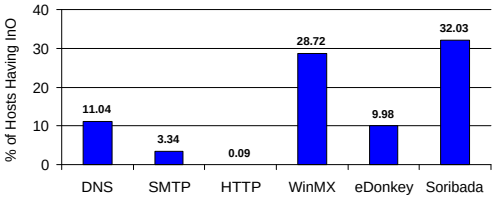
We also note that the percentage of nodes having only in-degrees or only out-degrees can be used to gain further insight into the interactions of the nodes. For example, if we observe an abnormally large number of degree one nodes, we can use the prevalence of in-degree or out-degree to distinguish DDoS attacks (where we observe a large number of out-degree only nodes), from worm scanning activity (where we observe a large number of in-degree only nodes).

Example: Let us see how TDGs can help in identifying a flow. An unclassified flow Z (with unregistered destination port number) exists between hosts X and Y. None of the two hosts has the InO property and they both have small degree. However, the two hosts belong to a TDG which has 20% of its nodes with the InO property, it also has a GCC that covers the 95% of all nodes and the majority of its nodes have degree higher than two. As we can see from Fig. 5, such highly connected communities with unregistered ports belong to P2P applications, similar conclusions can be derived from the degree distribution (Fig. 6) and from Fig. 7 due to high percentage of nodes with InO. Therefore, by examining nodes X and Y in isolation, we don't have any information regarding flow Z, however, by correlating them with an identified as P2P TDG we have an indication regarding the type of application that generated flow F.

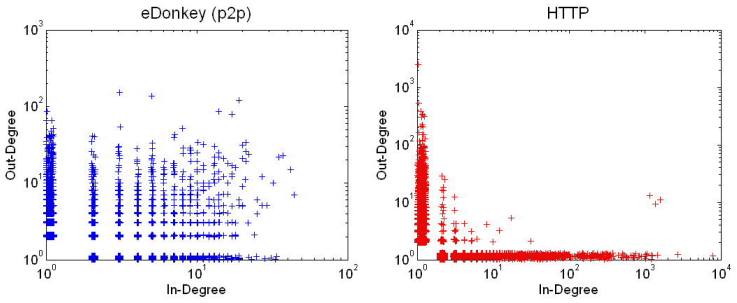
4.2 TDGs for Outbreak Detection

The problem of rapid detection of Internet worms and related address scan behavior is not new [25]. However, TDG's provide a good visualization mechanism to quickly identify such outbreaks; further, the same TDG measures used for application identification augmented with one further metric (Max Depth) can help separate benign port scanning, infectious outbreaks, and P2P applications. Note that the problem of separating P2P applications from say worms is a difficult one; for example, the automated worm signature scheme of [25] cites P2P applications as some of their worst false positives. We will see that a very simple graph measure, graph depth, accomplishes this separation. Note that the use of graph depth for worm identification was pointed out earlier by *Ellis et al.* [9] but we find that it is the combination of

[6]Similar results where found to hold for other traditional client-server applications such as FTP, POP3 and IMAP.



(a) InO is a highly discriminating metric for separating P2P applications. For SMTP, HTTP and eDonkey the EFSP filter was used. On the other hand WinMX, DNS and Soribada use UDP and their TDGs were created with the EFP filter.



(b) Scatter plot of the in- and out-degree of each node of the TDGs (EFSP links) of eDonkey (TCP) and HTTP. The scatter plots capture the correlation between the in- and out-degrees of hosts. To indicate the density of points in the 2-D plane, small random noise was added to each dimension in order for multiple overlapping points to be distinguishable.

Figure 7: In- and Out-degree correlations. All data are from the OC48 backbone trace.

InO and graph depth that works best. Further, we go further than *Ellis et al.* and show in Section 5 that both these metrics can be estimated efficiently in hardware at 20 Gbps.

Visualizing Outbreaks: We start in Fig. 8 with a visualization of the TDG of the MS-Blaster infection. The UCSD trace was captured by a real Blaster infection in a controlled honeypot environment. While the trace is only 150 nodes, it is reflective of a TDG collected within an enterprise because it shows connectivity between all nodes as opposed to the bipartite TDG graphs we showed for our access and backbone traces. Also, unlike the Slammer TDGs shown earlier which has several disconnected Star subgraphs, this trace has actual infections and not just Slammer scanning. Notice the clear tree like structure with a large depth and high InO.

P2P versus Outbreaks using Depth: In Table 3 we compare the standard graph metrics for 3 TDGs: the first is the WinMX TDG (P2P) from the wide area traces, the second is the Slammer trace from the wide area traces (scanning but no infection), and the third is the Blaster trace we described (scanning plus infection). Notice that the Max Depth for Slammer with scanning is 1, the Max Depth for WinMX is 4, while the Max Depth for Blaster is 8 even for a very small TDG! Notice also the high InO for Blaster and WinMX that separates them from Slammer scanning. Notice also the high max degree for Slammer scanning (1248).

The Table suggests that high Depth and high InO are strong characteristics of outbreaks. On the other hand, high average and Max degree with small InO and very small depth (close to 1) is characteristic of scanning. Finally, high InO (10 to 20%), moderate depth (2 to 4), and fairly large node degrees (> 8) is characteristic of P2P applications.

The benefit of using TDG's over Ellis' approach [9] is that we use a more refined discriminator (Average Degree, InO, Max Depth) and we calculate all of these using a single TDG as opposed to the per node connectivity graph suggested in [9]. Further, the same TDG can be used for application identification and not just for worm detection. Finally, we will show in Section 5 that these metrics can be calculated efficiently without even storing the entire TDG: instead, they can be approximated well using an order of magnitude less storage by employing sampled TDGs.

Depth in Enterprise versus Access Link Traces: While enterprise traces are the preferred data engine to generate TDGs for application and outbreak identification, ISPs may only have access to access traces. The notion of depth in an access trace needs to be modified compared to the standard notion in [9]. This is because the access link only sees infection attempts coming from outside the organization to inside and vice versa. If a successful infection attempt comes in, the access link will not see the resulting internal infection, but can only observe the rare case when the same internally infected node then turns around and sends an infection attempt to the outside. This suggests that the Max Depth observed in such access is likely to be 2 which is very different from the numbers like 8 observed in our Blaster trace and in [9]. We have carried out simulations that verify this and show there will also be a significant number of nodes with depth 2 during an infection but we omit these results for lack of space.

5 Sampling TDGs for Hardware Implementation: Theory and Practice

A hardware implementation of TDGs must minimize the amount of high speed memory devoted to storing TDG state. TDGs can be large, with the number of nodes equal to the number of IP addresses being monitored (say 64,000 in a large enterprise). A second multiplicative factor is that several different TDGs may need to be monitored concurrently (e.g., for several port numbers). Bloom filter representations can be used to compress the TDG by a constant factor of around 3 (the space required to store a node reduces from say 32 bits per node to around 10 bits per node [27]). For an

	Nodes	*Edges*	*Avg. Degree*	*In-&-Out(%)*	*MAX Degree*	*MAX Deg. Ratio*	*Size of GCC(%)*	*Avg Depth*	*MAX Depth*
OC48-WinMX	8,890	33,593	7.560	28.68	315	0.035	98.99	1.273	4
WIDE-Slammer	8,209	7,921	1.929	0	1248	0.0685	85.56	1	1
UCSD-Blaster	239	246	2.0587	21.56	30	0.1255	100	4.369	8

Table 3: Comparing the standard graph metrics for WinMX, Slammer(scanning) and Blaster(infection) TDGs.

order of magnitude reduction, the required graph measures (e.g., InO, degree distribution, MaxDepth) must be estimated accurately from a small sample of each TDG.

Unfortunately, most basic graph properties are hard to estimate in the *data stream* setting where edges are simply passing by [28, 29]. Moreover, we require an approach that is implementable in router hardware. We therefore propose to use *online graph sampling* to reduce storage and restrict ourselves to graph measures that can be efficiently calculated using such sampling.

For concreteness, consider the InO property defined earlier: If we can uniformly sample nodes from the TDG graph, we can approximate the fraction of nodes which satisfy the InO property. For example, suppose the TDG has 64,000 nodes and say 10% of the nodes have the InO property. First, we randomly sample and store a small number of TDG nodes; then, we watch the data stream for in-edges or out-edges for the sample nodes, and record with one bit if in-edges or out-edges exist. At the end of some specified period, the algorithm can calculate the fraction of nodes in the random sample that have the InO property. For example, if 20 nodes are found with this property in a sample of 250, a reasonable estimate for the whole graph is $20/250$. As in the Gallup Poll, the distribution for the random sample follows a Bernoulli distribution that tends (by the Central Limit Theorem) to the normal distribution. Since the standard deviation tends to fall off as the square root of the sample size, fairly small samples give small confidence intervals.

Concrete results can also be obtained by using one of a variety of standard Chernoff bounds, although with a small number of samples they may not be sufficiently tight. For example, if we choose m sample nodes from a graph independently and uniformly at random and find a fraction $\hat{p}$ of the nodes have some property, and we consider the actual fraction p of the nodes that have that property, we find (see, e.g., [30])

$$\Pr(p - \hat{p} \geq \epsilon) \leq \left(\frac{e^{-\epsilon/p}}{(1-\epsilon/p)^{(1-\epsilon/p)}} \right)^{mp}. \quad (1)$$

(Here we choose distinct nodes, without replacement; the bound still holds.) For example, if at most 20 nodes out of 250 appear to be spreaders, this gives that the actual fraction of spreaders is more than 16% with probability about than 0.2%; analysis of the Bernoulli distribution gives a stronger bound of 0.015%. In practice, good and easy to compute approximations can also be had using normal approximations to the binomial distribution.

We have tested the effectiveness of sampling on our traces from both backbone and access links, calculating the fraction of nodes that have the InO property for various TDGs. We found, as expected, that results with a random sample of approximately 1% of the nodes matched very closely the results using the whole graph. For example, for DNS on the OC48 trace, the InO degree fraction was 0.1097 for the whole graph (22,710 nodes) but the estimator with 225 sampled nodes was 0.1037 with a 95% confidence interval of 0.0092.

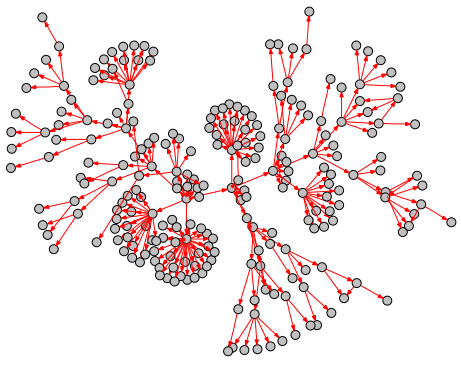


Figure 8: The tree-like structure of MS-Blaster Worm infection.

5.1 Sampling via Wegman's Algorithm

While these results are encouraging, the really hard part, as with the Gallup poll, is *how* to pick a *random* sample. The problem is particularly challenging in the network setting because **(a)** the TDG nodes arrive in online fashion (every received packet potentially adds an edge) **(b)** the size of the TDG is unknown and can vary dramatically **(c)** some packets duplicate already existing nodes and edges.

Suppose one wants to keep track of 250 randomly sampled nodes from a TDG. Unfortunately, the TDG may vary in node size from say 64,000 (all to say 100. If we knew the size was 64,000 in advance then we could pick a node for the sample with probability 250/ 64,000. Without knowing the population size in advance, a simple online algorithm that does the job is reservoir sampling [31]. Unfortunately, reservoir sampling does not really work in a setting where the online algorithm receives the same node multiple times. Since no memory is kept of nodes that have already been discarded, a node that has been discarded earlier can be added again if it appears as a duplicate.

Essentially, we have a multiset of incoming values (node IP addresses) and we wish to sample s (e.g., 250) distinct values. Wegman [32] has proposed an elegant reservoir algorithm to sample from multisets using close to optimal

storage. Wegman's algorithm uses a single reservoir and an adaptive sampling probability. The intuition is that hash based sampling (which suppressed duplicates) is done at some probability p and samples are stored in the reservoir. When the reservoir fills up, the algorithm moves to a sampling probability of $p/2$. To make space for new incoming samples, Wegman's algorithm goes through the existing reservoir and removes all currently stored samples that would have not been accepted had the sampling probability been $p/2$.

More precisely, the algorithm samples a node with probability $p = 2^{-l}$ only storing node IDs X such that the last $l = \log_2(1/p)$ bits of $Hash(X)$ are 0. A node ID X is also stored only if X is not already in the reservoir. For example, to sample with probability 1, we select all nodes ($l = 0$), to sample with probability $1/4$ we select all nodes whose hash values end with 00 ($l = 2$). Initially, $p = 1$. At any stage when the reservoir (say of size 250 as described above) fills up, p is halved by incrementing l. When this happens, Wegman's algorithm requires going through the reservoir and removing all nodes whose last l bits are not equal to 0. Since such nodes were in the sample because their last $l - 1$ bits were 0, their l-th bit must be a 1. Since this happens with probability half, this deletion has the effect of stochastically splitting the sample, leaving roughly half the reservoir free to store new samples at the new sampling probability.

The sudden halving of the sample size can leave the resultant sample off by a factor of approximately 2. For example, if the reservoir can hold 250 nodes, the algorithm may end up with a sample of 125 (or fewer) by the halving operation when the last distinct sample is received. This is because the halving is done on behalf of possibly future samples, and the algorithm, being online, cannot know that no more samples exist. Also, the halving is approximate and may sometimes remove more than half of the reservoir. Wegman [32] proves, however, that if the reservoir size is roughly 2.5 times the size of the required sample, then with high probability the algorithm will end with an unbiased sample of at least the required size (which can then be subsampled if an exact size is desired). Thus for a final sample of size 250, one has to keep a reservoir of roughly 625.

It is worth noting the following key property of Wegman's algorithm that we use later: if a node is seen and *not* put in the reservoir, duplicates of that node will never be put in at a later time. Hence, in the case of an online data stream, we know that an element in the reservoir was there since its first appearance in the stream.

High Speed Sampling Implementation: Wegman's algorithm has two major issues for high-speed implementation. First, the reservoir must implement some form of CAM or hash lookup to prevent the same sample X from being stored twice. Since CAMs take at least 5 times the transistors, a hash table implementation is better. Second, the effort required to reduce the size of the reservoir on a specific sample is much too large. Recall that we care about the worst-case packet processing overhead.

We solve the first problem using a hash table adapted to high speed processing. For example, d-left hashing uses d parallel hash tables each of which has b slots in each bucket. A new ID is hashed in parallel into a bucket in each of the d hash tables using d independent hash functions. The ID is stored in any vacant slot in the least loaded bucket breaking ties to the left. Search proceeds similarly, but all slots in each relevant bucket is searched in parallel. Although this requires a small increase in storage (a usual loading factor would be 70%), the probability of rejecting a new entry is very small. Because of the bounded processing and parallelism, d-left is easy to implement in hardware and is, indeed, used in routers today.

The second problem could be solved using a separate garbage collection process that periodically walks through the hash table entries marking entries that would be deleted if the sampling probability halves. Unfortunately, these elements cannot be deleted before the reservoir fills up, and deleting them in constant time after the halving requires marked nodes to be linked together which is problematic with hash tables.

The alternative we propose is to modify Wegman's algorithm to do lazy deletion. When p is halved and l is increased, we do nothing special. However, on any insert, when we retrieve d buckets in parallel and search among their slots looking for empty slots, we consider a bucket empty even if there is an entry X in the bucket but $Hash(X)$ does not have the last l bits equal to 0.

The only remaining issue is knowing when to halve the sampling probability p. Since Wegman's algorithm does a distinct deletion step, it can keep track of the number of distinct samples k and halve p when $k > R$, where R is the reservoir size. Unfortunately, with lazy deletion, we have no count as to the number of distinct samples. However, we can keep track of the number of distinct samples added, say k', after the last probability doubling. The next doubling is done is done in the modified algorithm when $k' > R/2$.

This slightly increases the probability that the reservoir will not be able to hold a valid incoming sample when, for example, the last deletion step deleted less than half of the reservoir. When an incoming sample has no space to be stored in the d-left hash table because of d-left hashing overflow a new halving should be initiated. Proper sizing of the hash table, however, avoids this problem.

We note that the factor of two waste in Wegman's algorithm can be reduced by using different downsampling factors other than 2, so that the size gap between the reservoir before and after the change in p value is less severe. For example, on a downsampling one could check if the next two bits of the hash were 00, 01, or 10, reducing the sample by a factor of $3/4$. One could also use more general values of p although this is somewhat more difficult in a bit-based hardware architecture. If we store node IDs in the hash structure this change requires no additional space in the hash table, but

only longer hashes (in bits). However, even with the simpler factor of 2 waste, it seems clear that one could feasibly implement the modified Wedgman algorithm for around 1000 concurrent TDGs in hardware at say 20 Gbps.

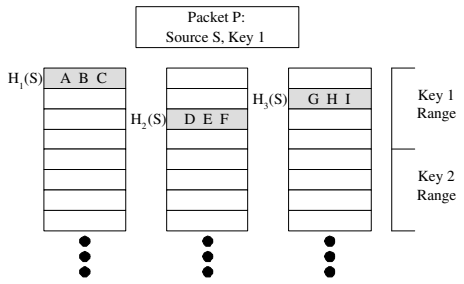


Figure 9: The modified Wegman algorithm using d-left hashing and lazy deletion and assuming node sampling of sources. When a packet comes in with a TDG key $Key1$, the key is used to find an offset into the overall d-left hash table. If a hash $H(S)$ of the source has l consecutive suffix bits equal to 0, S is inserted into the sample by inserting into the d-left hash table. Lazy deletion is done on insertion.

5.2 Graph Properties via Sampling

Fig, 9 shows an example where source S in a packet with TDG key K (which, recall, may correspond to a destination port number or content string and allows the packet to pass the edge filter, Section 2) is inserted into the sample. Assume that currently $l = 2$ and $H(S)$ has the last 2 bits equal to 0. Then S is hashed using say $d = 3$ hash functions $H_1(S)$, $H_2(S)$ and $H_3(S)$ and the corresponding buckets are retrieved. Each bucket has say 3 slots to hold nodes. Since S is not already in any slot, S should be inserted. Before inserting S, any implicit deletions are done. We know that if the last l bits of the hash of any node ID in any bucket is not 0 then that node can be deleted. For example, suppose $H(A) = 000100, H(B) = 000111, H(C) = 101100$, and $H(D) = 111011, H(E) = 111000, H(F) = 111111$, and $H(G) = 011000, H(I) = 101111, H(J) = 111101$. Then B can be deleted from the first bucket, D and F can be deleted from the second bucket, and I and J from the third bucket. After deletions, the least loaded buckets are the second and third buckets. Breaking ties to the left (this is where the phrase d-left comes from), S is inserted into the second bucket replacing either D or F. All of these steps can be done in parallel by reasonably sized combinatorial logic. Notice also that the same logic can be used for all TDG keys by dividing a large d-left hash table into subtables for each key and using a lookup on the key to map to an offset. Then the offset must be added to the value returned by the hash functions used for indexing ($H_1(), H_2(), H_3()$ in 9) to retrieve the corresponding bucket for a key.

We now consider what kinds of measures can be accurately determined via sampling based on Wegman's algorithm. It appears that estimating component size is hard in the online setting; however, the other measures (Degrees, InO, and Depth) have fairly compact implementations.

Node and Edges: Wegman's algorithm can be used to take a uniform random sample of distinct nodes or distinct edges. Any property of nodes or edges that can then be determined easily offline, based on the sample taken.

Node Degrees: Using Wegman's algorithm, one can sample nodes independently and uniformly at random. Because each node, if it is to appear in the final sample, is placed into the sample *on its first appearance*, we can keep counters for each node ID in the sample to track its in- and out-degree. This is essentially a generalization of the approach we described to obtain an estimate of the InO property.

We can further take advantage of the *temporal* nature of the stream to correctly find the fraction of spreaders, where a spreader sends a message out *after* receiving one. For this, we assume our stream correctly arrives in temporal order; that is, we see an edge (A, B) in the stream in the correct relative temporal order for when node A sent a message to node B. We then only set the out-degree bit for a node if its in-degree bit is already set; this properly labels nodes as spreaders if they send a message after receiving one.

Degree Distributions: With small samples as we aim for here, attempting to determine the entire degree distribution by sampling node degrees is impractical. We can, however, use the sampling to obtain a coarse histogram of the node degrees. We start a priori with bounds $a_1, a_2, \ldots, a_b$ for some small b; after sampling, we determine the fraction of nodes with degree $d \leq a_1$, with degree $a_1 < d \leq a_2$, and so on. By splitting nodes into a small number of groups, we can ensure the Bernoulli/Chernoff bounds are meaningful in this context. We have found that five groups with quickly increasing ranges performs well in our experiments (Fig. 6).

Random Temporal Walks to Estimate Depth: To estimate depth, we start by sampling node temporal out-components: choose nodes randomly, and then find all nodes potentially reached via a sequence of messages from that node. Keeping an entire node component is rather expensive in terms of space. A natural substitute is to keep a random walk from a given node. For example, when we sample a random node A, we select uniformly a random out-neighbor A_2 of A to follow. We subsequently choose a random out-neighbor A_3 of A_2 to follow, and so on. We again emphasize that we are restricted to out-neighbors that occur in the proper temporal order. Interestingly, we can sample such random walks even in the limited on-line setting.

The key is to keep temporary walks as we go; these walks may be replaced in part or in whole at any step, by using reservoir sampling at each node in the path. For example, suppose we have a current path from A consisting of A, A_2, A_3, A_4. For each node on the path, we need to know its out-degree up to this point in time. Suppose now a node (A_2, B_3) arrives in the stream. According to reservoir sampling we should replace the edge (A_2, A_3) in the path by (A_2, B_3) with probability $1/(\text{out-degree}(A_2) + 1)$; if this occurs, we change our walk to A, A_2, B_3, deleting the final edge.

Note that for this to work using reservoir sampling, we must take care; for example, it seems we should keep a list of out-edges from each node in the path, to avoid problems caused by the same edge appearing multiple times. (Wegman's algorithm is highly *inefficient* for keeping a *single* sample, so we cannot use it effectively here.) A better alternative is to keep edges adjacent to any node ever made part of a path in a global Bloom filter; then there is some chance that a new edge is not included in a walk because of false positive, but as the scheme is already randomized and approximate the noise introduced by this should be small. Note that the space used by this filter is not proportional to the size of the complete graph, but just to the much smaller number of edges we have to track through this process.

We may think of sampling random temporal paths as a way of getting some insight into the depth of the graph, starting at certain nodes. While the paths are a fairly weak substitute, it is difficult to approximate depth or related graph properties in small amounts of space. If one is interested in the depth from certain nodes, instead of considering the depth of the final random walk, one could consider the largest depth found from a node during this process. Alternatively, one could consider other replacement rules (in order to lessen the chance of shortening a currently long path, for instance). Also, if one is interested in depth, it is best not to choose a starting node uniformly at random. For example, consider a complete binary tree on a levels. A random node is likely to be near the bottom of the tree, yielding a short path; starting near the root, one will obtain a path of length nearly a.

For example, in the Blaster TDG, while only a few nodes have depth 8, roughly 60% of the nodes had depth 5. While a random walk technique is highly unlikely to find a depth of 8, our simulations over the UCSD Blaster trace, show it is very likely to find many nodes with depth within the range 2-5. However, we omit these results for lack of space. Thus the sampled TDG estimate of depth can be used but the value of depth used to discriminate outbreaks from say P2P traffic must be set lower. Once again, this suggests why it is so useful to obtain additional evidence from other measures such as InO and degree distributions.

Implementation Summary: Overall, other than component size and assortativity, all the other metrics described for application and outbreak in Section 4 can be estimated by storing only a very small amount of samples (say 250 nodes) for a TDG. Assuming a factor of 2 increase in storage for the hash table and Wegman implementation, and 64 bits per node (32 bits ID, 32 bits of state), this is roughly 32K bits per TDG. Assuming 10 Mbits of high speed memory, this provides space for monitoring 320 TDGs concurrently using 3 parallel memory accesses per packet. This can easily be implemented at 20 Gbps.

6 Related Work

We present an overview of related literature and compare it with our work. The overview highlights the most relevant work and indicative efforts across the different areas that our paper brings together.

Application classification and abnormality detection. Early efforts focused on classifying network flows using the well-known[7] port numbers [13]. However, recent studies show that such methods fail in correctly classifying new applications, like P2P protocols, that use ephemeral port numbers [33, 34]. In order to overcome this problem, deep packet inspection approaches were used in order to capture the invariant signatures of network applications [35, 36, 2]. High processing overhead, sensitive privacy issues and failure in the presence of payload encryption are the main disadvantages of this approach.

To increase classification accuracy more information regarding each flow must be extracted. By grouping packets into flows, metrics such as: packet interarrivals, average packet length, total bytes transferred and flow duration can be additionally used. This level of aggregation was utilized by a plethora of Machine Learning (ML) classification methods such as [37, 38, 39, 40, 41, 42] and the references therein. The most important challenge of these approaches is to identify the right set of attributes that will provide robust results in low computational cost. In general ML methods suffer from high processing overhead which renders them impractical for real-time classification at Gigabit speeds.

User behavior profiling gives an additional level of aggregation by grouping the set of flows belonging to a particular end host. BLINC [6] works by characterizing the "social behavior" of hosts at the Transport Layer, and henceforth accordingly labeling their flows. For example, if a host is found to participate in a P2P application, then all its flows that follow a particular behavior-profile are characterized as carrying P2P traffic. In order to achieve that, BLINC uses graph models called graphlets to capture the behavior of a *single* node. A fundamental difference with our work is that, in graphlets, port numbers appear as nodes in the graph. Thus, graphlets do not represent *network-wide* host interaction as we use them here. We emphasize that BLINC hints at the benefit of analyzing the node interaction at the "social" level, but it ultimately follows a different path focusing on the behavior of one node at a time.

Other related work towards the direction of host behavior profiling are [26, 5, 7]. In [5] clusters of interest are first extracted from multidimensional flow records and later modeled so as to represent various behaviors. To summarize, the basic concept of these approaches advocate towards the increase of classification accuracy, by using prior knowledge regarding the behavior of an end-host, when trying to classify its flows.

Our Approach: Our approach goes one step further than the host-based granularity and operates on aggregated sets (graphs) of related nodes. This allows us to extract information such as the popularity distribution of the set, the component distribution and all other features described in Section 3. After an edge filter (Section 2) is defined, the TDG can be

[7] For example, Port 80 denotes Web traffic.

constructed and then characterized. For example, Sumeet et al. in their evaluation of EarlyBird [3] automatic worm-signature extraction system, reported that a large number of false positives was caused by the *BitTorrent* P2P application. Our approach can be used to mitigate this number of false positive. In order to achieve that, we can first generate the TDG using as edges all packets carrying the derived signature of *BitTorrent*. An automated IDS can then characterize the derived TDG and assert if the observed behavior (abnormal content prevalence) is due to a worm outbreak or a benign application (e.g., *BitTorrent*).

Another advantage of our approach is that by monitoring a group of nodes, even if the behavior and characteristics of an individual node changes over a new window of observation, the behavior of the community as a whole will in general remain stable. Therefore, our proposed level of aggregation also deals with the variability of monitoring a single node's behavior at a time.

Network Monitoring and Visualization of high dimensional network data. Network monitoring, based on visualization of high dimensionality traffic data, is becoming of increasing importance for network management solutions. The problem is due to the huge volume of packets and flows that traverse today's IP networks making it computationally impossible to process, keep state and provide information for millions of connections. Thankfully, practice shows that the majority of connections carry benign content and most of the time the global traffic behavior is "normal". Therefore, the focus of network monitoring solutions is now shifted in filtering-out regular traffic behaviors and be able to spot and visualize abnormalities, like the *Autofocus* tool [43].

Yin et al. [44] presents a tool for near-real-time visualization of network NetFlow data. The grouping of nodes in the graph is performed based on the traffic volumes exchanged between them. In [45], the authors use manifold learning to visualize, in low dimension scatter plots (2-D or 3-D), very high-dimensional NetFlow traffic measurements. A plethora of works aiming at visualization of network data can also be found in the same paper [45]. The work in [46] belongs to the family of graph visualization literature presenting a visually friendly methodology of projecting individual nodes to a low dimensional (2-D) scatter plot. All these works can be used in parallel to our approach in order to enhance and complement the visual (e.g., Fig.2) articulation of our scheme.

Worm and virus detection. Although there has been a lot of work on worm and virus detection [47], most such work relies on the identification of signatures in the content of the worm, and thus differs significantly from the work here.

An earlier approach to use graph-based methods was proposed in 1996 [48, 8]. This early effort inspired the most relevant work to our effort [49]. Ellis et al. [49] employed host interaction graphs for worm detection within an enterprise network. By assuming global knowledge of packets exchanged, they constructed host interaction graphs where nodes consisted of local (intra-enterprise) end-hosts and links between nodes were formed upon fulfillment of some link predicate. Their key contribution is based on revealing the importance of a three-like communication structure, formed by a self propagating code, in the detection and containment of such threat.

The original work of [49] was later continued in [9] with the designing of a software Intrusion Detection System (IDS) aiming at real-time detection of any worm-generated tree-form structure. In contrast to our work, the authors in [9] only consider the case where global knowledge of all packets between all hosts is known and they focused their work on detection of worm activities. Namely, they did not generalize any findings in the concept of traffic monitoring, or application specific social profiling. In [50], the authors provide preliminary results illustrating the importance of the temporal appearance of hosts in a trace, for identifying P2P communities. However, none of these work uses the diversity of graph metrics we introduce nor they have hardware primitives taken under consideration.

Identifying the origin of the attack. Post-mortem trace analysis for the identification of a worm outbreak and the discovery of its origin was the focus of study in [51] and later in [10]. Both works exploit the tree-like structure of a propagating worm, but they only focus on offline forensic analysis while they explicitly target worm detection. Tolle et al. [52] also reveal the importance of host communication graphs in network monitoring and intrusion detection. In [52], they extract patterns from the traffic matrix of an enterprise network and then use clustering to group intra-communicating node in order to visualize the inter-cluster-graph. Again, no hardware primitives were introduced and only a simple link predicate of "A sends an IP packet to B" was used.

Communities of Interest in Data Networks. The study of Community of Interests (CoI) in data networks [53, 54] targets on capturing historical communication patterns of hosts, and then use them to build a model for "normal behavior". Deviations from normal can then be used to trigger an alarm. Another main focus of CoI is to achieve automatic extraction of significant communities from a set of communicating hosts (e.g., within an enterprise network). If the extraction step leads to meaningful communities, that follow a great deal of regularity and structure, then this information can be used to form parsimonious models which can constitute the foundation of a management policy [53].

These basic concepts were used in [55] for the design of a CoI based enterprise security scheme. In [55], the profile of users is derived together with thresholds that control the allowed deviation from "normal" host behavior. This ultimately makes it harder for a malware to exploit vulnerabilities in the enterprise. These methods are quite different from our approach. In our work, we first extract the communication graph of hosts, based on some edge-filter, and then use graph metrics in order to characterize the nature of the network application being used (e.g., if is a peer-to-peer or a client-server application).

Research in social network and other disciplines. Social graph analysis has drawn the attention of a plethora of researchers in social and business studies, and datamining. An exhaustive survey can be found in several books and recent articles [19] [56]. In [57], communication patterns of call-graphs, from a cellular telephony company, are analyzed in view of profiling the behavior of customers. The authors in [58] mine the inner structure of the Web graph[8] that can be further used for devising better crawling and browsing strategies. Social graph methodologies were also studied in business interaction graphs [59], where customer communication patterns are used in order to increase the profitability an organization. Similarly, graph mining teachings are used in [60] for the extraction and understanding of complex purchase behaviors by consumers. The herein referenced works cover some of the latest publications in the otherwise enormous context of social graph analysis.

The communication graph of e-mail users is studied in [61] as well as in other works referenced therein. Drineas et al. [61] present a brief study of the spectral characteristics of an e-mail graph revealing the message exchange patters from a small community of university users. Their work support similar finding as in [53] where the social behavior of hosts is found to be stable over time. However, Drineas used e-mail log files ranging across multiple days and is therefore different from our study of packet level behavior of hosts.

Finally, note that TDGs, as defined here, are significantly different than trust propagation networks. The problem there is to identify intruders in social networks, which represent trust relationships as discussed in the recent work of SybilGuard [62]. For example, the solution there relies heavily on the fact that in trust relationships a "bad" guy cannot have many trust edges with good guys, which is true in trust networks, where nodes are humans, like ebay sellers. However, this is not true when nodes are IP machines. By contrast, an edge in a TDG can be an unsolicited transmissions of a packet

7 Conclusions

Two essential features in a network monitoring tool dealing with vast amounts of network data are *aggregation* and *the ability to spot patterns*. TDGs represent a natural extension of previous approaches that have aggregated at the packet, flow, and host levels by aggregating across nodes. The aggregation across nodes also reveals patterns of social interaction across nodes that are specific to applications. These interaction patterns or graph structures can then be used to visually and quantitatively monitor existing applications and detect concealed applications and malcode.

Our paper has the following major findings. First, TDGs are interesting as objects of study in their own right, with degree distributions and rich club connectivity that is very different from other scale-free graphs such as Internet topology. Second, TDGs can provide a visualization perspective that complements other perspectives (such as the visualization of flow data in tool such as Plonka's FlowScan). Third, TDGs of different applications and malcode can be effectively discriminated using a small number of metrics such as average and max degree, InO, max depth, and the number and size of components. Fourth, many of these metrics (component sizes being a notable exception) can be computed at high speeds using very small samples of a TDG. Fifth, it is difficult to extrapolate results from offline sampling of graphs to online sampling; for example, assortativity is easy to compute offline but is difficult to estimate online. Despite this, the metrics that can be efficiently estimated (degree distributions, InO, and max depth) suffice to discriminate many interesting features such as P2P behavior and both dormant (scanning) and active infections.

Ultimately, these measures could be input as features to a learning algorithm that could be implemented offline in a management station based on data from sampled TDGs supplied by routers. The algorithm could be trained with the features of known applications to then predict concealed applications. Such an algorithm could help move beyond the visual appeal of TDGs to a more rigorous science for monitoring and detecting applications.

[8] Nodes are web pages and the directed links between pages are hyperlinks connecting them.

8 Acknowledgments

The authors thank Flavio Bonomi and Ronak Desai, from Cisco Systems, Inc., for their support throughout this work. Support for this work was provided by a Cisco URP grant.

References

[1] Susan Orr. DDoS Threatens Financial Institutions, 2005. www.cisco.com/web/strategy/docs/finance/fin_DOSS.pdf.

[2] Justin Ma, Kirill Levchenko, Christian Kreibich, Stefan Savage, and Geoffrey M. Voelker. Unexpected means of protocol inference. In *ACM IMC*, 2006.

[3] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *USENIX OSDI*, 2004.

[4] A. Lakhina, M. Crovella, and Christophe Diot. Mining Anomalies Using Traffic Feature Distributions. In *ACM SIGCOMM*, 2005.

[5] K. Xu, Z. Zhang, and S. Bhattacharyya. Profiling Internet Backbone Traffic: Behavior Models and Applications. In *ACM SIGCOMM*, 2005.

[6] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. BLINC: Multilevel Traffic Classification in the Dark. In *ACM SIGCOMM*, 2005.

[7] T. Karagiannis, D. Papagiannaki, N. Taft, and M. Faloutsos. Profiling the end host. In *PAM*, 2007.

[8] S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, J. Rowe, S. Staniford-Chen, R. Yip and D. Zerkle. The Design of GrIDS: A Graph-Based Intrusion Detection System. *UCD Technical Report CSE-99-2*, 1999.

[9] D. Ellis, J. Aiken, A. McLeod, and D. Keppler. Graph-based worm detection on operational enterprise networks. *Technical Report MITRE Corporation*, 2006.

[10] Y. Xie, V. Sekar, D. Maltz, M. Reiter, and H. Zhan. Forensic analysis of epidemic attacks in federated networks. In *IEEE ICNP*, 2006.

[11] Mazu Networks, http://www.mazunetworks.com.

[12] Arbor Networks, http://www.arbornetworks.com.

[13] CAIDA The CoralReef Project. http://www.caida.org/tools/measurement/coralreef/.

[14] CAIDA. Skitter project. http://www.caida.org/tools/measurement/skitter/.

[15] CAIDA OC48 Trace Project 20030115-100000-0-anon.pcap. http://imdc.datcat.org/data/1-3N3T-4=20030115-100000-0-anon.pcap+%2860+min%29.

[16] MAWI Project from the WIDE Backbone. http://csl.sony.co.jp/mawi/.

[17] National Laboratory for Applied Network Research (NLANR) http://www.nlanr.net.

[18] Graphviz – graph visualization software. http://www.graphviz.org/.

[19] Mark Newman, Albert-Laszlo Barabasi, and Duncan J. Watt. *The Structure and Dynamics of Networks*. Princeton Press, 2006.

[20] P. Mahadevan, D. Krioukov, B. Huffaker, X. Dimitropoulos, kc claffy, and A. Vahdat. The Internet AS-level topology: Three data sources and one definitive metric. In *ACM SIGCOMM CCR*, 36(1), 2006.

[21] Lun Li, David Alderson, Reiko Tanaka, John C. Doyle, and Walter Willinger. Towards a Theory of Scale-Free Graphs: Definition, Properties, and Implications. In *Internet Mathematics*, 2005.

[22] Mark Crovella and Balachander Krishnamurthy. *Internet Measurement: Infrastructure, Traffic and Applications*. John Wiley and Sons, Inc, 2006.

[23] P. Mahadevan, D. Krioukov, K. Fall, and A. Vahdat. Systematic Topology Analysis and Generation Using Degree Correlations . In *ACM SIGCOMM*, 2006.

[24] M. Faloutsos Q. Yang, G. Siganos and S. Lonardi. Evolution versus Intelligent Design: Comparing the Topology of Protein-Protein Interaction Networks to the Internet. In *LSS CSB*, 2006.

[25] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cuts. In *ACM SIGCOMM*, 2003.

[26] T. Karagiannis, A. Broido, M. Faloutsos, and kc claffy. Transport layer identification of p2p traffic. In *ACM SIGCOMM/USENIX IMC*, 2004.

[27] A. Broder and M. Mitzenmacher. Network Applications of Bloom Filters: A Survey. In *Internet Mathematics*, 1(4):485–509, 2004.

[28] N. Alon, Y. Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *ACM STOC*, pages 20–29, 1996.

[29] S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Now Publishers, 2005.

[30] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[31] J. Vitter. Random sampling with a reservoir. In *ACM Transactions on Mathematical Software*, 1985.

[32] Michael J. Maher, editor. *Advances in Computer Science - ASIAN 2004.*, Vol. 3321 *Lecture Notes in Computer Science*. Springer, 2004.

[33] T. Karagiannis, A. Broido, N. Brownlee, kc claffy, and M. Faloutsos. Is p2p dying or just hiding? In *IEEE Globecom*, 2004.

[34] Andrew Moore and Konstantina Papagiannaki. Toward the accurate identification of network applications. In *PAM*, 2005.

[35] Subhabrata Sen, Oliver Spatscheck, and Dongmei Wang. Accurate, scalable in-network identification of p2p traffic using application signatures. In *WWW*, 2004.

[36] Patrick Haffner, Subhabrata Sen, Oliver Spatscheck, and Dongmei Wang. Acas: Automated construction of application signatures. In *ACM SIGCOMM MineNet Workshop*, 2005.

[37] Augustin Soule, Kave Salamatian, Nina Taft, Richard Emilion, and Konstantina Papagiannaki. Flow classification by histograms: or how to go on safari in the Internet. In *ACM SIGMETRICS*, 2004.

[38] Andrew Moore and Denis Zuev. Internet traffic classification using Bayesian analysis techniques. In *ACM SIGMETRICS*, 2005.

[39] Nigel Williams, Sebastian Zander, and Grenville Armitage. Evaluating machine learning algorithms for automated network application identification. Technical Report CAIA-TR-060410B, 2006.

[40] J. Erman, M. Arlitt, and A. Mahanti. Traffic Classification Using Clustering Algorithms. In *ACM SIGCOMM MineNet Workshop*, 2006.

[41] S. Zander, T .Nguyen, and G. Armitage. Self-learning ip traffic classification based on statistical flow characteristics. In *PAM*, 2005.

[42] L. Bernaille, R. Teixeira, I. Akodjenou, A. Soule, K. Salamatian. Traffic classification on the fly. In *ACM SIGCOMM CCR*, 36(2):23–26, 2006.

[43] Cristian Estan, Stefan Savage, and George Varghese. Automatically inferring patterns of resource consumption in network traffic. In *ACM SIGCOMM*, 2003.

[44] X. Yin, W. Yurcik, and A. Slagell. VisFlowCluster-IP: Connectivity-based visual clustering of network hosts. In *IFIP SEC*, 2006.

[45] N. Patwari, A. Hero, and A. Pacholski. Manifold learning visualization of network traffic data. In *ACM SIGCOMM MineNet Workshop*, 2005.

[46] P. Wong, H. Foote, G. Chin, P. Mackey, and K. Perrine. Graph signatures for visual analytics. In *IEEE Transactions on Visualization and Computer Graphics*, 12(6), December 2006.

[47] David Brumley, Li-Hao Liu, Pongsin Poosankam, and Dawn Song. Taxonomy and effectiveness of worm defense strategies. TR CMU-CS-05-156, 2005.

[48] S. Staniford-Chen et al. GrIDS – A graph-based intrusion detection system for large networks. In *National Information Systems Security Conference*, 1996.

[49] D. Ellis, J. Aiken, K. Attwood, and S. Tenarglia. A Behavioral Approach to Worm Detection. In *ACM CCS WORM*, 2004.

[50] Fivos Constantinou and Panayiotis Mavrommatis. Identifying known and unknown peer-to-peer traffic. In *IEEE NCA*, 2006.

[51] Y. Xie, V. Sekar, D. Maltz, M. Reiter, and H. Zhang. Worm origin identification using random moonwalks. In *IEEE Symposium on Security and Privacy*, 2005.

[52] J. Tolle and O. Niggenmann. Supporting intrusion detection by graph clustering and graph drawing. In *RAID*, 2000.

[53] W. Aiello, C. Kalmanek, P. McDaniel, S. Sen, O. Spatscheck, and J. Merwe. Analysis of communities of interest in data networks. In *PAM*, 2005.

[54] Godfrey Tan, Massimiliano Poletto, John Guttag, and Frans Kaashoek. Role classification of hosts within enterprise networks based on connection patterns. In *USENIX Annual Technical Conference*, 2003.

[55] P. McDaniel, S. Sen, O. Spatscheck, J. Van der Merwe, B. Aiello, and C. Kalmanek. Enterprise Security: A Community of Interest Based Approach. In *NDSS*, 2006.

[56] Albert-Laszlo Barabasi. The origin of bursts and heavy tails in human dynamics. In *Nature*, 435:207, 2005.

[57] A. Nanavati, S. Gurumurthy, G. Das, D. Chakraborty, K. Dasgupta, S. Murkerjea, and A. Joshi. On the structural properties of massive telecom call graphs: Findings and implications. In *CIKM*, 2006.

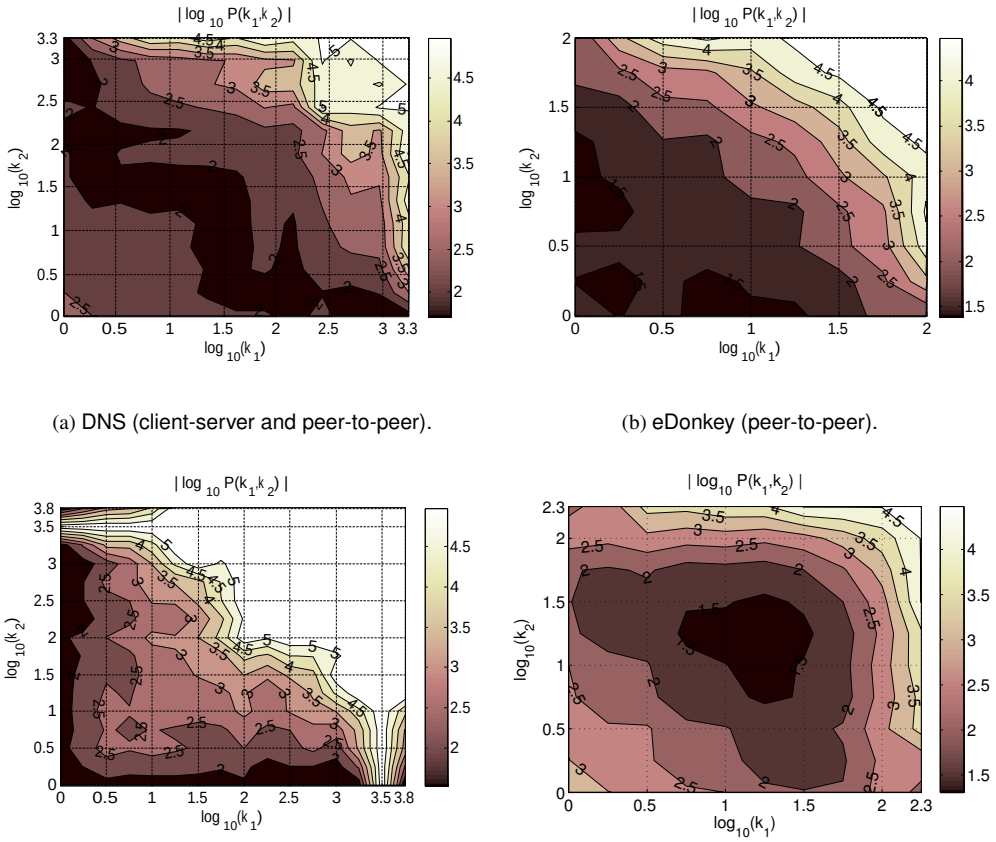
[58] D. Donato, S. Millozzi, S. Leonardi, and P. Tsaparas. Mining the inner structure of the web graph. In *International Workshop on the Web and Databases*, 2005.

[59] K. Ong, W. Ng, and E. Lim. Mining relationship graphs for effective business objectives. In *PAKDD*, 2002.

[60] K. Yada, H. Motoda, T. Washio, and A. Miyawaki. Consumer behavior analysis by graph mining technique. In *Book of Knowledge-Based Intelligent Information and Engineering Systems*, p. 800-806, 2004.

[61] P. Dineas, M. Krishnamoorthy, M. Sofka, and B. Yener. Studying e-mail graphs for intelligence monitoring and analysis in the absence of semantic information. In *IEEE ISI*, 2004.

[62] H. Yu, M. Kaminsky, P. B. Gibbons, and A. Flaxman. Sybilguard: Defending against Sybil attacks via social networks. In *ACM SIGCOMM*, 2006.



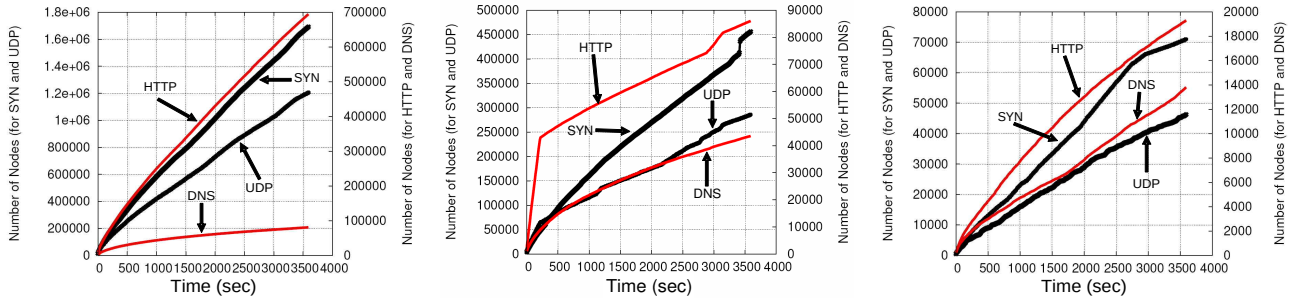
(a) DNS (client-server and peer-to-peer).

(b) eDonkey (peer-to-peer).

(c) HTTP (client-server).

(d) WinMX (peer-to-peer).

Figure 10: Joint Degree Distribution(JDD): $P(k_1, k_2)$ gives the probability of a randomly selected edge to connect nodes of degree k_1 and k_2. Same-colored areas in the contour plots, give regions where an edge is equal likely to exists. A dark colored region shows the areas with the higher concentration of edges, and white intricate areas with no edges. All probability matrices have white regions at the top right parts of the figures, showing that nodes with the highest degrees are not directly connected with each other. *Note:* The contour plots are symmetric in that $P(k_1, k_2) = P(k_2, k_1)$. All TDGs are derived from the first 300 sec of the OC48 trace.



(a) Trace OC48.

(b) Trace MAWI. The two modes in HTTP are caused by heavy scanning activity during the first 250 seconds of the trace.

(c) Trace AUCK.

Figure 11: Cumulative plot of the number of encountered hosts versus time, (for four representative TDGs) over the one hour duration of each of the three traces. Note that the two y-axis (left and right) correspond to different TDGs due to size differences. The SYN TDG is calculated by using the EFSP filter with all destination port numbers and the UDP TDG is calculated with the use of EFP filter on every UDP packet irrespective of its destination port number. *Observation:* The number of new nodes increases near linearly over the duration of the trace.