Appmon: An Application for Accurate per Application Network Traffic Characterization

Demetres Antoniades[1], Michalis Polychronakis[1], Spiros Antonatos[1],
Evangelos P. Markatos[1], Sven Ubik[2], and Arne Øslebø[3].

1. {danton,mikepo,antonat,markatos}@ics.forth.gr. Institute of Computer Science Foundation for Research and Technology, Hellas PO Box 71110, Heraklion, Crete, Greece
2. ubik@cesnet.cz CESNET, 160 00 Prague 6, Czech Republic
3. Arne.Oslebo@uninett.no UNINETT S.A. N-7465 Trondheim, Norway.

Abstract

Accurate per-application network traffic characterization is becoming increasingly difficult in the face of emerging applications that use dynamically negotiated port numbers. At the same time, information about the contribution of different network applications and services to the traffic mix is highly demanded by network administrators for facilitating effective network management and traffic engineering. In this paper we present *appmon*, a passive monitoring application for per-application network traffic classification. *Appmon* uses deep packet inspection to accurately attribute traffic flows to the applications that generate them, and reports in real time the network traffic breakdown through a Web-based GUI. *Appmon* is easy to configure and deploy, and is publicly available as an open source application.

1 Introduction

One of the most frequent requests of network administrators is to identify the applications and hosts that generate the largest amount of network traffic. The emergence of peer-to-peer file sharing, multimedia streaming, and conferencing applications has resulted to a substantial increase in the traffic volume, since they transfer a large amount of data. However, monitoring the traffic generated from such applications is becoming increasingly difficult.

Traditionally, traffic attribution to the corresponding applications is performed using the statically assigned port numbers. Widely used network services, like the Web, Telnet, SSH, and many others, are associated with well-known port numbers which can be used for identifying the traffic related with each application. However, many major new applications, including popular, bandwidth-hungry file sharing applications and widely used video and voice conferencing applications, do not use well-known port numbers. Instead, they allocate and use dynamically negotiated ports. Furthermore, some applications masquerade their traffic using pervasive, firewall-friendly protocols, like HTTP, in order to bypass firewall restrictions and make the identification of their traffic harder. Indeed, several widely used applications like BitTorrent [6] and Skype [7] can be configured to operate through port 80, which is usually left open even in environments with strict firewall configurations. Nowadays, the assumption that port 80 traffic is solely HTTP Web traffic is hardly true.

It is clear from the above that traditional network monitoring methods for determining per-applications network usage are not effective anymore for accurate traffic categorization [17]. Having identified this issue, several researchers have conducted significant work towards alternative ways for network traffic classification. Due to the popularity and high bandwidth demands of peer-to-peer file sharing applications, a significant body of work has focused on the identification and categorization of peer-to-peer application traffic. Initial approaches used deep packet inspection and application signatures for attributing traffic flows to the corresponding applications [19, 20]. Recent approaches identify the applications that generate the traffic either by deriving statistical models for certain protocols [18] or by characterizing the behavior of the host generating this traffic [22].

Motivated by the significance of traffic categorization for effective network management and traffic engineering and aiming at gaining a better understanding of Internet traffic, we have developed *appmon*, a passive network monitoring application for accurate per-application traffic identification and categorization. *Appmon* uses three different approaches for attributing flows to the applications that generate them. First, it searches inside application messages for characteristic application protocol patterns. For certain applications that dynamically negotiate the ports that are going to be used, *appmon* fully decodes the applications protocol to identify the new, dynamically generated port number and then tracks further traffic flows through these ports. Finally, legacy applications that do not match above filters are categorized based on well-known port numbers and protocols using BPF filters.

The rest of the paper is organized as follows. Section 2 gives a more extensive description of the tool. Section 3 discusses performance and evaluation issues. Section 4 refers to currently deployed *Appmon* sensors.

2 Application Design

This section presents the overall design of *appmon*, including a detailed description of the traffic classification algorithm, implementation details, and the graphical user interface.

2.1 Traffic Classification

Appmon passively monitors the traffic that passes trough the monitored link and categorizes the active network flows according to the application that generated them. A network flow is defined as a set of IP packets with the same protocol, source and destination IP address, and source and

destination port (also known as a 5-tuple). Traffic categorization is performed using information from both the packet header and payload.

The classification algorithm operates as follows: *appmon* processes each captured network packet sequentially. For each captured packet, it first checks if the packet belongs to an already categorized network flow. Information about the network flows seen so far is stored into a hash table, along with information about the matching application. *Appmon* keeps the minimal state required in order to reduce the packet processing time. This allows for a "fast path" processing of subsequent packets of an already categorized flow, since they will only result to a look up in the hash table for finding the record of the network flow in which they belong, and, consequently, the matching application, without the need for any further processing..

Packets that do not have a matching entry in the hash table are passed down to the next processing level, where each packet is sequentially processed by a set of modules called *application trackers*. Each tracker is responsible for identifying the traffic of a particular application or protocol. There are three different types of application trackers, depending on method used for classifying traffic: *packet inspection* trackers, *protocol decoding* trackers, and *header filtering* trackers.

Packet inspection trackers are used for tracking application-level protocols, mainly used in peer-to-peer file sharing applications such as Gnutella [8] and BitTorrent. The packet inspection tracker searches inside packet payloads for characteristic application messages or binary byte sequences that are used by application protocols. These application messages where selected by extensively reverse-engineering the network traffic of popular file sharing applications, as well as by studying the related work on signature-based traffic classification [9, 19, 20]. Although pattern matching inside packet payloads is a quite CPU intensive operation, in most cases the characteristic application patterns, usually protocol control messages, are present in the first 100 bytes of the packet payload, and thus the pattern matching is performed only to this portion of the payload, reducing significantly the processing overhead.

Protocol decoding trackers are used for publicly documented application level protocols that operate through well known control ports, but use a dynamically assigned port for data exchange. For example, in passive FTP, control messages are exchanged through port 21, but actual data transfers are made through a dynamically negotiated port. Protocol decoding trackers operate by fully decoding the application-level messages exchanged through the well-known control port, trying to identify the messages related with the negotiation of port numbers that will be used for future data transfers. When such a message is identified, the number of the dynamic port is extracted and then the tracker will correctly classify the new network flow that is going to be used for the data transfer, since the flow will use this dynamically negotiated port.

If none of the above groups of trackers succeeds in identifying a given packet, then the packet is passed to the header filtering trackers. Filtering trackers classify traffic based on packet header information such as identifying predefined registered ports [2] and other protocol information. Filtering trackers are implemented using BPF filters [16].

Table 1: Implemented Protocol Trackers.

Layer 4 Protocols	Application Protocols	
TCP	BitTorrent	eDonkey
UDP	Direct Connect	Gnutella
ICMP	FTP	HTTP
IP-in-IP	SSH	SMTP
	DNS	NetBIOS
	RTSP	OpenVPN

As we have already discussed, several applications masquerade their traffic using widely used, firewall-friendly protocols, like HTTP, in order to bypass firewall restrictions and make identification of their traffic harder. To avoid potential traffic misclassification due to such tricks, trackers are prioritized, with packet inspection trackers applied first, then the protocol decoding trackers, and finally header filtering trackers. When a packet is matched by a tracker, then it is not processed further by subsequent trackers. For example, the BitTorrent tracker has higher priority than the HTTP Web tracker. Thus, the flow of a BitTorrent packet through port 80 will be correctly attributed to the BitTorrent protocol, and not to Web traffic.

If none of the above methods manages to classify the flow in which the packet belongs, then the packet is temporarily considered as unknown, and the application waits for more packets of the same flow in order to classify it.

It worth mentioning that since most of the application specific patterns are located at the beginning of a flow, the vast majority of the monitored packets will belong to an already active – and thus categorized – network flow. As a result, expensive deep packet inspection operations are performed only to a small subset of the traffic, and *appmon* manages to process traffic loads of several hundred Mbit/s

Table 1 presents the currently implemented protocol trackers in *appmon*. We split these protocols into two broad categories. The first contains the main layer 4 protocols, while the other contains application-level protocols, including those used by several popular traffic-dominating peer-to-peer applications.

2.2 Graphical User Interface

Appmon reports the classification results through two different user interfaces, depending on the requirements of the user. For quick and easy network monitoring, there is a console-mode version which can report the results either through a batch text mode printout, or a more user-friendly ncurses [11] version. For long-term usage, *appmon* provides a powerful GUI accessible using any web browser. Due to space restrictions, in this paper we describe only the Web interface, since it provides a superset of the information provided by the console-mode versions.

Appmon reports the per-application traffic distribution through the web interface presented in Figure 1. The main page is split into three frames. The central frame presents a graph of the incoming and outgoing traffic distribution for the last hour. The graph presents the traffic portion of each categorized application with a different color, while any remaining non-categorized traffic is shown in grey. The topmost/bottommost line represents the total observed traffic load.

The information of this frame is better viewed in Figure 2, which presents the per-application distribution of the incoming and outgoing traffic at the University of Crete in Greece. The values are expressed in Mbit/s, and the graph isupdated every 10 seconds. A detailed per-application breakdown of the traffic load is presented underneath the graph.

The application offers five different time period views of the traffic distribution. The main view presents the per-application traffic distribution of the last hour. Links also exist for the time period of the last three hours, last day, last week, last month and last year.

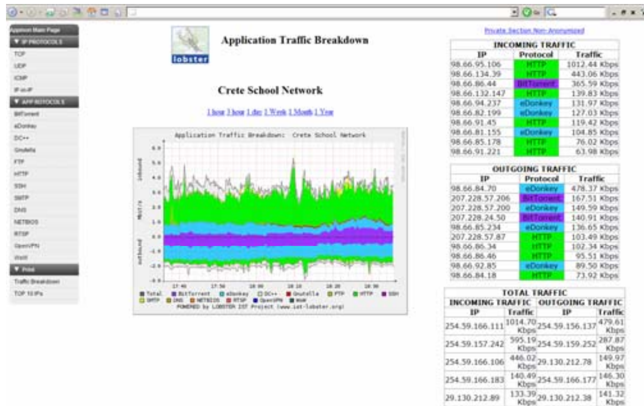

Figure 1: Appmon Web Interface.

Besides traffic classification, *Appmon* also reports the K top bandwidth consuming IP addresses. This is done by accumulating the traffic of each IP address after every packet is categorized at a specific application. In order to achieve this some extra state is needed. For every protocol we keep a hash table with all the IP addresses that belong to flows marked as belonging to this protocol. For every IP address we keep the number of bytes it transmitted, and the addresses are sorted in descending order according to the amount of traffic seen so far.

The top bandwidth consuming IP addresses are showed in three tables in the right frame of the Web interface. The first two tables contain the IP addresses of the K (10 by default) flows that consumed the largest portion of bandwidth during the last measurement period. Each record contains information about the application in which the flow belongs to and the exact amount of bandwidth that it consumed. The third table presents the same information at the IP level, which corresponds to the top K IP addresses that consumed the largest portion of bandwidth irrespective of application. Figure 3 shows an example of how the top 10 IP addresses are presented through the web interface.

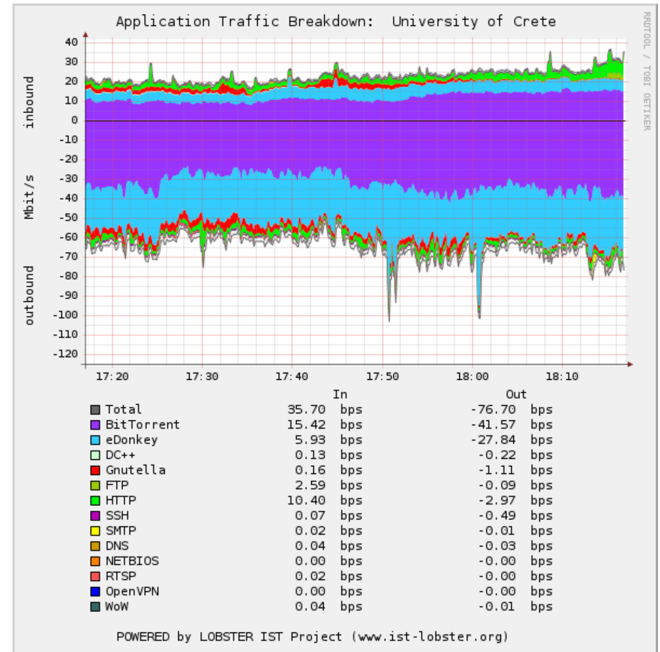

Figure 2: Per-application bandwidth usage.

Since information about IP addresses is sensitive and in some cases it may not be desirable to be exposed, appmon can *anonymize* all the IP addresses presented by the Web interface. Address anonymization is performed using prefix-preserving anonymization [23, 24, 25], which preserves subnet information. A non-anonymized version of the TOP IP address information is also available for view only by authorized personnel using a login procedure.

Finally, the left frame of the Web interface gives the user the ability to view the traffic of only selected protocols through a menu with all available protocols.

INCOMING TRAFFIC		
IP	Protocol	Traffic
98.66.95.106	HTTP	1012.44 Kbps
98.66.134.39	HTTP	443.06 Kbps
98.66.86.44	BitTorrent	365.59 Kbps
98.66.132.147	HTTP	139.83 Kbps
98.66.94.237	eDonkey	131.97 Kbps
98.66.82.199	eDonkey	127.03 Kbps
98.66.91.45	HTTP	119.42 Kbps
98.66.81.155	eDonkey	104.85 Kbps
98.66.85.178	HTTP	76.02 Kbps
98.66.91.221	HTTP	63.98 Kbps

Figure 3: Top 10 incoming traffic IP addresses as presented by appmon.

2.3 Implementation

To be freely available and easy installable, we have implemented *appmon* using only a few external libraries. *Appmon* is build in C language and uses the Libpcap packet capturing library [13], which supports live traffic capture using standard Ethernet interfaces, as well as DAG cards.

The crucial pattern matching operation within the packet payloads is performed using an implementation of the Boyer-Moore [15] string searching algorithm.

Appmon uses the RRDtool suite [10] for storing measurement data and graphing the traffic distribution. The Round Robin Database provided by RRDtool efficiently stores time-series data for very long periods in very little space using data aggregation. The database used by *appmon* has a size of a few megabytes and can store measurements for a period as long as one year.

The installation of the Web interface requires a web server like Apache with no extra packages. The results are rendered using simple CGI scripts and plain html code.

3 Performance

Our first experiment aims at exploring the performance of our application. We used a local testbed consisting of three PCs connected to a gigabit switch, as shown in Figure 4. The "Sender" PC generates traffic destined to the "Receiver" PC using the nttcp [1] tool. The traffic from both hosts is mirrored to the third monitoring machine which is running *appmon*.

The configuration of the measurement machine is as follows. We used an Intel Xeon 2.4 MHz, with 512 KB cache and 512 MB memory. The Operating System was Debian Linux with 2.6.15 kernel version. Two kinds of network interfaces were used. A regular Gigabit Ethernet interface (NIC), and a specialized DAG 4.3GE packet capturing card [12].

It is important to mention that nttcp produces artificial traffic by filling the packet payload with random bytes. This is a worst-case traffic load for *appmon* since none of the packets matches any of the monitored protocols. Thus, every packet passes through the "slow" processing path, going through all tracker functions, since none of the packets has a matching entry in the hash table, and none of the trackers is able to find a matching packet.

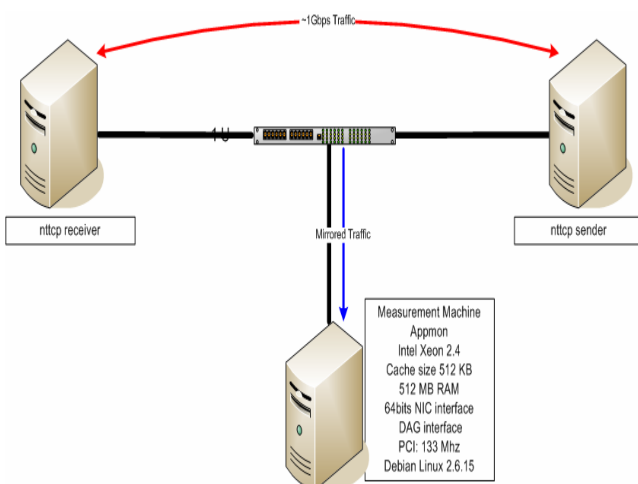

Figure 4: Testbed Environment

We stressed *appmon* by sending traffic in various speeds. Figure 5 shows the results for both NIC and DAG interfaces. As we can see *appmon* can process up to 500 Mbit/s without any packet loss when running on a regular NIC interface (blue line), while it is able to process all 900 Mbit/s when running on top of the DAG card (green line). The results imply that the application can fully monitor a Gigabit link using a DAG card.

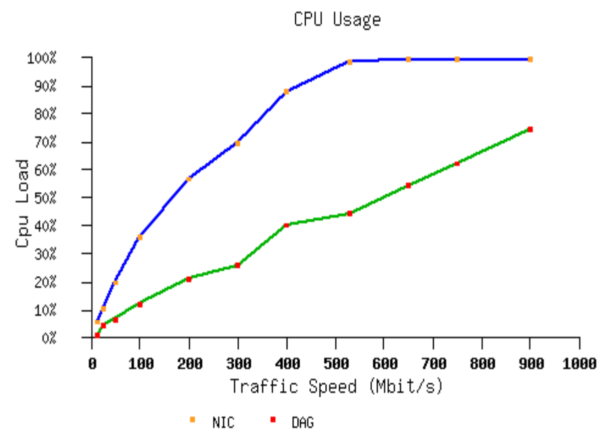

Figure 5: CPU Usage for Incoming TCP Traffic

For our second experiment we deployed *appmon* in a real network environment, aiming at verifying the performance results of the first experiment. *Appmon* was installed on a sensor at University of Crete, monitoring the incoming and outgoing traffic from the campus to the Internet. The monitoring machine was an Intel Xeon 3.2GHz, with 2MB cache memory and 1GB main memory, running a Debian Linux, kernel version 2.6.15. The traffic is captured using a DAG 4.2GE passive monitoring card. Along with the traffic load reported by the application, we measured the CPU load of the machine.

A new measurement result was produced every 5 minutes for a measurement period of four days.

Figure 6 presents the CPU load (y-axis) of the monitoring sensor as a function of the monitored traffic load (x-axis). Each point corresponds to a five minute interval, computed as the average of the measurements performed every 10 seconds in that interval. *Appmon* has a steady behavior, since the CPU load increases as the traffic load increases. Some corner cases in which the load is increased significantly while the traffic load is low are probably caused due to the almost simultaneous arrival of many new traffic flows that have not yet been categorized.

4 Deployment

Appmon can also operate on top of the Monitoring Application Programming Interface (MAPI) [3,4]. MAPI is an expressive programming interface for network monitoring that has been developed in the context of the LOBSTER Project [14]. MAPI gives the ability for remote

and distributed monitoring [5] without the need of user access to the remote monitoring sensors.

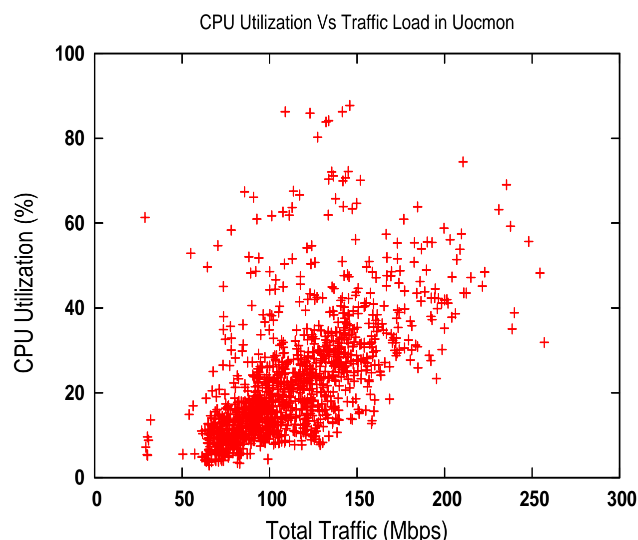

Figure 6: Appmon CPU Load Vs. Traffic Load while running on University of Crete

Using MAPI we have deployed *appmon* in several monitoring points around the world. Currently we have deployed *appmon* sensors in four institutions in Greece; the Foundation of Research and Technology Hellas, the University of Crete, the Greek School Network and the Node of HellasGRID in Crete. We have also deployed sensors in Czech Republic, as shown in Figure 7, and several sensors in Norway.

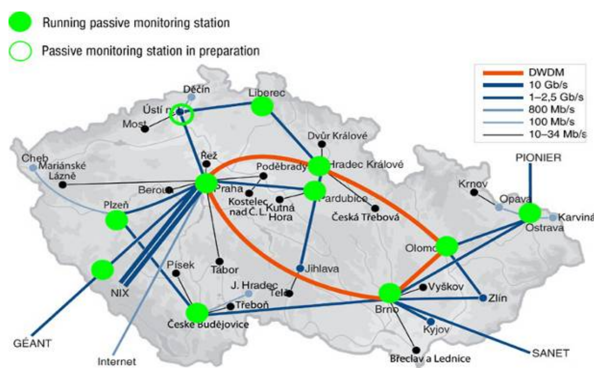

Figure 7: appmon deployment in Czech Republic through LOBSTER.

Conclusions

In this paper we have presented *appmon,* an application for real time per-application network traffic categorization. The main goal of the application is to visualize the network traffic usage in order to help in effectively monitoring the network traffic usage. As we have shown, *appmon* is able to categorize traffic in speeds that reach the one Gbit/s. *Appmon* uses a large set of protocol trackers for the classification of traffic from many emerging applications, while its module design allows for the easy addition of more protocol trackers in the future.

With several "bandwidth-hungry" applications increasingly trying to make their traffic difficult to detect, we expect that the use of encrypted traffic is on the way. In order to address this problem, we plan to explore whether non payload traffic classification methods can be used to identify and classify network traffic in real time.

Acknowledgments

This work was supported in part by the IST project LOBSTER funded by the European Union under contract number 004336.

References

1. Nttcp. http://sd.wareonearth.com/~phil/net/ttcp/.
2. Internet Assigned Numbers Authority. http://www.iana.org/.
3. MAPI official homepage. http://mapi.uninett.no.
4. M. Polychronakis, K. G. Anagnostakis, E. P. Markatos, and A. Oslebo. *Design of an application programming interface for IP network monitoring*. In Proceedings of the 9th IEEE/IFIP Network Operations and Management Symposium (NOMS), pages 483-496, April 2004.
5. P. Trimintzios, M. Polychronakis, A. Papadogiannakis, M. Foukarakis, E. P. Markatos and A. Oslebo. *DiMAPI: An application programing interface for distributed network monitoring*. In Proceedings of the 10th IEEE/IFIP Network Operations and Management Symposium (NOMS), April 2006.
6. The BitTorrent protocol. http://www.bittorent.org.
7. Skype. http://www.skype.com.
8. Gnutella. http://www.gnutella.com/.
9. http://protocolinfo.org/.
10. http://oss.oetiker.ch/rrdtool/.
11. The Ncurses library. http://www.gnu.org/software/ncurses/ncurses.html.
12. ENDACE. DAG Network Monitoring Interface Card. http://endace.com/networkMCards.htm.
13. The Packet Capture Library http://www.tcpdump.org.
14. The LOBSTER IST project http://www.ist-lobster.org.
15. R. S. Boyer and J. S Moore. *A fast string searching algoritm.* In Communication of ACM, Volume 20, pages 762 – 772, October 1977.
16. S. McCanne and V. Jacobson. *The {BSD} Packet Filter: A New Architecture for User-level Packet Capture.* In USENIX 1993.
17. A. Moore and K. Papagiannaki. *Toward the Accurate Identification of Network Applications*, In PAM, March, 2005.
18. L. Bernaille, R. Teixeira, I. Akodkenou, A. Soule, K. Slamatian. *Traffic Classification On The Fly.*
19. T. Karagiannis, A.Boido, N. Broenlee, kc claffy and M. Galoutros. *Is P2P dying or just hiding?* In IEEE Globecom 2004, GI

20. S. Sen, O. Spatscheck and D. Wang. *Accurate, Scalable In-Network Identification of P2P Traffic Using Applications Signatures*. In WWW, 2004.
21. T. Karagiannis, A. Broido, M. Faloutsos and kc claffy. *Transport layer identification of P2P traffic*. In ACM/SIGCOMM IMC, 2004.
22. T. Karagiannis, K. Papagiannaki, and M. Faloutsos. *Blinc: multilevel traffic classification in the dark*. SIGCOMM. Comput. Commun. Rev., 35(4):229–240, 2005.
23. A. Slagell, J. Wang and W. Yurcik. *Network log anonymization: Application of crypto-pan to cisco netflows*. NSF/AFRL Workshop on Secure Knowledge Management (SKM), 2004.
24. J. Xu, J. Fan, M. Ammar, and S. B. Moon. *On the design and performance of prefix-preserving ip traffic trace anonymization*. Internet Measurement Workshop (San Francisco, CA, USA: 2001), pages 263–266, 2001.
25. J. Xu, J. Fan, M. Ammar, and S. B. Moon. *Prefixpreserving ip address anonymization: Measurement-based security evaluation and a new cryptography-based scheme*. ICNP 2002, 2002.