A Framework for the Evaluation of Flow-based Traffic Monitoring Systems

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Peter Siska
2010

Leiter der Arbeit:
Professor Dr. Torsten Braun
Institut für Informatik und angewandte Mathematik

Acknowledgment

First of all I would like to thank my academic advisor, professor Prof. Dr. Torsten Braun, for agreeing to my rather unorthodox request and giving me the opportunity to write my master thesis as part of my stay at the IBM Zurich Research Laboratory. Furthermore, I would like to thank my project advisor Dr. Andreas Kind as well as all of my group members at the research laboratory, especially Marc Ph. Stoecklin, for their valuable input on the topic, their indispensable help allowing me to solve many problems along the way, but also for all the off-topic discussions we had during my stay. Moreover, I would like to thank Markus Nufer, my manager, for all the time he allowed me to invest into my studies, and for his indispensable support throughout the thesis. Lastly, I would like to thank Jana, for all her patience and love.

Summary

Systematic evaluation of flow-level network monitoring and analysis systems is crucial for quantifying their performance and accuracy under various traffic conditions. As many conditions critical to the proper operation of such systems occur only rarely, or produce overloads on the collector side when capturing the network traces corresponding to these conditions, such as network attacks, scans, or similar, the evaluation of traffic monitoring systems is typically performed on a set of synthetic test traces comprising simplistic traffic structure. Moreover, the automated assessment of the accuracy of such flow-based systems is further aggravated due to the missing tools to compare the exact values of traces generated with values analyzed by the network monitoring and analysis systems.

In this thesis, we propose a novel methodology to generate realistic traffic traces on the network flow level, allowing the combination of normal background traffic and customized traffic conditions. Our technique uses a graph-based approach to model and extract traffic-behavior templates from communication patterns observed in real-world traces. Furthermore, by combining extracted and user-defined traffic templates, realistic traces with parameterizable background traffic that also comprise critical borderline cases are generated in a scalable manner.

A proof-of-concept implementation demonstrates the utility and simplicity of our method to produce a variety of evaluation scenarios. Moreover, the framework introduced in our thesis comprises a pluggable architecture, allowing user-defined plugins to programmatically examine the exact values of the generated traces and perform operations before, during, and after the flow trace generation process, such as the automated evaluation and comparison of values in network monitoring and analysis systems processing the traces.

Our evaluation sections show that the extraction of traffic templates from real-world traces leads to a manageable number of intuitive models that still enable an accurate re-creation of the original communication properties. Moreover, our initial implementation of the framework is able to achieve satisfactory high flow generation rates, extending its use not only as a means of evaluating the accuracy of flow-based systems when processing traces with realistic traffic structure, but also for assessing these system's processing performance.

We believe that our approach is useful not only for the evaluation of flow-level network monitoring and analysis systems but also for a wider spectrum of general validation tasks based on network traces on the flow-level.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The complexity and importance of computer networks has risen considerably over the past decades as the world becomes increasingly connected. In the course of such development, new challenges in planning, deployment, and maintenance of computer networks arise. In order to ensure the best possible operational quality, and an error-free and continuous operation of their network infrastructure, businesses and Internet service providers employ a plethora of tools that monitor, measure, analyze and evaluate traffic that flows through their networks.

Traffic-monitoring and anomaly detection systems are widely used in corporate and service provider networks to gather network-related information of business critical applications, analyze prevalent communication patterns of the traffic, collect data for volume-based accounting, or detect abnormal traffic patterns. In addition to performing these analyses on the packet level, many of today's traffic monitoring systems use flow-based information of the network traffic, e.g., NetFlow or IETF IPFIX, exported by routers, switches, or software-based probes. Even though these protocols summarize observed traffic flows into a compact representation, the sheer amount of flows as well as particular traffic conditions may lead to an overload and inaccurate results produced by monitoring systems.

Systematic testing of monitoring systems to ensure accuracy and performance is therefore crucial, but, at the same time, poses a number of challenges. Flow-based traces are usually in a fixed, binary format, leaving them hard to parameterize in terms of the number of flows or the number of unique network hosts present in traces used for evaluation. Similarly, the flow record field values, such as the start and end timestamps, the number of packets or bytes, or the flow duration and similar, in such traces cannot easily be adapted to contain desired values that model specific, desired traffic conditions, without rewriting each flow record in existing traces.

Therefore, the evaluation of a traffic monitoring system is typically performed on a collection of traces known to contain complex constellations such as attacks, network scans or high item cardinalities, e.g., many different service ports or IP addresses, or very high flow rates, e.g., the number of network flows present in a trace during a fixed time interval.

Many traffic conditions detrimental to a monitoring system's performance are, however, difficult to collect as they are either observed rarely or may produce overloads on the collector side. Critical traffic scenarios must be generated synthetically. This is achieved with a set of manually parameterized scripts to imitate expected traffic conditions. Consequently, the flow attributes and

the structure of the traffic are limited by simplifications and assumptions resulting in unrealistic traffic properties. Therefore, evaluations using these traces only rarely provide satisfying results.

1.1 Problem Statement

In this thesis, we consider the problem of flow-level trace generation to support the evaluation in terms of performance, as well as in terms of correctness of traffic analysis performed, of monitoring systems under realistic conditions. Flow-level network-monitoring systems are faced with very high flow export rates from one or more export devices (e.g., routers) in the network. The systems are equipped with high-speed insertion algorithms and specially designed in-memory database systems. The performance of the flow processing is bounded by hardware constraints as well as the structure of the monitored traffic. If the analyzer process cannot keep up with the rate of incoming flow records, the quality of the results degrades. A proper evaluation of the performance limits of monitoring systems is therefore indispensable, both in terms of export rates and nature of the observed traffic. We propose a theoretical and practical framework comprised of different parts that, either used each on its own or in combination, help to evaluate such flow-based traffic-monitoring systems.

First, we propose a template-based approach using graph-theoretic metrics to define traffic patterns present in traffic scenarios such as attacks, anomalies, and other borderline cases. By combining different templates and streaming the generated traffic trace to a monitoring system, the performance and data processing accuracy of such system can be studied under various conditions. Our approach enables easy-to-use customization of traffic features and characteristics in terms of the number of hosts and flows, as well as the evaluation time.

Moreover, we present a self-parametrization technique that extracts traffic templates from real-world traffic traces. These templates enable the recreation of flow traces that closely resemble those of the original traffic, and allow for parameterization of certain traffic aspects such as the number of flows, the number of unique network hosts, or values of other network flow attributes such as the number of packets and bytes, as well as flow durations. The template library created can be used as background traffic and combined with customized templates to produce specific evaluation scenarios.

1.2 Contributions

The contribution of our work is threefold. First, we introduce the concept of traffic templates, a compact representation of network traffic conditions using a set of graph metrics and their evolution over time. By means of such templates, network flow traces containing desired traffic conditions can be composed and generated. Several different traffic templates can be combined to generate parameterizable background traces with realistic traffic structure. Furthermore, our template-based approach allows for definitions of custom, specific traffic scenarios, such as network scans, attacks, or similar. The combination of thereby defined templates containing user-defined traffic conditions with traffic templates established from real-world traces, enables

the generation of traces comprising borderline cases and realistic background traffic which can be used to evaluate the accuracy of traffic-monitoring systems.

Second, we present a self-parametrization technique that extracts traffic templates and their respective template parameters from existing flow traces and enables the creation of a library of realistic and parameterizable background traffic. We demonstrate that the traffic structure of flows generated from these templates exhibits characteristics close to those observed in the original traffic.

Finally, we introduce a flow-based traffic generator which generates flow records in NetFlow format from a collection of traffic templates. To the best of our knowledge, we are the first to generate parameterizable flow records with traffic structure established from real-world traces directly, without the detour of a packet-based traffic generator in conjunction with a flow collecting probe device or software application, as a means of evaluating flow-based monitoring systems. Moreover, the realistic traffic structure in the flow traces generated and the ability to easily define custom traffic scenarios addresses the simplistic approach and some drawbacks of the existing flow-based generators described in Section 2.2. Finally, the pluggable architecture of our flow generator implementation allows for an automated processing of the generated data, hence providing a valuable instrument to assess the accuracy of flow-based systems that process and analyze the flow trace generated.

1.3 Thesis Outline

This thesis is organized as follows. In Chapter 2, we review existing techniques to produce evaluation traces as well as other techniques related to our work. Moreover, we describe the graph-theoretical traffic classification approach, which serves as the main building block of our methodology. We specify important background information related to our objectives, and present observations that justify our design choices in Chapter 3. Chapter 4 presents our template-based approach and the self-parametrization technique. Moreover, we introduce details about our trace generator implementation and provide information about the algorithms used. The flow trace generation, the self-parametrization technique, as well as the definition of custom traffic scenarios and the performance of our trace generator are evaluated in Chapter 5.

Chapter 2

Related Work

2.1 Packet-based Generators

A vast number of packet-based network traffic generators and models have been proposed over the past years. The work by Rolland et al. [1, 2, 3] introduces LiTGen, a lightweight network traffic generator. LiTGen uses a hierarchical description of semantically meaningful traffic entities, where each entity is expressed by a set of random variables in order to statistically model IP traffic on a per user and application basis. This hierarchical model is made up of four distinct levels.

First, in a per-user defined session level, the traffic for each user is characterized by a series of session and inter-session periods, expressed by the size of each session in terms of the number of objects downloaded, and the inter-session durations. Next, on the object level each session is split into several objects, a set of user requests and server responses. Finally, a packet level further divides each session object into a series of packets and is expressed by a random variable for the packet inter-arrival times. LiTGen uses existing packet traces to extract the relevant parameters for each level and traffic entity in a bottom-up approach.

When generating the output, the packet stream for a each user and application is composed in a top-down fashion across the different levels, starting with the session level, moving to the generation of single packets on the packet level. The number of users is fixed for each application. By superimposing the synthetic trace of all users and all applications, the final trace is obtained.

Vishwanath et al. introduced an other packet-based traffic generator called Swing [4]. Similar to LiTGen, it uses a structural model to emulate the distributions for user, application, and network behavior. The model in Swing consists of four categories (layers), where each category is expressed by a set of suitable parameters. A users layer characterizes the application-specific user behavior in the traffic. The properties of different user sessions, such as the number and target of individual connections within a session, are expressed in the sessions layer. Then, the authors furthermore consider the connections layer, where connection-specific traffic properties such as the request size or the packet size distribution are characterized. Finally, they include a fourth layer in their model to model the low-level network characteristics such as link loss-rates and delays, path latencies, and similar.

Several other similar packet-based network traffic generators exist. Most commonly, such traffic generators preserve traffic properties on the packet-level, such as inter-packet gaps, file size distributions, or traffic burstiness. They have been applied in application-specific traffic generation, soft- or hardware performance evaluation [5, 6], and the evaluation of anomaly detection systems [7, 8].

We focus on flow-level rather than packet-level trace generation: our aim is to produce traces that preserve traffic properties on the network flow level (e.g., connectivity patterns), which are essential for the evaluation of flow-based network monitoring and analysis systems. Even though a packet-level generator combined with a flow-exporting probe may be employed produce flow traces, the overhead incurred and the risk of potential measurement errors arising in the probe are not justified. Furthermore, some packet-based generators, e.g., Swing, require the use of cutting-edge hardware to properly emulate all network characteristics and therefore generate realistic flow traces.

Moreover, our aim is to model traffic properties that are important for the evaluation of flow-level network monitoring systems. These properties include a useful traffic structure, the flow-level traffic properties, such as the number of flows, or the application (service) port diversity.

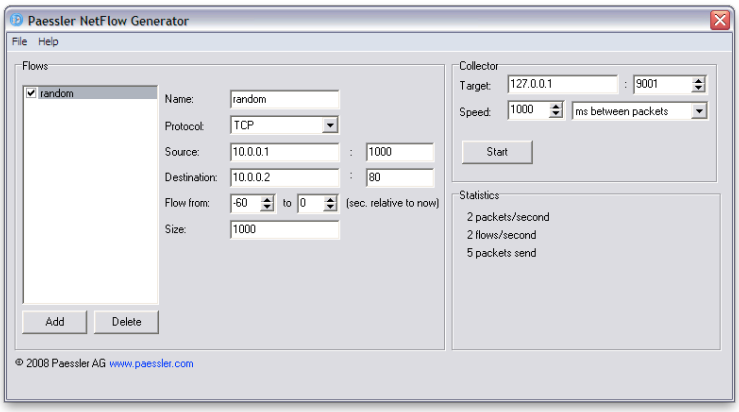
2.2 Flow-based Generators

To the best our our knowledge, only a few open-source or commercial tools for generating network traces in a flow-based format, e.g., NetFlow, are available. Paessler NetFlow Generator [9] and Flowalyzer NetFlow & sFlow Generator [10] are two freely available, closed-source software packages provided by businesses for rudimentary testing of their commercial network analysis products. Usually, these tools are applied as a means of testing the capability of flow-based monitoring systems to receive data. Consequently, only a small subset of the possible flow-level properties can be parameterized.

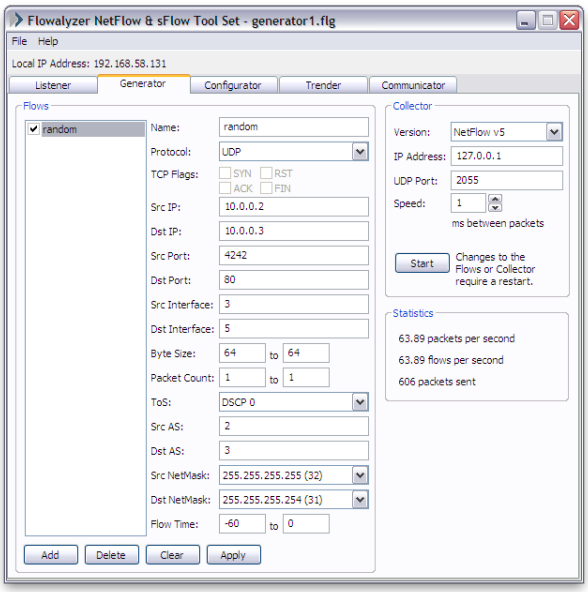
In the two software packages mentioned, for example, source and destination ranges for IP addresses as well as for application ports for the flow records to be generated can be defined through a graphical user interface. An example of the interfaces of both tools is shown in Figure 2.1. Furthermore, in some of the tools, the values of certain NetFlow record fields such as packet and byte count, source and destination interface, type of service, TCP flags, or source and destination autonomous system can be specified, either as fixed values or value ranges.

The generated NetFlow packets are sent as datagrams over the UDP protocol to the destination IP address and port specified in the user interface. An open-source alternative to these applications is the NetFlow Simulator in C# [11]. Similar to the previously mentioned tools, it provides a graphical user interface for configuration. However, this application only allows the user to specify a small portion of the flow record fields available. Namely, the NetFlow datagram version and a traffic rate, in terms of the number of packets per second, to use when sending the NetFlow packets over UDP. Values for the flow record fields are not user-definable but chosen at random for each record.

Although these tools generate flow records directly, without the detour of a packet-level traffic generator in combination with a flow-based data collecting router device or software application,



(a) Paessler NetFlow Generator



(b) Flowalyzer NetFlow & sFlow Generator

Figure 2.1: Graphical user interfaces of two NetFlow generators

they only support generation of simplistic traffic structure or trace properties. For example, the user can only specify source and destination ranges for IP addresses and application ports, which does not lead to realistic traffic structure in the trace generated. Moreover, custom traffic scenarios, such as network scans, attacks, or other borderline cases, cannot easily be modeled and underlaid with realistic background traffic. Consequently, the ability of these existing tools to evaluate flow-based systems under realistic traffic conditions is very limited.

Furthermore, some of the tools [9, 10] include only one flow record per each NetFlow packet

generated, which is another serious drawback as in real-world applications, router devices usually export up to 30[1] [12] flow records in a single UDP datagram. These tools, moreover, are not able extract traffic information from real network traces to parameterize the configuration and are limited to graphical user interfaces, which impede automation and are usually restricted to one specific operating system.

Flow-based network monitoring systems are usually capable of achieving flow record processing rates of thousands or even ten thousands of flows per second. This is where the current set of flow trace generators fails to provide satisfactory results. Although the NetFlow generators mentioned here can be used for basic inspections of data processed by a network monitoring system, the low flow record generation rates achieved constitute a serious drawback.

Finally, the automated processing and comparison of data generated with data collected by a network monitoring system is missing in these generators. Although some generators, such as the Paessler NetFlow Generator or Flowalyzer NetFlow & sFlow Generator, provide elementary statistics about the data generated (cf. Figure 2.1), there is no notion of any other flow record attributes which have been generated. As such, the collecting network monitoring system cannot compare the data collected, such as the total number of bytes in the flow records, or the total number of unique IP addresses present in the trace, with the data generated.

In our work, we propose a pluggable flow generator architecture, which allows for an automated processing of the generated data. Consequently, specific plugins can collect and compare actual flow attributes of generated data with values processed by the network monitoring system.

2.3 FLAME

Brauckhoff et al. [13] introduced a flow-level anomaly modeling engine. Their work focuses on providing a framework for injecting user-defined, realistic and parameterizable anomalies into existing flow traces. By operating on existing flow traces captured from real network traffic, the authors are able to provide realistic background traffic while having control over the simulated, explicitly defined anomalous traffic. As opposed to defining anomalies by a descriptive model, e.g., in terms of an anomalous effect on monitored network data, they apply a constructive model by building a list of known anomaly events. As such the authors distinguish between three classes of anomalies: additive, subtractive, and interactive. For additive anomalies, such as network scans or bot activities, FLAME injects new flow records into the existing traces, while not interacting with the baseline traffic. Subtractive anomalies, e.g., outages in the network or ingress shifts, remove existing flow records from the existing traces. In the case of interactive anomalies, such as denial of service attacks, new records that interact with the baseline traffic are added to the trace.

The FLAME prototype consists of five functional building blocks. A reader takes an existing flow trace as input and converts it to an internal format. The flows are then passed to a deleter unit, which selects flows to be deleted based on a deleter model passed as an additional input parameter. For additive and interactive anomalies, a flow generator outputs new flows based on

[1] The number of flow records exported in a single UDP datagram depends on the version of the protocol used. For different NetFlow versions for example, the number of records in a packet can vary from 24 to 30.

a generator model. Depending on the type of an anomaly, the flow generator either generates packets or entire flows and groups the output into one flow stream. In a next step, a flow merger combines the two output streams from the deleter and generator into one flow stream. Finally, the combined stream is written to a trace file and output by the flow writer.

The work presented in FLAME emphasizes on injecting user-defined anomalies into existing flow traces as a mean to evaluate anomaly detection systems. Our work differs in that we do not primarily target the evaluation of such anomaly detection systems, but more generally flow-based traffic monitoring systems. Moreover, as opposed to operating directly on existing flow traces, we extract parameters from existing traffic that allow us to generate new flow traces with traffic structure and host connectivity properties similar to that in the original traces.

2.4 Harpoon

Sommers et al. proposed Harpoon, an application-independent and configurable network traffic generator [14, 6]. Their approach aims at generating representative packet traffic at the IP flow level. The packet traffic is generated by means of unicast file transfers using either the TCP or UDP protocol. Harpoon implementation comprises a three-level hierarchical model.

On the lowest hierarchical level in Harpoon, the file level, file transfers are modeled by the distribution parameters for the the file sizes and the time intervals between consecutive file transfers. The middle level in their implementation is referred to as the session level, comprising three components: the IP address spatial distribution, the inter-session start times, and the session duration. Here, series of file transfers take place between network hosts drawn from the IP address distribution parameter, each series of file transfers being separated by a time interval determined by the inter-session start time parameter. Finally, the highest level is described as the user level, which models network users conducting consecutive sessions using either TCP or UDP. The session level is parameterized by the number of active users and the time period lengths during which users are active.

By modulating the parameters in each level in the model, Harpoon can generate packet traffic that exhibit bytes and packets volume, as well as temporal and spatial characteristics similar to those found in real traffic. Related to our work is the ability of Harpoon to extract the model parameters from existing traces. In Harpoon, the relevant distribution parameters for each level, such as the file sizes, inter-connection times, IP addresses, or the number of active sessions and similar, are extracted from existing NetFlow (or packet) traces in a self-parametrization process.

However, our approach differs in that we model and generate flow records (as opposed to packet traffic) and preserve flow-level properties (as opposed to per-flow properties), such as the distinct connection structures on service ports (applications), as well as the service port diversity found in real-world traffic traces. Moreover, while the approach presented in Harpoon seems suitable for router device and network hardware testing [6], it suffers similar drawbacks with regard to the evaluation of flow-based systems as described in Section 2.1, since the trace is being generated on the packet-level. Also, their approach, similar to some other packet-based generators, requires the use of several different computers as well as router hardware for the generation and collection of traces, while the framework introduced in this thesis can be operated from a single, desktop-

class commodity machine without any additional hardware requirements.

2.5 Topology Generators

Related to our problem of generating flow traces, with structural traffic properties similar to those found in real networks, is the generation of realistic network topologies, a topic which has been widely covered in the area of network research. Network topologies map the physical interconnections between network elements, which can be either single hosts on the host-level, network routers on the router-level, or entire Autonomous Systems (AS) which are connected groups of one or more IP network prefixes run by a specific network operator [15].

The need for generated, synthetic but realistic network topologies arises mainly due to the difficulty of obtaining topologies of operational networks. Although network protocols, applications, and algorithms should be designed to be independent of the underlying network topology, the latter often has an impact on the performance of network protocols or applications. Consequently, realistic network topologies are important for a proper evaluation of network-based technologies, especially in order to draw accurate conclusions when performing simulation-based evaluations.

The topological structures of networks are usually expressed as graphs in which vertices either constitute network hosts, routers, or ASs, based on the network hierarchy level modeled. Tangmunarunkit et al. distinguish between two categories of network topology generators: *structural* and *degree-based* topology generators [16].

Structural topology generators, such as Transit-Stub [17] or Tiers [18], are based on a hierarchical model of the Internet, which is viewed as a collection of interconnected routing domains. The size of each domain determines its type and also its position in the hierarchy, with wide area networks being on the top level, and fine-grained networks such as campus networks or local area networks being on the bottom. The graphs are generated top-down by interconnecting the routing domains based on predefined parameters, while the intranetwork connectivity in each routing domain is handled separately.

Faloutsos et al. [19] later showed that the degree distributions of graphs of router- and AS-level Internet topologies are power-laws, a structural property which the hierarchical topology generators were not able to model. Consequently, degree-based topology generators were introduced. These topology generators are designed to mainly match the power-law distributional properties of the Internet topology graph degrees.

A prominent and widely used topology generator is BRITE [20]. Similar to other topology generators, BRITE builds the graphs that model the topologies by first placing the vertices (nodes) in the plane. Then, the nodes are interconnected by forming edges in the graph. Furthermore, additional attributes relevant to topological components, such as the link delay, AS ids for router nodes, etc., can be assigned. Finally, the topology generated is output to a specific format. Due to different topology generation models that can used during the node placement, as well as the edge forming process, BRITE is capable of generating both, degree-based and hierarchical topologies.

Nevertheless, our work is different from topology generators. First, we model the traffic structure

on the network host-level, as opposed to the coarser level of traffic between network routers or ASs. Moreover, the degree distributions of graphs generated from the traffic structure for each service port are not of specific types or power-laws. Therefore we try to learn the traffic structure found in real traces and model it with graphs generated from the empirical degree distribution.

Finally, the physical topology of the network, such as the node placement and inter-distances, link-delays, or similar, are not of particular interest since we do not generate traffic on the packet level, but rather abstract to trace generation on the flow-level and concentrate on the logical structure of the traffic seen in a network, i.e., the number of clients and servers, and their interconnectivity.

2.6 Traffic Dispersion Graphs

The traffic modeling approach introduced in our work uses graph-theoretic means. We apply the concept of *Traffic Dispersion Graphs* (TDGs) introduced by Iliofotou et al. [21, 22] as a method for network traffic analysis and classification. TDGs are graphs in which each node is a host in the network and every edge represents an interaction between two hosts. Figure 2.2 shows an example TDG visualization established from an analyzed 300 seconds long flow trace. The number of nodes in the graph has been limited to 600 to better depict the resulting visualization.

Edges between nodes in TDGs can be established in various ways and their formation depends on the type of analysis the graphs are used for. For example, edges between nodes can be created based on the application service port of the traffic between two network hosts, the number of bytes transferred, or the IP protocol used, and similar. Moreover, by using directed edges in TDGs, the graphs can be used to capture the "social" behavior among hosts in a network. TDGs can therefore be used to successfully model the roles of network hosts.

When using directed edges in the graphs, there is a clear distinction between "who talks to whom"[22] in the traffic trace. As such, network hosts acting as service initiators and as service providers can be unambiguously identified and modeled using TDGs. We provide a more detailed definition of TDGs, motivate our design choices when modeling interactions between network hosts with edges in the graph, as well as depict additional visualizations in Chapter 3.

TDGs have successfully been applied as a means to network monitoring and traffic classification [22]. The authors create port-based TDGs from real network traffic traces from a set of twelve consecutive, disjoint intervals of fixed length of 300 seconds[2]. The edges between nodes in port-based TDGs are created for traffic between network hosts on specific application ports on the respective IP protocols.

They analyze the computed graphs by means of several graph metrics, such as the number of graph vertices and edges, the average degree, the assortativity coefficient, and similar, and show that the average metric values computed over the set of analyzed intervals introduce only small standard deviations of each metric for each network traffic type. Consequently, they suggest that the TDGs can be used to characterize and distinguish specific classes of traffic. Some graph metrics for HTTP (port 80) traffic, for example, significantly differ from the graph properties of

[2]The sum of these twelve consecutive, disjoint intervals leads to a total analyzed trace length of one hour of traffic.

TDGs created from DNS (port 53) traffic and vice versa.

The authors extend these findings in a more recent work and introduce Graption [23, 24], an application classification framework for automated detection of P2P applications. First, Graption eliminates network traffic for known legacy applications such as Web, DNS, or SMTP by filtering out these application-specific flows from the trace in a pre-processing step. Then, network flows in the remaining traffic are grouped based on numerous packet and flow features by clustering and cluster merging. Finally, a TDG is created for each group obtained. A set of rules applied to the graph metrics computed from these TDGs can successfully identify P2P traffic. Although their work focused on P2P application detection, they propose that Graption can be used for general application classification by choosing appropriate set of metrics in the application of rules [24].

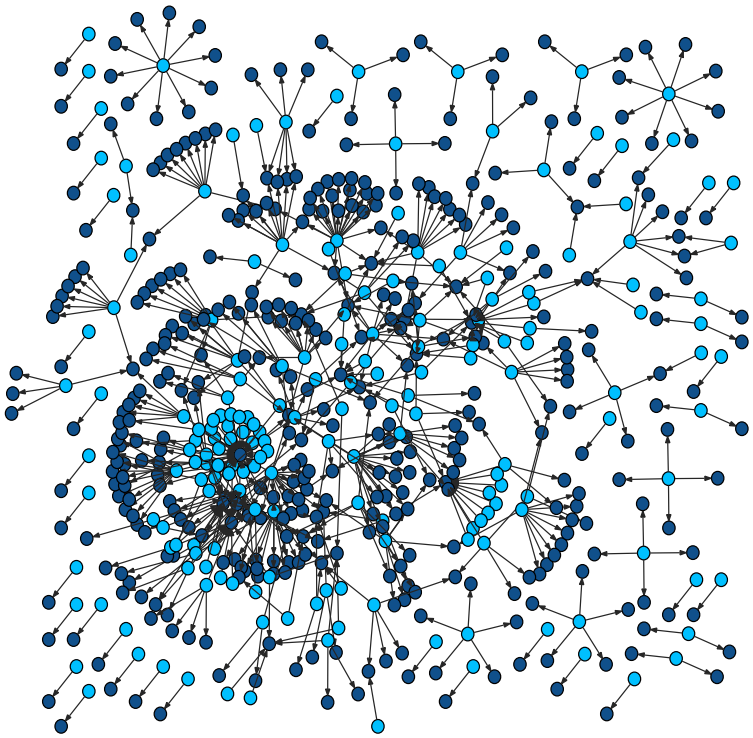


Figure 2.2: Example visualization of a TDG.

2.7 NetFlow / IETF IPFIX

NetFlow is a network protocol implemented on specific hardware router devices and provides a set of features that allows network operators to gain IP flow information about traffic in networks. Although initially implemented by Cisco and as such available only routers running Cisco's Internetwork Operating System (IOS), NetFlow has since become a widely spread network monitoring solution adopted by other vendors. The initial and to date the most commonly used NetFlow version is the protocol version 5 [12]. An updated and more flexible version 9 [25] introduced later by Cisco has been standardized by the Internet Engineering Task Force (IETF) and defines the IP Flow Information Export (IPFIX) Protocol [26, 27].

The most basic description of a flow is a set of packets that share common characteristics. More precisely, a network flow is defined as a unidirectional stream of packets which have the same source and destination IP address, use the same Layer 4 network protocol, and have an equal source and destination port in the IP packet headers [28]. Formally we express a flow as a 5-tuple f with equal key fields $f = (srcIP, dstIP, srcPort, dstPort, protocol)$.

The NetFlow protocol specification extends this 5-tuple flow definition by imposing additional bounds on the packet stream. In NetFlow, the set of packets must also arrive at the router on the same ingress interface and comprise the same Type of Service (ToS) byte in the IP headers [12]. For the remainder of this thesis we will use the term "flow" to refer to an entry in the NetFlow record, using the values which constitute the 5-tuple definition f, rather than to the series of packets defining a flow. Furthermore, a flow record in NetFlow can contain additional flow fields such as packet and byte counts, source and destination AS, or similar. The inclusion of these fields depends on the NetFlow version used as well as the configuration of the network device exporting flow-based information.

For each packet passing a measuring point in the network, e.g., a NetFlow-enabled router or switch, the device (exporter) keeps a flow cache, a list of all active flows. The NetFlow implementation then determines whether a packet is part of an already existing flow and updates the respective flow properties, or creates a new flow entry in the cache. Items in the flow cache are periodically expired based on a fixed set of rules such as flow inactivity timeout, FIN or RST TCP flags, or if the device cache becomes full. Expired flows are grouped together and packed into NetFlow datagrams and exported over the UDP protocol to a collecting device (collector) for further analysis.

Each NetFlow datagram consists of a header and a sequence of one or more flow records. The export packet header contains information about the NetFlow version used, the number of records in the datagram, a flow sequence number, as well as router-related information such as the system uptime and a UNIX timestamp. The number and type as well as the order of flow record fields is fixed in the NetFlow version 5. In NetFlow version 9 or IETF IPFIX the contents of a flow record can be described by a template flowset. This allows for a flexible configuration of the record format as opposed to the fixed format in earlier versions.

2.8 Flow Record Attributes

The main goal of our thesis is the generation of flow traces with realistic traffic structure, rather than an accurate reproduction of flow-level traffic patterns such as the flow durations, flow volume (in bytes or packets), or per-flow attributes such as the number of packets or bytes. Usually, flow-based network monitoring systems are mainly affected by the number of hosts, their inter-connections, as well as the network protocol and port diversity present in the traffic they process. Therefore, the reproduction of flow-level traffic patterns is, in our case, of secondary interest. Nevertheless, in order provide at least a reasonable approximation flow record attributes, a statistical distribution for modeling flow record attributes needs to be identified.

Current literature suggests the existence of heavy-tailed distributions, such as the Pareto, Weibull, or Lognormal distribution, for flow durations and flow lengths in network traffic [29, 30, 31]. Olivier et al. observed that the flow length in terms of flow duration and the number of packets or bytes often shows a very high degree of variability. Therefore, the statistical distribution of these values behaves differently in the head and the body of the distribution and is as such often best expressed by a mixture of two different distributions, such as, for example, the Pareto and the Lognormal distribution [32]. Other authors also suggest that flow-level properties for certain application-specific traffic such as web or peer-to-peer traffic can be approximated by a Pareto or Lognormal distribution [33, 34].

We describe the statistical distribution chosen in our approach in the definition of traffic templates in Section 4.2 as well as in the self-parametrization process outlined in Section 4.3.

Chapter 3

Background

3.1 Graph-based Connectivity Pattern Modeling

Our work has been inspired by the work of Iliofotou et al. [22]. They analyze the interactions between hosts on a given service port by means of graph-based metrics in *Traffic Dispersion Graphs* (TDG), described previously in Section 2.6. However, while they use the graph metrics for application-based traffic classification, we use them as a basis for trace generation.

A TDG is a directed graph $G = (V, E)$ that consists of a collection of vertices V (hosts) and a collection of edges E (connections) that connect pairs of vertices. Figure 3.1 (adapted from [22]) depicts an example TDG composed of six vertices $H_1, \ldots, H_6 \in V$. In general, the directed graph that defines a TDG is not simple as an edge (H_1, H_4) and an edge (H_4, H_1) can be created as shown in Figure 3.1. Furthermore, a TDG by definition evolves in time and space as new connections between hosts that interact with each other emerge, and new vertices and edges are being added to the graph. As such, each TDG is associated with a fixed time interval over which it was formed, or with a fixed number of network hosts used to establish the graph. The edge labels $1, \ldots, 7$ in Figure 3.1 depict the order in which the connections between hosts in the network were observed.

A vital part of TDGs is the definition of an edge, a process called "Edge Filtering" [24]. Generally, a connection or traffic flow between two hosts in a network has a clear notion of a service initiator and a service provider. For example, service initiators can be network nodes with specific applications such as web browsers, downloading data from and transferring data to web servers that act as service providers. Therefore, the edges between vertices in a TDG are directed. The authors in [24] distinguish between two basic edge filters for TDGs:

- **Edge of First Packet (EFP)**
  Since UDP is a connection-less protocol, we cannot properly differentiate between the service provider and service initiator due to the absence of TCP flags. Therefore, this filter, mainly used to translate UDP flows between hosts into directed edges, adds an edge (u, v) between two vertices (hosts) when the first packet is sent from u to v.
- **Edge on First SYN Packet (EFSP)**
  Contrary to UDP, the roles of two network hosts involved in an interaction can be

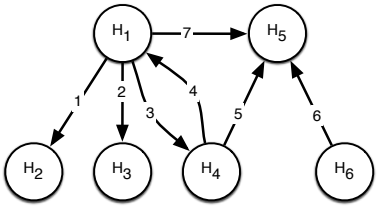


Figure 3.1: Example TDG with six vertices and seven directed edges

determined based on the values of the TCP flags. This filter creates an edge (u, v) between two vertices when the first SYN packet is sent from u to v.

This basic set of edge filters can be further extended by an additional set of rules to model specific goals of a study. We can for example chose to create edges between vertices in the graph only when a certain amount of packets or bytes has been transferred between two hosts. Another possible filter can consist of adding an edge only when traffic on a specific service port or a range of ports has been seen between pairs of network nodes. Furthermore, a protocol-based edge filter could be established, creating edges between vertices only when traffic is being exchanged using a specific protocol such as UDP, TCP, ICMP, or similar.

In our work we focus on *protocol- and port-based*[1] TDGs. We add a directed edge $(u, v) \in E$ between two vertices u and v when a flow between the two hosts is first seen over a given protocol and on a given service port. Since we analyze flow records in NetFlow format, we use the term "flow" instead of "packet" and name the edge filter applied **Edge on First Flow (EFF)**, based on the definition of the EFP filter.

Although the NetFlow protocol is capable of storing the value of TCP flags of flows in the records, most traces used in our analysis do not contain TCP flags. More generally, we need to assume that TCP flags are often not exported in the flow records by the respective devices, hence we cannot rely on these. Furthermore, the UDP protocol being connectionless does not provide any direct notion of service initiators and service provides, making it harder to determine the direction of a flow between two hosts.

Therefore, to determine the direction of an edge, we apply a simple heuristic. First, we compare the source and destination port numbers in the flow record with the service or application port number the port-based TDG is being established for. When the source port number matches the port number we filter on, the source IP address in the flow is most-likely the service provider and we create a directed edge in the TDG from the destination to the source IP address. Inversely, when the destination port number of the flow equals the filtered port number, the destination IP address is assumed to be the service provider and an edge from the source to the destination IP address is added to the graph. In some rare cases, the source and destination port number can be the same in a flow record. Therefore, in the cases, we identify the source IP address as the

[1] For the remainder of this thesis we will use the term "port-based" TDG when referring to these graphs. However, while leaving out the explicit "protocol-based" prefix, we note that each such graph models the traffic structure on exactly one service port and one network protocol.

Table 3.1: Traffic Statistics for the top 50 service ports, TCP and UDP traffic combined

(a) Campus Network Traffic

Length	% Packets	% Bytes	% Flows	% Hosts
1h	97.16 (1.43)	98.54 (1.10)	89.76 (1.96)	74.38 (11.67)
2h	96.90 (1.33)	98.30 (0.99)	90.23 (1.68)	75.76 (14.14)
4h	96.10 (0.91)	97.75 (0.67)	89.89 (1.45)	69.60 (12.72)
8h	96.43 (0.66)	98.20 (0.40)	89.72 (0.96)	64.89 (13.32)
24h	96.62 (0.45)	98.22 (0.36)	90.56 (0.48)	65.90 (12.29)
Average Values				
5h	96.85 (1.30)	98.32 (0.98)	89.96 (1.71)	72.95 (13.22)

(b) Hosting Environment Traffic

Duration	% Packets	% Bytes	% Flows	% Hosts
1h	87.35 (3.76)	90.57 (3.09)	91.14 (10.28)	98.49 (0.47)
2h	87.50 (3.16)	90.78 (2.89)	91.01 (9.26)	98.74 (0.30)
4h	86.72 (2.63)	89.77 (2.35)	91.66 (8.10)	98.95 (0.21)
8h	86.25 (2.04)	89.06 (1.26)	92.48 (4.38)	99.07 (0.15)
24h	86.14 (0.78)	89.00 (0.58)	89.86 (2.90)	99.10 (0.13)
Average Values				
6h	87.46 (3.15)	90.62 (2.82)	91.35 (8.89)	98.75 (0.34)

service initiator and the destination IP address as the service provider.

Consequently, we can exclude the existence of self-loops in the definition of a TDG since network connections are always established between two distinct hosts. Furthermore, once an edge (u, v) has been added to the graph using the EFF filter, new flows from u to v are ignored. Consequently, a TDG cannot have multiple edges between two vertices in the same direction.

3.2 Traffic Portion of The Top Service Ports

In our approach, we model the traffic structure for each service port and network protocol by means of one port-based TDG. Therefore, in a preliminary study, we explored how many port-based TDGs are needed to achieve a high coverage of the connectivity patterns. We analyzed flow traces collected in two different networks. The first set of flow traces comprises NetFlow records of the internal traffic from an average-sized campus network, collected over a period of 10 days between May 1, and May 9, 2009. The second flow trace collection consists of NetFlow records accumulated at a large hosting environment between April 14, and April 20, 2008.

The analysis was conducted over several disjoint trace lengths, ranging from one hour up to an entire day of traffic. We determined the top 50 service ports, for TCP and UDP traffic combined, present in the trace, and computed the relative amount of traffic these ports accounts for in terms of various metrics. Table 3.1 shows the percentages of the total amount of the number of packets and bytes, the number of flows, and the number of unique hosts for the top 50 service ports. For each analyzed trace length, we list the average value computed over a set of disjoint intervals of that length, and depict the corresponding standard deviations in parentheses. The bottom row shows the values averaged over all analyzed intervals with the corresponding average interval length on the left. The values for the internal campus network traffic are specified in Table 3.1(a), the statistics for traces collected at a hosting environment are listed in Table 3.1(b).

We observe that the majority of flow records that define the structure of the traffic are associated with only a small subset of dominant service ports. In the campus network traffic, for example, an average of 89.95% ($\sigma = 1.71$) of all flows are related to only 50 service ports. Similarly, 72.95% ($\sigma = 13.22$) of all hosts can be attributed to this set of ports. The dominant service ports in the hosting provider traces account for even a larger relative amount of the total values in the

traffic. Particularly, 95.46% ($\sigma = 1.29$) of all flows and 98.95% ($\sigma = 0.10$) of all unique hosts are found on the top 50 service port in the traces. We further note that the percentages of packets and octets transmitted on the top service ports account for a large portion of the traffic in both traces analyzed. However, the values for the number of packets and octets do not contribute to the traffic structure determined by the number of unique hosts and the flows between these hosts. Hence, in order to preserve the traffic structure found in a trace, we focus on the relative amount of flows and IP addresses covered by the set of dominant service ports.

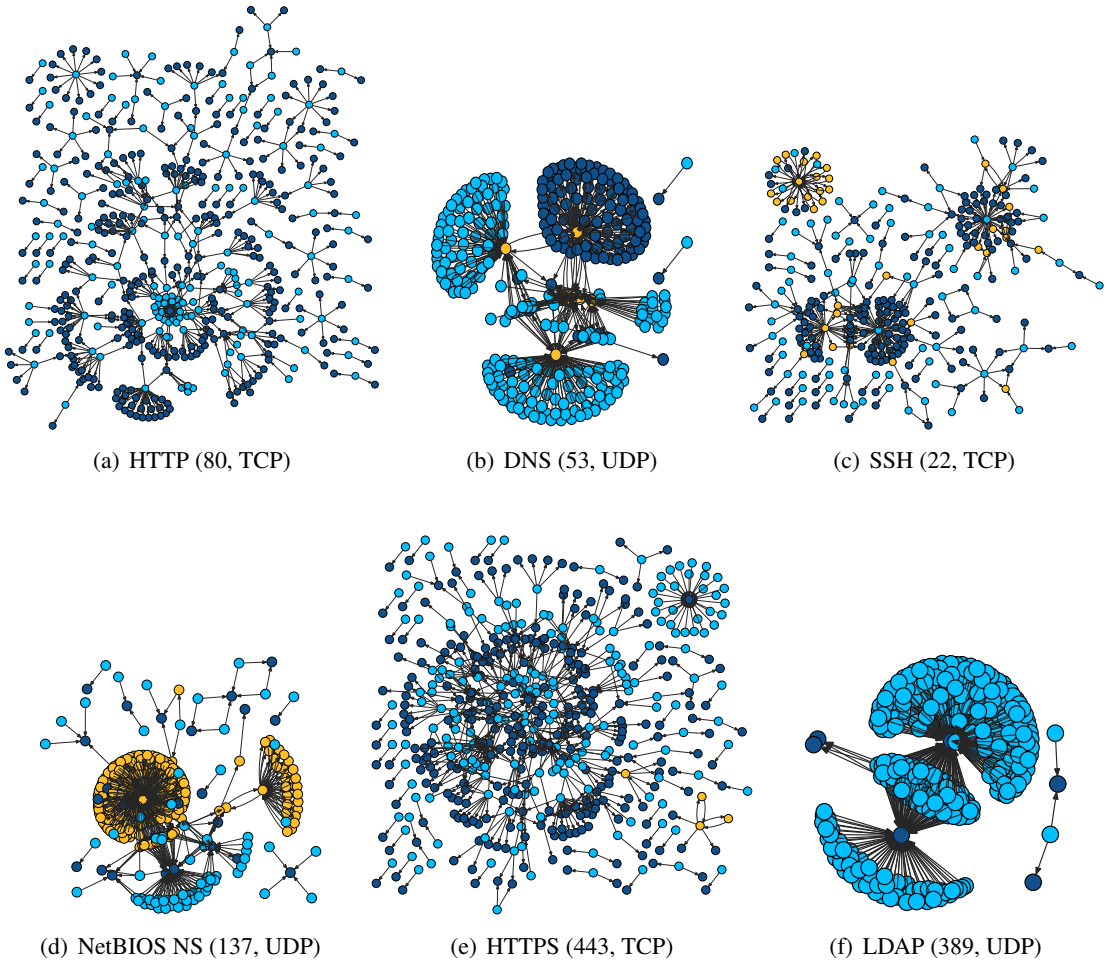
As a result of the lesser port diversity in the hosting provider traces, the percentage values for the number of unique hosts are higher compared to the values found in the campus traffic. The latter exhibits a increased number of different applications that run on a wider range of service ports. For example, the portion of HTTP traffic on the service port 80 (TCP) and DNS traffic on port 53 (UDP) on average accounts for more than 70% of the traffic in terms of the number of flows, as well as in terms of the traffic volume in packets and bytes transmitted. Consequently, traffic on dominant service ports in the hosting traces contains a greater amount of unique IP addresses compared to the total hosts in the traffic.

We conclude that the combination of the top n service ports (e.g., with $n = 50$) in terms of the number of flows and the number of unique hosts provides a sufficiently high coverage of the connectivity patterns present in the traces analyzed. As such, a combination of port-based TDGs established for the top n service ports can model the service port diversity and the traffic structure with respect to the number of unique hosts and flows well.

3.3 Visualizations of Traffic Dispersion Graphs

Visualizations of Traffic Dispersion Graphs provide an additional level of information with regard to the traffic analyzed. In this thesis, we use the GraphViz [35, 36] graph visualization software to depict the TDGs. GraphViz provides a set of tools capable of converting graph definitions from a text-based format to images of directed and undirected graphs. Furthermore, the graph layout is automatically optimized to produce lesser overlaps of vertices and to place denser graph structures around the center of the image.

Figure 3.2 shows visualizations of TDGs for six different service ports generated from the campus network traffic traces. The number of unique hosts has been limited to 300 ($|V| = 300$) to improve the quality of visualization of the underlying traffic structure. The graphs for HTTP and HTTPS traffic in Figures 3.2(a) and 3.2(e) respectively have been generated from 500 unique hosts ($|V| = 500$). A visual inspection shows that the graphs exhibit distinct structures that are characteristic of each service port on a given protocol. The TDG of service port 80 (HTTP) in Figure 3.2(a) and port 443 (HTTPS) in Figure 3.2(e) contains many vertices that only connect to one or a few other vertices. This is a typical property of web traffic where usually various different clients connect to several web servers. Other service ports, such as service port 53 (DNS) or service port 389 (LDAP) using the UDP protocol, exhibit much denser structures containing many low out-degree vertices connecting to only a few extremely high in-degree vertices like in (c.f. Figures 3.2(b) and 3.2(f)). This corresponds to the properties of DNS traffic, where numerous clients connect to usually only a few name-servers. Similarly, the traffic



(a) HTTP (80, TCP) (b) DNS (53, UDP) (c) SSH (22, TCP)

(d) NetBIOS NS (137, UDP) (e) HTTPS (443, TCP) (f) LDAP (389, UDP)

Figure 3.2: Visualization of port-based TDGs from six service ports. The direction of edges defines the service initiators (from) and providers (to). Characteristic for DNS traffic is the existence of a few high in-degree and out-degree nodes, in contrast to HTTP and SSH traffic which exhibits significantly more low-degree vertices.

structure for applications using the LDAP protocol contains many clients that connect to only a few designated LDAP servers.

Moreover, we find that in many TDGs a large number of vertices have either zero out-degree (sinks) or in-degree (sources) and only a small portion of the hosts act as both, service initiators and service providers. We apply colors to vertices in order to better depict the role of the corresponding hosts in the graphs. Sinks, vertices with incoming edges only, are colored dark-blue. Sources, vertices that have outgoing edges only, have a light-blue color. Additionally, vertices with both incoming and outgoing edges are colored yellow and represent hosts in the traffic that are service initiators as well as service providers. The latter, for example, can be seen in peer-to-peer or terminal services traffic, such as on service port 22 (SSH) in Figure 3.2(c).

Similarly, due to the peer-to-peer nature of the NetBIOS Name Service (NBNS) [37], traffic on the service port 137 (NBNS) in Figure 3.2(d) comprises nodes that act as both, clients and servers. In contrast to these TDGs, graphs for the service ports 80 (HTTP) and 443 (HTTPS) in Figures 3.2(a) and 3.2(e) respectively, as well as the service port 389 (LDAP) graph in Figure 3.2(f), contain sinks and sources only. Such graphs are a clear indication of traffic structure found in pure client-server applications.

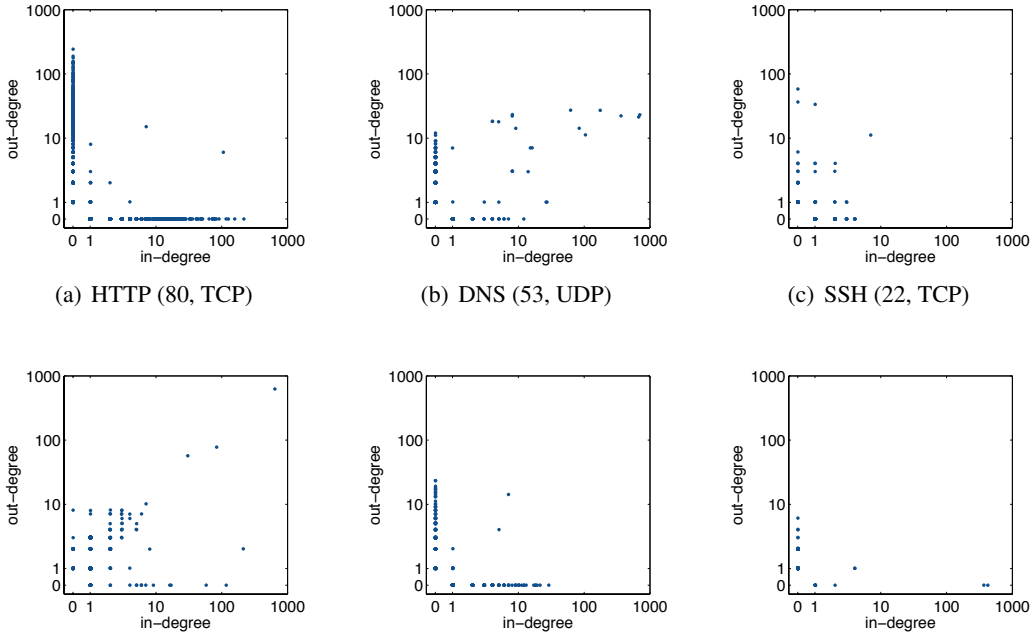
3.4 Graph Degree Properties

Traffic Dispersion Graphs contain information about the structure of the network traffic analyzed. However, in order to find a way of generate flow traces with similar structural properties as found in these graphs, we need to find a different representation of TDGs to achieve this goal. In this section, we analyze the degree properties of port-based TDGs in order to establish the basis for a method to capture the inter-connectivity between network hosts.

The out-degrees and in-degrees of vertices in TDGs capture the connectivity between the underlying network hosts and hence describe the structure of the traffic captured by the graphs. We analyzed the vertex out-degree and in-degree properties of various port-based TDGs in various traffic traces. In Figure 3.3 we visualize the graph vertices in a scatter plot, where the number of out-degrees and in-degrees defines the position of each vertex. The plots for six different service ports were created from TDGs established from one hour long traces from the campus network traffic. Vertices with zero out-degrees and in-degrees have been artificially added to the plot in order to include them on the log-log scale.

We find that graphs for most ports, such as web traffic on port 80 and port 443 in Figures. 3.3(a) and 3.3(e) respectively, exhibit a distinct separation of vertices along both axes. These vertices have either zero out-degree (sinks) or zero in-degree (sources). Similarly, the plots for the graph on the service port 389 (LDAP) in Figure 3.3(f) show the same separation of vertices along the axes and therefore primarily of sinks and sources. This is typical characteristic of client-server applications. The points along the x-axis are vertices with zero out-degree values and act as the servers, whereas the points along the y-axis represent vertices in the graph with zero in-degrees and are the clients.

While other services, such as DNS in Figure 3.3(b) and SSH in Figure 3.3(c), still exhibit a concentration of vertices along the axes, they are also characterized by points distributed to a greater extent in the first quadrant of the Cartesian plane. These points reflect vertices with non-zero out-degree and in-degree values. The peer-to-peer structure of the NetBIOS Name Service is well visible in the plot in Figure 3.3(d), where numerous points are scattered throughout the Cartesian plane. Nevertheless, the traffic structure on a considerable number of service ports consists primarily of sinks and sources, whereas a smaller portion of vertices is located in the center of the plane. The vertex degree values in TDGs reflect the communication patterns between hosts on a given service port. We use these findings to define a method to capture the inter-connectivity between network hosts based on the out-degree and in-degree values of vertices in TDGs.



(a) HTTP (80, TCP) (b) DNS (53, UDP) (c) SSH (22, TCP)

(d) NBNS (137, UDP) (e) HTTPS (443 TCP) (f) LDAP (389, UDP)

Figure 3.3: Scatter plots of out-degree and in-degree values of each TDG vertex for different ports on a log-log scale over a one-hour interval. Zero out-degree and in-degree vertices have been artificially added to the scatter plot.

Chapter 4

Flow Trace Generation Framework

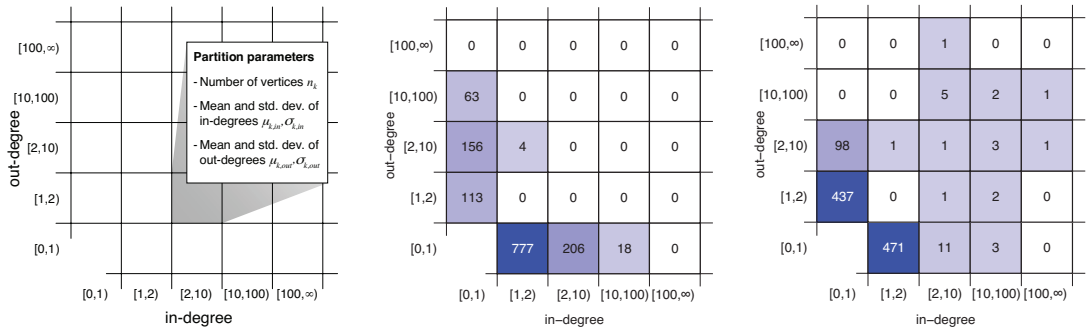
In this chapter we introduce the concept and methodology of your flow trace generation and evaluation framework. Our flow trace generation technique comprises four building blocks. We model the connection patterns between network hosts on distinct service ports and network protocols in terms of out-degree and in-degree distributions, which are parametrized by *partitioning* the joint degree distribution. The partitioning comprises information about the inter-connectivity of hosts, based on which TDGs whose edges comprise connections between hosts can be established.

Then, we introduce the concept of *traffic templates* used to express the traffic structure characteristics of a service port on a given network protocol. Traffic templates include various distributional parameters of flow record attributes, such as the parameters for the flow duration and for the number of packets and bytes, as well as parameters for the aforementioned partitioning of the joint degree distribution. Collections of traffic templates can be used to generate flow-based traces with useful traffic structure, as well as specific, user-defined traffic conditions.

Therefore, flow traces are generated from a collection of traffic templates in two steps. First, for each template, a graph generation algorithm builds a set of admissible connections from the partition parameters, based on which the *trace generator* produces the flow records with meaningful attribute values. Moreover, the pluggable architecture of our framework allows for automated processing of the generated trace by user-defined plugins which interact with the trace generation process during different phases.

Moreover, we describe a *self-parametrization* technique which can extract traffic templates with their respective parameters from an existing set of flow traces. The thereby extracted traffic templates can be used to produce evaluation traces comprising “normal” and parameterizable background traffic together with critical borderline conditions.

This chapter is organized as follows. First, we describe the basic ideas behind the partitioning of the joint degree distribution. Next, we introduce the concept of traffic templates and establish definitions for the set of template parameters used. We explain the self-parametrization process used to extract a relevant set of traffic templates with their corresponding parameters from existing traces. The process of generating flow records from traffic templates with a trace generator, along with specific characterization of the algorithms used, as well as the



(a) Degree distribution partitioning (b) Partitioning of HTTP (80, TCP) (c) Partitioning of DNS (53, UDP)

Figure 4.1: 4.1(a) shows the out-degree and the in-degree plane of a TDG divided into 24 distinct, fixed-sized partitions. 4.1(b) and 4.1(c) depict the partitioning of a 300-s time interval for traffic analyzed on service port 80 (HTTP) and service port 53 (DNS) respectively. Values in each each partition indicate the number of vertices.

implementation details of the pluggable architecture of our framework is outlined at the end of this chapter.

4.1 Partitioning

In Section 3.4 we introduced scatter plots of out-degree and in-degree values of TDGs. The vertex values for out-degree and in-degree contain information about how hosts interconnect with each other. Based on these scatter plots, we use the joint out-degree and in-degree distribution of vertices derived from TDGs to capture this social behavior in terms of connectivity between hosts in the network. We use this empirical degree distribution to establish TDGs from the partitionings when generating flow traces.

For a given service port, protocol, and fixed time interval, we divide the plane spanned by the out-degree and in-degree values of vertices in TDG established from a traffic trace into fixed-sized partitions (quantization). The partitions boundaries are determined by the non-linear, left-closed intervals $[0, 1)$, $[1, 2)$, $[2, 10)$, $[10, 100)$, and $[100, \infty)$. This leads to a set K of 24 relevant[1] partitions as shown in Figure 4.1(a). Hence the out-degree and the in-degree value of a vertex determine its distinct placement into exactly one partition.

The interval boundaries are chosen such that vertices are separated into meaningful partitions. We distinguish between zero, low, medium, and high degree vertices for both, out-degrees and in-degrees. Vertices with zero out-degree fall into the $[0, 1) \times z$ partitions, whereas vertices with zero in-degree fall into the $z \times [0, 1)$ partitions. Similarly, the low degree vertices in the partitioning are in the $[1, 2)$ and $[2, 10)$ partitions, while the medium degree vertices are in the

[1] We ignore the partition $[0, 1) \times [0, 1)$ in the bottom left corner in Figure 4.1(a) as it does not contain any valid nodes. By definition, a TDG does not have any isolated vertices, i.e., nodes with both zero out-degree and zero in-degree.

$[10, 100)$ partitions and the high degree vertices fall into the $[100, \infty)$ partitions for both, out-degrees and in-degrees.

For example, a web server that is accessed by a only few clients falls into one of the low in-degree partitions (e.g., $[1, 2)$). A popular web server, in contrast, is likely to be assigned to a high in-degree partition (e.g., $[100, \infty)$). Similarly, in the case of DNS traffic for example, a domain name resolver is probably assigned to a high-out degree partition while DNS clients usually only query one or a few servers to resolve domain names. Such clients will are probably to be assigned one of the zero in-degree and low out-degree partitions (e.g. $[1, 2)$ or $[2, 10)$). In general, clients are likely to exhibit zero in-degrees, but non-zero out-degrees.

The vertex population in a partition $k \in K$ is expressed by a set of five parameters $G_k = (n_k, \mu_{k,out}, \mu_{k,in}, \sigma_{k,out}, \sigma_{k,in})$. The parameter n_k is the number of vertices in a partition k. Furthermore, to capture the average values of vertex degrees and the deviations thereof, $\mu_{k,out}$ and $\mu_{k,in}$ are the mean out-degree and in-degree of the vertices in a partition k, whereas $\sigma_{k,out}$ and $\sigma_{k,in}$ are the respective standard deviations.

Figure 4.1 shows two examples of a partitioning created from TDGs of HTTP traffic (service port 80) in Figure 4.1(b) and DNS traffic (service port 53) in Figure 4.1(c) respectively. Both partitionings were generated from graphs established from the campus network traces measured in a 300-s time interval. The values and the color intensity in each partition indicate the number of vertices n_k. The zero out-degree partitions (bottom row) in Figure 4.1(b) contain most likely web servers, where popular servers with high in-degrees are in partitions on the right-hand side and less frequently visited servers are in partitions on the left-hand side. In this particular case, for example, we identified eight of the 18 hosts in the $[0, 1) \times [10, 100)$ partition to be highly frequented Google web servers. Similarly, the clients in Figure 4.1(b) are in the zero in-degree partitions (left column), with clients connecting to several web servers are in the upper partitions and clients accessing few servers are in the lower partitions. Typical for a client-server architecture, as seen in web applications, is the absence of vertices with non-zero out-degrees and non-zero in-degrees.

Partitioning of DNS traffic in Figure 4.1(c) shows similar concentration of nodes along both axes. Similarly to web traffic, clients reside in the left column and servers are located in the bottom row. However, in contrast to HTTP traffic, we also find vertices in the high out-degrees and high-in-degrees partitions (e.g. $[10, 100) \times [100, \infty)$). These are most likely to be domain name resolvers propagating resolution queries to other name servers. To sum up, we conclude that the information about the vertex out-degree and in-degree in these partitionings can be used to model connections (flows) between network hosts based on the traffic structure found in real traces.

4.2 Traffic Templates

A traffic template captures the structural properties of the connection patterns between hosts, as well as the distribution parameters of flow record attributes, such as the flow duration or the number of packets and bytes, for a time period. Furthermore, each traffic template is associated with exactly one service port and one transport layer protocol, such as TCP or UDP. A template

consists of three parts: a representation of the degree distribution partitioning representing the traffic structure found on a service port, a collection of distribution parameters defining admissible flow attributes, as well as a set of optional configuration parameters related to the flow trace generation process. Moreover, a period length T is assigned to each template to maintain the temporal dimension associated to the template definition. The list of all template components is depicted in Table 4.1.

Table 4.1: Traffic Template Parameters

Parameter	Description	Example
T	Period length associated with the template parameters.	300-s
R	Protocol number for the associated service port p_{dst}	17 (UDP)
$\vec{a}_{g,k}$	Polynomial coefficients for each partition parameter $g \in G_k$ for every partition $k \in K$.	
p_{dst}	Destination (service) port for service initiators.	80
P_{src}	Source port range for service initiators.	1024–32 768
IP_k	Ranges for source and destination IP addresses for every partition $k \in K$.	10.2.19.0/24
$d_\mu, d_\sigma,\ p_\mu, p_\sigma,$ b_μ, b_σ	Lognormal distribution parameter for the duration, packets, and octets flow record field.	2.03, 1.16
μ_H, σ_H	Mean value and standard deviation of the number of hosts.	812, 22.75
μ_F, σ_F	Mean value and standard deviation of the number of flows.	58 732, 210.25
	Optional Parameters	
$\min_H$	Minimum number of hosts for a period T	780
I_{active}	Range of intervals T the traffic template is active at.	1-3

4.2.1 Distribution Parameters

Number of Hosts and Flows The average number of hosts present during a period of length T for a given service port is parameterized by μ_H. The variability in the number hosts over a set of consecutive periods of length T is expressed by the associated standard deviation value σ_H. Similarly, we parameterize the number of network flows present during a period by μ_F and express the deviation of the number of flows by σ_F.

Flow Record Attributes The length of network flows is expressed by the duration flow record field value. Equally, the flow size in terms of the number of packets and the number of bytes transmitted is quantified by the flow record field values for packets and octets. In terms of the evaluation of flow-based monitoring systems, the importance of flow-level properties such as the duration, the number of packets, as well as the number of bytes, is secondary. These systems are mainly affected by the traffic structure determined by the

number of hosts and their inter-connections, the number of flows such systems need to process, as well as the diversity of service ports present in the traffic analyzed. In our approach, we concentrate on modeling the traffic structure as opposed to the flow-level properties. Nevertheless, the distribution parameters for flow lengths and flow sizes in the traffic templates are close to values observed in real network traces.

We outlined some valid statistical distributions to model the flow-level traffic properties in Section 2.8. Based on these findings, we chose to model the flow-level properties in traffic templates with a Lognormal distribution. The distribution parameters d_μ, d_σ, p_μ, p_σ, and b_μ, b_σ denote the duration, packet, and octet attributes, respectively.

We further note that the Lognormal distribution used by default can be easily replaced by different distributions, such as the Normal distribution, the Pareto distribution, or the Uniform distribution, in combination with the appropriate distribution parameters to model other flow properties requested. Other possible distributions, provided in our initial implementation of the framework, that can be used to replace the Lognormal distribution when generating flow records, are described in Section 4.4.1.

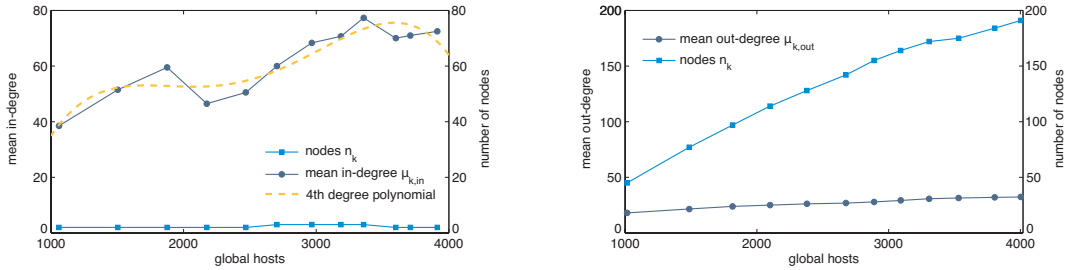
Source and Destination Ports Service initiators in the network use service ports of providers to establish connections. Web servers, for example, run as a service on one of the well-known ports (e.g., service port 80). Connections from clients to these web servers are established over the network using a random source port (on the client) and the application-specific service port on the server. The parameter P_{src} is the range of source ports the service initiators use to connect to service providers on the destination (service) port p_{dst} the traffic template is associated with.

IP Address Ranges The partitioning of the plane spanned by the out-degrees and in-degrees of TDG vertices comprises 24 partitions. The traffic templates contain definitions of IP address ranges IP_k for each partition $k \in K$. We use the Class Inter-domain Routing (CIDR) [38] notation to list IP address ranges in the templates and allow for several CIDR prefixes to be specified. Traffic templates for web traffic (service port 80), for example, can contain prefix definitions for several Class C networks (e.g. 172.134.16.0/24 and 82.14.2.0/24) in the partition $[0, 1) \times [10, 100)$, which comprises vertices (hosts) with zero out-degrees and in-degree values greater than ten (popular web servers).

4.2.2 Representation of Partitioning

The partitioning introduced in Section 4.1 captures the inter-connectivity between network hosts and their roles (e.g., clients or servers), in the traffic. The partitioning of the plane spanned by the out-degrees and in-degrees of vertices in TDGs allows us to employ this information to recreate the graphs, whose edges determine the connections between network hosts, during the flow trace generation process.

The analysis of parameters from partitions generated over a number of time intervals of varying length allows us to determine the temporal behavior of the number of hosts active on service port p_{dst} and of their connectivity properties. With increasing numbers of distinct hosts observed, the



(a) $[10, 100) \times [10, 100)$ partition parameters (DNS)

(b) $[10, 100) \times [0, 1)$ partition parameters (HTTP)

Figure 4.2: Evolution of different partition parameters $g \in G_k$ over a set of twelve time intervals of increasing length. 4.2(a) Shows the number of host, as well as the mean in-degree values including the approximating polynomial of the partition $k = [10, 100) \times [10, 100)$ for DNS traffic. 4.2(b) Depicts the number of hosts and the mean out-degree of the partition $k = [10, 100) \times [0, 1)$ for HTTP traffic.

connectivity patterns change depending on the service port. For example, for DNS (53), the in-degree of servers generally increases with the number of clients observed whereas the number of servers remains constant. For web traffic, on the other hand, many new distinct servers appear in low in-degree or low out-degree partitions, whereas only the in-degree of popular servers increases.

Figure 4.2 shows an example of the partition parameters evolution for two partitions and two DNS and HTTP (80) traffic generated over a set of twelve time intervals of increasing length. In Figure 4.2(a) the number of hosts in the high in-degree partition $[10, 100) \times [10, 100)$, likely to comprise DNS servers, remains almost constant for DNS traffic, whereas the mean in-degree values for hosts increases. The high out-degree and zero in-degree partition $[10, 100) \times [0, 1)$ depicted in Figure 4.2(b) for HTTP traffic, in contrast, shows an increasing number of hosts while the mean out-degree value rises only gradually. With the increasing numbers of distinct hosts in the traffic, new web clients likely appear in this partition while, for each client, the number of connections established remains almost constant.

To address this dependency of the traffic structure on the host population, we express the partition parameters as a function of the number of hosts. As such, we approximate the parameters in G_k of each partition $k \in K$ with a polynomial function. Each partition $k \in K$ is then expressed as a collection of coefficients $\vec{a}_{g,k}$ for each partition parameter in $g \in G_k$. For example, in Figure4.2(a), we depict the approximating polynomial for the partition parameter $g = \mu_{k,in}$ (mean in-degree) for the particular partition $k = ([10, 100) \times [10, 100))$. The polynomial in this particular case is of degree four, leading to a collection of five coefficients $\vec{a}_{g,k}$. The resulting polynomials allow us to compute partition parameters for a desired number of hosts when generating flow traces.

4.2.3 Optional Parameters

In contrast to the mandatory distributional parameters of flow attributes and source and destination IP addresses and ports, as well as the polynomial coefficients representing the partitioning, we complement the definition of traffic templates with the description of two optional parameters.

Minimum number of hosts The hosts generated from a traffic template for a given service port and network protocol for each interval T is defined by the distributional parameters for the number of hosts μ_H, σ_H. In some rare cases, the number of hosts to be generated, drawn from the normal distribution defined by these parameters, can be insufficient. For example, when the average number of hosts is $\mu_H = 820$ and the standard deviation equals $\sigma_H = 80.4$, the number of hosts in one particular interval can be much smaller (e.g. 540) than the mean value. The parameter $\min_H$ imposes a lower bound on the number of hosts generated for a time interval of length T.

Active intervals In order to achieve a coverage of the traffic structure and port diversity similar to real-world traces, a certain amount of service ports must be present when generating flow traces as described in 3.2. Therefore, the number of traffic templates employed must be reasonably high. Usually, a collection of several traffic templates associated with various service ports and network protocols is used to generate traces for a set of time intervals of length T. In some cases, however, traces for certain service ports must be excluded from the traffic mix. For example, a network scan on the service port 22 (SSH) is likely to be present only during certain specific time intervals, while being absent in others when the hosts performing the scan are not active. The I_{active} parameter in traffic templates addresses this requirement.

The I_{active} parameter can be specified using three different notations. First, a set of numbers corresponding to the time intervals during which flow records are being generated for a given traffic template can be specified. For example, for a traffic template to be active during the first, third, and fourth time interval when generating flow records, the I_{active} parameter is set to $I_{active} = 1, 3, 4$. Furthermore, the parameter notation also supports the specification of time interval ranges, e.g. $1-4$, during which a traffic template is active, or a combination of the interval ranges and specific time intervals, e.g., $1, 3-5, 7$. In the latter example, flow records are generated from the template during the first, third to fifth, and the seventh time interval.

Additionally, we introduce a third template-specific notation in order to account for cases where a specific traffic template is active at "every n-th" time interval, the parameter is set to $I_{active} = /n$. For example, to activate a template during every second time interval in the flow trace generation process, the parameter is set to $I_{active} = /2$.

4.3 Self-Parameterization

In addition to manually defining traffic templates, our approach provides the ability to automatically establish a set of traffic templates that provides a sufficient coverage of the traffic structure, as well as automatically extract the template parameters defined in Table 4.1 for each each template from existing flow traces. We refer to this process as *self-parametrization*. The self-parametrization comprises two main steps:

1. First, a collection C_{top} of the top service ports, a combination of sets of top n service ports in terms of the number of unique hosts and the number of flows for TCP and UDP, is established from a set of flow traces.

2. Then, for each port in the collection of top service ports C_{top}, a traffic template with the relevant distribution parameters and partitioning representation through polynomial coefficients is established.

The self-parametrization process operates on a set of existing flow traces. These traces are usually flow records in NetFlow version 5, version 9, or IETF IPFIX format respectively, as exported by routers and collected by NetFlow probes such as nProbe [39], either into a database or binary-format files. Each trace file is associated with a time interval of fixed length T. The campus network and hosting provider trace files used in our particular case each constitutes a time interval of length $T = 300$ seconds. For example, the analysis of twelve consecutive trace files represents a one hour long network traffic trace. The output of the self-parametrization is a set of traffic templates. In addition to the the flow trace input, five parameters further refine the self-parametrization process and output. These additional parameters are specified in a configuration file that looks as follows:

```
1  [configure]
2
3  top = 50
4  udp = 53
5  tcp = 80,445,8080
6
7  [create]
8
9  destination = /some/directory
10 cidr = 24
```

The configuration file is divided into two sections: the `[configure]` section related to the process of establishing the collection of the top service ports C_{top} for both, TCP and UDP, as well as the `[create]` section applied to the second part of the self-parametrization, the generation of traffic templates and estimation of the relevant parameters thereof. The self-parametrization configuration file parameters are described as follows:

top This parameter determines the size n of the top ports lists when calculating the top service ports in terms of the number of unique hosts and the number of flows for both protocols, TCP and UDP. We describe the top service ports calculation in Section 4.3.1.

tcp This parameter specifies TCP service ports to be explicitly included in the self-parametrization of templates, even though they may not be part of the collection of the top ports C_{top} established in the first step of the process. For example, to always include web traffic when using the self-parametrization, the service ports 80 and 443 can be specified.

udp Similarly, description of a collection of UDP specific service ports for which traffic templates may be generated and parameterized, independently of the port numbers present in the set of top service ports C_{top}.

cidr The self-parametrization allows for different levels of granularity, which can be specified by one of the values 8, 16, 24, or 32, when grouping IP addresses into CIDR prefixes.

destination This is the destination directory the parameterized traffic templates are saved in.

4.3.1 Top Ports Calculation

The analysis of the campus network and hosting provider traffic traces over different time intervals in Section 3.2 showed that the top n service ports account for the major portion of the traffic with respect to the number of unique hosts as well as the number of flows. We use these findings in the first part of the self-parametrization process, which analyzes flow traces to determine a collection of relevant service ports traffic templates are generated for.

First, we distinguish between TCP and UDP service ports. For each protocol, we establish two separate sets of service ports: a set which contains the top n service ports with regard to the number of unique hosts, as well as a second set that comprises the top n ports with respect to the number of flows present in the trace analyzed. The result of this process are four sets of n service ports, two for each protocol. The union of these four sets of n ports yields a collection C_{top} of relevant service ports that account for the major portion of the traffic, both in terms of hosts and flows.

In most cases, the average values from Section 3.2 for the top n service ports (e.g. for $n = 50$) with regard to the number of unique hosts and the number of flows present a lower bound on the traffic portion covered by the established collection C_{top} of service ports. The service ports, which usually account for a significant portion of the total number of flows, may differ from the service ports with the highest fraction of the number of unique hosts. Similarly, the service ports with the highest number of unique hosts do not necessarily contain the most flows. Therefore, the collection C_{top} of service ports with the highest number of hosts and flows, for both TCP and UDP, in the worst case comprises at least $n = 50$ service ports. However, the size of C_{top} is typically several times higher than n.

4.3.2 parameterization of Templates

The parameter extraction in the second step of the self-parametrization is performed over the combination of all flow trace segments of time intervals of length T (e.g. $T = 300$ seconds), for each port in the collection of top service ports C_{top} associated with the corresponding TCP or UDP network protocol.

Grouping of IP Addresses

The IP addresses of network hosts present in the flow trace over the sum of analyzed time intervals are grouped into CIDR prefixes for each partition $k \in K$. We simplify the super-netting process by imposing a restriction on the grouping granularity. Different **top** parameters, namely 8, 16, 24, and 32, can be set in the configuration file and determine how many IP addresses are grouped together into one subnet prefix. The lower the parameter specified, the more IP addresses are grouped together into one subnet.

For example, when the self-parametrization process should retain every single IP address present in the trace for each partition, the parameter 32 corresponding to the prefix notation `x.x.x.x/32` needs to be specified. Similarly, when we chose to group IP addresses in the same Class C network together, a parameter value of 24 which leads to the CIDR notation `x.x.x.0/24` is more appropriate. The output of this process is a set of network prefixes in CIDR notation present in the analyzed trace for each partition $k \in K$.

Estimation of Distribution Parameters

In Section 4.2.1 we described the distribution parameters of the flow duration and flow size, in packets and bytes, and chose to model these flow record attributes with a Lognormal distribution. We analyzed different service ports and found that, for some service ports, the Lognormal distribution is not always suitable and the flow parameters, such as the duration, the number of packets, or the number of bytes, might be better approximated by a different distribution or even constant values.

However, as described in Section 4.2.1, in our approach we emphasize parameters related to the traffic structure, which is relevant for the evaluation of flow-based systems, rather than the flow-level traffic properties. Therefore, in order to simplify the self-parametrization process, we chose to model these parameters with only one distribution. However, we note that the distribution used for flow attributes in the estimation process can easily be replaced by other distributions, value ranges, or constant values as described in Section 4.4.1.

The Lognormal distribution is a continuous probability distribution of a random variable X whose logarithm $Y = \ln(X)$ is normally distributed with mean μ and standard deviation σ [40]. The probability density function of X is defined as

$$f_X(x;\mu,\sigma) = \begin{cases} \frac{1}{\sqrt{2\pi}\sigma x} e^{-\frac{1}{2\sigma^2}[\ln(x)-\mu]^2}, & x \geq 0, \\ 0 & x < 0. \end{cases} \tag{4.1}$$

For each service port in the collection of relevant top ports C_{top}, the Lognormal distribution parameters μ and σ for the flow duration (d_μ, d_σ), the number of packets (p_μ, p_σ), and the number of bytes (b_μ, b_σ) are estimated over all analyzed intervals using the maximum likelihood estimation. The method of maximum likelihood estimation is an estimation of the parameters that maximize the likelihood function of a given distribution, in the case of the Lognormal distribution, approximating the μ and σ parameters [40]. For a simple discrete distribution with only one parameter, for example, where $X_1, X_2, \ldots, X_n$ denote the independent random

variables from the discrete distribution represented by $g(X, \alpha)$, and α is the single parameter of the distribution to be estimated, the likelihood function L is defined as the joint distribution of the independent random variables

$$L(X_1, X_2, \ldots, X_n; \alpha) = g(X_1, X_2, \ldots, X_n; \alpha) = g(X_1; \alpha) \cdot f(X_2; \alpha) \cdot \ldots \cdot f(X_n; \alpha) \quad (4.2)$$

In the case of the Lognormal distribution with the independent, log-normally distributed, random variable X, the logarithm $Y = \ln(X)$ of the random variable X is normally distributed. Therefore, the Lognormal probability density function can be rewritten as

$$f_{logn}(x; \mu, \sigma) = \prod_{i=1}^{n} (\frac{1}{x_i}) f_{norm}(ln(x); \mu, \sigma) \quad (4.3)$$

where f_{logn} denotes the probability density function of the Lognormal distribution (4.1), and f_{norm} constitutes the probability density function of the normal distribution [41]. Consequently, the log-likelihood estimation method for normal distributions can be applied. Therefore, using the same indices for the two distributions as in (4.3) and denoting the log-likelihood normal distribution function as l_{norm} (defined in [42]), the log-likelihood function l_{logn} for the Lognormal distribution can be written as

$$\begin{aligned} l_{logn}(\mu, \sigma | x_1, x_2, \ldots, x_n) &= -\sum_k \ln(x_k) \; + \; l_{norm}(\mu, \sigma | \ln(x_1), \ln(x_2), \ldots, \ln(x_n)) \\ &= \mathrm{c} \; + \; l_{norm}(\mu, \sigma | \ln(x_1), \ln(x_2), \ldots, \ln(x_n)) \end{aligned}$$

where c is a constant term with regard to μ and σ. Applying the formulas for the normal distribution maximum likelihood parameter estimators from [42] using the equality established in 4.4, the estimated maximum likelihood parameters $\hat{\mu}$ and $\hat{\sigma}^2$ are defined [41] as

$$\hat{\mu} = \frac{\sum_k \ln x_k}{n}, \quad \hat{\sigma}^2 = \frac{\sum_k (\ln x_k - \hat{\mu})^2}{n}. \quad (4.4)$$

Furthermore, the average number of flows μ_F and the average number of unique hosts μ_H, as well as the respective standard deviation values σ_F and σ_H observed over all analyzed time intervals is established.

Estimation of Partition Parameters

For each time interval analyzed, a TDG is created and the associated partitioning of the joint out-degree and in-degree distribution of vertices in the graph is established. In this process, the partitions for each interval consist of the traffic structure accumulated over all previously analyzed intervals. Consequently, the partitionings established from the graphs capture the

dependency of the traffic structure on the number of hosts observed. As such, we express each partition parameter $g \in G_k$ as a function of the number of hosts.

We apply the mathematical procedure of least squares polynomial fitting to determine coefficients of a polynomial function we derive the partition parameters from, for a given number of hosts. The least squares fitting method is a form of linear regression and provides a solution to the problem of finding the best fitting line (or polynomial) through a set of points. The method minimizes the sum of squares of the vertical offsets (residuals) between the fitted values and the actual values evaluated [43, 44]. The polynomial P_j of degree j with coefficients $c_0, c_1, \ldots, c_j$ is defined as

$$P_j(x) = y = c_0 + c_1 x + \cdots + c_j x^j. \tag{4.5}$$

Given a set of n points $x_1, \ldots, x_n \in \mathcal{X}$ (hosts) and their respective values $y_1, \ldots, y_n \in \mathcal{Y}$ (partition parameters $g \in G_k$) the method of least squares minimizes the sum of squares error defined as

$$SSE = \sum_{i=1}^{n} [y_i - P_j(x)]^2 = \sum_{i=1}^{n} [y_i - (c_0 + c_1 x_i + \cdots + c_j x_i^j)]^2. \tag{4.6}$$

To minimize the sum of squares error SSE the first partial derivatives for all polynomial coefficients $c_0, c_1, \ldots, c_n$ are set to zero. This leads to a system of normal equations of the least squares fit whose solution vector comprises the polynomial coefficients with the minimal sum of squares error SSE. We write the equation system to be solved in matrix notation $Xc = y$ where

$$X = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^j \\ 1 & x_2 & x_2^2 & \cdots & x_2^j \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^j \end{bmatrix}, c = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_j \end{bmatrix}, y = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix} \tag{4.7}$$

The premultiplication of the equation (4.7) on both sides with the matrix transpose X^T yields

$$X^T X c = X^T y. \tag{4.8}$$

The matrix equation (4.8) can be either solved numerically to obtain the desired polynomial coefficients, or, if the matrix $X^T X = Z$ is well formed, directly inverted to obtain the desired solution vector $\vec{a}$ [44]. Hence, the set of polynomial coefficients is established by

$$\vec{a} = (X^T X)^{-1} X^T y = Z^{-1} X^T y. \tag{4.9}$$

Furthermore, the approximating polynomial degree j is chosen to be variable for each partition parameter $g \in G_k$. Depending on the gradient of the partition parameters, the goodness of fit of the approximating polynomial obtained can vary with respect to its degree j. For HTTP traffic in Figure 4.2(b), for example, the number of vertices n_k in the high out-degree

Algorithm 1 Calculation of $\vec{a}_{g,k}$ for a partition parameter $g \in G_k$ for a partition $k \in K$

Input: The number of hosts $\mathcal{X} = \{x_1, \ldots, x_n\}$, partition parameter values $\mathcal{Y}_g = \{y_1, \ldots, y_n\}$.
Output: Polynomial coefficients $\vec{a}_{g,k}$ for partition parameter g.

```
1:  a_best ← ∅                                   ▷ coefficients of best polynomial
2:  SSE_min ← INT_max                            ▷ lowest global SSE_min, initialized with a high value
3:  if y_i = 0, ∀y_i ∈ Y then                    ▷ return zero when all values in Y equal zero
4:      return 0
5:  end if
6:  for j ← 4 to 1 do                            ▷ iterate from higher to lower degree
7:      (Ŷ, a_j) ← POLYFIT(X, Y_g, j)            ▷ least squares fitting with degree j
8:      SSE_j ← 0
9:      for i ← 1 to n do                        ▷ compute local SSE_j
10:         SSE_j ← SSE_j + (y_i − ŷ_i)^2
11:     end for
12:     if SSE_j ≤ SSE_min then                  ▷ compare global with local SSE
13:         a_best ← a_j                         ▷ update coefficients
14:     end if
15: end for
16: return a_best
```

partition $k = [10, 100) \times [0, 1)$, computed over all intervals analyzed, increases linearly with the total number of hosts observed. Consequently, a lower degree polynomial provides a better approximation. For DNS traffic in Figure 4.2(a), on the other hand, the gradient of the mean in-degree $\mu_{k,in}$ of vertices in the high in-degree partition $k = [10, 100) \times [10, 100)$ is likely to be better approximated by a higher degree polynomial.

Although the points $x_1, \ldots, x_n$ can be approximated by a direct fit polynomial with degree $n - 1$ which runs exactly through all points, such as the Lagrange polynomial [45], this kind of polynomial tends to oscillate due to its high degree for values that lie between the points $x_1, \ldots, x_n$. Therefore, we limit the maximum degree j of the polynomial computed to $j = 4$ and determine the best polynomial degree for each partition parameter $g \in G_k$ by comparing the sum of squares error SSE.

In Algorithm 1 we outline the process of determining the polynomial with the lowest SSE along with its corresponding coefficients for a partition parameter $g \in G_k$ for a partition k. The number of unique hosts is expressed as $\mathcal{X}$, the corresponding partition parameter values as $\mathcal{Y}_g$. Initially, the lowest global sum of squares error value SSE_{min} is set to a reasonably high number (e.g. highest integer value possible) and an empty set of polynomial coefficients $\vec{a}_{best}$ is initialized.

Next, if all values in $\mathcal{Y}$ equal zero, the value zero is returned since there is nothing to interpolate. Then, iterating from the highest polynomial degree allowed ($j = 4$) in decreasing order up to a degree of one ($j = 1$), the least square approximating polynomial of degree j is computed. The POLYFIT function thereby returns the interpolated partition parameters $\hat{\mathcal{Y}}$, as well as the corresponding polynomial coefficients $\vec{a}_j$. The sum of squares error value SSE_j for the current

polynomial degree j is iteratively computed as defined in (4.6) and compared to the minimal sum of squares error SSE_{min}. As a result, the lowest-degree polynomial with the minimal SSE value is preferred, hence the size of the polynomial coefficients $\vec{a}_{best}$ returned can vary from five (polynomial degree four) to two (linear function).

4.4 Flow Trace Generation

The goal of this step is the generation of flow traces in a flow-based format (e.g., NetFlow), from a collection of traffic templates. The thereby generated traces can be either streamed directly over the network to a flow-based collector device, a flow-based network monitoring or anomaly detection, or saved in binary format on the filesystem, either for further analysis or offline processing (as opposed to analyzing a directly streamed trace).

The flow trace generation process consists of three consecutive steps. First, the templates to produce the desired traffic scenario are selected by the user and customized accordingly if desired. In a next step, for each template selected, a TDG of the underlying traffic structure, which represents the admissible connections between network hosts, is generated. Finally, the flow records, which constitute the underlying traffic structure and whose order is shuffled across all traffic templates, are output by a trace generator. In this chapter, we first describe how traffic templates are customized. Furthermore, we describe how TDGs are established from the partition parameters, determined by the polynomial coefficients in the traffic templates, and outline details about the flow record generation procedure, where flow records are output by a trace generator.

4.4.1 Template Customization

Each traffic template is associated with one specific service port and network protocol. The template parameters, which determine the structural properties as well as flow record attributes of the traffic trace to be generated, can be changed to achieve desired traffic scenarios present in the trace. We divide the customization of traffic templates into three parts and distinguish between the following operations:

1. Parameterization of flow record attributes (fields), such as the number of packets, octets, or duration values.
2. Scaling of the number of flows and the number of unique hosts generated during a fixed time interval of length T.
3. Definition of the time intervals during which each traffic template is active.

To parameterize flow record fields, their respective distributional parameters may be set in the traffic templates, based on the definitions of the template parameters established in Section 4.2. As such, we set the Lognormal distribution parameters d_μ, d_σ to parameterize the flow durations, as well as the parameters p_μ, p_σ for the number of packets and b_μ, b_σ for the number of bytes.

Furthermore, to allow for flexibility with respect to the flow attributes, it is also possible to alter the traffic templates by replacing the Lognormal distribution with different distributions. Our framework currently supports the following definitions of additional distributions or sampling methods for the duration, packets, and bytes values:

Normal Distribution Similarly to the Lognormal distribution, the Normal distribution $\mathcal{N}(\mu, \sigma^2)$ is defined (and customized) by two parameters μ and σ in the traffic templates. We omit the separate notation of the parameter for the duration, packets, and bytes but note that the respective parameters μ and σ are defined for each flow attribute separately (e.g., d_μ and d_σ for the flow duration attribute, where μ and σ are the Normal distribution parameters).

Uniform Distribution Furthermore, for the flow record fields to exhibit values distributed with equal probability between two values, the minimum parameter a and the maximum parameter b for the flow durations, the number of packets, or the number of bytes are specified. The values for the respective flow record attributes are hence distributed uniformly at random, with probability $1/n$ where $n = b - a + 1$, between the parameters specified. Again, we omit the subscript notation for the distribution parameters but note that they are specified for each flow record field separately (e.g., p_a and p_b for the number of packets in the flows records).

Constant Value Additionally, to simulate specific scenarios where the values for flow durations, the number of packets, or the number of bytes desired must be held constant, one single numerical parameter is specified for the respective fields.

Random Value Finally, the distributional properties a flow record field can be omitted. In this special case, the values for the respective fields are chosen at random.

The number of flows (and its variability over time) to be generated during a time interval is controlled by the parameters μ_F and σ_F. Similarly, the number of hosts to be present in the trace is customized with μ_H and σ_H. Finally, the time intervals during which a traffic template is active are specified with the I_{active} parameter using the notation introduced in Section 4.2.3. For example, to appoint a traffic template to be active at every third interval, the I_{active} parameter $I_{active} = /3$ is specified. Similarly, to activate a traffic template for the first three intervals, as well as the last out of twelve intervals flow traces are generated for, the I_{active} parameter can be set to $I_{active} = 1 - 3, 12$.

Naturally, the entire body of the remaining traffic template parameters from Table 4.1 can be adjusted according to specific needs. For traffic templates established in the self-parametrization process, the parameters from the previous paragraphs are the most relevant. However, for user-defined templates, the definition of the polynomial coefficients $\vec{a}_{g,k}$ which determine the partition parameters $g \in G_k$ (and hence the traffic structure), as well as the specification of IP address ranges IP_k for each partition for each partition $k \in K$ is important. We give an example of user-defined templates to model specific traffic scenarios (e.g., a network scan) in Section 5.4.

4.4.2 Generating TDGs from Templates

The edges in a TDG are used during the flow record generation as a list of admissible connections between hosts to preserve the desired structural properties for a given template (e.g., in terms of degree distributions, the number of vertices, and the number of edges). We establish a TDG from the joint degree distribution given by the partitioning. The parameters, i.e., the number of vertices in each partition and the mean out-degrees and in-degree and their respective deviations, are derived from the polynomial functions evaluated for the parameterized number of hosts. This allows the traffic to be parameterized with respect to the number of hosts generated, while maintaining a realistic traffic structure for each time interval.

To establish a TDG, we apply a random graph algorithm, the *matching algorithm* proposed by Newman et al. [46], which efficiently generates random graphs with an arbitrary degree distribution. In general, random graphs are graphs generated by a random process [47]. Bollobás et al. describe two commonly known, basic random graph models [47, 48]: the $\mathscr{G}(n, M)$ model and the $\mathscr{G}(n, p)$ model. The first model comprises all graphs with set of vertices $V = \{1, 2, \ldots, n\}$ and M edges, where each of the graphs occurs with equal probability. The second model consists of all graphs with vertex set $V = \{1, 2, \ldots, n\}$ in which every possible edge in the graph is chosen independently with probability $0 < p < 1$.

In contrast to choosing edges in TDGs at random or with a fixed probability p, we need to generate directed graphs with an arbitrary degree distribution. The joint degree distribution is defined implicitly in the partitioning of the out-degree and in-degree plane, established from the polynomial coefficients $\vec{a}_{g,k}$ in the traffic templates. We divide the algorithm used to establish the TDGs from the partitioning in two steps:

1. First, all graph vertices are initialized for each partition and assigned a set of out-stubs or in-stubs[2] (or both), depending on the degree setting determined by the respective partition parameters.

2. Second, the out-stubs and in-stubs of all vertices are picked randomly in pairs and joined to form edges in the graph.

Algorithm 2 depicts the algorithm used to initialize the vertices with their respective stubs in the graph. First, an empty graph G_{tmp} is created. Then, for each partition $k \in K$, an empty vertex set V_k is initialized and the following set of operations is performed, while iterating over the total number of vertices n_k. Each vertex v_k is assigned a random IP address from the IP address ranges IP_k. This procedure is repeated in case a vertex v_k with the IP address drawn already exists in the set of added vertices V_k for the partition k, until the vertex is assigned a unique IP address, not present in the vertex set V_k.

Then, the vertex v_k is assigned a number of out-stubs and in-stubs drawn from the normal distribution defined by $\mathcal{N}(\mu_{k,out}, \sigma^2_{k,out})$ and $\mathcal{N}(\mu_{k,in}, \sigma^2_{k,in})$. The partition boundaries, determined by the left-closed intervals for both, out-degree and in-degree, impose a restriction on the number of out-stubs and in-stubs allowed for each vertex v_k. Therefore, if the number

[2] Stubs can be considered as open ends of out-going or in-coming edges.

Algorithm 2 Graph generating algorithm

Input: Partitioning of the degree distribution with partitions $k \in K$, IP address ranges definitions IP_k

Output: Graph $G_{tmp} = V$ with vertices $v \in V$ with assigned stubs.

$G_{tmp} \leftarrow \emptyset$ ▷ empty graph
for all $k \in K$ **do** ▷ loop through all partitions $k \in K$
  $V_k \leftarrow \emptyset$ ▷ added vertices for the partition k
  for $i \leftarrow 0$ **to** n_k **do** ▷ iterate over total number of vertices n_k
    repeat ▷ assign unique IP addresses to vertices
      $v_k \leftarrow$ RANDOMELEMENT(IP_k)
    until $v_k \notin V_k$

    repeat ▷ assign out-stubs
      STUBSOUT$(v_k) \leftarrow \mathcal{N}(\mu_{k,out}, \sigma^2_{k,out})$
    until STUBSOUT$(v_k) \geq min_{k,out}$ and STUBSOUT$(v_k) < max_{k,out}$

    repeat ▷ assign in-stubs
      STUBSIN$(v_k) \leftarrow \mathcal{N}(\mu_{k,in}, \sigma^2_{k,in})$
    until STUBSIN$(v_k) \geq min_{k,in}$ and STUBSIN$(v_k) < max_{k,in}$

    $V_k \leftarrow V_k \cup v_k$ ▷ add vertex to set of added vertices V_k

    if added and expected number of stubs do not match **then**
      for all $v_k \in V_k$ **do**
        SCALESTUBS(v_k) ▷ linear scaling of the number of stubs
      end for
    end if
  end for
  $G_{tmp} \leftarrow G_{tmp} \cup V_k$ ▷ add all vertices in V_k to the graph
end for
return G_{tmp}

of stubs assigned lies outside of the partition boundaries $min_{k,out}, max_{k,out}$ for out-degrees or $min_{k,in}, max_{k,in}$ for in-degrees respectively, the stub assignment is repeated. Finally, each vertex v_k is added to the set of added vertices V_k for the partition k.

The expected total number of out-degrees and in-degrees (stubs) for each partition can be easily derived from the mean values for out-degrees and in-degrees respectively (e.g., $\mu_{k,out} \cdot n_k$ for the total number of expected out-degrees in a partition k). Due to the randomness of the normal distribution, the total number of added stubs may significantly differ from these expected totals. Therefore, at the end of the algorithm, the number of out-stubs and in-stubs assigned for each vertex v_k in the partition k is linearly downscaled or upscaled by the SCALESTUBS function.

Algorithm 3 Algorithm to form edges from vertex out-stubs and in-stubs.

Input: Graph $G_{tmp} = V$ with out-stubs and in-stubs assigned to vertices $v \in V$
Output: Final graph G with edges $(u, v) \in E$ (connections).

1: $stubs_{out} \leftarrow \emptyset$ ▷ set of all out-stubs
2: $stubs_{in} \leftarrow \emptyset$ ▷ set of all in-stubs
3: **for all** $v \in V$ **do**
4: $stubs_{out} \leftarrow stubs_{out} \cup \text{STUBSOUT}(v)$ ▷ add all out-stubs from v to $stubs_{out}$
5: $stubs_{in} \leftarrow stubs_{in} \cup \text{STUBSIN}(v)$ ▷ add all out-stubs from v to $stubs_{in}$
6: **end for**
7: **if** $|stubs_{out}| \leq |stubs_{in}|$ **then**
8: **for all** $s_{out} \in stubs_{out}$ **do** ▷ iterate over all out-stubs
9: **repeat**
10: $s_{in} \leftarrow \text{RANDOMELEMENT}(stubs_{in})$ ▷ random in-stub
11: **until** $s_{in} \neq s_{out}$ and $(s_{out}, s_{in}) \notin E$ ▷ prevent self-loops and multiple edges
12: $stubs_{in} \leftarrow stubs_{in} \setminus s_{in}$ ▷ remove from in-stubs $stubs_{in}$
13: $E \leftarrow E \cup (s_{out}, s_{in})$ ▷ connect stub pair
14: **end for**
15: **else**
16: **for all** $s_{in} \in stubs_{in}$ **do** ▷ iterate over all in-stubs
17: **repeat**
18: $s_{out} \leftarrow \text{RANDOMELEMENT}(stubs_{out})$ ▷ random out-stub
19: **until** $s_{out} \neq s_{in}$ and $(s_{out}, s_{in}) \notin E$ ▷ prevent self-loops and multiple edges
20: $stubs_{out} \leftarrow stubs_{out} \setminus s_{out}$ ▷ remove from out-stubs $stubs_{out}$
21: $E \leftarrow E \cup (s_{out}, s_{in})$ ▷ connect stub pair
22: **end for**
23: **end if**
24: **return** G

The scaling factor is computed from the comparison of the added and expected number of out-stubs and in-stubs.

To complete the definition of the graph generating algorithm, we outline how edges between vertices in the graph prepared G_{tmp} are formed in Algorithm 3. First, the sets of all out-stubs $stubs_{out}$ and in-stubs $stubs_{in}$ are initialized from the vertices $v \in V$ in the graph G_{tmp} created in Algorithm 2. Although the partition boundaries impose restrictions on the number of stubs created for a vertex v_k in a specific partition k, the total number of out-stubs and in-stubs assigned to all vertices in the graph may differ. Therefore, the algorithm iterates over the smaller of the two quantities to ensure it terminates. Here, we illustrate the part of the algorithm that iterates over the set of out-stubs $stubs_{out}$, that is if the total number of out-stubs $|stubs_{out}|$ is smaller than the total number of in-stubs $|stubs_{in}|$. The part of the algorithm for the case where the number of out-stubs is greater that the number of in-stubs $|stubs_{out}| > |stubs_{in}|$ is processed similarly and outlined in Algorithm 3 in line 16 to 22.

For each out-stub s_{out}, the algorithm randomly choses an in-stub s_{in} and connects them to form an edge $(s_{out}, s_{in}) \in E$ in the graph, while removing the in-stub from the set of available in-stubs $stubs_{in}$. In order to generate graphs without self-loops and multiple edges, as defined in Section 3.1, we use a modified version of the matching algorithm proposed by Milo et al. [49]. Therefore, if the vertex at the end of a randomly chosen in-stub s_{in} equals the vertex at the end of the current out-stub, a new in-stub s_{in} is chosen at random. Similarly, to prevent the formation of multiple edges between the same vertices, a new in-stub s_{in} is randomly chosen if an edge between the two vertices of the stubs (s_{out}, s_{in}) already exists in the edge set E.

4.4.3 Flow Record Generation

Flow records are generated for consecutive time intervals of length T by a trace generator, whose implementation details we outline in this section. The trace generator operates during two consecutive phases. First, during an *initialization phase*, the generator processes input parameters, i.e., the traffic templates, and initializes various value registries used when generating flow records. Then, in a *generating phase*, flow records whose fields are populated with appropriate values, i.e., the source and destination IP addresses, source and destination ports, values for durations, packets, and similar, are output.

Initialization Phase

The generator implementation comprises various registries for values used when generating the flow records. However, two value registries are of special importance when generating flow records. First, for each traffic template, the flow generator must be aware of the time intervals the traffic template is active at and hence the flow records for a specific service port are being generated. Second, the range of valid source ports P_{src} for each traffic template must be determined and available during the entire flow generation process.

The values in the registries for active intervals as well as for the valid source port ranges are associated with a template specific, unique key. Since each traffic template is associated with exactly one service port and one specific network portal, the template specific registry key is a combination thereof. The template specific registry key for DNS traffic templates, for example, is a combination of the service port 53 and the UDP protocol number 17.

Active Intervals Registry The number of time intervals to be generated is a user-defined input parameter. The active intervals registry R_{int} contains interval numbers for each traffic template during which flow records for the respective service port are being generated. The specification of the I_{active} parameter in traffic templates is optional, hence by default a template is active during all intervals to be generated.

Source Ports Registry The admissible source port range is specified by the P_{src} parameter in each traffic template flow records are generated for. The simplest solution to set the source port flow record field, when generating records for a specific traffic template, is to randomly sample from the entire range of source ports P_{src}. However, this simplification may lead to undesirable traffic structure in the generated trace. When flow records are

generated for a collection of self-parametrized templates, the thereby emerged set of template-specific service ports $P_{service}$ may intersect with the source port ranges P_{src} of some templates. Therefore, we exclude all ports $p \in P_{service}$ from the source port range P_{src} for each template. The source port registry R_{ports} comprises the adjusted range of source ports for each template.

Generating Phase

During the second phase of the flow generation process, flow records in NetFlow format are populated with the appropriate values and output, in the current implementation, in NetFlow version 5 format. The records are either saved in binary format to files on the filesystem, or streamed over the UDP protocol to an IP address and the respective destination port specified. The latter simulates the functionality of a flow-based exporting device, such as routers. Algorithm 4 outlines the entire flow trace-generating algorithm, including both, the initialization and the generating phase in which flow records are output. The respective steps of the algorithm are described as follows:

1. First, the active intervals registry R_{int} and the source port registry R_{ports} are initialized in the INITIALIZE procedure. Then, the algorithm iterates over the number of time intervals m and outputs the flow records for all traffic templates $tpl \in TPL$ active during each interval, performing the following steps.

2. The number of hosts and the number of flows are sampled for each active traffic template from the normal distribution determined by the μ_H, σ_H parameters for the number of hosts, and μ_F, σ_F for the number of flows respectively. Additionally, the optional minimum number of hosts parameter $\min_H$ is respected if present in a template and the sampled number of hosts adjusted accordingly (not depicted in Algorithm 4).

3. Then the partitioning of the degree distribution is derived from the corresponding polynomial parameters $\vec{a}_{g,k}$ for each traffic template, based on the number of hosts drawn in RANDOM from the respective normal distribution $\mathcal{N}(\mu_H, \sigma_H^2)$. Then, the port-based TDG $G = (E, V)$ is generated in CREATEGRAPH from the partitioning as described in Algorithm 2 and Algorithm 3. The edges $(u, v) \in E$ in the thereby established graph comprise admissible connections between network hosts u and v.

4. Next, for each edge $(u, v) \in E$ flow records f are created. The source and destination IP address are determined by the respective edge vertices u and v. Values for duration, packets, and octets flow record fields are chosen by sampling from the log-normal distribution (or a different distribution if specified) defined by the parameters d_μ, d_σ, p_μ, p_σ, and b_μ, b_σ respectively. The end timestamps of flows are distributed uniformly at random over the period marked by the start and end timestamp of the current time period of length T (current interval i in Algorithm 4). The thereby derived end timestamps, in conjunction with the duration value, yield the start time and thus the time-dependent interleaving of flows. The template-specific destination port p_{dst} and source port, chosen

Algorithm 4 Algorithm to generate flow trace from traffic templates.

Input: Collection of traffic templates TPL, number of m time intervals to be generated
Output: Flow records generated for templates TPL.

INITIALIZE() ▷ initialization of R_{int} and R_{ports} registries
for $i \leftarrow 0$ **to** m **do** ▷ iterate over m time intervals
  $edgeRegistry \leftarrow \emptyset$ ▷ admissible connections for each template
  $connectionFlows \leftarrow \emptyset$ ▷ connections with basic traffic structure $\forall tpl \in TPL$
  $remainingFlows \leftarrow \emptyset$ ▷ remaining connections $\forall tpl \in TPL$
  for all $tpl \in TPL$ **do**
    if $tpl \in R_{int}(i)$ **then** ▷ processing active templates only
      $hosts \leftarrow$ RANDOM($\mathcal{N}(\mu_H, \sigma_H^2)$) ▷ number of hosts to generate for tpl
      $flows \leftarrow$ RANDOM($\mathcal{N}(\mu_F, \sigma_F^2)$) ▷ number of flows to generate for tpl
      $G = (E, V) \leftarrow$ CREATEGRAPH($hosts$) ▷ establish TDG
      for all $(u, v) \in E$ **do** ▷ flow records for each $(u, v) \in E$
        $duration \leftarrow$ RANDOM(d_μ, d_σ)
        ...
        $sport \leftarrow$ RANDOM(R_{ports})
        $f \leftarrow (u, v, sport, p_{dst}, R, duration, \ldots)$
        $connectionFlows \leftarrow connectionFlows \cup f$
        $flows \leftarrow flows - 2$ ▷ decrease count (flow and responder flow)
      end for
      for $j \leftarrow 0$ **to** $flows$ **do** ▷ save template keys for remaining flow count
        $remainingFlows \leftarrow remainingFlows \cup tpl$
      end for
      $edgeRegistry \leftarrow edgeRegistry \cup E$ ▷ save edges for tpl
    end if
    $connectionFlows \leftarrow$ SHUFFLE($connectionFlows$) ▷ randomize order
    for all $f \in connectionFlows$ **do** ▷ output flow records
      OUTPUT($f \leftarrow (u, v, sport, p_{dst}, R, \ldots,)$)
      OUTPUT($f \leftarrow (v, u, p_{dst}, sport, R, \ldots)$)
    end for
    $remainingFlows \leftarrow$ SHUFFLE($remainingFlows$) ▷ randomize template keys
    for all $tpl \in remainingFlows$ **do** ▷ get template key for each remaining flow
      $(u, v) \leftarrow$ RANDOM($edgeRegistry(tpl)$)) ▷ get random edge (u, v) for tpl
      $duration \leftarrow$ RANDOM(d_μ, d_σ)
      ...
      $sport \leftarrow$ RANDOM(R_{ports})
      OUTPUT($f \leftarrow (u, v, sport, p_{dst}, R, \ldots)$) ▷ output flow records
      OUTPUT($f \leftarrow (v, u, p_{dst}, sport, R, \ldots)$)
    end for
  end for
end for

from the port range specified in the source ports registry R_{ports}, are set for each flow record f to be generated.

5. The body of thereby created flow records is added to the list of connections $connectionFlows$ for the current interval i. This list contains the basic traffic structure (comprising all edges $(u, v) \in E$ from the generated graph G) for all service ports flow records are generated for.

6. Usually, the number of flows for a template is greater than the number of connections between hosts. Therefore, for the remaining number of flows, we add the template key (combination of service port and network protocol) to the list of remaining flows $remainingFlows$ for an interval. Furthermore, to be able to generate flow records from this list of remaining flows for a template, we add the list of admissible connections defined by the TDG edges E to a template-based edge registry $edgeRegistry$.

7. Then, the entry order in the prepared list of flow records for all active traffic templates is randomized in SHUFFLE, and two corresponding flow records[3] are output for each element.

8. Similarly, for the remaining number of active traffic template flows for the current time interval i, flow records are generated by first randomizing the order of the templates (template keys) in the list of remaining flows $remainingFlows$. Then, for each template key, the algorithm randomly chooses an edge (u, v) from the list of admissible connections for the respective template, and generates the flow records whose fields have been populated with random values sampled from the corresponding distributions.

Important to say is that the individual flow records output by the flow trace generator in the OUTPUT function in Algorithm 4 are properly buffered, such that the generated NetFlow packets contain a realistic number of flow records (e.g., 30), as opposed to some of the current flow-based trace generators which incorrectly output NetFlow packets that contain only one flow record as described in Section 2.2.

4.5 Pluggable Architecture

In Section 2.2 we described the currently existing flow-based trace generators and presented their drawbacks. Although these tools are capable of generating flow records directly, without the detour of a packet-based traffic generator, one of their disadvantages is the missing ability to determine the exact values of the data generated. For example, they do not provide any instruments to programmatically determine the exact number of flows, the number of unique hosts, the top service ports, or other flow-related attributes such as the generated number of packets or bytes. Therefore, when streaming the generated trace to a flow-based monitoring

[3] As a simplification, we assume that for every flow between two hosts a responder flow exists. The attributes of the responder flow are set by reversing the source and destination IP addresses and ports.

system, its analysis accuracy cannot be reliably determined since the values analyzed by the system cannot be compared to the values generated.

In this section, we introduce the pluggable architecture of our framework. Our flow trace generator provides several entry points, called "hooks", for plugins to interact with the generator during different phases of the generation process. Therefore, the data generated by our framework can be subjected to additional examinations if desired. For example, the values effectively generated can be compared to the values established during analysis of the generated trace, performed by a flow-based network monitoring system.

The plugins are implemented as object-oriented Perl modules. Each plugin must provide at least one plugin instantiation method to be recognized and activated by the trace generator. Furthermore, a set of additional plugin methods can be implemented for added functionality. The flow trace generator invokes each method during different phases of the flow record generation process. We describe the collection of mandatory and optional plugin methods as follows.

`new()` This is the mandatory plugin instantiation method. Plugins without an instantiation method are not activated and thus cannot be used by the trace generator.

`active()` This method determines whether the implementing plugin is activated by the trace generator or not. The definition of this method is optional, plugins without an `active()` method are ignored by the generator, for a plugin to be activated, the value returned by the method must evaluate to `TRUE` (in Perl, the Boolean data type representing the truth value is `1`). This method also simplifies the activation and deactivation of different plugins employed for different evaluations.

`init(@args)` This plugin method is called exactly once during the initialization phase of the flow trace generator. Plugins that require internal configurations should implement this method. The entire generator parameters array is passed as a method parameter to the plugin. For example, to determine the existence of certain generator flags (e.g. verbose), each plugin can inspect the `@args` parameter array.

`record($record)` This plugin method is called each time a flow record is output. The flow record is passed in anonymous array `$record` to all active plugins. The practical application of this hook is the possibility for plugins to establish exact counters of flow record field values generated. For example, to determine the exact number of bytes present in the trace generated, a plugin can increase internal counters for the number of bytes each time its `record()` method is called with the respective value for the number of bytes in the `$record` parameter.

`finit()` The last plugin method is called exactly once after the flow records for all intervals have been generated. The method targets post-processing plugin operations and is parameterless. For example, a user-defined plugin can compare values generated with values analyzed by a flow-based system the flow records have been streamed to.

Chapter 5

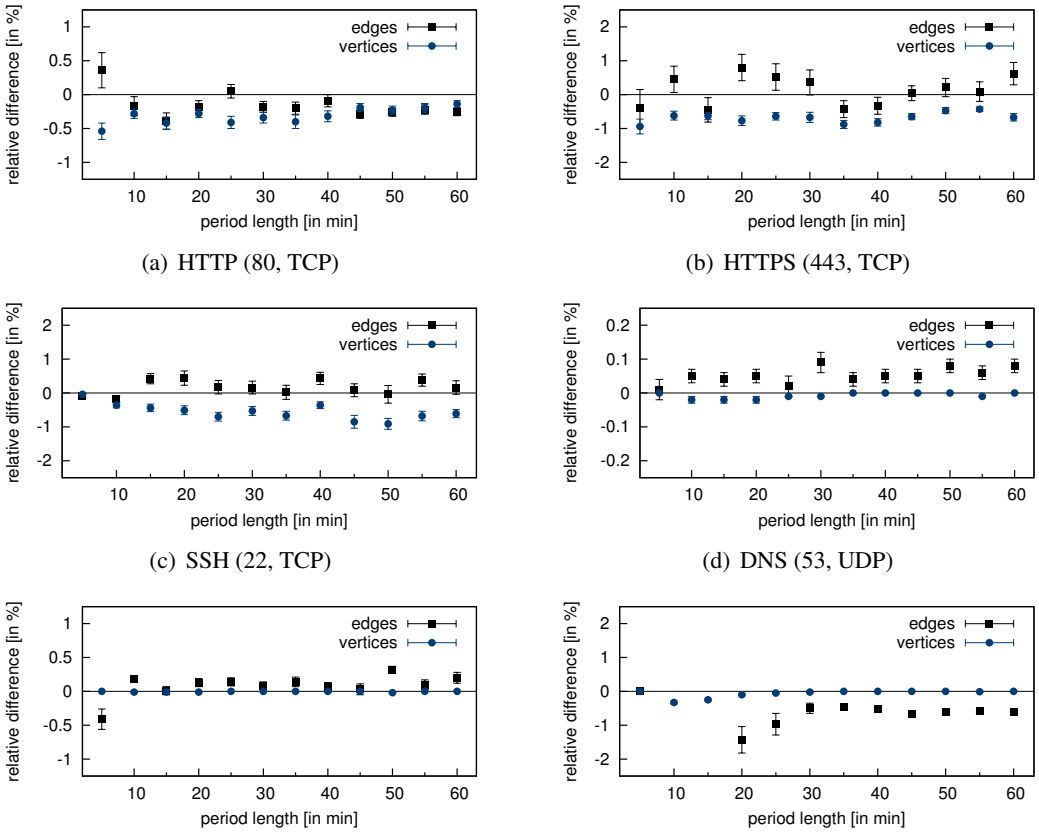
Evaluation

In this chapter, we evaluate the flow generation and self-parametrization technique introduced in Sections 4.3 and 4.4. We assess the accuracy of graph generation algorithm by creating TDGs from partition parameters and comparing the graph metrics of the generated graphs and graphs established from real traffic traces. Moreover, we show that the flow records generated by our technique exhibit similar structural properties as the records from original traffic and provide performance measurements of the record generation. In a case study, we demonstrate the ability of using templates from background traffic in combination with user-defined templates to generate new traces containing abnormal traffic events. Furthermore, we examine the performance of the flow trace generator in Section 5.5 and outline the limitations of our approach at the end of this chapter in Section 5.6.

The evaluation testbed consists of a desktop-class commodity Core 2 Duo processor with 3 MB shared L2 cache running at 3.06 GHz with 4 GB RAM. We implemented a prototype of our technique in Perl 5. We use two different data sets, introduced in Section 3.2, for the evaluation of our framework. The first data set consists of 10 days of NetFlow records of the internal traffic from an average-sized campus network, collected between May 1, and May 9, 2009. The second data set comprises 7 days of NetFlow records collected at a large hosting environment between April 14, and April 20, 2008.

5.1 Graph Generation

We evaluate the graph generation technique in terms of the number of vertices and the number of edges as well as the degree distribution of TDGs created from the partition parameters G_k. First, we generate TDGs directly from static partition parameters G_k. Then, we study the effect of approximation of the partition parameters using a polynomial function with parameters $\vec{a}_{g,k}$. We analyze flow traces from the campus network data set for a time period of one hour, divided into 12 intervals 300 s length. The trace analyzed comprises 15 353 unique IP addresses and 1.2 million flows. We note that the experiments in this section performed on traces from different time periods show quantitatively and qualitatively similar results, hence we only display results for the trace and time interval previously described.



(a) HTTP (80, TCP)

(b) HTTPS (443, TCP)

(c) SSH (22, TCP)

(d) DNS (53, UDP)

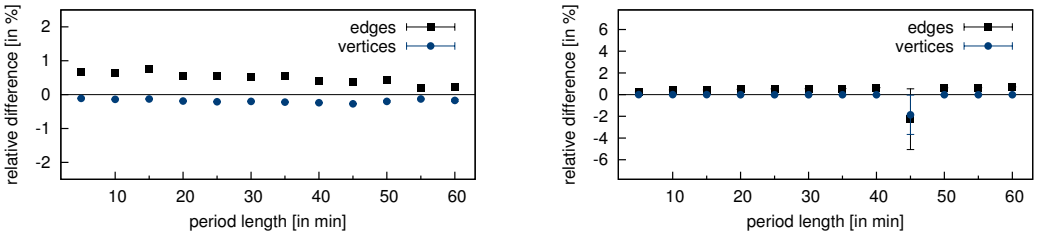
(e) NetBIOS NS (137, UDP)

(f) LDAP (389, UDP)

Figure 5.1: Relative difference between the number of connected vertices and edges of the original graph and the graph generated from partitions, established from the campus network traffic (20 repeated runs of the graph algorithm for each period length).

For each interval, we extract the partitioning with the corresponding partition parameters G_k from the traffic accumulated over the analyzed intervals. Then, we generate a TDG for each partitioning by means of the graph-generating algorithm described in Section 4.4.2. We repeat the random graph algorithm for different ports and protocols 20 times to increase the stability of the measurements and to introduce meaningful average values. In Figure 5.1 we show the relative difference in the number of connected vertices $|V|$ (without isolated vertices) and the number of edges $|U|$ between the original and the generated graphs for six different service ports for both, TCP and UDP protocols.

We observe that the random graph algorithm introduces an average error of 0.31% ($\sigma = 0.11$) for HTTP traffic in Figure 5.1(a) in the number of vertices, and 0.22% ($\sigma = 0.09$) in the number of edges respectively. Similarly, the difference in the number of vertices and edges for DNS traffic in Figure 5.1(d) is on average 0.01% ($\sigma = 0.01$) for vertices, and 0.06% ($\sigma = 0.02$) for edges. In some cases, other service ports analyzed exhibit slightly increased average errors.



(a) HTTP (80, TCP)

(b) DNS (53, UDP)

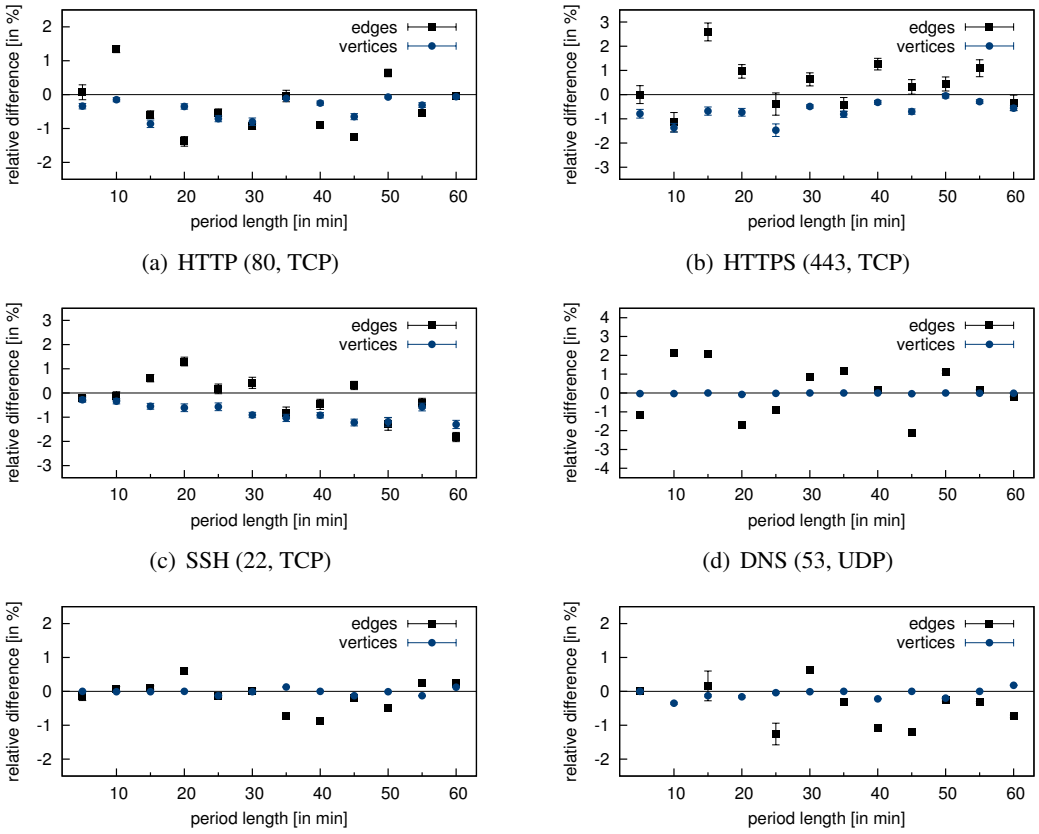
Figure 5.2: Relative difference between the number of connected vertices and edges of the original graph and the graph generated from partitions, established from the hosting provider traffic (20 repeated runs of the graph algorithm for each period length).

Web traffic on the service port 443 (HTTPS) in Figure 5.1(b), for example, in contrast to web traffic on service port 80 (HTTP), shows an average error 0.68% ($\sigma = 0.14$) for the number of vertices and 0.39% ($\sigma = 0.20$) for the number of edges. This can be attributed to lower traffic portion on this service port. In contrast to HTTP traffic, the number of vertices in the graphs for HTTPS traffic is considerably lower. Therefore, a difference in the number of out-stubs and in-stubs, drawn from higher-degree partition (e.g., $[10, 100)$) when generating graphs from the partitionings, has a stronger impact on the relative difference in vertices and edges.

Furthermore, for the sake of completeness, we note the numerical values for the remaining service ports. For SSH traffic on service port 22 (TCP) in Figure 5.1(c), the average error is 0.55% ($\sigma = 0.23$) in the number of vertices, and 0.21% ($\sigma = 0.15$) in the number of edges. For LDAP service port 389 (UDP) in Figure 5.1(f), the difference is 0.07% ($\sigma = 0.11$) in vertices and 0.99% ($\sigma = 0.85$) in edges. Finally, the values for service port 137 on UDP (NetBIOS NS) in Figure 5.1(e) are 0.01% ($\sigma = 0.01$) for the number of vertices, and 0.15% ($\sigma = 0.11$) for the number of edges respectively.

For most service ports, the relative difference in the number of vertices is smaller than the difference with respect to the number of edges. This is likely due to the mismatch of the total number of out-stubs and in-stubs assigned to vertices in partitions. Although the number of stubs assigned to a vertex is linearly scaled in the graph-generating algorithm (c.f. Algorithm 2), in order to match the total number of stubs in a partition as closely as possible, the number of stubs assigned can still differ. We attribute this difference to the nature of the normal distribution and to the rounding applied to the sampled values (degrees values are integers). Vertices with assigned stubs, even though the number of stubs might differ from the vertex degrees in the original graphs, are successfully connected by the edge forming algorithm. Therefore, the number of isolated vertices is minimal in the graphs generated.

Next, we perform an evaluation of the graph-generating technique on the hosting provider network traces, which contain a considerably higher number of unique IP addresses and flows. Particularly, the data set analyzed comprises 5.75 million flows and 158 931 unique IP addresses. Figure 5.2 shows the relative differences in the number of vertices and edges between the original and generated graph for each interval for DNS (53, UDP) and HTTP (80, TCP) traffic. The average error for HTTP traffic in Figure 5.2(a) is 0.18% ($\sigma = 0.05$) in vertices and 0.48%



(a) HTTP (80, TCP)

(b) HTTPS (443, TCP)

(c) SSH (22, TCP)

(d) DNS (53, UDP)

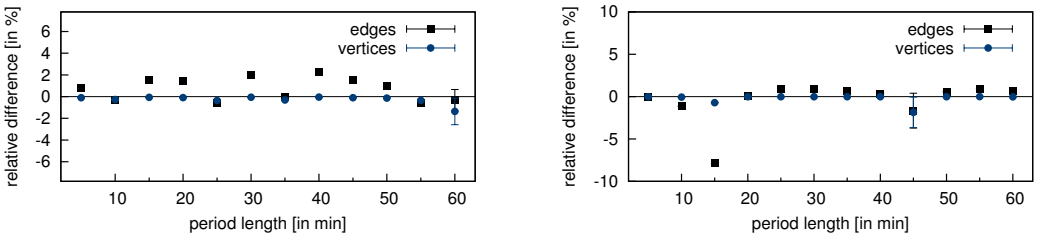
(e) NetBIOS NS (137, UDP)

(f) LDAP (389, UDP)

Figure 5.3: Relative difference between the number of connected vertices and edges of the original graph and the graph generated from partitionings established from polynomial coefficients $\vec{a}_{g,k}$ in the campus network traffic (20 repeated runs of the graph algorithm for each period length).

($\sigma = 0.17$) in edges. The higher difference in the number of vertices and edges observed for DNS traffic in Figure 5.2(b) is explainable due to the previously mentioned mismatch between the number of out-stubs and in-stubs, assigned from the partition parameters, which can lead to isolated vertices and fewer connected edges. However, the graph shows that the average difference is around 2% (in some cases even around zero) which leads to acceptable approximation of the number of edges and vertices. The average error for DNS traffic computed over the sum of intervals analyzed is 0.17% ($\sigma = 0.51$) in the number of vertices and 0.68% ($\sigma = 0.49$) in the number of edges.

Now, we apply the self-parametrization technique to extract the polynomial coefficients $\vec{a}_{g,k}$ over the same intervals. Based on the thereby derived coefficients, the partition parameters G_k are computed from the polynomials and the TDGs are generated from the thereby derived partitioning. We depict the results for different service ports for the campus network traffic in Figure 5.3, and for DNS and HTTP traffic from the hosting provider traces in Figure 5.4. We



(a) HTTP (80, TCP)

(b) DNS (53, UDP)

Figure 5.4: Relative difference between the number of connected vertices and edges of the original graph and the graph generated from partitionings established from polynomial coefficients $\vec{a}_{g,k}$ from the hosting provider traffic (20 repeated runs of the graph algorithm for each period length).

observe that the graphs generated in this manner exhibit slightly higher average errors.

For the campus network traffic, the additional approximation step of using the polynomial parameters introduces an average error of 0.39% ($\sigma = 0.28$) for the number of vertices, and 0.69% ($\sigma = 0.47$) for the number of edges for HTTP traffic in Figure 5.3(a). For DNS traffic in Figure 5.3(d) the average error is 0.02% ($\sigma = 0.02$) for the number of vertices, and 1.15% ($\sigma = 0.71$) for the number of edges. Similarly, the average difference in vertices is 0.69% ($\sigma = 0.80$) and in the number of edges 0.78% ($\sigma = 0.66$) for HTTPS traffic in Figure 5.3(b). The absolute values for the average errors for the remaining service ports are 0.79% ($\sigma = 0.33$) for vertices and 0.66% ($\sigma = 0.51$) for edges for SSH traffic in Figure 5.3(c), 0.06% ($\sigma = 0.06$) for vertices and 0.32% ($\sigma = 0.27$) for edges in the service port 137 (NetBIOS NS) traffic in Figure 5.3(e), and finally 0.11% ($\sigma = 0.11$) for vertices and 1.15% ($\sigma = 1.29$) for edges in the LDAP (service port 389) traffic in Figure 5.3(f).

The average errors for the graphs generated from the hosting provider network traffic, comprising a much higher number of unique hosts and flows for each time interval analyzed, are 0.26% ($\sigma = 0.35$) for the number of vertices and 1.03% ($\sigma = 0.68$) for the number of edges for HTTP traffic in Figure 5.4(a). For DNS traffic in Figure 5.4(b), the average different in the number of vertices is 0.23% ($\sigma = 0.53$) and the average error in the number of edges is 1.32% ($\sigma = 2.03$).

These increased differences can be explained through the polynomial approximation of the partition parameters. For higher degree polynomials, the approximating polynomial function does not run through the original values (cf. Figure 4.2(a)). Consequently, the number of out-stubs and in-stubs are sampled from partition parameters that differ from the original values. This could be improved by using the actual partition parameters observed at each interval in the traffic templates, instead of approximating the values with a polynomial function. Partition parameters G_k could then be derived for numbers of hosts by linearly interpolating between partitionings at two intervals (two values for the number of hosts). However, the complexity of traffic templates and the user-based definition thereof would increase.

Similar to graphs generated from static partition parameters G_k, the difference in the number of vertices, for most service ports, is distributed steadily around the zero value. Hence, the graphs generated contain roughly the same number of vertices (hosts), and usually only a few isolated

vertices (hosts without any connections) are present.

To sum up, we have shown that graphs generated by means of the random graph algorithm exhibit only small differences in the number of hosts and the number of vertices, compared to graphs generated from original traffic traces. Furthermore, the polynomial coefficients, although introducing slightly higher average errors compared to the static partition parameters, provide good approximations of the partition parameters for arbitrary host populations.

5.2 Graph Degree Distribution

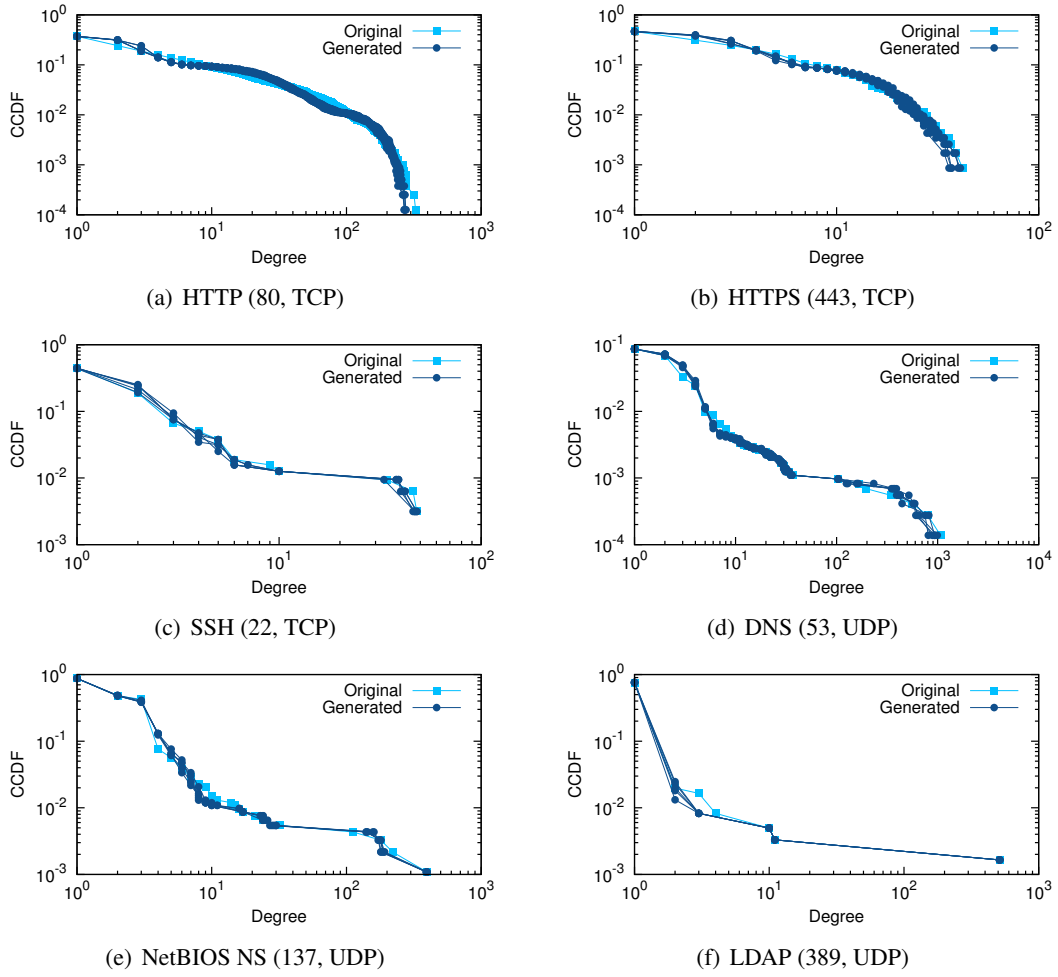
We further compare the degree distribution of the underlying undirected graph for both, the original and generated graphs on different ports by computing the empirical Complementary Cumulative Distribution Function (CCDF) for the degree, determined by the number of adjacent edges (connections), of each vertex $v \in V$. In Figure 5.5 we depict the CCDF of the original graphs and those generated from partition parameters, all graphs created from an analysis of four hours of the campus network traffic trace. The campus network trace analyzed consists of 43 385 unique IP addresses and 5.1 million network flows. Similar to the experiments in Section 5.1, we note that the experiments in this section performed on the hosting provider traces, as well as on the campus network traces, from different time periods show quantitatively and qualitatively similar results. Therefore, we only depict results for the trace and time interval previously described.

Similarly, in Figure 5.6 we depict the CCDF comparison of original graphs and graphs generated from the hosting provider data set. The analysis comprised a one hour long traffic trace, worth of 158 931 network hosts and 5.7 million flows. To show the stability of our measurements, we repeated the graph-generating algorithm five times and plot all curves (hence the overlapping points) for both traces analyzed.

We observe that the degree distribution curves for most service ports match well. Especially for the the campus network traces graphs, only slight variations between the vertex degrees in the original and generated graphs are present. In some cases, the probability of vertices having a certain degree is different in the graphs generated when compared to the original graphs, hence small sections of some distribution curves do not match exactly.

For traffic on service port 389 (LDAP) on the UDP protocol in Figure 5.5(f), for example, the mismatch of the two curves around vertex degree $x = 3$, where $P(X > x)_{original} > P(X > x)_{generated}$, suggests that the original graph comprises more higher-degree vertices for degrees values around three. This can be attributed to the sampling of the number of out-stubs and in-stubs from the partition parameters, i.e., in partitions $[2, 10)$ where the vertex degrees in the generated graphs can take a wider range of values than in the lower-degree partitions, such as $[0, 1)$ or $[1, 2)$.

Similarly, the distribution curves for graphs generated from the campus network traffic on service port 80 (HTTP) on the TCP protocol in Figure 5.6(a) exhibit similar deviations from the original values. For degree values from three to ten, the generated graph contains more lower-degree vertices. Moreover, the graph generated comprises more higher-degree vertices as indicated by the marginal slope difference in distribution curves for degree values $x \in$



(a) HTTP (80, TCP)

(b) HTTPS (443, TCP)

(c) SSH (22, TCP)

(d) DNS (53, UDP)

(e) NetBIOS NS (137, UDP)

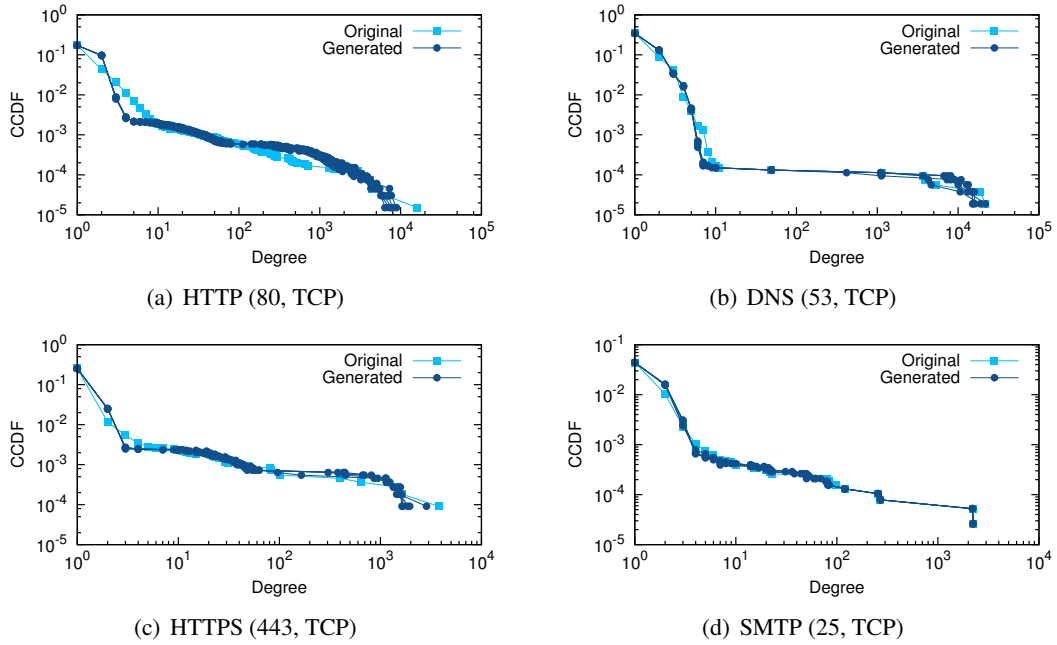
(f) LDAP (389, UDP)

Figure 5.5: CCDF, $P(X > x)$, of the degrees of each vertex in the original graphs and graphs generated from an analysis of four hours of the campus network traffic trace.

$\{100, \ldots, 1000\}$. We explain this, in analogy to the curve slope difference for lower-degree values, with the higher variance of vertex stubs drawn for high-degree partitions, e.g., $[100, \infty)$, especially for the hosting provider traffic where vertex degree values are generally much higher than in graphs generated from the campus network traffic (c.f. Figure 5.5(a)).

The distribution curve differences for high-volume[1] traffic traces suggests that an adaptation of left-closed intervals, which determine the partition boundaries of the partitioning of the out-degree and in-degree plane, could help increasing the accuracy of the degree distribution in the graphs generated. Furthermore, the intervals could be determined dynamically in the self-parametrization process, depending on vertex degree values for graphs established from the trace analyzed.

[1] High-volume traces in terms of the number of unique IP addresses (network hosts) in the traffic.



(a) HTTP (80, TCP)

(b) DNS (53, TCP)

(c) HTTPS (443, TCP)

(d) SMTP (25, TCP)

Figure 5.6: CCDF, $P(X > x)$, of the degrees of each vertex in the original graphs and graphs generated from an analysis of one hour of the hosting provider traffic trace.

To sum up, we find the graph-generating algorithm and our partitioning approach reproduces the original degree distribution and thus the host connectivity properties accurately. The number of vertices with a specific out-degree and in-degree in the generated graphs is close to values found in the original graphs.

5.3 Traffic Structure

5.3.1 Graph Metrics

We use the self-parametrization process on the campus network traces to extract the traffic template parameters. The flow records analyzed span over 48 consecutive 300 s long time intervals, comprising a total of 5.15 million flows and 43 385 unique IP addresses. We generate traffic traces for a period of 4 hours from the templates and create the TDGs. Similarly, we establish the TDGs from the original traffic traces, with equal period length, from three different days of the campus network data set, comprising an average of 4.76 ($\sigma = 0.22$) million flows and 38 014 ($\sigma = 4\,968$) unique IP addresses. Then, we compare the TDGs by means of eight graph metrics, that capture the structure of network traffic, which have been introduced by Iliofotou et al. [22] and recently refined [50].

Table 5.1 shows a comparison of the graph metrics of the original and generated traces for different service ports for both, TCP and UDP network protocols. The traces have been generated from the self-parametrized traffic templates 10 times, the values in the table depict the

generated graph metrics averaged over these 10 runs. Similarly, the values for the original graph metrics have been averaged over the three different days analyzed. The values in parentheses, for both the graphs of the original traffic and the traces generated, show the respective standard deviations. The graph metrics used are described as follows.

Average Degree The average degree metric is computed by counting the number of edges, both out-going and in-coming, adjacent to a vertex. Therefore, this metric ignores the directionality of the graph and considers the underlying undirected graph of a TDG. High average degree values are an indication of how tightly connected a graph is.

OnlyIn, OnlyOut, InO In Section 3.3 we introduced TDGs visualizations and distinguished between different roles of vertices in the graphs. Each TDG usually comprises vertices with different degree properties. Vertices with zero in-degree values (sources) have out-going edges only, modeling hosts in the network traffic which act as service initiators (clients). Similarly, vertices with zero out-degree values (sinks) have only incoming edges and represent hosts which act as service providers (servers). The third class of graph vertices are those with non-zero out-degrees and non-zero in-degrees, hence vertices with both, outgoing and incoming edges. Therefore, we capture the directionality of the graph by the percentage of sources (OnlyOut), sinks (OnlyIn), and vertices with both outgoing and incoming edges (InO).

LWCC Each graph can be split into a number of connected subgraphs, called components. The Largest Weakly Connected Component (LWCC) indicates the number of vertices in the maximal connected subgraph as a percentage of the total number of vertices $|V|$. In their work, Iliofotou et al. use the Giant Connected Component (GCC) metric to quantify the connectivity of a graph. When considering the underlying undirected graph of a TDG, this metric equals the LWCC metric in the directed graph. Densely connected graphs exhibit higher LWCC values.

Max Degree Ratio (MDR) The Max Degree Ratio (MDR) is the the maximum vertex degree in the graph normalized by the total number of vertices $|V|$ minus one. Therefore, this metric provides information about the maximum possible degree of a node in the graph. High MDR value suggests the existence of a dominant high-degree vertex, such as, for example, in graphs established from DNS traffic where high MDR values are likely due to the presence of high-degree DNS servers.

RU The Relative Uncertainty (RU) metric measures the uniformity of the vertex degree distribution. The probability of a randomly selected vertex having a degree j is defined by the graph degree distribution $P(j) = n(j)/j$, for $j \in \{1, \ldots, j_{max}\}$, where the number of vertices with degree j is defined as $n(j)$, and j_{max} is the maximum vertex degree in the graph. The entropy of the degree distribution $H(X)$ is defined as $H(X) = -\sum_{j}^{j_{max}} P(j)\log(P(j))$ with $\log(P(j)) = 0$ if $P(j) = 0$. The Relative Uncertainty is subsequently computed from the formula $H(X)/\log_2 j_{max}$ [50]. The RU value one denotes a uniform degree distribution, values closer to zero denote higher variety in the degrees.

Table 5.1: Comparison of graph metrics of TDGs established from original and generated traffic.

Port	Trace type	Vertices	Edges	Avg. Degree	InO, %	OnlyOut, %	OnlyIn, %	LWCC, %	MDR	RU	r
					TCP						
80	orig	8389	26248	3.13 (0.09)	0.52 (0.02)	10.36 (0.67)	89.12 (0.66)	99.62 (0.21)	0.07 (0.00)	0.26 (0.01)	-0.37 (0.01)
	gen	8157	24860	3.05 (0.01)	0.52 (0.00)	10.63 (0.04)	88.84 (0.04)	98.74 (0.17)	0.04 (0.00)	0.28 (0.00)	-0.34 (0.00)
1352	orig	622	1389	2.23 (0.06)	32.62 (1.94)	47.82 (1.52)	19.56 (2.07)	94.38 (1.44)	0.49 (0.01)	0.30 (0.00)	-0.55 (0.02)
	gen	645	1479	2.29 (0.02)	33.04 (0.17)	49.02 (0.12)	17.93 (0.17)	98.36 (0.57)	0.49 (0.03)	0.31 (0.00)	-0.50 (0.01)
443	orig	1150	2079	1.81 (0.05)	2.05 (0.33)	32.20 (0.32)	65.75 (0.53)	89.47 (0.65)	0.07 (0.00)	0.39 (0.01)	-0.15 (0.03)
	gen	1154	2150	1.86 (0.01)	2.15 (0.03)	32.83 (0.13)	65.02 (0.13)	95.74 (0.68)	0.04 (0.00)	0.45 (0.01)	-0.05 (0.02)
139	orig	671	1374	2.05 (0.04)	11.60 (1.38)	83.89 (1.29)	4.51 (0.78)	97.07 (1.91)	0.69 (0.02)	0.24 (0.01)	-0.63 (0.01)
	gen	670	1371	2.05 (0.01)	14.48 (0.26)	80.02 (0.10)	5.50 (0.20)	99.24 (0.43)	0.58 (0.05)	0.26 (0.00)	-0.59 (0.02)
					UDP						
53	orig	7265	8961	1.24 (0.03)	0.42 (0.05)	19.16 (1.25)	80.43 (1.23)	99.99 (0.01)	0.80 (0.01)	0.05 (0.01)	-0.57 (0.01)
	gen	7584	9447	1.25 (0.00)	0.42 (0.00)	20.12 (0.00)	79.46 (0.00)	99.84 (0.07)	0.78 (0.00)	0.05 (0.00)	-0.54 (0.01)
137	orig	920	2570	2.79 (0.01)	81.93 (0.56)	9.53 (0.37)	8.54 (0.51)	97.17 (0.29)	1.45 (0.01)	0.21 (0.00)	-0.52 (0.01)
	gen	929	2632	2.83 (0.01)	83.47 (0.10)	9.42 (0.10)	7.12 (0.03)	100.00 (0.00)	1.48 (0.01)	0.22 (0.00)	-0.46 (0.01)
389	orig	589	1054	1.79 (0.01)	0.40 (0.07)	97.46 (0.09)	2.15 (0.04)	97.68 (0.04)	0.89 (0.01)	0.11 (0.00)	-0.93 (0.01)
	gen	594	1010	1.70 (0.01)	0.00 (0.00)	99.66 (0.00)	0.34 (0.00)	100.00 (0.00)	0.87 (0.00)	0.09 (0.00)	-1.00 (0.00)
123	orig	1281	3634	2.84 (0.02)	80.85 (0.69)	17.85 (0.73)	1.30 (0.06)	96.94 (0.50)	0.60 (0.02)	0.21 (0.00)	-0.70 (0.01)
	gen	1264	3603	2.85 (0.01)	80.60 (0.06)	18.28 (0.07)	1.12 (0.03)	99.97 (0.06)	0.49 (0.04)	0.23 (0.00)	-0.72 (0.01)

r The assortativity coefficient $r \in [-1, 1]$ provides a summary metric of the relationship between the degrees of adjacent vertices. If the assortativity coefficient value is zero, the vertex degrees have random correlations and there no relationship between the degrees of adjacent vertices. When the assortativity coefficient is greater than zero, $r > 0$, the graph is assortative with respect to the vertex degrees, hence high-degree vertices are likely to be connected to other high-degree vertices. If the assortativity coefficient is smaller than zero, $r < 0$, on the other hand, low-degree vertices are likely to be connected to high-degree vertices.

We observe that the differences in the metrics in Table 5.1 are very small. In general, the directionality of the original and the generated graphs is reproduced accurately. The relative amount of sinks, sources, and vertices with both outgoing and incoming edges are a close match. Similarly, the correlation between the degree values of two adjacent vertices are almost equal in the original and generated graphs. Graphs generated from DNS traffic, for example, exhibit a negative assortativity coefficient (approx. -0.5), meaning low-degree vertices (DNS clients) are likely to connect to high-degree vertices (DNS servers). Conversely, for HTTP traffic, the assortativity coefficient has values closer to zero (approx. -0.3) which suggests the presence of web servers with lesser incoming connections.

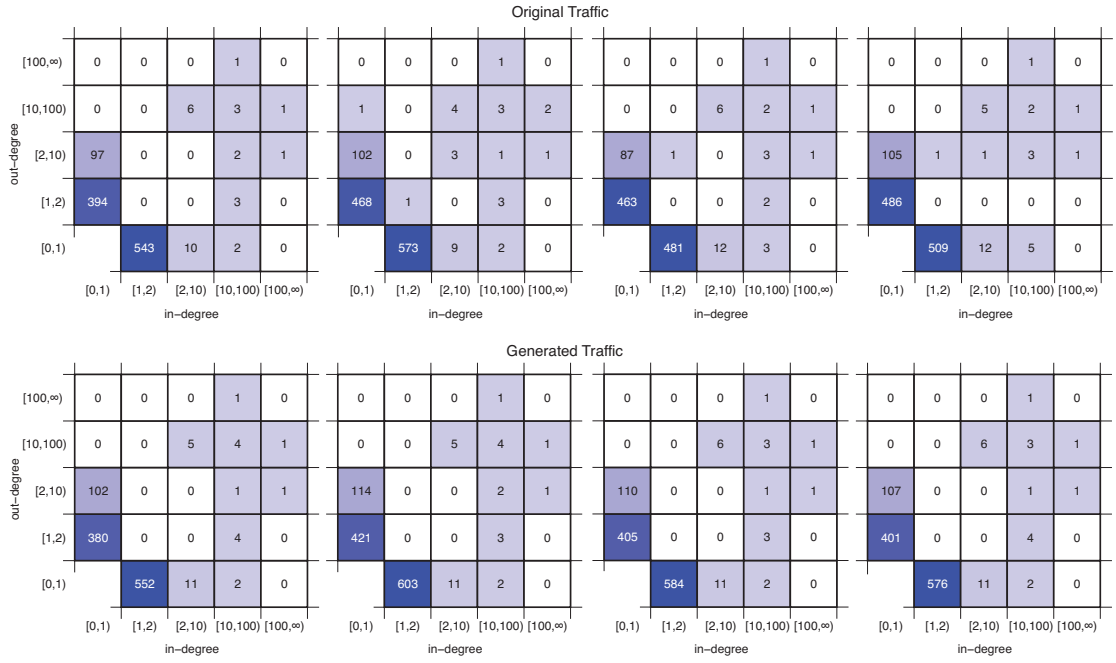
Interestingly, for traffic on service port 137 on the UDP protocol, a discrepancy exists between the number of sinks, as well as the number of vertices with out-degrees and in-degrees, in the original and generated graphs. Consequently, the graph for this service port is completely connected, as indicated by a LWCC value of 100%. In this particular case, this is likely due to the matching algorithm which spuriously connects out-stubs and in-stubs for a small amount of vertices, leading to an increased number of connection between vertices that are not connected in the original graph.

To sum up, we have shown that the traffic structure of the original and generated traffic is highly comparable by means of various graph metrics. We conclude that our approach leads to traces with realistic traffic structure for time intervals of fixed length.

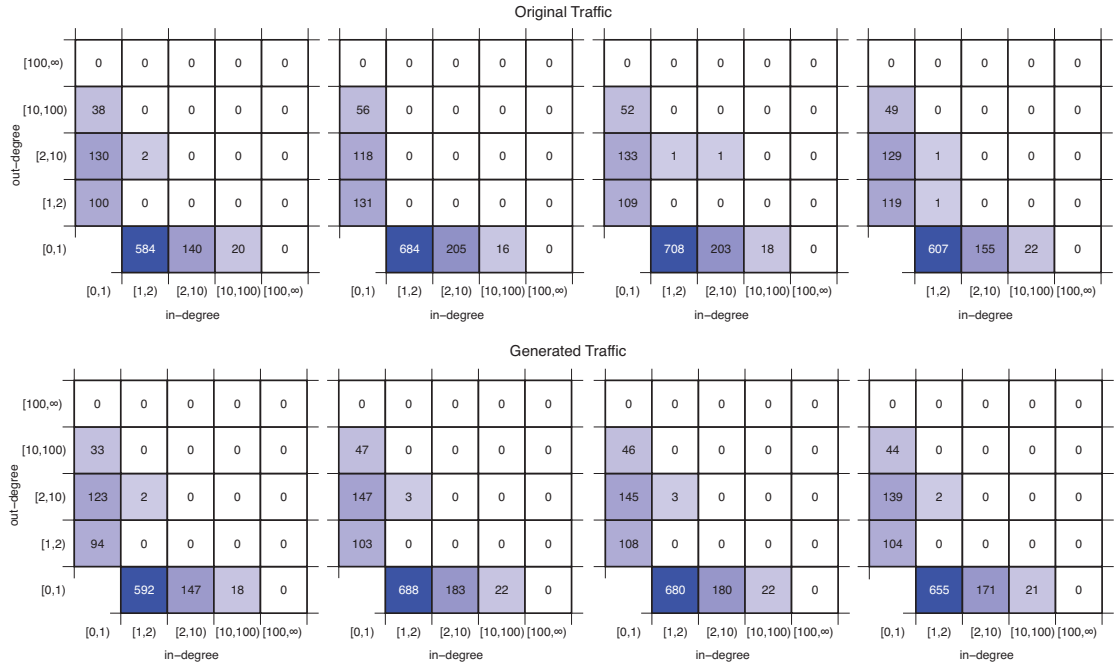
5.3.2 Partitioning

We use the self-parametrization process on the campus network traces to extract the traffic template parameters. The flow records analyzed span over 36 consecutive 300 s intervals, each comprising an average of 110 313 ($\sigma = 8804$) flows and 4 216 ($\sigma = 468$) unique IP addresses. The self-parametrization parameter for the number of top service ports n is set to $n = 50$, establishing the top 50 service ports in terms of unique IP addresses and the number of flows in the trace spanned by the 36 time intervals. The combination of the collections of all top service ports yields 132 traffic templates, 71 templates for the TCP protocol and 61 templates for the UDP protocol, based on which we generate the flow records for 36 consecutive time intervals.

In Figure 5.7 we depict the partitioning from four consecutive intervals established from the original and generated trace. The number (and color) for each partition indicate the count of hosts observed in the partition. Figure 5.7(a) shows the partitioning visualizations for DNS (service port 53) traffic. We observe that our method generates traces with highly similar traffic



(a) DNS (53, UDP)



(b) HTTP (80, TCP)

Figure 5.7: Comparison of graph partitionings between original and generated traffic created from four consecutive time intervals of 300 s length. Figure 5.7(a) shows graph partitionings for DNS (service port 53) traffic on the UDP protocol. Figure 5.7(b) shows graph partitionings for HTTP (service port 80) traffic on the TCP protocol.

structure. For example, the graphs established from the original trace comprise a high number of hosts in the zero in-degree and low out-degree ($[1, 2) \times [0, 1)$) partition, which are likely to be DNS clients.

The high number of hosts in the zero in-degree and low out-degree partition ($[0, 1) \times [1, 2)$) are most likely DNS servers. Furthermore, a number of hosts exists in the high out-degree and high in-degree partitions. These hosts are likely servers propagating name server resolution requests to other DNS servers. The partitioning established from the generated trace exhibit highly similar values (and color intensities), hence the traffic structure is reproduced accurately.

Similarly, for HTTP (service port 80) traffic in Figure 5.7(b), we observe that the number of hosts in the client partitions, i.e., the zero in-degree partitions in the left column, is very similar in the generated trace compared to the original partitioning. Furthermore, the number of web servers in the generated trace, i.e., in the zero out-degree partitions the bottom row, is close to the values in the original traffic.

There is a marginal mismatch between the number of hosts in some of the non-zero out-degree and non-zero in-degree partitions for both service ports, DNS and HTTP. This is mainly due to marginal differences between the original out-degree and in-degree values and the values approximated by the polynomial parameters, the graphs, and hence the traffic structure, are generated from. The values for the mean out-degree and the mean in-degree, as well as the respective standard deviations thereof, are left out to depict the visualizations better. However, the note that values for these quantities in the partitionings established from generated traces are similarly close to values in the original trace.

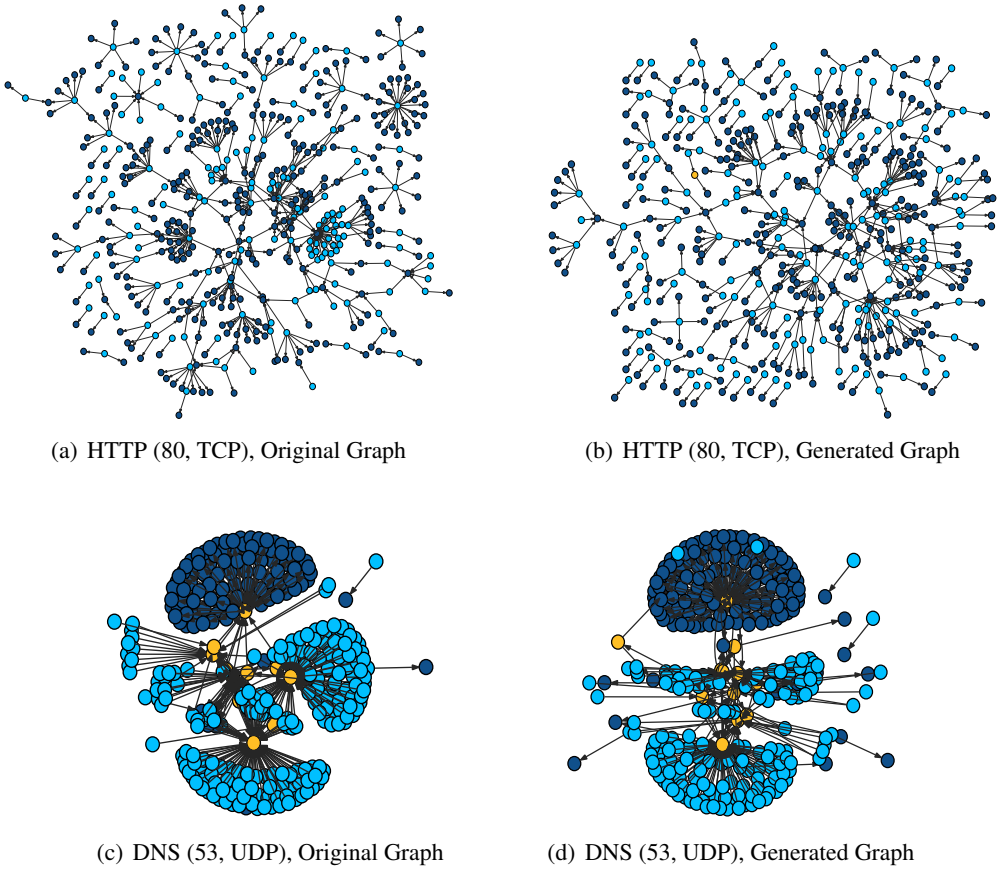
To sum up, we have shown that flow traces generated by our technique exhibit traffic structure highly similar to the structure found in original traffic compared at each time interval. Although minimal differences in the number of hosts for some partitions exist, given the approximating character our our approach, we find this to be negligible in real world applications.

5.3.3 Visualizations

In addition to the evaluation of traffic structure by means of various graph metrics, as well as visualizations of the partitioning of the degree distribution plane, we use visualizations of port and protocol based TDGs of generated traffic. We generate flow records from self-parametrized traffic templates established from the campus network traffic in Section 5.3.2. Then, we establish the TDGs from both, the original and generated traces, by employing the GraphViz tool introduced in Section 3.3.

Traffic Dispersion Graph visualizations of original and generated HTTP traffic on service port 80 (TCP), as well as visualizations of graphs established from DNS traffic on service port 53 (UDP) are depicted in Figure 5.8. The layout and the vertex positions in the visualized graphs has been entirely determined by the algorithms employed in GraphViz. The number of vertices in the graphs has been limited to 500 ($|V| = 500$) for HTTP traffic, and to 300 ($|V| = 300$) for DNS traffic to depict the results better.

By inspecting these visualizations of TDGs for the generated flows, we observe that the graph structure for HTTP traffic in Figure 5.8(b) is similar to the TDG shown in Figure 5.8(a). We



(a) HTTP (80, TCP), Original Graph

(b) HTTP (80, TCP), Generated Graph

(c) DNS (53, UDP), Original Graph

(d) DNS (53, UDP), Generated Graph

Figure 5.8: Visualizations of TDGs generated from real traces and graphs established from flow traces generated from self-parametrized traffic templates.

observe that the quantities of connections between low out-degree vertices and zero in-degree vertices (clients) connecting to higher in-degree vertices (popular servers) are in the same order.

Furthermore, the TDG established from the traffic templates is visually similar to the HTTP traffic TDG in Figure 3.2(a) in Section 3.3, generated from real flow traces collected during a different time period on a different day of the week.

Likewise, the structure of the graph for generated DNS traffic in Figure 5.8(d) matches the original graph in Figure 5.8(c) closely. The characteristic access pattern from clients (zero in-degree vertices) to a few DNS servers (high in-degree vertices) is well represented in the graph. Moreover, the graph structure determined by the high number of low in-degree servers in the upper part of both visualizations for DNS traffic is reproduced well.

To sum up, we have shown that the graph structure, and therefore the underlying traffic structure in the traces generated, closely matches the structure in graphs established from original traces. Although some minor differences are present in the graphs, given the approximating nature of the polynomial coefficients of the partition parameters, as well as the random graph algorithm, the

Table 5.2: Network Scan Template Parameters

Parameter	Value	Description
T	300-s	Period length
R	6	Network protocol
P_{src}	22	Service port
p_{dst}	1024-65535	Source port range
μ_H, σ_H	603, 0	Mean and standard deviation value for the number of hosts
μ_F, σ_F	5 000, 500	Mean and standard deviation value for the number of flows
$\vec{a}_{n_{k_s},k_s}$	3	Number of vertices for the partition $k_s = ([100,\infty) \times [0,1))$ (scanners)
$\vec{a}_{\mu_{\mu_{k_s,out}},k_s}$	200	Mean out-degree for vertices in the partition for k_s
$\vec{a}_{n_{k_t},k_t}$	600	Number of vertices in the partition $k_t = ([0,1) \times [1,2))$ (targets)
$\vec{a}_{\mu_{\mu_{k_t,in}},k_t}$	1	Mean in-degree of vertices in the partition k_t
IP_{k_s}	10.46.0.1/32, 10.46.0.2/32, 10.46.0.3/32	IP address ranges for the scanners
IP_{k_t}	9.9.0.0/16	IP address ranges for the targets
I_{active}	/2	Active intervals

structure of the graphs is highly comparable. Moreover, the algorithms in GraphViz, controlling how vertices in a graph are laid out, implicitly provide an additional valuable metric as graphs with different structures visually differ.

5.4 Definition of Traffic Scenarios

The definition of specific borderline or traffic scenarios such as network scans, attacks, or abnormal surges of traffic is of special interest when evaluating flow-based network-monitoring systems. In this section, we demonstrate that such events can easily be defined using our technique by manually specifying traffic templates and their corresponding parameters.

We chose to model a network scan on the service port 22 (SSH) on the TCP protocol using our template-based approach. Network scans on the packet level involve sending TCP packets with specific TCP flags to a collection of hosts. The scan performing host determines whether the targeted hosts are running a specific service, i.e., a terminal service such as SSH, based on the TCP flags in the packets received. On the flow level, these packet flows are grouped into flow records.

Therefore, our aim is to generate flow records containing a network scan performed by three different hosts (*scanners*), each conducting a scan on 200 different hosts (*targets*) on the terminal services destination port 22. Furthermore, the generated trace should comprise realistic background traffic without anomalies. Table 5.2 depicts the traffic template parameters used to model the network scan. We describe the parameters set as follows:

Protocol The protocol template parameter R is set to the TCP protocol number $R = 6$.

Ports The service (destination) port for all flows in the scan is the SSH service port 22. Therefore, we set the appropriate parameter p_{dst} in the template to $p_{dst} = 22$. The range of source ports for the service initiators, hence the scan performing hosts, is set to a reasonable value $P_{src} = (1024, \ldots, 65535)$ with the corresponding template parameter P_{src}.

Hosts and Flows The network scan is to be performed by three distinct network hosts targeting 200 other hosts. Consequently, we set the mean values and the respective standard deviation for the number of hosts to $\mu_H = 603$ and $\sigma_H = 0$, leading to a constant number of hosts generated for each time interval. Similarly, the number of flows and the respective standard deviation are set with $\mu_F = 5\,000$ and $\sigma_F = 500$, leading to a varying number of flows for each time interval.

Traffic Structure Next, the structure of the traffic trace to be generated is parameterized in the templates. We start with the partition definitions: the scanners are vertices in the partition $k_s = ([100, \infty) \times [0, 1))$ (out-degree higher than 100, zero in-degree), whereas the targets are vertices in the low-degree partition $k_t = ([0, 1) \times [1, 2))$ (zero out-degrees, one in-degree only).

Furthermore, the polynomial coefficients $\vec{a}_{g,k}$ for two partition parameters $g \in G_k$ must be set for each partition, k_s and k_t. Three scanners in the k_s partition are defined by a constant value (polynomial of degree zero) for the number of hosts $n_{k_s} = 3$. Similarly, the 200 targets in the partition k_t are expressed as a constant value $n_{k_t} = 200$.

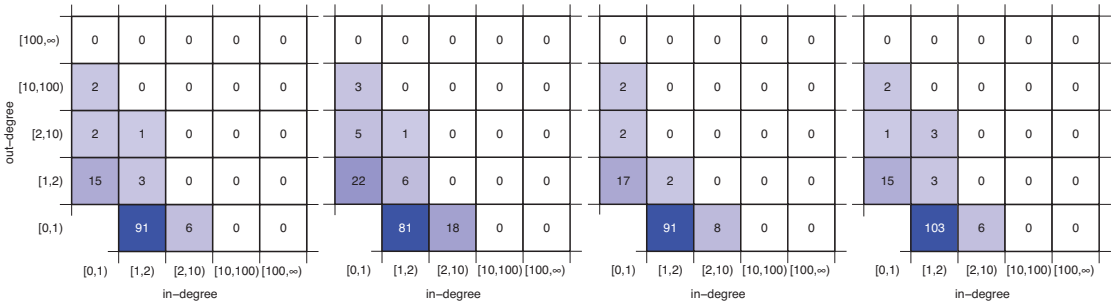
The structure in the template is then completed by setting the mean number of out-degrees (connections) $\mu_{k_s,out}$ for the scanners in the partition k_s to $\mu_{k_s,out} = 200$, and the mean in-degree value $\mu_{k_t,in}$ for the targets in k_t to $\mu_{k_t,in} = 1$. The respective standard deviations, as well as the polynomial coefficients for the remaining partition parameters G_k are omitted[2].

IP Addresses Additionally, we define IP address for both, the scanners and the targets by the respective parameters for vertices in each partition. Particularly, the scanners are assigned IP addresses by setting the IP_{k_s} parameter to the list of three IP address ranges in CIDR notation for single hosts, $IP_{k_s} = (10.46.0.1/32,\ 10.46.0.2/32,\ 10.46.0.3/32)$. Similarly, a Class B network IP address range, comprising 65 535 possible hosts, is assigned to the targets by setting the IP_{k_t} parameter to $IP_{k_t} = 9.9.0.0/16$. The targets are assigned random IP address from the range determined by this network prefix.

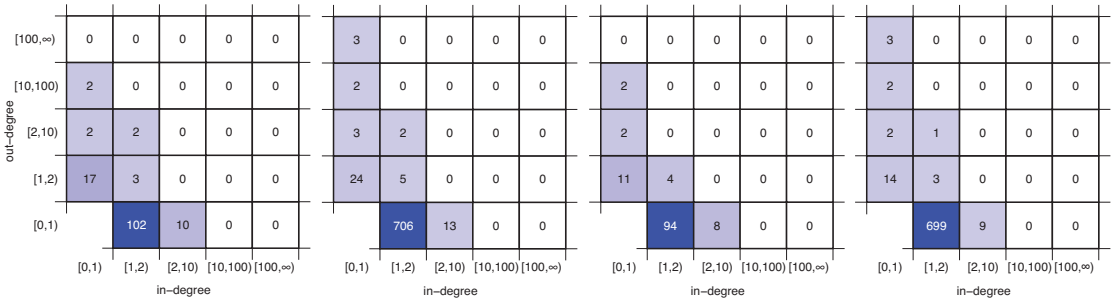
Active Intervals The network scan is to be performed on certain intervals, as opposed to being active during all time intervals in the trace generated. We set the network scan traffic template to be present in ever second time interval by specifying the respective I_{active} parameter to $I_{active} = /2$.

[2]Polynomial coefficients omitted for partition parameters G_k default to zero for each partition k and each parameter $g \in G_k$.

The remaining traffic template parameters, such as parameters for the flow durations d_μ, d_σ, the number of packets p_μ, p_σ, or the number of bytes b_μ, b_σ, are left out in this traffic scenario. As such, random values are assigned for these flow record fields in the trace generation process.



(a) Original Traffic



(b) Generated Traffic

Figure 5.9: Comparison of graph partitionings between original and generated SSH (service port 22) traffic, created from four consecutive time intervals of 300 s.

Then, we use the collection of 132 traffic templates established in Section 5.3.2 by self-parametrization, while adding the network scan traffic template, to generate flow records from a total of 133 traffic templates. In Figure 5.9, we depict the partitioning from four consecutive time intervals established from the original and generated traces for SSH (service port 22) traffic.

First, we observe that the traffic structure in the generated flow trace (first and third time interval) depicted in Figure 5.9(b) is similar to the to the partitioning in Figure 5.9(a) established from original SSH traffic. The number of clients (left column) and servers (bottom row) are approximated well by the flow generator. Moreover, the number of hosts with both, outgoing and incoming connections (vertices in the non-zero out-degree and non-zero in-degree partitions) is similar in the original and generated trace.

The partitionings in Figure 5.9(b) clearly depict the time intervals during which the custom-defined network scan is present in the trace. The number of clients in the $[0, 1) \times [1, 2)$ partition in every second interval is increased by the number of hosts targeted by the scan. Similarly, the three hosts performing the scan are depicted in the partitionings of the second and fourth time interval in Figure 5.9(b) in the top-left partition $[100, \infty) \times [0, 1)$.

To sum up, we have demonstrated that custom traffic scenarios comprising specific structural

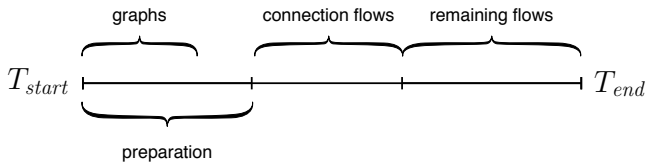


Figure 5.10: Three phases of the flow-generation process for an interval T.

properties can be defined by our template-based approach. In addition to the network scan scenario presented here, other scenarios such as attacks performed by a large number of network hosts connecting to one specific hosts or to a group of network hosts can be easily specified according to the methodology presented. Furthermore, by choosing different partition parameters for partitions of varying out-degree and in-degree, the partitioning approach also allows to model varying host connectivity. This can be used, for example, to model worm-like structures in the generated traces.

Finally, the advantage of the traffic scenario templates and the flow traces generated thereof is, that they can be generated in combination with parameterizable background traffic from templates established from real traces (e.g., in the self-parametrization process). Consequently, this allows for an evaluation of flow-based network monitoring systems under realistic conditions.

5.5 Performance

The flow-generation process for a time interval of length T, determined by the start timestamp T_{start} and the end timestamp T_{end}, can be divided into three parts depicted in Figure 5.10. First, the *preparation* part comprises operations needed to prepare the traffic structure by establishing the TDGs from the partitioning computed from the polynomial coefficients in the traffic templates. Additionally, the flow record field values for the *connection flows*, determined by the edges in all TDGs which model the basic underlying traffic structure for the trace to be generated, are set in this preparation part of the flow-generation.

The second part of the entire flow-generation process comprises two parts: the generation of all connection flows (one flow for each edge (u, v) in all TDGs established), as well as the generation of the *remaining flows*, which model the remaining number of flows for each traffic template active during the current interval. In our evaluation, the body of self-parametrized traffic templates each has a time length intervals of $T = 300$ s, over which we generate the flow records.

Consequently, we divide the performance evaluation of the flow-generation into three parts. First, we measure the time needed to create TDGs for all templates for different number of network hosts in the preparation part. Then, we quantify the flow generation rates achieved when iterating over the list of connection flows, as well as when generating the remaining number of flows for all traffic templates.

We use the collection of 132 self-parametrized traffic templates established in Section 5.3.2,

comprising 71 templates for the TCP protocol and 61 templates for the UDP protocol, to generate flow records in the performance evaluation. The thereby generated NetFlow packets are sent over UDP protocol to a collector. We also evaluated the performance of the flow generator saving the generated NetFlow packets to files (in binary format) on the filesystem. We note that the performance results were quantitatively similar to the UDP output results presented in this section.

The graph generation during the preparation phase for an average number of 8548 ($\sigma = 418$) network hosts took 3.4 s ($\sigma = 0.25$) generated from all 132 traffic templates. The respective number of unique hosts in the generated trace (combined for all templates) is lower, namely on average 7630 ($\sigma = 382$), since the IP addresses of hosts, determined by the IP address ranges in the traffic template, may overlap across different templates.

Increasing the number of hosts for a period to an average of 15 763 ($\sigma = 707$) yields a mean processing time of 7.31 s ($\sigma = 0.44$), with an average of 13 316 ($\sigma = 578$) unique IP addresses in the resulting trace. Further increasing the average number of network hosts to 24 864 ($\sigma = 1113$), on average resulting in 20 522 ($\sigma = 1016$) unique IP address in the generated trace, yields a mean processing time of 14.3 s ($\sigma = 0.99$).

Obviously, in order to model the underlying traffic structure, the graph generation constitutes the bottle-neck in the flow-generation process by introducing waiting times between two consecutive time intervals flow records are generated for. However, the number of unique IP addresses in the hosting provider data set used, is on average around 20 000 during 300 seconds long time intervals. The graph algorithm for a comparable number of unique IP addresses (although the number of hosts drawn from the respective distribution parameters in the traffic templates is generally higher) in our evaluation took 14.3 s, on average, which is about 5% of time constituting a time interval. Therefore, we conclude that this additional time window introduced, during which no flow records are streamed to the collector, is negligible.

Next, we evaluate the flow generation rate achieved during both, the phase when the connection flows are being generated, as well as during the generation of the remaining flow records for traffic templates active during an interval T. When generating flow records for the set of connection flows, the generator achieves an average rate of 100 493 flows/s ($\sigma = 4093$). Flow records for the set of remaining flow records are generated at an average rate of 46 051 flows/s ($\sigma = 397$). The rate achieved over both phases combined was on average 45 641 flows/s ($\sigma = 462$).

We compared these rates for an average number of total flows generated for a time interval ranging from 1.32 million ($\sigma = 10321$) to 12.45 million ($\sigma = 1\,037\,089$) flows. The number of flows per second for both phases were remain constant[3].

The difference in these two quantities is due to the way the two sets of flows are processed. Since the values for flow record fields, such as the flow duration, the number of packets, or the number of bytes, are pre-set for all entries in the list of connection flows (as described in Algorithm 4), the generator can achieve much higher flow rates by simply iterating over this list. For the remaining flow records, on the other hand, values for the flow record fields are sampled

[3]Small differences in the flow generation rates were present, introduced mainly due to the different overall CPU load of the system the measurements were conducted on, as opposed to the increased number of flows.

from the respective distributions for each flow record generated. This introduced an additional overhead and increases the processing time.

In an earlier implementation of the prototype, we were able to achieve an average flow generation rate of 98 218 flows/s ($\sigma = 3907$) for all flow records generated. This was due to the different structure of the flow generation process, which consisted only of two steps: a preparation phase where TDGs were established and flow record fields pre-populated for all flow records to be generated, as well as a generating phase where flow records were output by iterating over the prepared flow lists.

While the flow rates in this approach were higher, the introduced overhead in terms of processing time in the preparation phase was not justified. Moreover, generating millions of flow records for a time interval introduced additional memory issues since the flow record fields were pre-populated for the entire number of flow records.

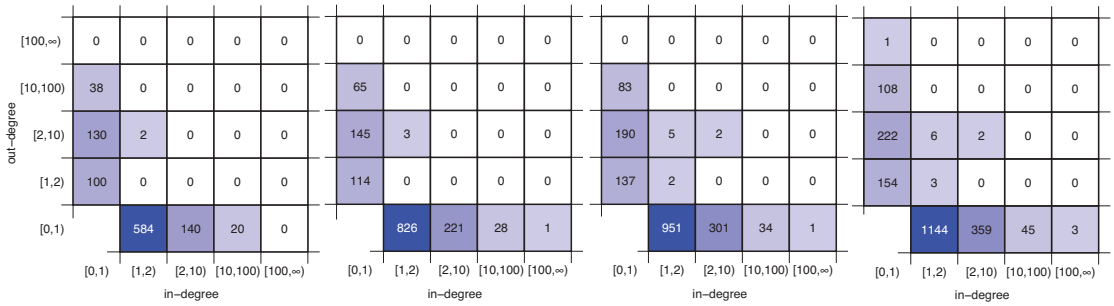
To sum up, we have shown that the graph algorithm scales fairly well in terms of the number of hosts. Moreover, the flow generator achieves satisfactory high rates when generating flow records. Also, the generator behaves especially well when generating millions of flow records for a time interval. Therefore, our framework can be used to stress-test flow-based network monitoring systems in corporate networks with generated traces comprising realistic traffic structure.

5.6 Limitations

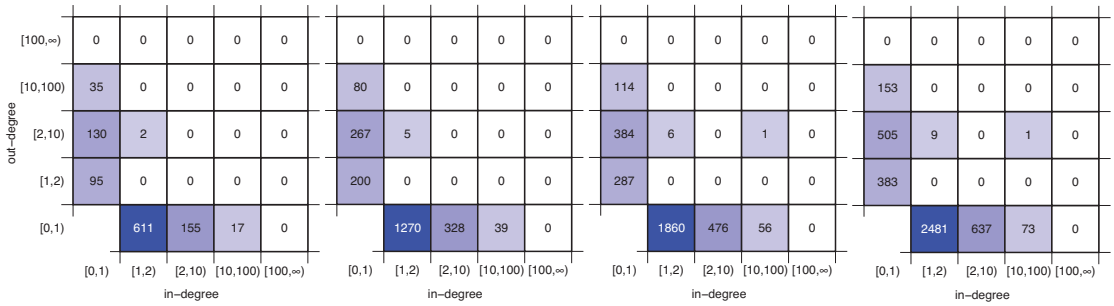
In the previous evaluation sections we showed that our graph-based approach reproduces the traffic structure for each time interval accurately, not only by means of the inter-connectivity between network hosts, but also with respect to more advanced graph metrics as shown in Section 5.3.1. However, in the current prototype implementation of our framework, a limitation with regard to the aggregated traffic structure of the generated traffic exists. In this section, we illustrate these limitations and outline some of the approaches taken in order to ameliorate these shortcomings.

Figure 5.11 shows a comparison of graph partitionings of HTTP traffic (service port 80) between the original traces and the traces generated by our flow trace generator from self-parameterized traffic templates, established from the campus network traffic in Section 5.3.2. The partitionings in Figure 5.11 were created from four consecutive time intervals of 300 length. Moreover, the partitioning at each time interval depicts the traffic structure accumulated over all previous time intervals. For example, the second partitioning in Figure 5.11 shows the traffic structure of the first and second interval, for both, the original and generated traffic. Similarly, the third partitioning depicts the traffic structure accumulated over the first, second, and third time interval, etc.

We observe that the number of vertices in partitions in the original traffic in Figure 5.11(a) and the traffic generated in Figure 5.11(b) differs, especially in the low degree partitions, such as for example $[0, 1) \times [1, 2)$ or $[1, 2) \times [0, 1)$. We attribute these differences in the traffic structure to the memory-less nature of the flow trace generation process and illustrate this limitation in the next paragraphs. We further note that this limitation in not related to specific service ports and



(a) Original Traffic (80, TCP)



(b) Generated Traffic (80, TCP)

Figure 5.11: Comparison of graph partitionings between original and generated HTTP (service port 80) traffic, created from four consecutive time intervals of 300 s length. Each partitioning depicts the traffic structure accumulated over the sum of all previous intervals.

network protocols, the HTTP (service port 80) traffic depicted in Figure 5.11 merely serves as an example of the limitation. We observed that other service ports and protocols, although not always in the same intensity, exhibit similar differences in the aggregated traffic structure.

Figure 5.12 depicts two partitionings (HTTP traffic) established from two consecutive time intervals T_1 and T_2 of 300 s length. Similar to the partitionings in Figure 5.11, the partitioning at each time interval depicts the structure of the traffic aggregated over all previous intervals. In Figure 5.12, the partitioning on the left-hand side depicts the traffic structure of one time interval T_1, whereas the traffic structure depicted by the partitioning on the right-hand side is from the intervals $T_1 + T_2 = T_{1,2}$.

Let us consider, for example, vertices in the zero out-degree and low in-degree partition $k_{low} = [0, 1) \times [1, 2)$, which in the case of HTTP traffic is likely to contain less popular web servers (hosts with low incoming connections and zero outgoing connections). Furthermore, we consider a vertex v (host), determined by its unique IP address, which is present in the partition k_{low} in both, the first time interval T_1, as well as the second time interval T_2 of the trace analyzed. Consequently, the vertex v has zero out-degree and one out-degree (connection) during each time interval. Therefore, when we establish the partitioning of both time intervals T_1 and T_2 separately, the vertex v is going to be present in the partition k_{low} in both partitionings. However, when we consider the partitioning of the traffic structure aggregated over all previous

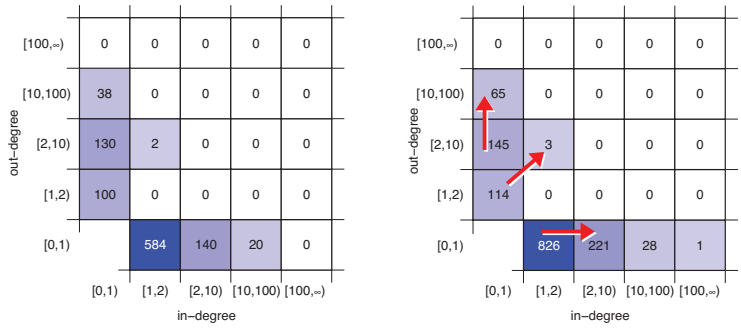


Figure 5.12: Left: partitioning of the first time interval T_1. Right: partitioning of traffic accumulated over the first and second interval $T_{1,2}$. The degree (connections) of vertices (hosts) which are present in the same partition in both intervals is likely to increase, hence these vertices change partitions (red arrows) between two consecutive intervals.

time intervals (e.g., for T_2 the partitioning depicts the structure of $T_1 + T_2$ aggregated), the vertex v will most likely have moved to a higher in-degree partition (e.g., $[0, 1) \times [2, 10)$) since the total number of out-degrees (connections) of v will have increased to two (one connections during each interval). The red arrows in Figure 5.12 illustrate the transition of vertices to different partitions between two consecutive intervals, as the degree values of vertices that are present during both time intervals increase.

The traffic templates in the current prototype implementation of our technique do not contain any information about this transition of vertices to different partitions between consecutive time intervals. The flow trace generator samples the IP addresses of hosts in each partition $k \in K$ from the range of IP addresses specified in IP_k for each time interval independently, without any notion of which IP addresses were generated in the previous intervals (hence memory-less). Consequently, the flow traces generated by our framework are likely to contain an increased number of network hosts as depicted in Figure 5.11, while the generated network hosts are likely to contain lesser connections (degrees). We have tried to ameliorate this difference in the vertices and vertex degrees in the aggregated traffic structure with several approaches.

Fixed vertices in high-degree partitions One approach to tackle the limitation of aggregated traffic structure was to fix the vertices in high-degree partitions between consecutive time intervals flow traces are generated for. The rationale behind this is the consideration that some vertices (hosts) in the high-degree partitions remain present during the entire duration of the trace generated. For example, DNS servers in high in-degree partitions (cf. Figure 4.1(c)) are likely to be present in each time interval in the flow trace. Similarly, the number of the popular web servers in HTTP traffic in the high in-degree partitions remains constant, while only web clients in the zero in-degree and low out-degree partitions change.

Therefore, we added a retentive component to the flow trace generator which memorized the vertices (IP addresses of nodes) in high-degree partitions, and, instead of resampling

the entire vertex population for these partitions, the memorized vertices were reused when the flow trace for new time interval. Although this approach worked well for some service ports (e.g., DNS or HTTP) and the high-degree partitions, it did not solve the aggregated traffic problem in general, independent of a service port. Moreover, the difference in the number of vertices and their degrees in lower degree partitions remained, since this approach only considered the high-degree partitions.

Discarding factor Based on the shortcomings of the previous approach of fixing vertices in high-degree partitions, as well as due to our observations that the vertices change partitions also in the lower degree intervals, we tried to improve the aggregated traffic structure by introducing a discarding factor for every partition. In this approach, we fixed the vertices for each partition $k \in K$ between consecutive time intervals, as opposed to retaining vertices in high-degree partitions only. Then, for each new time interval flow records were to be generated for, a certain amount of vertices retained from the previous interval in a partition $k \in K$ was discarded, based on a "discarding factor", while new vertices were drawn from the respective range of IP addresses IP_k and added. The undiscarded portion of the vertices reused for the new interval lead to increased vertex degrees for these vertices in the aggregated traffic, hence simulated the transition of vertices between partitions.

Even though this approach lead to generally better results with respect to the various service ports, the aggregated structure however still differed. Moreover, an additional challenge was to define a proper method to determine the discarding factor. Here, we pursued two different approaches. First, we defined different, fixed discarding factors for low-, medium-, and high-degree partitions and discarded the vertices between consecutive intervals based on these factors. While this lead to aggregated traffic structure closer to the original for certain service ports, the fixed discarding factors could not be applied generally as the aggregated traffic structure differs for different service ports. Consequently, in our second approach, we tried to compute a variable discarding factor dynamically for each partition $k \in K$ and each service port from partitionings of aggregated traffic of various, consecutive time intervals. However, also with a dynamic discarding factor, we were not able to achieve a general solution that reproduced the aggregated traffic accurately. Furthermore, the added performance penalty when generating flow traces was not justified by the slightly better aggregated structure.

Fixed IP address space The third approach taken was to introduce a fixed IP address space for each partition. As a means to achieve the retention of vertices in partitions $k \in K$ across consecutive time intervals, and as such to capture the transition of vertices between partitions, we fixed the size of the IP address space for each partition. For example, for a partition $k \in K$, the IP address space during all consecutive time intervals T would remain constant, comprising a fixed number of IP addresses (vertices), e.g., 1000. Moreover, the IP address space of each partition $k \in K$ was disjoint from all other partitions for a service port.

This implicitly added the retention of certain IP addresses (vertices) between consecutive time intervals. For example, when the number of vertices for a partition $k_x \in K$ was

400 for each interval and the IP address space was fixed to 1000 IP addresses, the IP addresses of some of the 400 vertices drawn from the 1000 IP addresses for each new time interval would "overlap" with the IP addresses of vertices from the previous interval. As such, IP addresses (vertices) present in both intervals would exhibit increased degrees (connections) in the aggregated traces.

While this approach somehow ameliorated the aggregated traffic structure, similar to the previous approaches, it also posed a different problem. The restriction on the total IP addresses in the entire trace generated, given by the disjoint nature of the IP address spaces across partitions $k \in K$, was not justified by the slightly better aggregated structure.

To sum up, we have shown that the aggregated traffic structure of the generated traces is likely to exhibit decreased out-degrees and in-degrees (outgoing and incoming connections) for vertices in certain partitions. Moreover, the number of vertices in certain partitions is likely to be higher in the traces generated, compared to the values in the original traces. Although we made several attempts to ameliorate the aggregated structure, the problem of finding a generation solution for these limitations remains.

Nevertheless, we believe that our approach provides a valuable addition to the currently existing flow-based generators, since the simplistic traffic structure generated by the latter, as well as the inability to define specific traffic scenarios, is less applicable for realistic evaluation of flow-based monitoring systems. Moreover, the pluggable architecture in our framework allows for programmatic and exact examination of the values generated. Therefore, in spite of the differences in the aggregated traffic structure, flow-based systems analyzing the trace generated can compensate these structural differences since they "know" what exactly has been generated.

Chapter 6

Conclusion

The role of computer networks today is more important than ever, not only for businesses and business critical applications, but also more generally for our quotidian live. However, the increasing complexity of existing and newly established computer networks introduces several new challenges on infrastructures, hardware devices, as well as on network monitoring and analysis applications and systems.

Today, network monitoring and analysis or anomaly detection systems are widely used in corporate and service provider networks. Often, these systems are needed to ensure an uninterrupted operation of the network infrastructure, as well as to collect network-related data used for accounting, detection of anomalies, or quality assurance. Consequently, a proper evaluation to ensure the accuracy of such systems is crucial. However, proper testing and evaluation of flow-based network monitoring systems remains challenging, since the generation of synthetic traces with realistic traffic structure on the flow-level is problematic, and real-world traces with borderline traffic conditions, vital for realistic evaluations, are either difficult to collect or rarely observed.

In this thesis, we considered the problem of generating flow-level traces with realistic, parameterizable traffic structure, as well as the ability to include custom, user-defined traffic scenarios comprising traffic conditions important for the evaluation of flow-based network monitoring and anomaly detection systems under realistic conditions.

We introduced the concept of traffic templates which model structural properties of the connection patterns between hosts by means of Traffic Dispersion Graphs, as well as the distribution parameters of several flow record attributes. We further described the self-parametrization process of our framework, capable of extracting a collection of relevant traffic templates with their respective parameters from existing network traces. Moreover, user-defined traffic templates with customized parameters can be used to model specific traffic conditions such as network scans, attacks, or similar. An additional advantage of our template-based approach is that the notion of IP address ranges in traffic templates can be easily removed, while leaving the structure of the generated traffic intact. Therefore, traffic templates can be easily anonymized and exchanged between different parties, without privacy-related issues.

A vital part of the framework introduced in our thesis is the trace generator, providing several advantages over the currently existing flow trace generation solutions. First, our generator can

be used on desktop-class computers using commodity hardware, while achieving satisfactory high flow generation rates, generating flow records directly without the detour of any additional soft- or hardware components. Second, the pluggable architecture of our framework presents an additional advantage over the existing solutions. User-defined plugins can collect exact values of the trace generated during different phases of the flow generation process, allowing an automated processing and comparison between the generated data and the values determined by the network monitoring and anomaly detection systems processing the trace generated.

Our evaluations showed that traces generated from the self-parametrized traffic templates comprise traffic structure highly similar to the structure in original traces for each time interval. Moreover, traffic templates with custom traffic scenarios can be used in combination with self-parametrized templates to generate traces comprising parameterizable background traffic with user-defined traffic scenarios. We also outlined the limitations of our template-based approach with respect to the aggregated traffic structure. Nevertheless, we believe that, in spite of these shortcomings, the high flow generation rate, as well as the more realistic traffic structure present in the traces generated, and the pluggable architecture present an advantage over the current flow trace generators available.

The possibilities for future work are threefold. First, the aggregated traffic structure in the traces generated could be ameliorated to more closely resemble the aggregated structure of the original traffic. The possible approaches presented in our evaluation, although not leading to a general solution to the problem, could act as a starting point for further research in this area.

Second, we believe that the performance of the flow generator, while already delivering satisfactory results, could be improved further. We chose the Perl for the implementation the initial version of our framework mainly due to the numerous additional modules available, enabling us to more rapidly develop a ready-to-use version of our framework. The use of lower-level programming languages is likely to improve the performance of certain application parts and algorithms. Furthermore, implementing threads for the different phases of the flow generation process could improve the flow generation rates on multi-core systems by pre-generating graphs for the subsequent time intervals, hence minimizing the time window described in Section 5.5 during which no flow records are output.

Third, the structure of traffic templates could be extended to allow for definitions of additional traffic scenarios. In Section 5.4 we showed an example definition of a horizontal network scan (one target port on different hosts). Other scenarios such as attacks, or even worm like structures can be defined similarly by choosing the appropriate template parameters. However, for the definition of a vertical network scan (different ports on one host), for example, many very similar traffic templates are needed. This could most likely be improved by introducing additional template parameters for similar traffic scenarios and conditions.

To sum up, we have presented a flow trace generation framework to enable and simplify the automated evaluation of flow-based network monitoring and anomaly detection systems under realistic traffic conditions. We further believe that our framework is also useful for other, more general validation tasks based on network flows.

Glossary

CCDF	Complementary Cumulative Distribution Function
CIDR	Class Inter-domain Routing
EFF	Edge on First Flow
EFP	Edge on First Packet
EFSP	Edge on First SYN Packet
FLAME	Flow-level Anomaly Modeling Engine
IETF	Internet Engineering Task Force
IOS	Internetwork Operating System
IPFIX	IP Flow Information Export
LWCC	Largest Weakly Connected Component
MDR	Max Degree Ratio
NBNS	NetBIOS Naming Service
RU	Relative Uncertainty
SSE	Sum of Square Error
TAG	Traffic Activity Graph
TDG	Traffic Dispersion Graph

Bibliography

[1] C. Rolland, J. Ridoux, and B. Baynat, "LiTGen, a Lightweight Traffic Generator: Application to P2P and Mail Wireless Traffic," in *PAM '07: Proceedings of the Passive and Active Measurement Conference*, 2007.

[2] ——, "Catching IP Traffic Burstiness with a Lightweight Generator," in *Networking*, 2007, pp. 924–934.

[3] C. Rolland, J. Ridoux, B. Baynat, and V. Borrel, "Using LiTGen, a realistic IP traffic model, to evaluate the impact of burstiness on performance," in *Simutools '08: Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, 2008.

[4] K. V. Vishwanath and A. Vahdat, "Realistic and responsive network traffic generation," in *SIGCOMM '06: Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2006.

[5] P. Barford and M. Crovella, "Generating representative web workloads for network and server performance evaluation," in *SIGMETRICS '98: Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, 1998.

[6] J. Sommers, H. Kim, and P. Barford, "Harpoon: a flow-level traffic generator for router and network tests," in *SIGMETRICS '04/Performance '04: Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, 2004.

[7] A. Rupp, H. Dreger, A. Feldmann, and R. Sommer, "Packet trace manipulation framework for test labs," in *IMC '04: Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, 2004.

[8] J. Sommers, V. Yegneswaran, and P. Barford, "A framework for malicious workload generation," in *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, 2004.

[9] Paessler, "Paessler NetFlow Generator," http://www.paessler.com/ (last accessed: Jan. 2010).

[10] P. International, "Flowalyzer NetFlow & sFlow Generator," http://www.plixer.com/products/netflow-sflow/flowalyzer-netflow-sflow-tester.php (last accessed: Feb. 2010).

[11] J. Juping, "Netflow Simulator in C#," http://sourceforge.net/projects/ netflowsim (last accessed: Jan. 2010).

[12] Cisco, "Netflow services solutions guide," http://www.cisco.com/en/US/docs/ios/solutions_docs/netflow/nfwhite.html (last accessed: Jan. 2010).

[13] D. Brauckhoff, A. Wagner, and M. May, "Flame: A flow-level anomaly modeling engine," in *CSET'08: Proceedings of the Conf. on Cyber Security Experimentation and Test*, 2008.

[14] J. Sommers and P. Barford, "Self-configuring network traffic generation," in *IMC '04: Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, 2004.

[15] J. Hawkinson and T. Bates, "Guidelines for creation, selection, and registration of an Autonomous System (AS)," RFC 1930 (Best Current Practice), Internet Engineering Task Force, Mar. 1996. [Online]. Available: http://www.ietf.org/rfc/rfc1930.txt

[16] H. Tangmunarunkit, R. Govindan, S. Jamin, S. Shenker, and W. Willinger, "Network topology generators: degree-based vs. structural," *SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 4, pp. 147–159, 2002.

[17] K. Calvert, M. B. Doar, A. Nexion, E. W. Zegura, G. Tech, and G. Tech, "Modeling internet topology," *IEEE Communications Magazine*, vol. 35, pp. 160–163, 1997.

[18] M. B. Doar and A. Nexion, "A better model for generating test networks," in *Proceeding of IEEE GLOBECOM*, 1996, pp. 86–93.

[19] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*. New York, NY, USA: ACM, 1999, pp. 251–262.

[20] A. Medina, A. Lakhina, I. Matta, and J. Byers, "Brite: an approach to universal topology generation," in *9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2001*, 2001, pp. 346–353. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=948886

[21] M. Iliofotou, P. Pappu, M. Faloutsos, M. Mitzenmacher, S. Singh, and G. Varghese, "Network traffic analysis using traffic dispersion graphs (tdgs): Techniques and hardware implementation," 2007.

[22] ——, "Network monitoring using traffic dispersion graphs (TDGs)," in *IMC '07: Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, 2007.

[23] M. Iliofotou, P. Pappu, M. Faloutsos, M. Mitzenmacher, G. Varghese, and H.-C. Kim, "Graption: Automated detection of p2p applications using traffic dispersion graphs (tdgs)," 2008.

[24] M. Iliofotou, H.-C. Kim, M. Faloutsos, M. Mitzenmacher, P. Pappu, and G. Varghese, "Graph-based p2p traffic classification at the internet backbone," in *INFOCOM Workshops 2009, IEEE*, 2009. [Online]. Available: http://dx.doi.org/10.1109/INFCOMW.2009.5072151

[25] B. Claise, "Cisco Systems NetFlow Services Export Version 9," RFC 3954 (Informational), Internet Engineering Task Force, Oct. 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3954.txt

[26] ——, "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information," RFC 5101 (Proposed Standard), Internet Engineering Task Force, Jan. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5101.txt

[27] J. Quittek, S. Bryant, B. Claise, P. Aitken, and J. Meyer, "Information Model for IP Flow Information Export," RFC 5102 (Proposed Standard), Internet Engineering Task Force, Jan. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5102.txt

[28] K. C. Claffy, H. werner Braun, and G. C. Polyzos, "A parameterizable methodology for internet traffic flow profiling," *IEEE Journal on Selected Areas in Communications*, vol. 13, pp. 1481–1494, 1995.

[29] T. Asaka, K. Ori, and H. Yamamoto, "Method of estimating flow duration distribution using active measurements," *IEICE transactions on communications*, vol. 86, no. 10, pp. 3030–3038, 2003. [Online]. Available: http://ci.nii.ac.jp/naid/110003221525/en/

[30] W. Liu, J. Gong, W. D. 0001, and G. Cheng, "An algorithm for estimation of flow length distributions using heavy-tailed feature," in *International Conference on Computational Science (4)*, 2006, pp. 144–151.

[31] V. Paxson, "Empirically derived analytic models of wide-area TCP connections," *IEEE/ACM Transactions on Networking*, vol. 2, no. 4, pp. 316–336, August 1994. [Online]. Available: http://dx.doi.org/10.1109/90.330413

[32] P. Olivier and N. Benameur, "Flow level ip traffic characterization," available online: http://perso.rd.francetelecom.fr/roberts/Pub/OB01.pdf.

[33] T. Mori, M. Uchida, and S. Goto, "Flow analysis of internet traffic: World wide web versus peer-to-peer," *Syst. Comput. Japan*, vol. 36, no. 11, pp. 70–81, 2005.

[34] M. Pustisek, I. Humar, and J. Bester, "Empirical analysis and modeling of peer-to-peer traffic flows," in *Electrotechnical Conference, 2008. MELECON 2008. The 14th IEEE Mediterranean*, 2008, pp. 169–175.

[35] "Graphviz," http://www.graphviz.org (last accessed: Mar. 2010).

[36] J. Ellson, E. Gansner, E. Koutsofios, S. North, and G. Woodhull, "Graphviz and dynagraph – static and dynamic graph drawing tools," in *Graph Drawing Software*, ser. Mathematics and Visualization, M. Junger and P. Mutzel, Eds.

Berlin/Heidelberg: Springer-Verlag, 2004, pp. 127–148. [Online]. Available: http://www.springer.com/math/cse/book/978-3-540-00881-1

[37] N. W. G. in the Defense Advanced Research Projects Agency, I. A. Board, and E. to End Services Task Force, "Protocol standard for a NetBIOS service on a TCP/UDP transport: Concepts and methods," RFC 1001 (Standard), Internet Engineering Task Force, Mar. 1987. [Online]. Available: http://www.ietf.org/rfc/rfc1001.txt

[38] V. Fuller and T. Li, "Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan," RFC 4632 (Best Current Practice), Internet Engineering Task Force, Aug. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4632.txt

[39] L. Deri, "nprobe: an open source netflow probe for gigabit networks," *IN PROC. OF TERENA TNC 2003*, 2003.

[40] R. E. Walpole, R. H. Myers, S. L. Myers, and K. Ye, *Probability and Statistics for Engineers and Scientists*, 8th ed. Pearson Prentice Hall, Dec. 2007.

[41] Wikipedia, "Log-normal distribution — Wikipedia, the free encyclopedia," http://en.wikipedia.org/w/index.php?title=Log-normal_distribution&oldid=344020073 (Last accessed: Feb. 2010), 2010.

[42] ——, "Normal distribution — Wikipedia, the free encyclopedia," http://en.wikipedia.org/w/index.php?title=Normal_distribution&oldid=346208688 (Last accessed: Feb. 2010), 2010.

[43] E. W. Weisstein, ""Least Squares Fitting." from mathworld–a wolfram web resource." http://mathworld.wolfram.com/LeastSquaresFitting.html.

[44] ——, ""Least Squares Fitting–Polynomial" From MathWorld–A Wolfram Web Resource." http://mathworld.wolfram.com/LeastSquaresFittingPolynomial.html.

[45] J. D. Hoffman, *Numerical Methods for Engineers and Scientists*, 2nd ed. Marcel Dekker, 2001.

[46] M. E. J. Newman, S. H. Strogatz, and D. J. Watts, "Random graphs with arbitrary degree distributions and their applications," in *Phys. Rev. E 64(2)*, 2001.

[47] B. Bollobás, *Random Graphs*. Cambridge University Press, 2001.

[48] P. Erdös and A. Rényi, "On random graphs, I," *Publicationes Mathematicae (Debrecen)*, vol. 6, pp. 290–297, 1959.

[49] R. Milo, N. Kashtan, S. Itzkovitz, M. E. J. Newman, and U. Alon, "On the uniform generation of random graphs with prescribed degree sequences," Eprint arXiv:cond-mat/0312028, 2003.

[50] M. Iliofotou, M. Faloutsos, and M. Mitzenmacher, "Exploiting dynamicity in graph-based traffic analysis: techniques and applications," in *CoNEXT '09: Proceedings of the 5th international conference on Emerging networking experiments and technologies*, 2009.