Design of an IP Flow Record Query Language

by

Vladislav Marinov

a thesis for conferral of a Master of Science in Smart Systems

Juergen Schoenwaelder

Name and title of first reviewer

Aiko Pras

Name and title of second reviewer

Date of Submission: April 29th 2009

School of Engineering and Science

Design of an IP Flow Record Query Language

Vladislav Marinov

Computer Science
Jacobs University Bremen
Campus Ring 1
28759 Bremen
Germany

Type: M.Sc. Thesis Report
Date: January 29, 2009
Supervisor: Prof. J. Schönwälder

Executive Summary

Analyzing Internet traffic has become an extremely important and challenging task. NetFlow/IPFIX flow records are widely used to provide a summary of the Internet traffic carried on a link or forwarded by a router. Several tools exist to filter or to search for specific flows in a collection of flow records. However, there is a need for a framework (filter language) which allows certain types of traffic patterns to be defined and matched in a collection of flow records. In this M.Sc. thesis we research the various filter/query languages used by tools or proposed in the literature and extract a common basis for a new orthogonal flow record query language. A knowledge base of flow patterns that belong to common network services, applications and attacks was built to facilitate the design of the new query language and test its capabilities.

Contents

1 Introduction **6**

2 Background **8**
2.1 Network Monitoring Domain 8
2.2 NetFlow/IPFIX Protocol 9
2.3 NetFlow/IPFIX Example 11

3 State Of The Art **13**
3.1 SQL-based Query Languages 13
3.2 Tribeca Stream Query Language 15
3.3 SRL Language 16
3.4 BPF Filter Expressions 18
3.5 Flow-Tools 19
3.6 Script-Based Query Languages 21

4 Flow-level Analysis of Internet Traffic **25**

5 Statement and Motivation of Research **28**

6 Query Language Requirements **30**

7 Framework for IP Flow Filtering **32**
7.1 Framework 32
7.2 NetFlow/IPFIX Attributes 33
7.3 Keywords 33
7.4 Splitter 36
7.5 Filter 36
7.6 Grouper 38
7.7 Group-filter 44
7.8 Merger 46
7.9 Ungrouper 52
7.10 Linking the Elements 53

8 Traffic Patterns **55**

8.1 Web Traffic . . . 56
8.1.1 Packet-level Pattern . . . 56
8.1.2 Flow-level Pattern . . . 57
8.1.3 Query Language Specification . . . 58
8.2 FTP Download . . . 62
8.2.1 Packet-level Pattern . . . 62
8.2.2 Flow-level Pattern . . . 62
8.2.3 Query Language Specification . . . 63
8.3 W32/Blaster worm . . . 67
8.3.1 Packet-Level Pattern . . . 67
8.3.2 Flow-Level Pattern . . . 68
8.3.3 Query Language Specification . . . 69
8.4 Welchia/Nachi Worm . . . 75
8.4.1 Packet-Level Pattern . . . 75
8.4.2 Flow-Level Pattern . . . 75
8.4.3 Query Language Specification . . . 75
8.5 STUN . . . 82
8.5.1 Packet-Level Pattern . . . 82
8.5.2 Flow-Level Pattern . . . 82
8.5.3 Query Language Specification . . . 84
8.6 Skype . . . 91
8.6.1 Packet-Level Pattern . . . 91
8.6.2 Flow-Level Pattern . . . 92
8.6.3 Query Language Specification . . . 93

9 Conclusion and Future Work **100**

A FlowScan Reporting Module **106**

List of Figures

2.1 NetFlow Monitoring Architecture [1] 11
2.2 Active Flows Seen on a NetFlow Enabled Router 12

3.1 Identifying a Network Scan Attack through a SQL query . . . 14
3.2 An SQL Query for a DSMS Using a Sliding Window 14
3.3 GSQL Query for Counting the Number of Flows over 1-minute Intervals Grouping by DstIP 15
3.4 Tribeca Query Language 16
3.5 Multiplexing and Demultiplexing a Stream in the Tribeca Query Language 16
3.6 `NeTraMet` ruleset written in SRL 17
3.7 Filtering Expression for *tcpdump* 18
3.8 Filtering and Aggregating in *nfdump* 18
3.9 Example Access Control List Definition File for `flow-filter` 19
3.10 Example of *flow-tools* Filter Primitives and Filter Definition . 20
3.11 Example of *flow-tools* Report Definition 21
3.12 Displaying the Top 5 Busiest Destination IPs Connected to a Web Server 22
3.13 Aggregating Flows by Source IP and Destination Port Number 23
3.14 Grouping Flows in a Long Lived Connection 23
3.15 Grouping Flows that Belong to one TCP Connection 24

4.1 Characterization of Attack Traffic Patterns in [2] 26

7.1 IP Flow Filtering Architecture 33
7.2 Sample Filtering Architecture 54

8.1 Packet Level Breakdown of a Simple HTTP Transfer 56
8.2 Flow Level Breakdown of a Simple HTTP Transfer 57
8.3 Capturing Web Page Retrieval with the IP Flow Filtering Framework 60
8.4 Packet Level Breakdown of a Simple FTP Transfer 62
8.5 Flow Level Breakdown of a Simple FTP Transfer 63

8.6 Capturing FTP download of Big Files with the IP Flow Filtering Framework 65
8.7 Packet Level Breakdown of a Blaster Infection 67
8.8 Flow Level Breakdown of a Blaster Infection 68
8.9 Capturing Blaster Worm Infections with the IP Flow Filtering Framework 72
8.10 Capturing Nachi Worm Infections with the IP Flow Filtering Framework 79
8.11 Packet Level Breakdown of a STUN Activity 83
8.12 Flow Level Breakdown of a STUN Activity 84
8.13 Capturing STUN traffic with the IP Flow Filtering Framework 87
8.14 Packet Level Breakdown of an Unsuccessful Skype Login Attempt 92
8.15 Flow Level Breakdown of an Unsuccessful Skype Login Attempt 93
8.16 Capturing Skype traffic with the IP Flow Filtering Framework 96

A.1 *FlowScan Reporting Module for Detecting Napster Traffic* . . 107

Chapter 1

Introduction

The analysis of network traffic and more specifically Internet traffic has become an important area of research. Cisco has designed the Netflow/IPFIX protocol [3, 4] which allows to create a summary for the traffic flows that traverse a router. A network flow is defined as an unidirectional sequence of packets between given source and destination endpoints. Flow records include details such as IP addresses, packet and byte counts, timestamps, Type of Service (ToS), application ports, input and output interfaces, etc. Network elements (routers and switches) gather flow data and export it to collectors for analysis.

Although the flow records carried by NetFlow/IPFIX provide some sort of summary about the traffic traversing a specific router, this information still contains too many details for network administrators and is not useful unless processed by network analysis tools. Most of the existent tools provide mechanisms for searching of specific flows in a collection of flow records. This makes possible some simple tasks like filtering by an IP address or port number or generating Top N talkers reports. However, identifying more complex flow patterns resembles a search for a pin in a haystack. In order to describe complex traffic patterns and match a collection of flow records against the description one needs a useful flow record query language.

This paper is structured as follows: Section 2 provides background and comparison of the various approaches for analyzing network traffic data that exist in the network management domain. It also presents the NetFlow/IPFIX management protocol in more detail and gives a concrete usage example. Section 3 describes the state of the art in query languages used by network analysis tools. In Section 4 we present some further tools and algorithms used for analyzing traffic flow data. They do not use a well-defined query language or the query language that they use has not been documented. In Section 5 we present the motivation and the research statement of our

project and in Section 6 we discuss what capabilities a new IP flow record query language should have. Section 7 then introduces the IP flow filtering framework and the primitives that form our new IP flow record query language. We evaluate our approach in Section 8 by collecting a set of traffic patterns that belong to some popular network applications and services and write them down using the primitives of the new IP flow record query language. We conclude and discuss future work in Section 9.

Chapter 2

Background

2.1 Network Monitoring Domain

Network operators are very interested in understanding the ways in which traffic flows through their networks. Traffic information is vital for trouble shooting (detecting unusual behaviors), for collecting usage data (logging, security incident detection, billing) and for capacity planning. There exist several approaches for traffic measurement depending on the source and format of the management data and the way it is retrieved.

- **Analyzing Packet Layer Data** - one approach is to write copies of packets (or perhaps just packet headers) to a trace file on disk using tools such as *Wireshark* [5] and *tcpdump* [6]. Once a trace file has been collected it can be analysed offline. In case complete packets are captured one has access to the network and transport layer data as well as to the application layer data. This might give further insights into different aspects of network traffic behavior.

- **Analyzing Flow Data** - a second approach is to collect and analyze aggregated summaries of network traffic. Routers and switches inspect the packet headers of the traversing network traffic and produce flow records which contain certain accounting information (such as number of bytes, packets etc.) for all packets that have some common characteristics. The flow records are then exported to a flow collector and further analyzed by a network analysis tool. This method gives network administrators access to summarized network and transport layer data as well as to some statistics. The most widely used protocol used for this kind of network management is the Cisco NetFlow protocol which we describe in detail in Section 2.2. The IP Flow Information Export (IPFIX) protocol [3, 4] aims at being a standard

version of NetFlow while the Packet Sampling (PSAMP) specification [7] defines a packet sampling approach that is compatible with IPFIX. PSAMP supports several selection operations, including random selection, deterministic selection, and deterministic approximations to random selection. High-speed Ethernet switching engines typically found in Internet exchange points usually support the SFLOW protocol [8], a protocol that can be used to retrieve selected packet header information from a sampled stream of traffic passing through a switch.

NetFlow data provides a good view of traffic through a router or switch, but its flows are limited to unidirectional 5-tuples. The Real-time Traffic Flow Measurement (RTFM) working group of the Internet Engineering Task Force (IETF) has defined an architecture for traffic flow measurement [9], which provides a very general way for a user to specify which flows are to be measured. Furthermore, RTFM flows are bi-directional, having byte and packet counters for each direction.

- **Remote Monitoring (RMON)** - RMON, the IETFs remote network monitoring system, provides a third way to measure network traffic. An RMON agent (usually built into a switch or router) implements the RMON Management Information Base (MIB) [10], allowing a network manager to determine traffic levels in network segments, total traffic loads to/from busy hosts and traffic loads between host pairs, for different protocols, and so on. RMON, however, does not provide any flow measurement capability.

By using the first approach one obtains fine grained data and application specific information. However, collecting complete packets leads to high volume of data to be analyzed and does not scale with the current high-speed networks. The RMON-based approach leads to low volume, coarse-grained, non-application specific data. However, little information can be retrieved to do further network traffic analysis. The flow-based approach lies somewhere in the middle between both in terms of volume and level of detail. While flow traces carry less information than packet traces, there is indication backed up by experiments that flow traces if used with care well characterize the actual traffic [11]. Therefore, we will use the second approach to carry out network traffic analysis. The Cisco NetFlow definition of a flow will be implied when discussing flows in the rest of this writing.

2.2 NetFlow/IPFIX Protocol

Cisco Systems' NetFlow services provide network administrators with access to IP flow information from their data networks. Network elements (routers and switches) gather flow data and export it to collectors. The collected

data provides fine-grained metering for highly flexible and detailed resource usage accounting.

Many definitions exist for a flow. In general a flow is a set of packets which share a common property. In [12] Cisco defines a flow as a unidirectional stream of packets between a given source and destination both specified by a network-layer IP address and transport-layer source and destination port numbers. Specifically, a flow is identified as the combination of the following seven key fields:

- Source IP address
- Destination IP address
- Source port number
- Destination port number
- IP protocol type (TCP, UDP, etc.)
- ToS byte
- Input logical interface (ifIndex)

These seven key fields define a unique flow. If a flow has one different field than another flow, then it is considered a new flow. However, many NetFlow analysis tools use a 5-tuple which includes the first five fields listed above to identify a flow. In addition to the key fields, a flow contains other accounting fields which may differ slightly depending on the NetFlow version record format. According Cisco NetFlow version 5 [12], these data include the start and end times for the flow, the number of packets and octets in the flow, the source and destination Autonomous System (AS) numbers, the input and output interface numbers for the device where the NetFlow record was created, the source and destination net masks and, for flows of TCP traffic, a logical OR of all of the TCP header flags seen (except for the ACK flag). In the case of Internet Control Message Protocol (ICMP) traffic, the ICMP type and subtype are recorded in the destination port field of the NetFlow records. Cisco NetFlow version 9 [3] introduces a new template format, which includes all of these fields and optionally includes extra information, such as Multiprotocol Label Switching (MPLS) labels and IP version 6 addresses and port numbers. The IPFIX (IP Flow eXport Information) IETF working group has chosen NetFlow version 9 as a base for developing a common, universal standard of export for IP flow information from network devices. The IPFIX protocol [4] aims at being the standard version of NetFlow.

A NetFlow record is created when traffic is first seen by a router or switch that is configured for NetFlow services. The record ends and is exported to the collector when one of the following conditions are met:

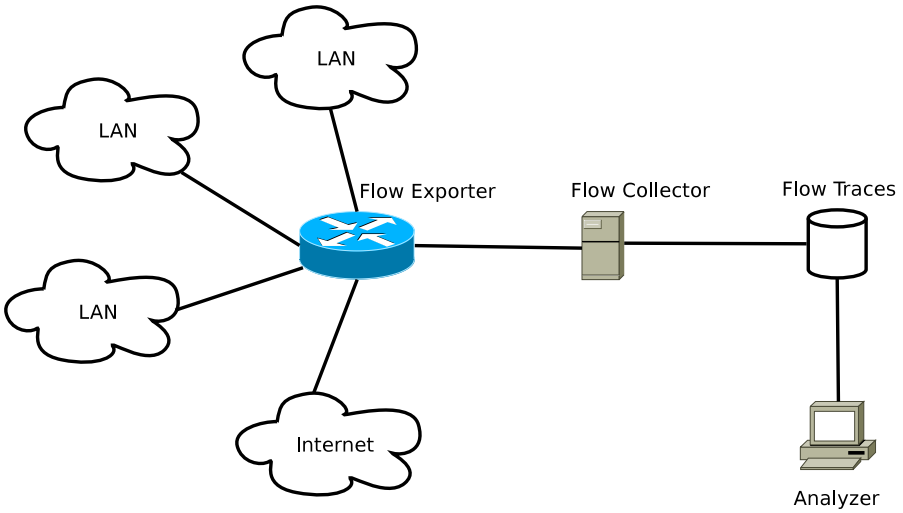

Figure 2.1: NetFlow Monitoring Architecture [1]

- If the exporter can detect the end of a flow. For flows representing TCP traffic this would be the occurrence of a FIN or RST flag.
- If the flow has been inactive for a certain period of time (Cisco default is 15 seconds).
- For long-lasting flows, the exporter should export the flow records on a regular basis (the Cisco default is 30 minutes after the start of the flow).
- If the exporter experiences internal constraints such as when the flow table fills.

A typical NetFlow architecture consisting of a NetFlow enabled router, a NetFlow collector and a host running a NetFlow analysis tool is shown in Figure 2.1

2.3 NetFlow/IPFIX Example

We illustrate the operation of Netflow with a specific example taken from [13]. Suppose that a SSH connection is established from a client on host `12.14.2.3` port 1234 to a server on host `13.18.5.6` port 22, and that the traffic passes through a router that has NetFlow processing enabled. We will simplify things and identify flows here by a tuple containing the IP Protocol type, source IP address, source port number, destination IP address and destination port number. The initial packet from the client to the

server causes the router to create a flow entry for {`TCP, 12.14.2.3, 1234, 13.18.5.6, 22`}. The response from the server to the client causes the router to create a related flow {`TCP, 13.18.5.6, 22, 12.14.2.3, 1234`}. Data from subsequent traffic will be aggregated in these two flow records until one of the ending conditions listed in Section 2.2 is seen, such as when the TCP session ends, or because there has been no traffic for 15 seconds. The active flows can be seen on Figure 2.2.

```
SrcIf SrcIP     DstIf DstIP     Pr SP   DP   Pkts
AT2/3 12.14.2.3 AT3/1 13.18.5.6 06 03FA 0016 4
AT3/1 13.18.5.6 AT2/3 12.14.2.3 06 0016 03FA 8
```

Figure 2.2: Active Flows Seen on a NetFlow Enabled Router

In the simplest case for a TCP session there will be a single flow representing the traffic from the client to the server, and a single flow representing traffic from the server to the client. The TCP flags field for both flows would typically have both the SYN and FIN bits set, indicating that packets with those flags had been seen traveling in both directions. This is not typical, however. Traffic for a single TCP connection is frequently represented by multiple flow records, due to timeouts, the flow table filling up, or the 30 minute flow maximum lifetime. This means that one often has to string multiple flow records together to get all of the data corresponding to an entire TCP session. In these cases, the TCP flags field can be used to determine whether a flow represents data from the start, middle or end of the TCP session. Flows from the start of a session will have the SYN (but not FIN or RST) bit set, flows from the middle of the session will typically have no flag bits set, and flows from the end of the session will have the FIN or RST bits set (but not SYN). Flows for UDP and ICMP traffic behave similarly, although it is important to note that since neither of these are connection oriented protocols, flows of UDP and ICMP traffic are just collections of similar packets.

Chapter 3

State Of The Art

In this section we try to classify the record query languages and split them in several groups based on their properties. In Section 3.1 we present query languages that use SQL-like syntax and in Section 3.4 we present query languages that use Berkeley Packet Filter (BPF) syntax. In Sections 3.2, 3.3 and 3.5 we present three network analysis tools that define query languages of their own. Finally, in Section 3.6 we present tools that use scripting languages to express record queries.

3.1 SQL-based Query Languages

Many of the early implementations of a network analysis tools used a Relational Database Management System (RDBMS) to store the data contained in flow records and therefore they use SQL-based query languages for retrieving flows. B.Nickless [14] describes a system which uses standard MySQL and Oracle DBMS for storing the attributes from the NetFlow records. Using powerful SQL queries, the tool was able to provide good support for basic intrusion detection and usage statistics. The query shown in Figure 3.1, for example, provides a network administrator with a list of IP addresses from external autonomous systems (AS) that have contacted a large number of internal IP addresses (which is a sign for a scan attack)

With the advance of high-speed links, however, network managers could not rely on pure DBMS anymore because of performance issues. There was also a semantic mismatch between the traffic analysis operations and the operations supported by the commercial DBMS. Traffic analysis does not require fast random access, transactional update or relational joins. Rather, it needs fast sequential access to a stream of traffic records and the ability to filter, aggregate, define windows, demultiplex and remultiplex the stream. The

```
SELECT src_ipn,COUNT(DISTINCT dst_ipn) AS num_anl_addrs
FROM netflows
WHERE (src_as>0) AND (start_time>date_sub(now(),interval 3 day))
GROUP BY src_ipn
HAVING num_anl_addrs > 64
ORDER BY num_anl_addrs
```

Figure 3.1: Identifying a Network Scan Attack through a SQL query

data used by network analysis applications can be best modeled as transient data streams as opposed to the persistent relational data model used by traditional DBMS. It is recognized that continuous queries, approximation and adaptivity are some key features that are common for such stream applications. However, none of these is supported by standard DBMS. Based on these requirements B.Babcock et al. [15] proposes the design of a Data Stream Management System (DSMS). Together with the model the authors also extend the SQL query language by adding the capability to define a sliding window. This allows operators to clearly specify which time periods in a never-ending stream of flow records should be used in aggregate queries and prevents problems related to high memory utilization and blocking operators as described in [15]. The query in Figure 3.2 computes the average call length considering the ten most recent long-distance calls placed by each customer. The underlying data source is a stream of telephone call records, each with four attributes: `customer_id`, `type`, `minutes` and `timestamp`.

```
SELECT AVG(S.minutes)
FROM   Calls S [ PARTITION BY S.customer_id
                 ROWS 10 PRECEDING
                 WHERE S.type='LongDistance']
```

Figure 3.2: An SQL Query for a DSMS Using a Sliding Window

Gigascope [16] is another stream database for network monitoring applications. It uses GSQL for query and filtering which is yet another modification of the SQL query language adopted in a way so that time windows can be defined inside the query. GSQL supports selection, join, aggregation and stream merge operations. The query execution is optimized by breaking the query plan into a low level processing part and high level processing part. The low level processing does some preliminary filtering, projection and simple aggregation. It is performed in hardware (if possible) or might involve the invocation of the Berkeley Packet Filter (discussed later as a separate filtering language) at the OS level. The high level processing performs the complex aggregation operations using data cube computation algorithms.

An example of a GSQL query is presented in Figure 3.3. It counts the number of flows carrying TCP/IPv4 traffic in every one-minute interval by grouping them according to destination IP. The first part of the query constitutes the low-level processing - it filters by IP version and protocol and projects three attributes from a flow record. In the second part aggregation operations such as grouping by time and counting are performed. The attribute time is a 1-second granularity timer, so `time/60` defines minute-long buckets. `getlpmid` is a simple function which performs longest prefix matching - that is, it defines which subnet an IP address belongs to.

```
DEFINE{query_name tcpDst0}
SELECT dstIP, dstPort, time
FROM   eth0.TCP
WHERE  IPVersion=4 AND Protocol=6

SELECT peerid, tb, COUNT(*)
FROM   tcpDst0
GROUP BY time/60 as tb, getlpmid(dstIP,'peerid.tbl') AS peerid
```

Figure 3.3: GSQL Query for Counting the Number of Flows over 1-minute Intervals Grouping by DstIP

3.2 Tribeca Stream Query Language

Tribeca [17] is another extensible, stream-oriented DBMS designed to support network traffic analysis. It is optimized to analyze streams coming from the network in real time as well as offline traces. It defines its own stream query language which is very similar to SQL and supports simple operations such as projection, selection and aggregation of streams. *Tribeca* takes streams as input, produces streams as output and uses pipes to feed streams from one operator to another. An example describing the basic capabilities of *Tribeca* is show in Figure 3.4.

The query reads a source stream of type *Atm-Trace* from tape. It uses projection to create a stream of *(time stamp, VCI)* pairs. That stream is then passed to two different *qual* operations. The first saves into a file all *(ts,VCI)* pairs with timestamp less than 1 million. The second finds all *(ts,VCI)* pairs in which the VCI is between 5 and 50. Finally, the aggregate finds the minimum time stamp from those pairs.

Tribeca can also perform demultiplexing and multiplexing of streams based on stream attributes. This allows network operators to partition a certain stream, perform component analysis and then reassemble the stream again.

```
source_stream s1 is {tape sample1 AtmTrace}
result_stream r1 is {file res1}
result_stream r2 is {file res2}
stream_pipe p1 p2
stream_proj {{s1.atm.ts s1.atm.vci}} p1
stream_qual {{p1.ts.lte 1000000}} r1
stream_qual {{p1.vci.gt 5} {p1.vci.lt 50}} p2
stream_agg {p2.ts.min} r2
```

Figure 3.4: Tribeca Query Language

Figure 3.5 shows an example of a query that uses the *demux* operator to divide the stream by *VCI* and counts the packets on each virtual circuit (VC). Note that `p1` is not a single stream but a set of substreams, each with the cells for one virtual circuit. The *(VCI,count)* pairs are then combined into a single stream (`p3`) that is qualified and aggregated.

```
stream_demux {s1.atm.vci} p1
stream_agg {p1.atm.vci p1.count} p2
stream_mux p2 p3
stream_qual {p3.count.lt 100} p4
stream_agg {p4.count.mean} r1
```

Figure 3.5: Multiplexing and Demultiplexing a Stream in the Tribeca Query Language

Finally, following the trends of a DSMS specified in Section 3.1 the *Tribeca* query language also defines a windowing mechanism to select a timeframe for the analysis. This mechanism also allows restricted join operations to be defined. At any given moment during the execution of a query, the records in a window can be compared to the current record in any stream.

3.3 SRL Language

The RTFM architecture [9] defines the SRL language [18] for describing traffic flows. SRL is used to specify filtering rulesets, which instruct flow collectors which traffic flows are of interest and which flow attributes are to be stored. SRL is defined in ABNF format and has its own primitives. NeTraMet [19] is first open source implementation of RTFM. It consists of meters, meter readers and managers. The managers use SRL for defining rulesets that get uploaded on the traffic meters.

Each SRL ruleset may start with declarations (`DEFINE` clauses), which are followed by a number of statements. There exist four types of statements - conditional (`IF/ELSE`), imperative (such as `SAVE/COUNT`), subroutine invocations or compound statements (sequence of the previous three types). The `COUNT` statement appears after all testing and saving is complete and is responsible for building the flow identifier from the attributes, which have been saved, find it in the meter's flow table (creating a new entry if this is the first packet observed for the flow), and incrementing its counters. An example of a ruleset for NeTraMet written in SRL is shown in Figure 3.6. This program counts only IPv4 packets, saving the protocol type (`SourceTransType` - tcp, udp or 0), the source and destination IP addresses (`SourcePeerAddress` and `DestPeerAddress`) and a flow identification variable (`FlowKind` - 'F' for ftp and, '?' for unclassified). The ruleset uses a `NOMATCH` action to specify the packet direction - its resulting flows will have the well-known ports as their destination.

```
DEFINE IPv4 = 1;
DEFINE ftp = (20, 21);
DEFINE tcp = 6;

IF SourcePeerType == IPv4 SAVE;
   ELSE IGNORE;  # Not an IPv4 packet

IF SourceTransType == tcp SAVE, {
   IF SourceTransAddress == ftp
      NOMATCH;
   ELSE IF DestTransAddress == ftp
           SAVE, STORE FlowKind := 'F';
   ELSE {
           SAVE DestTransAddress;
           STORE FlowKind := '?';
         }
   }
ELSE SAVE SourceTransType = 0;
SAVE SourcePeerAddress /32;
SAVE DestPeerAddress   /32;
COUNT;
```

Figure 3.6: `NeTraMet` ruleset written in SRL

3.4 BPF Filter Expressions

The *Berkeley Packet Filter (BPF)* [20] specifies simple rules which are widely used among network analysis tools to filter and query a collection of packet traces. BPF allows to construct simple logical expressions for filtering network traces by IP address, port number, protocol etc. One popular use of the BPF is in the `tcpdump` [6] utility. Figure 3.7 shows a BPF filter expression for *tcpdump* which filters all incoming or outgoing web traffic to the local network `192.168.0.1/24`.

```
(dst net 192.168.0.1) and (tcp src or dst port 80 or 443)
```

Figure 3.7: Filtering Expression for *tcpdump*

The BPF rules for constructing filter expressions are also used in `nfdump` [21], which is a powerful and fast filter engine used to analyze network flow data and generate traffic reports. `nfdump` is currently one of the *de facto* standard tools for analyzing NetFlow data and generating reports like TopN statistics for flows, bytes etc. The expression in Figure 3.8 shows a sample *nfdump* filter expression which lists all incoming flows on interface 5 that carry traffic from or to the local network (`10.0.0.0/24` except the host `10.0.0.1`) destined for port 1433. The filtered flows must also have a duration of less than 100 seconds and have a speed of more than 10kbps. Such an expression can be used to detect a Microsoft SQL Server attack (if the flow is directed to the local network) or an infection of a host with a worm (if the flow is originating from the local network).

```
nfdump -r foo -A srcip,dstport ‘in if 5 and
dst net 10.0.0.0/24 and not host 10.0.0.1 and
bps > 10k and duration < 100 and dst port 1433‘
```

Figure 3.8: Filtering and Aggregating in *nfdump*

nfdump also allows flows to be aggregated based on a combination of flow attributes such as source or destination IP address, port numbers and ASes. In the example in Figure 3.8 the listed flows are aggregated by source IP address and destination port number.

BPF expressions are also used in the *CoralReef* network analysis tool described in [22], [23] in order to generate traffic reports from collected trace files. The *Time Machine* tool described in [24] uses BPF expressions to define classes of traffic and construct queries for retrieval of interesting flows. *Ntop* described in [25] and [26] is yet another network usage monitor, similar to the Unix `top` application, for tracking and analyzing various network

activities such as network utilization, established connections etc. It also relies on BPF filter expressions for traffic filtering and classification.

3.5 Flow-Tools

`flow-tools` [13], [27] is the other widely-used collection of applications for collecting and analyzing NetFlow data. The applications are written in Perl, Python and C and, thus, there is lots of scripting involved in the implementation of network analysis modules. We discuss *flow-tools* in a separate section because the construction of filter expressions and report modules involves Cisco Access Control List (ACL) syntax [28] or syntax that is very close to Cisco ACL.

Several *flow-tools* applications are responsible for filtering flows and generating reports - `flow-filter`, `flow-nfilter`, `flow-report` and `flow-stat`. `flow-filter` uses the Cisco ACL format to specify a filter for IP addresses and command line arguments for specifying other filtering parameters such as port numbers, ASes etc. An example of a Cisco ACL which can be used with `flow-filter` is shown in Figure 3.9. The access control list `foo` only allows traffic that originates from `10.0.0.1` and `128.146.222.0/24`, while the access control list `bar` only allows TCP traffic from `10.1.1.2` to `172.16.1.1` and rejects all other IP traffic. As seen from the example `flow-tools` can use both standard ACLs, which can filter by source IP address only, and extended ACLs, which allow filtering by other attributes such as protocol, port number and destination IP address.

```
! match the attackers
ip access-list standard foo permit 10.0.0.1 0.0.0.0
ip access-list standard foo permit 128.146.222.0 0.0.0.255
ip access-list standard foo deny any

! only allow tcp traffic between these 2 hosts
ip access-list extended bar permit tcp host 10.1.1.2 host 172.16.1.1
ip access-list extended bar deny ip any any
```

Figure 3.9: Example Access Control List Definition File for `flow-filter`

The `flow-nfilter` utility filters flows based on user selectable criteria. Filters are defined in a configuration file and are composed of primitives and a definition. Filter primitives declarations begin with the `filter-primitive` keyword followed by a symbolic name. Each primitive definition has a `type` statement followed by a number of `permit` and `deny` statements. Some of the primitives types are `as`, `ip-protocol`, `ip-address-mask` etc. Filter

definitions begin with the `filter-definition` keyword followed by a number of `match` lines grouped to form logical AND and OR operations on the flow using the selected primitives. `flow-filter` has the same capabilities as `flow-nfilter`, however all filtering primitives have to be entered on the command line. The configuration file shown in Figure 3.10 will filter all flows with a destination port of 80 or source port of 25 (smtp) starting after Dec 12, 2001.

```
filter-primitive port80
  type ip-port
  permit 80

filter-primitive port25
  type ip-port
  permit smtp

filter-primitive dec12
  type time-date
  permit gt Dec 12, 2001

filter-definition foo
  match ip-source-port port80
  match start-time dec12
  or
  match ip-destination-port port25
  match start-time dec12
```

Figure 3.10: Example of *flow-tools* Filter Primitives and Filter Definition

While `flow-filter` and `flow-nfilter` do the filtering job in the *flow-tools* package, `flow-report` is the utility which allows to do some sort of aggregation based on the NetFlow attributes. The reports generated by `flow-report` are defined in a configuration file by the `stat-report` keyword followed by a report name. Each report has a type and may invoke a number of filters defined in the way described in Figure 3.10. Aggregation can be performed on a number of NetFlow attributes. If the report type is `ip-source/destination-address/input-interface`, for example, the flows will be aggregated on the three flow attributes. A definition of a report, which filters a collection of flows based on the filter defined in Figure 3.10 and aggregates the resulting flows by source IP address and a destination port number is shown in Figure 3.11.

```
stat-report bar
  type ip-source-address/ip-destination-port
  filter foo
  output
    format ascii
    options +header,+xheader,+totals,+names,+percent-total
    sort +pps
    tally 5
    path /tmp/output6
```

Figure 3.11: Example of *flow-tools* Report Definition

3.6 Script-Based Query Languages

There is a number of network analysis tools that are capable of generating powerful high-level traffic reports, which might help operators to detect interesting traffic patterns. However, reports must be specified as separate modules written in a scripting language. While this method allows to define some heuristics, writing such report modules is not trivial and might involve some heavy scripting.

FlowScan described in [29] is a collection of perl scripts which glues together a flow-collection engine such as the `flow-capture` application from `flow-tools`, a high performance RRD database, which is specifically designed for time series data [30], and a visualization tool. The reporting mechanism of *FlowScan* consists of a central perl module which is responsible for loading and executing report modules. The modules themselves are also written in Perl. Initially the *FlowScan* was distributed with its original report modules `CampusIO.pm` and `SubNetIO.pm`. Later more intelligent report modules such as `CUFlow` [31] were developed which allow some flexible configuration of IP addresses, services and other NetFlow parameters. However, the description of heuristic approaches for detecting high-level traffic patterns would still involve a reasonable amount of Perl scripting. An example function from the `CampusIO` class that detects Napster traffic is shown in the Appendix.

Stager [32] is another tool for presenting network traffic statistics. It consists of a number of backends which are responsible for collecting network management data from protocols such as NetFlow and SNMP. The data is then processed by a set of Perl scripts which generate traffic reports by doing filtering and aggregation. The set of reports that can be produced based on a collection of flow records is limited to displaying various traffic statistics for flows aggregated by a set of NetFlow attributes. The tool does

not allow to define analysis specific heuristics as shown in *FlowScan*.

AutoFocus [33] is a network analysis tool, written in Perl and C++, in which C.Estan et al. propose an approach for detecting high-level traffic patterns by aggregating NetFlow records in clusters based on the flow record attributes. Aggregation on several flow attributes results in a multidimensional cluster. The traffic mix is defined using the source and destination IP address, source and destination ports and protocol field. Initially all possible multidimensional clusters are constructed. Then an algorithm is executed which selects only clusters that are interesting to the network administrator. It aims at retaining clusters with the least degree of aggregation (so that a bigger number of flow attributes is contained). Interesting activities are considered to be exceeding a certain threshold of traffic volume of a cluster or significant change of the traffic volume inside the cluster. Finally, all clusters are prioritized by being tagged with a degree of *unexpectedness* and presented to the network administrator as a traffic report.

The *SiLK* Analysis Suite [34] is a collection of command-line tools for querying packed NetFlow data written in C, Perl and Python. By combining the various tools one can perform various query operations, ranging from per-record filtering to statistical analysis of groups of records. The analysis tools interoperate using pipes, allowing a user to develop a relatively sophisticated query from a simple beginning. The most important tool is `rwfilter`, an application for querying the central NetFlow data repository for NetFlow records that satisfy a set of filtering options. `rwstats` can be used to summarize flow records by one of a limited number of key/value pairs and display the results as a Top-N or Bottom-N list. `rwuniq` is a general purpose counting tool- it provides counts of the records, bytes, and packets for any combination of fields, thus allowing aggregation of flow records based on any number of NetFlow attributes. Figure 3.12 shows an example of using `rwfilter` and `rwstats` to display statistics about the most active IPs (according to number of flows), which are connected to a local web server. The invocation to `rwfilter` filters all TCP traffic to `10.0.0.1` and port 80. Then the second command aggregates the resulting flows by source IP address and displays the top 5 flows.

```
rwfilter --proto=6 --daddress=10.0.0.1 \
--dport=80 --pass=stdout someflows.raw \
| rwstats --sip --top --flow --count=5
```

Figure 3.12: Displaying the Top 5 Busiest Destination IPs Connected to a Web Server

In Figure 3.13 we show and example of a combination of `rwfilter` and `rwuiq`, which filters all TCP traffic, aggregates the resulting flows by fields

1 and 4 (source IP address and destination port number) and displays all entries that have more than 100 flows.

```
rwfilter --proto=6 --pass=stdout someflows.raw \
| rwuniq --field=1,4 --flows=100
```

Figure 3.13: Aggregating Flows by Source IP and Destination Port Number

Unlike other network analysis tools, *SiLK* contains two grouping tools that allow an analyst to label a set of flows that share common attributes with an identifier. The `rwgroup` tool walks through a file of flow records and groups records that have common attributes, such as source/destination IP pairs. This tool allows an analyst to group together all the flows in a long lived session (such as a FTP connection for example) as shown in Figure 3.14. In this example all flows that have identical source and destination IP addresses and ports (NetFlow fields 1,2,3,4) and timestamps that differ by at most one hour (field 9) are grouped together. In a similar manner one can tag web traffic flows from a single user that are closely related in time and then use that information to identify individual webpage fetches.

```
rwgroup --id-field=1 --id-field=2 --id-field=3 --id-field=4 \
--delta-field=9 --delta-value=3600
```

Figure 3.14: Grouping Flows in a Long Lived Connection

`rwmatch` creates matched groups, where a matched group consists of an initial record (a query) followed by one or more responses. A response is a record that is related to the query but is collected in a different direction or from a different router. As a result, the fields relating the two records may be different: for example, the source IP address in one record may match the destination IP address in another record. The most basic use of `rwmatch` is to group records into both sides of a bidirectional session, such as a Hypertext Transfer Protocol (HTTP) request. However, `rwmatch` is capable of more flexible matching, such as across protocols to identify `traceroute` messages. Figure 3.15 shows and example in which flows that belong to a single TCP connection are grouped together. Two records are considered related if all of their related fields are equal and their start times match within a value specified by `time-delta`. The example shown in Figure 3.15 states that the source IP from the first record (field 1) should match the destination IP of the second record (field 2) and the other way round. The same criteria must hold for the source and destination port numbers and the flows must have starting times that do not differ by more than 30 seconds.

```
rwmatch --relate=1,2 --relate=2,1 --relate=3,4 --relate=4,3 \
 --time-delta=30
```

Figure 3.15: Grouping Flows that Belong to one TCP Connection

Tool	Query Language	Input Data Format
B.Nickless et. al. [14]	SQL	RDBMS
B.Babcock et. al. [15]	extended SQL	DSMS
Gigascope	GSQL	DSMS
Tribeca	proprietary	DSMS
NeTraMet	SRL	flow files (bi-directional)
`tcpdump`	BPF	`pcap` files
`nfdump`	BPF	`nfcapd` raw NetFlow files
CoralReef	BPF	`pcap` and `crl` files
Time Machine	BPF	indexed `pcap` files
`ntop`	BPF	`pcap` files/raw Netflow files
Flow-Tools	ACL/proprietary	`flow-capture` raw NetFlow files
FlowScan	perl script	`flow-capture` raw NetFlow files
Stager	perl script	raw NetFlow files
AutoFocus	proprietary	packet header traces/raw NetFlow files
SiLK	proprietary	raw NetFlow files

Table 3.1: Query languages used by network traffic analysis tools

In addition to the properties described *SiLK* has other tools which allow aggregation by IP subnets and creation and filtering based on IP sets. As shown in the examples in Figure 3.14 and 3.15 *SiLK* is the only tool among the huge collection of network monitoring tools that we have reviewed which is capable of declaring the correlation of flows based on NetFlow record attributes. Therefore, we believe that it might serve as a good basis for a new flow query language.

A summary of the query languages used by the various network traffic analysis tools is presented in Table 3.1.

Chapter 4

Flow-level Analysis of Internet Traffic

Section 3 presented an overview of the existent query languages used by various network traffic analysis and monitoring tools. However, a number of tools, which perform flow-level analysis of Internet traffic do not utilize a well-defined flow query language or their query language has not been documented in the literature. In this section we present related work in the area of flow-based identification of traffic anomalies and flow-based classification of traffic.

In [35] D.Brauckhoff et. al. present the idea of extracting simple and realistic models of malicious and benign traffic anomalies from a NetFlow dataset and storing them in a database. Later on, these anomalies will be injected into background traffic traces, which can be used for testing and training flow-level anomaly detection systems. Unfortunately, there is no formal description of how the traffic anomaly models are built. In [36] P.Barford and D.Plonka describe a project focused on characterization of anomalous network traffic behavior by using flow-level data. They use FlowScan [29] to identify a variety of traffic anomalies. The flows that belong to these anomalies are further clustered into groups such as network operation anomalies, flash crowd anomalies and network abuse anomalies based on similarities observed in the flow behavior. The flows in each of these clusters are analyzed by using simple statistics, time series analysis and wavelet analysis to derive some flow anomaly pattern for each of the three groups. M.Kim et. al. [2] propose an approach for describing traffic anomalies such as DoS attacks, worms etc. by using flow-level patterns. The paper characterizes some simple scanning and flooding attacks by using flow record attributes. These are shown in Figure 4.1. In order to define the patterns of more complicated attacks, flows are aggregated by source or destination IP ad-

dress and some statistics on the flow record attributes is calculated. Then a similar approach to the one shown in Figure 4.1 is used to identify scanning or flooding attack. However, the implementation of this method in the NG-MON network monitoring tool [37] utilizes complex mathematical expressions where some of the parameters as well as the thresholds for small (S) and Large (L) are adjusted manually using the trial and error method.

L : Large S : Small

Attacks / Property	scanning		flooding				
	host	network	TCP SYN	smurf	fraggle	ping-pong	general (ICMP,UDP,TCP) flooding
flow count	L	L	L	L	L	S	L or S
flow size/flow	S	S	S	L or S	L or S	L	L or S
packet count/flow	S	S	S	L or S	L or S	L	L or S
packet size	S	S	S	L or S	L or S	L or S	L or S
total bandwidth	L or S	L or S	L or S	L	L	L	L
total packet count	L or S	L or S	L or S	L	L	L	L
property	1 destination	1 port		ICMP broadcast	UDP broadcast	reflecting port	

Figure 4.1: Characterization of Attack Traffic Patterns in [2]

In [38] T. Dubendorfer et. al. present a detailed flow-level analysis of the Blaster and Sobig worms. The paper contains a detailed discussion on the sequence of packets sent by an attacker infected with one of these worms. Later on in the analysis these packets are aggregated into flows to build a flow-level fingerprint of the two worms. We have used the flow fingerprint of the Blaster worm as one of our flow patterns in Section 8. In addition to the approach described in [38], T. Dubendorfer et. al. present another novel approach for detection of worms by using flow-level data. In [39] they describe how by analyzing backbone traffic at flow-level, they can attribute various behavioral properties to hosts like ratio of outgoing to incoming traffic, responsiveness and number of connections, which are strongly influenced by a worm outbreak. These properties are used to group hosts into distinct classes (traffic, connector, responder) and by tracking the cardinality of these classes over time worm outbreak events can be reliably detected. For example, the outbreak of the Sobig worm is characterized with a sudden increase of the cardinality of the traffic and connector classes. The entropy of some flow record attributes can also be used to detect worm outbreaks. In [40] A. Wagner et. al. present an approach for identification of worm outbreaks by measuring how random the distribution of various flow record attributes is. The Witty worm, for example, can be successfully detected by the fact that during its outbreak the destination IP address space and destination

port numbers become much less compressible (i.e more random) compared to a normal network behavior. The host-based approach from [39] and the entropy-based approach from [40] are implemented as plugins in the framework for real time worm attack detection UPFrame [41]. Another plugin for identification of worm attacks is based on activity detection by utilizing activity plots. An activity plot visualizes the IP address space or the port range in a collection of flows. The color of each pixel is associated with the traffic generated by the specific IP address or port. Such an activity plot can be used to successfully detect the outbreak of the Nachi worm, which scans a huge sequential IP address space but skips IP addresses containing the octet 197.

Finally, there has been quite some work in the direction of identifying Peer-to-Peer traffic based on flow-level data. In [42] A. Wagner et. al. present a *PeerTracker* algorithm, which identifies hosts participating in most common P2P networks based on Cisco NetFlow traces. The algorithm relies on the fact that although in P2P networks peers usually use ports above 1024, they sometimes exchange messages on some well-known ports. Thus, *PeerTacker* keeps statistics of the activity on the 100 most recent ports of each host and by using pre-defined ranges for the bandwidth of common P2P networks it is able to properly identify if a host is participating in one of these networks. In [43] M.Kim et. al. propose an algorithm for identifying P2P traffic using flow traces, which is implemented as part of the NG-MON tool [37]. In the initial phase, the algorithm tries to match and aggregate the flows that utilize well-known port numbers. In the Flow Relation Mapping phase of the algorithm the un-matched flows are correlated to the already matched ones based on flow record attributes. Each correlation is associated with a certain weight and based on that the flow trace is split into groups where each group identifies traffic that belongs to a certain P2P application. R. van de Meent and A. Pras follow a similar approach in [44] to identify unknown traffic in NetFlow traces. Their algorithm is based on the hypothesis that unidentified traffic on unknown ports is induced by already identified traffic on registered ports between the same peers. This leads to a significant portion of the unknown traffic being associated with standard applications.

Chapter 5

Statement and Motivation of Research

Given the large number of flow records collected on high-speed networks, it is necessary to reduce their number to a comprehensible scale using filtering and aggregation mechanisms. Each flow or aggregated flow has a set of properties attached to it that characterize the flow. It is to be expected that flows that correspond to similar network activities (certain applications or certain attacks) have similar properties. In addition to the properties recorded in flow records, one can derive further properties that are even more suitable to characterize the behavior of a flows. One objective when investigating traces is to detect traffic regularities such as repeating patterns, which can be associated with the usage of common network services. This approach can be further extended to detect traffic irregularities such as network anomalies or attacks, which also generate specific patterns. These patterns typically spread over several flows. For example, if an intensity peak in flow X always occurs after an intensity peak in flow Y with a fixed delay, they form a pattern describing a certain network behavior. The goal of network administrators is to detect such patterns of correlated flows.

For example, one would be interested in finding out where, when, and how often a certain Internet service is used. A concrete scenario is a network administrator who wants to detect VoIP applications by finding STUN flows generated by VoIP applications when they try to discover whether they are located behind a Network Address Translator (NAT). If one knew the pattern that is created when a service is trying to establish a connection, one could search for this specific pattern in the selected flows. We are aware that although the presence/absence of a certain pattern may be a hint for the presence/absence of a particular service this by no means proves that the service is really running/missing.

The goal of this project is to design a flow record query language, which allows to describe patterns in a declarative and easy to understand way. The language should be able to define filter expressions (needed to select relevant flows) and relationships (needed to relate selected flows). Another requirement is that it should be possible to express causal dependencies between flows as well as timing and concurrency constraints. Existent query languages as discussed in Section 3 are not suitable for detecting complex traffic patterns because of either performance issues (SQL-based query languages) [17, 15] or because they lack a time and concurrency dimension (BPF expressions and the other query languages we discussed). Furthermore, the new query language should provide support for network specific aggregation functions, such as IP address prefix aggregation, IP address suffix aggregation, port number range aggregations, etc. which are not part of many standard query languages. Using the new query language we will build a knowledge base of netflow fingerprints that belong to some common network services, applications and attacks.

Chapter 6

Query Language Requirements

During the proposal phase the following requirements have been identified for our new IP flow record query language:

- **Aggregation** - our query language should provide support for network specific aggregation functions, such as IP address prefix aggregation, IP address suffix aggregation, port number range aggregations, etc. For example, the Nachi worm can be identified by performing aggregation of IP addresses based on the IP octets. Furthermore, one should be able to calculate various statistics based on the aggregated flows. The capability to detect traffic anomalies and patterns in many of the existent tools analyzed in Section 3 is based on different aggregation operations.
- **Dependency** - the query language should be able to express dependencies and relationships between flows in several dimensions:
  - *correlation* - the query language should have the capability to correlate flows based on a number of flow record attributes such as matching flows where the source IP address of flow A is the same as the destination IP address of flow B. This can be used to identify request/response messages as in the SiLK example shown in Figure 3.15.
  - *time dependencies* - one should be able to define a time window in a collection of network flow records. Other time-related characteristics are delays between flows, start and end times of a flow, duration of a flow, flow concurrency etc. Allen's calculus [45] may be used as a base for building the necessary language primitives.

- *flow order* - one should be able to define the order in which flows occur in the trace file. In Section 8.3 we figured out that the Blaster worm can be identified as sequence of three flows (using ports TCP/135, TCP/4444 and UDP/69).
- *existence or non-existence of specific flows* - one should also be able to query for the existence or non-existence of a specific flow in a certain collection of flows. For example, existence of an unidirectional flow in a TCP connection might be a reason for inferring that there is a firewall on the network path.
- *dependencies between non-absolute flow attributes* - one should be able to define implied dependencies between flow attributes such as port numbers or IP addresses. For example, during the initial phase of a blaster worm infection the attacker scans 20 consecutive IP addresses. Thus, the presence of flows to port 135/TCP with a destination IP addresses `A.B.C.D`, `A.B.C.(D+1)`, `A.B.C.(D+2)`, ..., `A.B.C.(D+19)` might be a reason to infer that there is an occurrence of the first phase of Blaster infection.

- **Comparison operations on the flow record attributes** - In BPF one can perform comparison operations on the packet header attributes. Our query language should inherit the BPF capabilities for doing comparison operation on the flow record attributes. We have seen that port comparison can be used as a very powerful approach for service identification based on a list of well-known ports that are registered with IANA [46] and Grafitti [47] (to the extent that traffic uses well-known port numbers)

Chapter 7

Framework for IP Flow Filtering

In this section we present our stream-based framework for IP flow filtering and the primitives used for its definition. Section 7.1 provides an overall description of the IP flow filtering framework and its elements. Then in Section 7.2 and Section 7.3 we provide a list of the Netflow/IPFIX elements and the other keywords that are used within our framework. From Section 7.4 through Section 7.9 we describe each element from the IP flow filtering framework, show how it can be defined using our primitives and provide a short usage example. Finally, in Section 7.10 we show how the various elements can be linked together.

7.1 Framework

Our framework for IP flow filtering is shown in Figure 7.1. It follows a stream-oriented approach - it consists of a number of elements, which are connected with each other via pipes. Each element gets an input, performs some sort of operation on it (filtering, aggregation etc.) and the output is piped to the next element. The following elements are used within the framework:

- `filter`
- `grouper`
- `group-filter`
- `splitter`
- `merger`

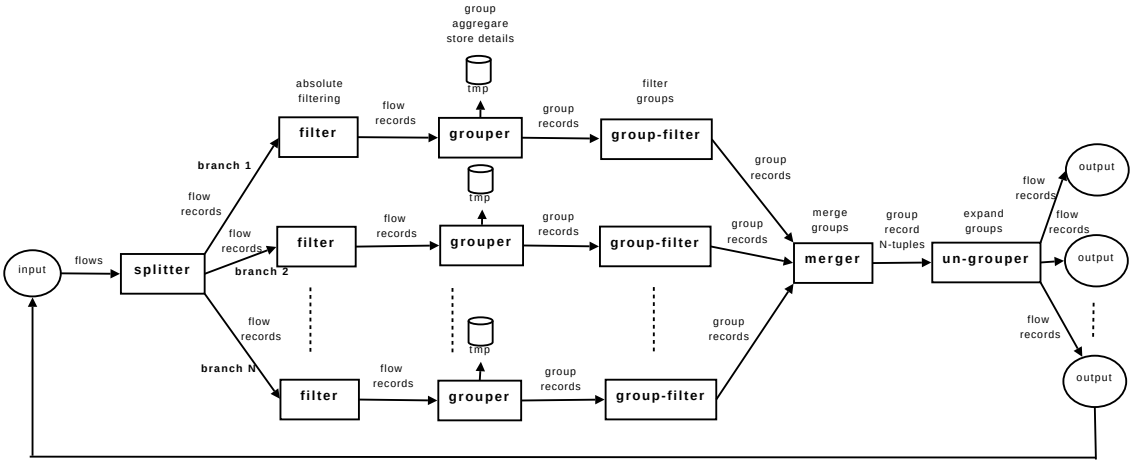

Figure 7.1: IP Flow Filtering Architecture

- `ungrouper`

7.2 NetFlow/IPFIX Attributes

The NetFlow/IPFIX attributes used within our framework as well as the filtering primitives used for their representation are listed in Table 7.1. The first column specifies the keyword used within our IP flow filtering framework, the second column provides a brief description of the flow record attribute and in the third column we describe the list of possible values for that attribute. Most of the flow record attributes used within our framework are taken from the IPFIX information model specified in RFC 5102 [48] and therefore in the third column we specify the corresponding IPFIX information model element. A detailed description of the allowed values for those elements can be found in [48]. For some of the flow record attributes, however, we have decided to use a different set of values since they seem more user-friendly.

7.3 Keywords

Table 7.2 lists the variables and the associated keywords that are used for the definitions in Sections 7.4 through 7.10. Variables are terminal symbols in the BNF context and used in the definitions to identify the occurrence of flow attributes, various operators etc. In the real examples they will be substituted by the respective keywords.

Filtering primitive	Description	Value
`srcip`	IPv4 source IP address	`sourceIPv4Address`
`srcmask`	IPv4 source mask	
`srcprefix`	IPv4 source prefix length	`sourceIPv4PrefixLength`
`dstip`	IPv4 destination IP address	`destinationIPv4Address`
`dstmask`	IPv4 destination mark	
`dstprefix`	IPv4 destination prefix length	`destinationIPv4PrefixLength`
`srcport`	Source port number	`sourceTransportPort`
`dstport`	Destination port number	`destinationTransportPort`
`proto`	Transport layer protocol	`tcp, udp, sctp, dccp..`
`bytes`	Size of data in bytes	`octetDeltaCount`
`packets`	Size of data in packets	`packetDeltaCount`
`flags`	Logical OR of TCP flags	a string of S,A,R,F

`stime`	Start time of a flow	`flowStartSeconds`
		`flowStartMilliseconds`
		...

`etime`	End time of a flow	`flowEndSeconds`
		`flowEndMilliseconds`
		...

`duration`	Duration of a flow	duration of a flow in `s, ms, ..` for example `1s, 5ms, etc.`
`tos`	Logical OR of TOS bytes	
`next-hop`	BGP next hop router	`ipNextHopIPv4Address`
`in-if`	Router input interface	`ingressInterface`
`out-if`	Router output interface	`egressInterface`
`src-as`	Source AS number	`bgpSourceAsNumber`
`dst-as`	Destination AS number	`bgpDestinationAsNumber`
`exporter`	Exporter IP address	`exporterIPv4Address`

Table 7.1: Netflow/IPFIX attributes

Variable	Keywords	Description

`splitter-id`
`filter-id`
`grouper-id`
`group-filter-id`
`merger-id`
`ungrouper-id`
`branch-id`
`module-id`
`name-id`
	A string identifier	An identifier for the operators of the IP flow filtering framework

`attribute`	A Netflow/IPFIX attribute from Table 7.1	Keywords for the flow record attributes
`value`	Value or a range of values for an `attribute`	Value or a range of values for a flow record attribute
`eq-op`	`=, !=, <, >, <=, >=, <<, >>, in, notin`	Equality/inequality operators
`log-op`	`AND, OR, NOT`	Logical operators
`comp-op`	`+, -, /, *, delta, mask, prefix`	Computational operators
`comp-op-grouper`	`+, -, /, *, relative-delta absolute-delta, mask, prefix`	Computational operators used by a grouper
`aggr-op`	`min, max, avg, sum, count bitAND, bitOR, mask, union`	Aggregation operators
`allen-op`	`<,>,m,mi,o,oi,s,si,d,di,f,fi,=`	Allen operators
`element-id`	`splitter, filter, grouper merger, ungrouper`	A common name for the elements listed in Section 7.1

Table 7.2: IP flow filtering framework keywords

7.4 Splitter

- **Description** The `splitter` is the simplest operator in the IP flow filtering framework. It takes the input stream of flow records/group records and copies them on each output stream without performing any changes on them. There is one input branch and several output branches for a `splitter`.
- **Definition**

  ```
  splitter <splitter-id> {}
  ```

- **Input** A stream of flow records/group records.
- **Output** Several streams of flow records/group records, one stream per output branch.
- **Usage Example** Defining a `splitter` is straighforward.

  ```
  splitter s {}
  ```

7.5 Filter

- **Description** The `filter` operator takes a stream of flow records as an input and copies to its output stream only the flow records that match the filtering rules. The flow records, which do not match the filtering rules are dropped. The `filter` operator performs *absolute* filtering, it compares the flow attributes of the input flow records with absolute values (or a range of absolute values). It can also perform comparison between the various fields of a single flow record, that is it can compare one field of a flow record against another field of the same flow record (for example source port number with destination port number). The `filter` operator does not support *relative* filtering between fields from different flow records i.e., it does not perform comparison between the flow attributes of different incoming flow records.
- **Definition**

  ```
  filter <filter-id> {
  <filter-id> <log-op> <filter-id> ...
  <filter-id> <log-op> <filter-id> ...
  ...
  ...
  <filter-id> <log-op> <filter-id> ...
  <filter-statement> [OR <filter-statement>] ...
  <filter-statement> [OR <filter-statement>] ...
  ```

```
...
...
<filter-statement> [OR <filter-statement>] ...
}

<filter-statement> = <attribute> <eq-op> <value> |
    <attribute> <eq-op> <attribute> [<comp-op> <value>]
```

The filtering rules are written one rule per line and are connected by an implicit AND. There are two types of filtering rules:

- Simple - several attributes can be combined via an OR operation on one line
- Composite - used to define filters by linking already defined filters with logical operators

- **Pseudocode** The following pseudocode illustrates how a filter F made of simple and composite rules decides whether an incoming flow record fr should be copied to the output or dropped.

```
evaluate_filter (F,fr) {
  output = fr;
  for each rule R in F {
    if R is simple {
      if for all filter statements fs in R check(fs,fr)==false
         output = NULL; // fr doesn't match any of the OR statements
    }

    else { // F is a composite rule
       for each filter statement fs in R {
         output = evaluate_filter(fs,output);
         if (output!=NULL)
            break; //a filter was matched
       }
    }

    if (output==NULL)
      break; //do not check further rules
    }
  return output;
}
```

- **Input** A stream of flow records/group records.
- **Output** A stream of flow records/group records, consisting of those input flow records/group records that match the filtering rules.

- **Usage Example** A definition of a `filter` that matches HTTP traffic between `80.91.34.1` and the network `212.201.49.0/24`

```
filter http {
 srcip = 80.91.34.1
 dstip in 212.201.49.0/24
 srcport = 80 OR dstport = 80
}
```

A definition of a `filter` that matches SMTP traffic between `80.91.34.1` and the network `212.201.49.0/24`

```
filter smtp {
 srcip = 80.91.34.1
 dstip in 212.201.49.0/24
 srcport = 25 OR dstport = 25
}
```

A definition of a `filter` that matches SMTP or HTTP traffic between `80.91.34.1` and the network `212.201.49.0/24`

```
filter http_smtp {
 http OR smtp
}
```

7.6 Grouper

- **Description** The `grouper` operator takes a stream of flow records as an input and partitions them into groups and subgroups following grouping rules. The grouping rules themselves are organized into rule modules, where each rule module contains a number of rules logically linked by an implicit AND. The different rule modules on the other hand are logically linked by an implicit OR. The rules reflect some relative dependencies and patterns among the attributes of the input flow records. The `grouper` tags each flow record with a group label and each group consists of flow records tagged with the same group label. Internally, each group consists of several not necessarily non-overlapping subgroups, which correspond to the different rule modules. The `grouper` also tags each flow record with a rule module identifier (also called a subgroup label) if the flow record satisfies the set of rules within the corresponding rule module. In order to be added to a group a flow record must satisfy the rules from at least one rule module. In case a flow record satisfies the rules from several rule modules it is tagged with the rule module identifier of all matching rule modules and thus belongs to several subgroups. The way group and subgroup labels

are stored into flow records is implementation specific, for example the `SiLK` tool [34] stores the labels in the `next-hop` field of the flow records.

For each group of flow records a group record is created. It may consists of the following attributes:

- Flow record attributes according to which the grouping was performed. This is usually a set of attributes that are unique for a subgroup and form a *key* for that subgroup (if we speak in terms of a database record). If there is a single rule module within a `grouper` definition then there will be a single subgroup and these flow record attributes will be unique for the group as well.
- Aggregated values for other flow record attributes from a single subgroup. If the subgroups within a single group are identified by `g1`, `g2` etc. then the group record may contain members such as `g1.sum(attr1)`, `g1.min(attr2)`, `g2.max(attr3)` etc.
- Aggregated values for other flow record attributes from the whole group. In such a case the aggregation is performed over the flow records of the whole group (as opposed to aggregation over a single subgroup). For example, the group record may contain members such as `sum(attr1)`, `min(attr2)`, `max(attr3)` etc. Note that in this case we can drop the subgroup label.

Once the group record is created the subgroup labels are not needed anymore and can be deleted. Finally, the group records are copied over the output stream. During the grouping operation some information from the original flow record trace is lost because of the aggregation operation during the creation of the group records. Therefore, after the grouping and tagging is performed and before the aggregation phase the tagged flow records are copied to a temporary storage so that they can be later retrieved by the `ungroup` operator.

During the grouping phase a greedy approach is followed to partition the flow records into groups. Initially, we start with an untagged trace of flow records ordered by their timestamps. We take the first flow record into consideration and tag it with a unique group label. The first flow record is also tagged with the module identifiers of all rule modules (i.e it belongs to all subgroups). Then we iterate through the trace file checking if the grouping rules from the various rule modules are satisfied for each other flow record. For each match between a flow record and rule module we tag the corresponding flow record with the same group label and the corresponding module identifier (subgroup label). The next step is to take into consideration the next (second) flow record in the trace file. If it has already been tagged, then we skip it and proceed with the next flow record. If it is untagged

then we repeat the same procedure for matching rules and tagging the respective matching flow records with a unique group label and the respective module identifiers. The matching and tagging procedure is repeated until we reach the end of the flow record trace file. We end up with a set of flow records where each flow record is tagged with a unique group label and a number of module identifiers (subgroup labels). Usually, the grouping rules within a rule module specify that subgroups cannot span more than a certain time window and thus we do not have to walk through the complete trace file checking for a match for every single flow record. According to our algorithm a flow record may belong to only one group but may be part of several subgroups (if it matches the rule from more than one rule module).

- **Pseudocode** Our greedy approach can be written as follows in pseudocode:

```
flow_record = file->first;
do {
   if (flow_record.group_label == NULL){
     gl = generate_group_label();
     tag(flow_record, gl);
     for each rule module m {
        sl = generate_subgroup_label(m);
        tag(flow_record, sl);
     }
     for (r=flow_record->next; r != EOF; r=r->next)
        for each rule module m
           if (match(flow_record, r, rules(m)) && (r.group_label==NULL)){
              tag(r,gl);
              sl = generate_subgroup_label(m);
              tag(r,sl);
           }
   }
   flow_record = flow_record->next;
} while (flow_record != EOF)

for each group label gl {
  for each subgroup label sl {
    do_subgroup_aggregations(sl);
  }
  do_group_aggregations(gl);
  create_group_record(gl);
}
```

- **Definition**

```
grouper <grouper-id> {
<group-module-definition>
<group-module-definition>
...
...
<group-module-definition>
<aggregate-statement>
}

<group-module-definition> =
module <module-id> {
<relative-statement>
<relative-statement>
...
...
<relative-statement>
}

<relative-statement> =  \
<attribute> <eq-op> <attribute> [<comp-op-grouper> <value>]
<aggregate-statement> =                                                    \
aggregate [<module-id>.]<attribute> as [<name-id>],                        \
          [<module-id>.]<attribute> as [<name-id>],                        \
          ...                                                              \
          ...                                                              \
          [<module-id>.]<attribute> as [<name-id>],                        \
          [<module-id>.]<aggr-op>(<attribute>,[value]) as [<name-id>], \
          [<module-id>.]<aggr-op>(<attribute>,[value]) as [<name-id>], \
          ...                                                              \
          ...                                                              \
          [<module-id>.]<aggr-op>(<attribute>,[value]) as [<name-id>]  \
```

The definition of a `grouper` consists of a number of rule module definitions and an `aggregate` statement. Each rule module cosists of a number of statements, which specify the grouping rules (i.e the relationship between the flow record attributes). All statements are written one per line. The meaning of the `relative-delta` and `absolute-delta` operators is best illustrated through the usage example provided for the `grouper` operator. The aggregate statement specifies how the group record is constructed (i.e what information will be retained for a group).

- **Input** A stream of flow records.
- **Output** A stream of group records, one group record for a set of

flow records tagged with the same label. A group record consists of a number of flow record attributes (unique for a group/subgroup) and a number of aggregated flow record attributes.

- **Note** It is important to mention that a group is a set of flow records tagged with the same label, while a group record is the information that the `grouper` retains for a group. Only group records get copied as an output from the `grouper`. However, we may use both terms interchangeably (group record vs. groups) when we don't refer to one of the two specifically but rather want to refer to a group of flows that follow a certain pattern. Furthermore, we may refer to a group as `flow group` in order to be more specific in a certain context.
- **Usage Example** Partition a set of flow records into groups according to the following criteria:
  - Each group should represent a scanning attack.
  - We define a scanning attack to be a sequence of packets from a single source IP address to a block of consecutive IP addresses and a fixed port number.
  - A scanning attack cannot last more than 5 seconds i.e if we observe scanning activity from a particular host more than 5 seconds after we observed a previous scanning activity originating from the same host then this is considered to be a separate scanning attack and should be placed in a new group.
  - A scanning attack cannot be interrupted for more than 5 ms (usually scanners generate the traffic to its victims at a very high rate). That is if we observe a pause of more than 5 ms between the flow records that belong to the scanning activity of a particular host then the new scanning activity should be considered to be a new scanning attack and thus placed in a separate group.
  - For each scanning attack retain the source IP address (i.e the attacker), the destination port number (the port on the victim), the union of the IP addresses scanned, the start and end times of the scan, the number of flows, packets and bytes as group record attributes.

```
grouper scan {
   module g1 {
      srcip = srcip
      dstip = dstip relative-delta 1
      dstport = dstport
      stime = stime relative-delta 5ms
      stime = stime absolute-delta 5s
```

```
   }
   aggregate srcip, dstport, union(dstip),      \
             min(stime) as stime,               \
             max(etime) as etime, count,        \
             sum(packets) as packets,           \
             sum(bytes) as bytes
}
```

This grouper contains only one rule module and therefore we may omit the module identifier when writing down the group record attributes in the `aggregate` statement. The grouping rules state that a new flow record x should be added to an existent group G, originated by a flow record g (that is the first flow record added to that group) and having members $(g, g_1, g_2, ...g_n)$ if the following conditions are met:

- x must have the same source IP address and destination port number as g (as well as the other members of the group).
- The destination IP address of x must differ by at most 1 from the destination IP address of the last added flow record of the group.
- The start time of x must differ by at most 5ms from the start time of the last added flow record of the group.
- The start time of x must differ by at most 5s from the start time of the first flow record from the group g.

The following grouper contains two grouping rule modules and is usually used to partition the incoming set of flow records into groups in such a way that each group corresponds to the bi-directional communication between two transport endpoints. In the case of TCP this could be used to aggregate all flow records that correspond to a single TCP connection in one group. For each group we retain the amount of data (in term of bytes) in each direction of the communication as well as the overall amout of data exchanged between the two transport endpoints.

```
grouper g_bidir_communication {
   module g1 {
      srcip = dstip
      dstip = srcip
      srcport = dstport
      dstport = srcport
      stime = stime relative-delta 5ms
      stime = stime absolute-delta 5s
   }
   module g2 {
```

```
      srcip = srcip
      dstip = dstip
      srcport = srcport
      dstport = dstport
      stime = stime relative-delta 5ms
      stime = stime absolute-delta 5s
   }
   aggregate g1.srcip as srcip,           \
             g1.dstip as dstip,           \
             g1.srcport as srcport,       \
             g1.dstport as dstport,       \
             g1.sum(bytes) as bytes_out,  \
             g2.sum(bytes) as bytes_in,   \
             sum(bytes) as bytes_total,   \
             min(stime) as stime,         \
             max(etime) as etime,
}
```

Each of the rule modules contains grouping rules which match the corresponding direction of the communication. When creating the group record we use the first subgroup to extract the transport endpoints (IP addresses and port numbers) and then take the amount of bytes in each of the two directions of the communication by aggregating the data exchanged within each of the two separate subgroups. At the end, in order to get the overal amount of data exchanged we perform aggregation over the whole group.

7.7 Group-filter

- **Description** The `group-filter` operator takes a stream of group records as an input and copies to its output stream only those group records that match the filtering rules. The group records, which do not match the filtering rules are dropped. The `group-filter` operator performs *absolute* filtering on the flow record attributes or the aggregated flow record attributes contained within the group records. It compares the flow record attributes (aggregated flow record attributes) of the input group records with absolute values (or a range of absolute values). It can also compare various group record attributes within the same group record. It does not support *relative* filtering i.e it does not perform comparison between the flow record attributes (aggregated flow record attributes) of different incoming group records.
- **Definition**

```
group-filter <group-filter-id> {
<group-filter-statement> [OR <group-filter-statement>]...
<group-filter-statement> [OR <group-filter-statement>]...
...
...
<group-filter-statement> [OR <group-filter-statement>]...
}

<group-filter-attr> = <attribute> | <aggr-op>(<attribute>, [value])
<group-filter-statement> = <group-filter-attr> <operator> <value>           |
                           <group-filter-attr> <operator> <group-filter-attr>
```

The group filtering rules are written one rule per line and are connected by an implicit AND. Each group filtering rule may consist of several group filtering statements linked by a logical OR.

- **Pseudocode** The following pseudocode illustrates how a group filter G decides whether an incoming group record gr should be copied to the output or dropped.

```
evaluate_group_filter(G,gr) {
  output = gr;
  for each rule R in G {
    flag = false;
    for each grp-filter-statement S in R
      if match(S,gr)
        flag = true;
    if !flag {
      output = NULL;
      break; // one rule failed, don't check other rules
    }
  }
  return output;
}
```

- **Input** A stream of group records. Each group record consists of flow record attributes and aggregated flow record attributes. The flow record attributes and the aggregated flow record attributes used with-ing the filtering rules must be present in the incoming group records. That is these flow record attributes/aggregated flow record attributes must be present in the `aggregate` clause of the respective `grouper` operator.

- **Output** A stream of group records, consisting of the group records that match the filtering rules.

- **Usage Example** Filter group records, which refer to flow groups with

more than 20 flows, more than 1MB of traffic and destination IP address in `212.201.49.0/24`

```
group-filter foo {
count > 20
sum(bytes) > 1M
dstip in 212.201.49.0/24
}
```

This group filter assumes that the number of flows and bytes in a group as well as the destination IP address are contained in the attributes of the incoming group records i.e these attributes are present in the `aggregate` clause of the `grouper` operator.

7.8 Merger

- **Description** The `merger` operator takes several streams of group records as an input and copies to its output tuples of group records that satisfy some pre-defined merging rules. The merging rules are organized in merging rule modules. Each rule module specifies a set of input branches from all branches that meet at the `merger` and a number of rules, which use group record attributes to define certain relationships among the various flow groups. If there are N input branches as an input to a specific merging rule module, the output stream of that rule module will consist of N-tuples of group records, one group record per input branch. The output stream of one of the merging rule modules is the output of the whole `merger` operator. There is always exactly one rule module that produces the output stream for the `merger` operator and that rule module must be the first one defined in the `merger` definition.

  In most cases there will be only one merging rule module and the tuples of group records that it generates will produce the `merger` output stream. Using one merging rule module allows us to define the existence of certain patterns among the various flow groups. However, in order to be able to check for patterns that do not exist we will need more than one merging rule module. This feature is better illustrated through one of the usage examples presented. For example consider the following two queries:

  - *Query1* Find the flow records that correspond to a TCP connection between source IP address A and destination IP address B, port `ftp`, followed by a TCP connection between source IP address B, port `ftp-data` and destination IP address A.

– *Query2* Find the flow records that correspond to a TCP connection between source IP address A and destination IP address B, port `ftp`, followed by a TCP connection between source IP address B, port `ftp-data` and destination IP address A only if these two connections are not preceded by a TCP connection between source IP address A and destination IP address B, port *http*.

In more straightforward words, the first query aims at detecting a FTP file transfer between A and B, while the second query aims at detecting a FTP file transfer between A and B only if A did not already download the respective file using HTTP. While *Query1* is relatively easy to be defined using a single rule module, for *Query2* we will need a more complicated mechanism to check for the condition "A HTTP transfer did not already take place between these two entities".

In such a scenario we first perform the merging using the module rules that produce the `merger` output stream. Before copying the resulting tuples to the output stream, however, we feed them into the other merging modules and for each such tuple we check if the corresponding merging module would produce some output. These additional merging rule modules are used to define certain patterns that should not be observed and therefore a resulting tuple is only copied to the `merger` output stream if it does not generate any result when fed into any of the additional merging rules modules.

In general, the `merger` operator allows to perform grouping at a more general level compared to the `grouper` operator. The most powerful feature of the `merger` operator is its capability to express timing and concurrency constraints among various traffic groups by using Allen's interval algebra.

- **Pseudocode** The following pseudocode illustrates the function of a `merger` rule module.

```
get_module_output_stream(module m) {
(branch_1, branch_2, ..., branch_n) = get_input_branches(m);
for each g_1 in group_records(branch_1)
  for each g_2 in group_records(branch_2)
    ...
      ...
         for each g_n in group_records(branch_n)
            if match(g_1, g_2, ..., g_n, rules(m))
               output.add(g_1, g_2,..., g_n);

return output;
}
```

If there is only one rule module then its output stream is the `merger` output stream. However, if there are more merging rule modules then we need to apply the following algorithm:

```
get_merger_output_stream(module m_1, module m_2,..module m_k) {
merger_output_stream = 0;
m1_output_stream = get_module_output_stream(m_1);
B = get_input_branches(m_1);
for each tuple t in m1_output_stream {
   output = 0;
   for each rule module M in m_2 through m_k {
      S = get_input_branches(M);
      (b_1, b_2,..b_n) = S\B;  /*set difference*/
         for each g_1 in group_records(b_1)
           for each g_2 in group_records(b_2)
             ...
               ...
                 for each g_n in group_records(b_n)
                   if match(g_1, g_2, ..., g_n, t, rules(M))
                     output++;
   }
if (output == 0)
  add(merger_output_stream, t);
return merger_output_stream;
}
```

The complexity of this algorithm is exponential with the number of merging branches, but it is presented here for the matter of illustrating the function of the `merger` operator. Implementers can use some heuristics to optimize it based on the merging rules. Merging rules based on Allen's interval algebra are very beneficial in this aspect because they restrict the merging time window to a significant extent. We do not elaborate further on how to optimize a merger and make it more efficient from an algorithmical point of view since such optimizations are implementation specific.

- **Definition**

```
merger <merger-id> {
<merger-module-definition>
<merger-module-definition>
...
...
<merger-module-definition>
<export-statement>
}
```

```
<merger-module-definition> =
module <module-id> {
<branch-statement>
<relative-merge-statement>
<relative-merge-statement>
...
...
<relative-merge-statement>
<allen-merge-statement>
<allen-merge-statement>
...
...
<allen-merge-statement>
}

<branch-statement> = branches <branch-id>, <branch-id>, ...
<relative-merge-statement> = \
<branch-id>.<attr> <eq-op> <branch-id>.<attr> [<comp-op> <value>]
<attr> = <attribute> | <aggr-op>(<attribute>, [<value>])
<allen-merge-statement> = <branch-id> <allen-op> <branch-id>
<export-statement> = export <module-id> [if <condition>]
<condition> = (<module-id> = 0) AND (<module-id> = 0) ...
```

The `merger` definition contains a number of merging rule module definitions and an export statement. The merging `module` definitions consist of three types of statements (rules):

- `branch statement` - this statement states which branches provide the input for respective merging rule module. Branch names are global to the filtering framework and we describe how their names are specified in Section 7.10.
- `relative merge statements` - these statements express dependencies among the flow record attributes (aggregated flow record attributes) of the various group records. They do not compare the flow record attributes (aggregated flow record attributes) with absolute values but perform inter-group-record comparison.
- `Allen merge statements` - these statements define the timing and concurrency constraints among the various group records by using Allen's time interval algebra.

The export statement specifies the output stream of which rule module will produce the `merger` output stream and under what conditions a tuple from that particular rule module output stream will be sent to

the `merger` output stream. The conditions always require that the output of all additional merging rule modules is the empty set since these additional merging modules specify some patterns that should not be observed. The condition that the output of a merging module should be empty is specified by writing `<module-id> = 0`.

- **Input** - several streams of group records, one input stream of group records per input branch. The flow record attributes (aggregated flow record attributes) used in the definition of the mering rules must be present in the incoming group records from the respective branch (that is if `c.dstip` is present in the merging rules then `dstip` must be present in the group record attributes coming from branch c)
- **Output** - a stream of group record tuples, where the group records from the tuple satisfy the merging rules from the first merging module and produce no output when fed into the additional merging rule modules (if applicable); If there are N input branches as an input to the first merging rule module that meet at the `merger`, then the output stream will consist of group record N-tuples with the following constraints:
  - Each of the N input branches is represented by one group record in the N-tuple.
  - The group records in the N-tuple satisfy the merging rules specified in the first merging rule module of the `merger` definition.
  - The group records produce no output stream when fed into the additional merging rule modules.
- **Usage Example** The `merger` definition is very much dependent on the previous `filters`, `groupers` and `group-filters`, since they define the group records that arrive at the `merger`. Here is a sample `merger` definition where we want to express the following dependencies among the input group records from the four branches a, b, c and d
  - The destination port number of the group records from branch a is the same as the destination port number for the group records from branch c.
  - The source port number of the group records from branch b is the same as the source port number for the group records from branch d.
  - The destination IP address of the group records from branch a is the same as the source IP address of the group records from branch b, the destination IP address of the group records from

branch d and the source IP address of the group records from branch d.

- The source port of the group records from branch a differs by at most one from the source port number of the group records from branch c.
- Branch a overlaps in time with branch b.
- Branch c overlaps in time with branch d.

```
merger m {
  module m1 {
     branches a,b,c,d
     a.dstport = c.dstport
     b.srcport = d.srcport
     a.dstip = b.srcip
     a.dstip = c.dstip
     a.dstip = d.srcip
     a.srcport = c.srcport delta 1
     a o b
     c o d
  }
  export m1
}
```

This merger definition contains a single merging rule module and the output stream of that rule module constitutes the output of the merger. The following query illustrates the usage of a merger with more than one merging rule modules.

- *Query* Find the flow records that correspond to a TCP connection between source IP address A and destination IP address B, port `ftp`, followed by a TCP connection between source IP address B, port `ftp-data` and destination IP address A only if these two connections are not preceded by a TCP connection between source IP address A and destination IP address B, port *http*.
- Assume that branch a contains group records that correspond to the unidirectional TCP traffic between source IP address A and destination IP address B, port `ftp`. Each group record contains information about a single TCP connection.
- Assume that branch b contains group records that correspond to the unidirectional TCP traffic between source IP address B, port `ftp-data` and destination IP address A. Each group record contains information about a single TCP connection.

– Assume that branch c contains group records that correspond to the unidirectional TCP traffic between source IP address A and destination IP address B, port `http`. Each group record contains information about a single TCP connection.

The following contains the `merger` definition where branches a, b and c meet.

```
merger M {
   module m1 {
      branches a,b
      a.srcip = b.dstip
      a.dstip = b.srcip
      a < b OR a o b
   }
   module m2 {
      branches a,c
      a.srcip = c.srcip
      a.dstip = c.dstip
      c < a OR c o a
   }
   export m1 if m2 = 0
}
```

The first rule module ensures that the ftp control connection and the ftp data connection are established between the same network endpoints and enforces the condition that the control connection should take place before the data connection. The second rule module then takes the resulting pairs of group records and feeds them into the merging module $m2$. For each resulting group record from branch a, $m2$ checks if there exists a group record from branch c so that the merging rules (of $m2$) are satisfied. If $m2$ doesn't find a matching group record from branch c then the respective group record from branch a is sent to M's output stream (paired with the matching group record from b according to $m1$).

7.9 Ungrouper

- **Description** The `ungrouper` operator takes a stream of group record tuples as an input. For each group record tuple it expands the flow groups contained in the tuple using the labels and the flows stored in the temporary storage during the grouping phase of the `grouper` operator. Finally, for each group record tuple it outputs a separate stream of flow records. The flow records obtained from each group

record tuple are ordered by their timestamps and presented to the viewer in a capture order. Any flow record repetitions are eliminated, that is if a flow record is part of several flow groups within the group record tuple it is shown to the viewer only once. Each output stream is considered to be a result/match of the query operation performed by using the IP flow filtering framework. A query operation may return any number of results or no results at all and should clearly distinguish the flow records that belong to different results.

- **Definition**

  `ungrouper <ungrouper-id> {}`

- **Input** - a stream of group record tuples
- **Output** - several streams of flow records; one output stream of flow records per incoming group record tuple
- **Usage Example**

  `ungrouper foo {}`

7.10 Linking the Elements

The operators we described in Section 7.4 through Section 7.9 need to be linked via pipes to form the IP flow filtering framework shown in Figure 7.1. For this purpose we use the following rules:

```
<element-id> [branch <branch-id>]-> <element-id> [branch <branch-id>] ->  ...
```

An `element-id` is an identifier of an already defined operator. The optional `branch` part of the rule is used to specify the branch identifier that gets generated from a `splitter` operator. The input and output files/streams are specified with the following rules.

```
<input-file> -> <element-id>
<element-id> -> <output-file>
```

In order to distinguish file names from element identifiers the input and output file names should be surrouned by quotes. The rules below describe the model shown in Figure 7.2.

```
"trace.raw" -> S
S branch A -> X -> G1 -> GF1 -> M
S branch B -> Y -> G2 -> M
M -> U -> "result.raw"
```

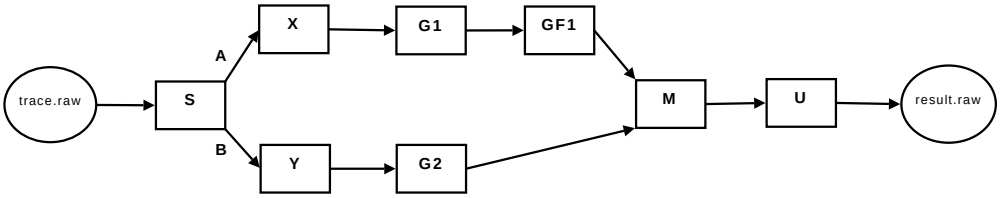

Figure 7.2: Sample Filtering Architecture

Chapter 8

Traffic Patterns

This section serves as an evaluation of our new IP flow record query language defined. We present a number of traffic patterns and define them using the primitives of the IP flow filtering framework defined in Section 7. Traffic patterns can be defined at the packet level as a sequence/combination of TCP/UDP IP packets or at the flow level as a sequence/combination of Cisco NetFlow flow records. In this context we will call the former *packet-based traffic patterns* and the latter *flow-based traffic patterns*. The flow-based traffic patterns may also be referred to as *flow fingerprints*. Simple traffic patterns can be easily expressed with the existent query languages described in Section 3. For example, email and web traffic can be easily identified by looking for flows with a destination port of 25 or 80 respectively. Therefore, we have selected a set of more complex traffic patterns as a requirement for our new flow record query language. Most of the patterns belong to applications where several flows are generated simultaneously or sequentially. Some of these flows carry specific source or destination port numbers. In some cases TCP and UDP flows are intermixed in a specific manner and the order uniquely identifies the traffic pattern of a specific application. We refer to the set of flows that a specific application generates as its flow fingerprint as defined above.

We have tried to compile a selection of common traffic patterns that belong to popular and well-known applications. However, in some of the examples presented we make certain assumptions and refer to special cases in order to be able to present the full capabilities of our flow record query language. For each traffic pattern in this section we present the packet-level pattern followed by its flow-level pattern (or flow fingerprint) and at the end we provide a definition using our new IP flow record query language. In order to get started the first two examples are quite simple, while the rest of the compilation refers to some more involved scenarios.

8.1 Web Traffic

While it is relatively easy to identify web traffic by looking for flow records with a source or destination port 80, it is a bit harder to match web traffic that belongs to a single application. In this example we try to describe the HTTP request response pattern of a browser. A Mozilla Firefox browser was used to retrieve the web page of Jacobs University Bremen (`http://www.jacobs-university.de`), which consists of an HTML part and a number of images. While downloading a web page some browsers may generate lots of extra traffic in addition to the packets carrying the contents of the web page. However in this example we only consider the traffic during the communication of a web client with a web server that belongs to the retrieval of the particular web page i.e we only look at the packets carrying the contents of the web page.

8.1.1 Packet-level Pattern

The packet level data exchange is shown in Figure 8.1. In the figure all packets that belong to a single flow are represented by an arrow.

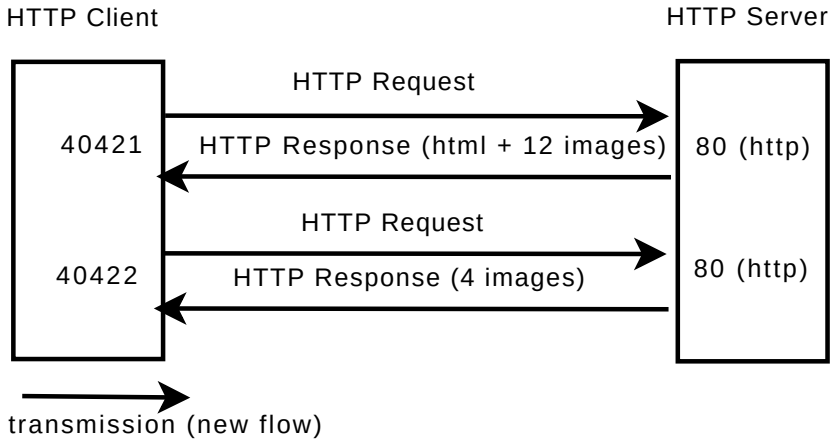

Figure 8.1: Packet Level Breakdown of a Simple HTTP Transfer

Two different TCP connections are used for transferring the contents of the web page. The first TCP connection is used for retrieving the HTML part and a number of images. The second TCP connection is started in the middle of the first HTTP transfer and is used to transfer images only. We are not interested in investigating why the web page retrieval was split over two different TCP connections and why one of the connections carried 12 images and the other 4 images. We rather take this as granted and try to describe this pattern using our flow record quesry language.

8.1.2 Flow-level Pattern

The flow level breakdown of the same HTTP transfer is shown in Figure 8.2.

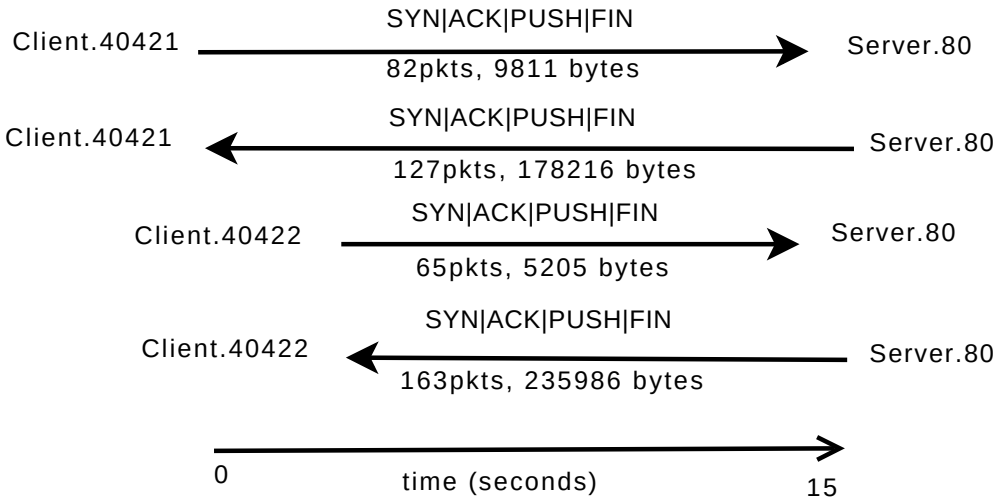

Figure 8.2: Flow Level Breakdown of a Simple HTTP Transfer

According to the NetFlow specifications each TCP connection consists of two different flows (one flow in each direction). The flow representing the second TCP connection starts slightly later (200ms) after the flow representing the first TCP connection. Both TCP connections are closed by the server at the same time (difference was in the range of 10 microseconds) and therefore the response flows end slightly after the request flows (again with a difference in the range of 20 microseconds). In Figure 8.2 the linearity of the time scale is not respected and the time difference between the start/end times of the flows may seem bigger. We have chosen this approach in order to illustrate the order in which the various flows start and terminate. Another feature that can be noticed in the flow diagram is that the flows from the client to the server carry much less traffic compared to the flows from the server to the client. This is as expected because the packets that originate from the HTTP client only carry the HTTP requests and the TCP acknowledgements while the packets that originate from the HTTP server carry the actual data. Therefore, we conclude that the following flow fingerprint may belong to a Mozilla Firefox browser which is downloading the web page of Jacobs University:

- A number of TCP flows with a source or destination port 80.
- The flows can be split into two distinct groups - one flow group rep-

resents the traffic for the HTTP requests and the other flow group represents the traffic for the HTTP responses.

- The transport endpoints of the two flow groups match, i.e the source IP address of the first flow group is the same as the destination IP address for the second flow group and vice verca. The same relationship holds for the port numbers.
- The flows in first group have consecutive source port numbers and a destination port number 80. The flows in the second group have a source port number 80 and consecutive destination port numbers. This pattern of consecutive port numers was observed after repeating the same measurement several times and it maybe specific to the specific machine or browser. We are not interested in justifying what causes the pattern but rather to describe it using our IP flow record query language.
- The flows representing the HTTP requests (first flow group) carry much less data compared to the flows representing the HTTP responses (second flow group).

8.1.3 Query Language Specification

The following is the specification of the retrieval of the Jacobs University web page using a Mozilla Firefox browser within our IP flow filtering framework. Two branches are defined - one for HTTP requests and one for HTTP responses. The specification is illustrated in Figure 8.3.

```
splitter S {}

filter f_www_req {
   dstport = 80
   proto = TCP
}

grouper g_www_req {
   module g1 {
      srcip = srcip
      dstip = dstip
      srcport = srcport relative-delta 1
      stime = stime relative-delta 500ms
   }
   aggregate srcip, dstip, union(srcport), sum(bytes) as bytes,
             min(stime) as stime, max(etime) as etime
}
```

```
filter f_www_res {
   srcport = 80
   proto = TCP
}

grouper g_www_res {
   module g1 {
      srcip = srcip
      dstip = dstip
      dstport = dstport relative-delta 1
      stime = stime relative-delta 500ms
   }
   aggregate srcip, dstip, union(dstport), sum(bytes) as bytes,
             min(stime) as stime, max(etime) as etime
}

merger M {
   module m1 {
      branches A,B
      A.srcip = B.dstip
      A.dstip = B.srcip
      A.union(srcport) = B.union(dstport)
      A.bytes << B.bytes
      A o B OR A f B
   }
   export m1
}

ungrouper U {}

"input" -> S
S branch A -> f_www_req -> g_www_res -> M
S branch B -> f_www_res -> g_www_res -> M
M -> U -> "output"
```

- The first branch detects the HTTP requests from the web client to the web server. It consists of a filter and a grouper. The filter `f_www_req` picks out only the flow records, which belong to TCP connections and have a destination port number 80 (well-known port number used for HTTP traffic). The grouper `g_www_req`, which contains a single group module, then partitions the flow records into groups, which have the same source and destination IP address. Another requirement is that all flow records within a group should have consecutive source port

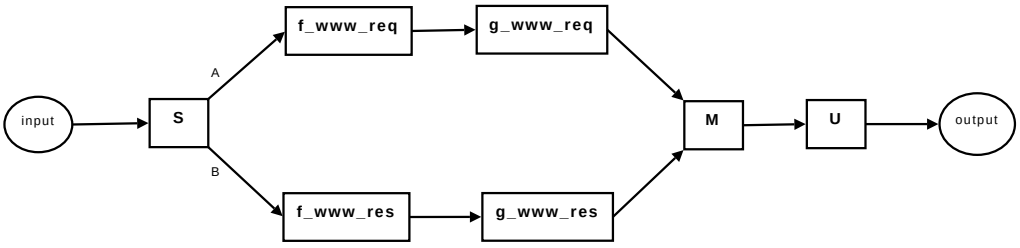

Figure 8.3: Capturing Web Page Retrieval with the IP Flow Filtering Framework

numbers (as specified in the netflow fingerprint). Last but not least, there might be a maximum of 500ms delay between the start times of the flow records in a single group, because during our observation we discovered that the second HTTP connection with the web server is not started at the same time as the first one. For each resulting group the grouper creates a group record, which consists of the source and destination IP address, the set of source port numbers, the start and end time of the HTTP requests and the amount of data transferred. These group records are then passed to the merger `M`.

- The second branch identifies the HTTP response traffic in an identical manner to the first branch. The filter `f_www_res` picks out the flow records that have a source port number 80/TCP. The grouper `g_www_res` partitions the flow records into groups, which have the same source and destination IP address. However, in this case the requirement is that flow records within the same group should have consecutive destination port numbers (instead of source port numbers). The group records are created by extracting the same attributes from each group as in the first branch (again the only difference is that we take the set of destination port numbers instead of the set of source port numbers).

- The merger `M` consists of a single merging rule module, which generates its output stream. The rules inside the module $m1$ specify the merging conditions for the two already defined branches `A` and `B`. The first three rules specify that the transport endpoints of the web client and the web server should stay the same within the same HTTP transfer i.e the source IP address during the requests should be the same as the destination IP address during the responses and vice versa. Furthermore, the set of port numbers used to originate the HTTP requests should be the same as the set of port numbers at which the responses are received. The next rule imposes the requirement that the data carried by the HTTP reuests should be much less compared to the data contained within the responses. In this concrete case we can as-

sume that the `<<` operator implies `10 times less` as we can observe from the flow diagram. The last rule specifies that the response traffic should end at about the same time as the request traffic. This should hold because any of the TCP connections used within the same HTTP transfer contributes to both the request and response group of flow records. Therefore, the tear-down of the last TCP connection will coincide with the end times of a flow record from each of the two groups. In practice we observed that the server starts the TCP tear-down procedure for both TCP connections and thus the server was the last one to send packets that belong to the response flows. For this reason we can assume a small delay between the termination of the request flow group and the termination of the response flow group, thus allowing overlap in the group records (in the sense of the Allen's interval algebra).

- The last part of our definition consists of defining the ungrouper and linking the already defined elements using pipes to build a model for our IP flow filtering framework.

8.2 FTP Download

An FTP session usually consists of a control connection and several data connections. The FTP server usually runs the control connection on port 21 and the data connection on a random port above 1024. In rare cases the data connection is run on the port 20, which is reserved for FTP data. The data connection is established after the control connection and is technically initiated by the FTP server. In some cases the FTP client is located behind a firewall or a NAT, which does not allow the opening of a TCP connection by the server. In such cases the data connection is initiated by the FTP client and the server port used is negotiated in the control connection. In this example we try to describe the FTP download pattern of a big file. An FTP client was used to retrieve a 750MB file from the FTP server at `ftp.eecs.jacobs-university.de`. In our example the FTP server was able to initiate a data connection to the client.

8.2.1 Packet-level Pattern

Figure 8.4 shows the packet level breakdown of an FTP transfer. In the figure all packets that belong to a single flow are represented by an arrow.

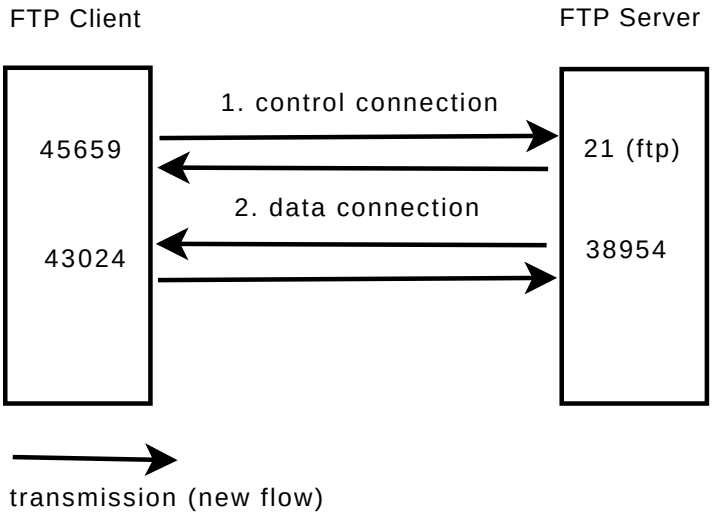

Figure 8.4: Packet Level Breakdown of a Simple FTP Transfer

8.2.2 Flow-level Pattern

Figure 8.5 shows the flow level breakdown of the FTP transfer.

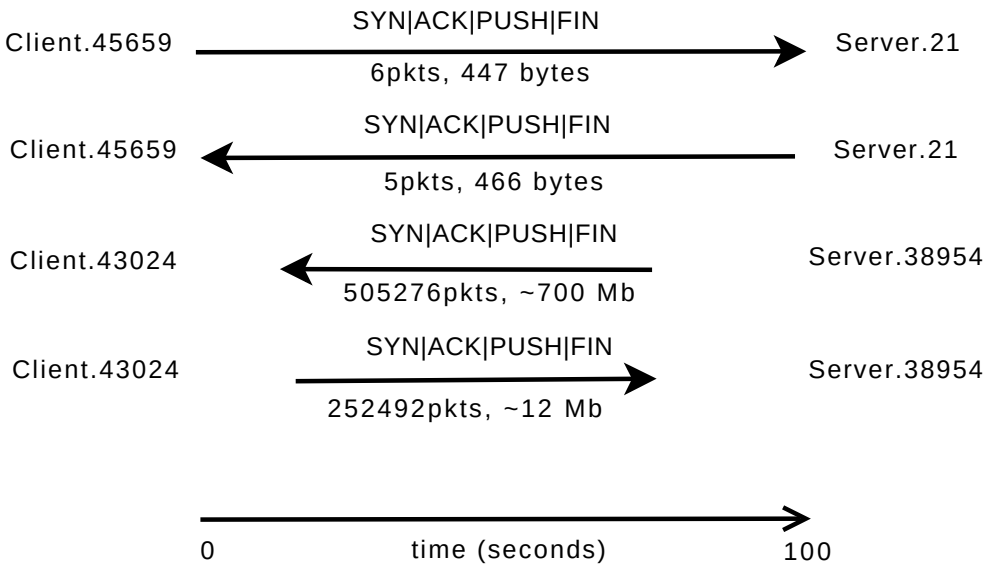

Figure 8.5: Flow Level Breakdown of a Simple FTP Transfer

From this example we determine the netflow fingerprint of an FTP download:

- Two pairs of TCP flows that are very close in time.
- The first pair of flows represents the FTP control connection. Therefore, one of the flows in this pair has a destination port of 21 and the other has a source port of 21.
- The second pair of flows represents the FTP data connection. It is usually initiated by the FTP server and thus the flow from the FTP server towards the FTP client should slightly precede the flow from the FTP client towards the FTP server. Furthermore, the data connection should carry much more data than the control connection.
- The pair of flows representing the FTP control connection should start before the pair of flows representing the data connection.

8.2.3 Query Language Specification

The following is the specification of FTP download of a big file within our IP flow filtering framework. Two branches are defined - one for the FTP control connection and one for the FTP data connection. The specification is illustrated in Figure 8.6.

```
splitter S {}
```

```
filter f_control {
   srcport = 21 OR dstport = 21
   proto = TCP
}

grouper g_group_tcp {
   module g1 {
      srcip = dstip
      dstip = srcip
      srcport = dstport
      dstport = srcport
      stime = stime relative-delta 500ms
   }
   module g2 {
      srcip = srcip
      dstip = dstip
      srcport = srcport
      dstport = dstport
      stime = stime relative-delta 500ms
   }
   aggregate g1.srcip as srcip, g1.dstip as dstip,
             g1.srcport as srcport, sum(bytes) as bytes,
             min(stime) as stime, max(etime) as etime
}

filter f_data {
   proto = TCP
}

group-filter gf_data {
   bytes > 700MB
}

merger M {
   module m1 {
      branches A,B
      A.srcip = B.dstip
      A.dstip = B.srcip
      A.bytes << B.bytes
      B d A
   }
   export m1
}
```

```
ungrouper U {}

"input" -> S
S branch A -> f_control -> g_group_tcp -> M
S branch B -> f_data -> g_group_tcp -> gf_data -> M
M -> U -> "output"
```

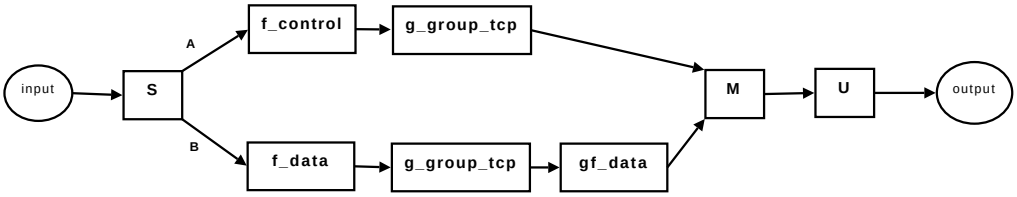

Figure 8.6: Capturing FTP download of Big Files with the IP Flow Filtering Framework

- The first branch detects the traffic corresponding to the FTP control connection. The filter `f_control` picks out only the flow records that belong to TCP connections where either the source or destination port number is 21 (`ftp`). The grouper `g_control` then partitions the resulting flow records into groups where the flow records within a single group belong to the same TCP connection. The group module $g2$ adds to the flow group the flow records that have the same source and destination transport endpoints as the flow record that has generated the group. The group module $g1$ adds to the group the flow records that correspond to the traffic in the opposite direction i.e it requires that the destination transport endpoint of the flow records to be added is the same as the source transport endpoint of the flow record that generated the group and vice versa (the source transport endpoint is the same as the destination transport endpoint of the flow record that generated the group). At the end for each group (i.e for each TCP connection) a group record is created, which contains the transport endpoints for each TCP connection (extracted from the flow records that belong to $g1$), the start and end time as well as the amount of data exchanged (taken by performing aggregation over all flow records in the group).

- The second branch aims at detecting the traffic corresponding to the FTP data connection. The filter `f_data` is a relaxed version of `f_control`. It picks out flow records, which belong to TCP connections. We cannot narrow the flow records in a data connection to any port number since usually the FTP server and the FTP client use random source and destination port numbers. Therefore, a lot of flow records will successfully traverse `f_data`. In this branch we use the same grouper as in the first branch since our goal is to partition the flow records

into groups such that the flow records in each group correspond to the same TCP (FTP data) connection. Finally, the group filter only allows group records with more than 700MB of traffic to reach the merger since we are interested at detecting FTP download of big files (where our custom definition for big is more than 700MB).

- The merger `M` contains a single merging rule module, which specifies the merging conditions for the two already defined branches `A` and `B`. The first two rules of $m1$ specify that the network endpoints of the FTP client and the FTP server in the control connection should be the same as those in the data connection. Note that the requirement states that the source IP address of the first branch should be the same as the destination IP address of the second branch and vice versa. This is because we aim at capturing FTP transfers where the server originates the FTP data connection (so we need to swap source and destination in the control and data connection). The third rule in the merger specification imposes the requirement that the traffic exchanged in the data connection should be much bigger compared to the traffic exchanged in the control connection. Finally, we use the Alen's interval algebra statement to specify that the data connection should be completely contained within the control connection.
- The last part of our definition consists of defining the ungrouper and linking the already defined elements using pipes to build a model for our IP flow filtering framework.

8.3 W32/Blaster worm

In this section we present the traffic pattern generated by a computer infected with the Blaster.A worm. A [38] is a recent Internet worm, which exploits the Microsoft Windows Remote Procedure Call DCOM vulnerability. Upon successful execution, the worm attempts to retrieve a copy of the file `msblast.exe` from the compromising host. Once this file is retrieved, the compromised system then runs it and begins scanning for other vulnerable systems to compromise in the same manner. In [49] Dübendorfer et. al. describe the various stages of the Blaster worm infection and perform an in-depth packet and flow level analysis.

8.3.1 Packet-Level Pattern

Figure 8.7 shows the packet-level breakdown of a Blaster attack performed by an already infected host. The characteristic network activity (an infected attacker trying to infect other hosts on the network) associated with such an attack consists of the following steps:

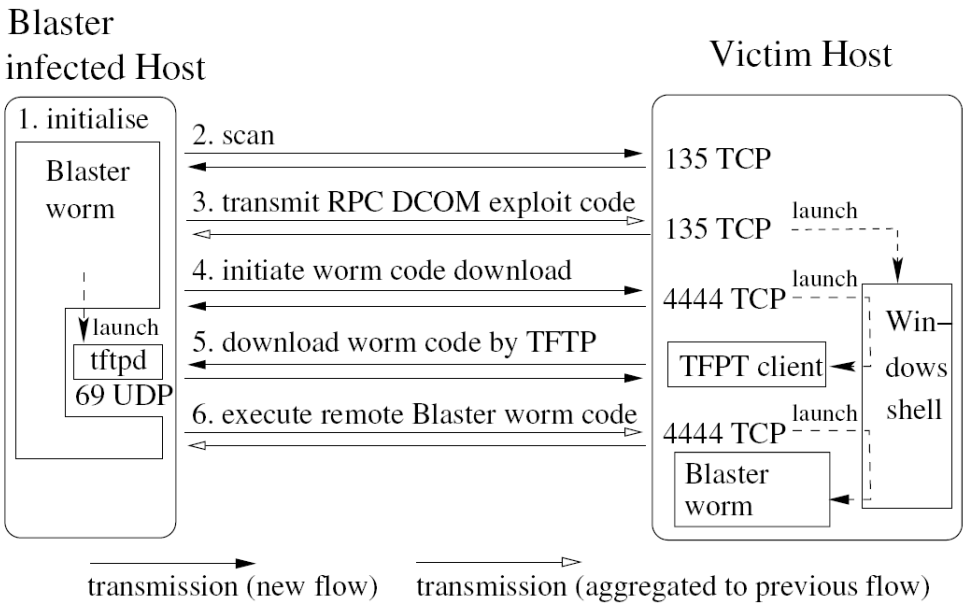

Figure 8.7: Packet Level Breakdown of a Blaster Infection

1. Initialize
2. **Victim Scanning on Port 135/TCP** - The attacker performs a scan on port 135/TCP of 20 sequential IP addresses at a time.

3. **Transmission of RPC Exploit Code** - If a TCP connection to destination port 135 can be opened, the exploit code is sent to the victim.
4. **Initiation of Worm Code Download** - Blaster initiates a TCP connection to port 4444/TCP and the command `tftp -i attacker-IP GET msblast.exe` is executed to start a Trivial File Transfer Protocol (TFTP) download of `msblast.exe` from the Blaster-infected host.
5. **Download of Worm Code by TFTP** - The Blaster-infected host is contacted on port 69/UDP for a download of the worm code.
6. **Blaster Worm Code Execution** - The Blaster-infected machine stops its TFTP daemon after the transmission of the worm code or after 20 seconds of TFTP inactivity. In case of success, it sends a command to start `msblast.exe` on the already open TCP connection to port 4444 of the victim. Now, the victim is running Blaster and starts to infect other machines.

8.3.2 Flow-Level Pattern

The flow-level breakdown of the Blaster attack is shown on Figure 8.8.

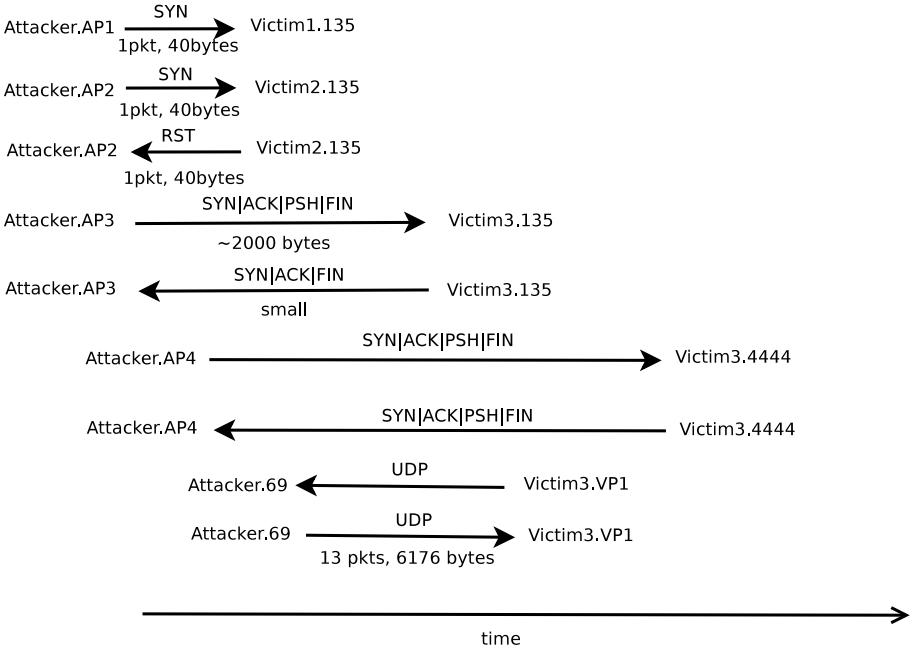

Figure 8.8: Flow Level Breakdown of a Blaster Infection

Therefore, the netflow fingerprint of a Blaster infection consists of the following sequence of flows (order is important):

- More than 20 flows originating from the attacker directed to port 135 of different hosts. These flows are small since they only carry a SYN packet. This represents the scanning activity of the attacker. Some of these scans may trigger a reverse flow consisting of RST packets.
- In a successful attack there will be a pair of bigger flows (longer and carrying more data) to and from port 135/TCP of the victim
- The pair of flows representing the TCP connection to port 135 of the victim is followed by a pair of flows representing the TCP connection to port 4444.
- The next step is a pair of flows to and from port 69/UDP of the attacker representing the TFTP transfer of `msblast.exe`. The flow to the attacker slightly precedes the flow from the attacker since the connection is initiated by the infected host. These two flows end before the flows representing the TCP connection on port 4444 from the previous step

Furthermore, Table 8.1 shows how the different stages of a Blaster infection can be identified based on the presence or absence of the respective flows defined above.

Stage	135/TCP	135/TCP	4444/TCP	4444/TCP	69/UDP	69/UDP
	A → V	A ← V	A → V	A ← V	A → V	A ← V
2	X	-	-	-	-	-
3	X	X	-	-	-	-
4	X	X	X	X	-	-
5	X	X	X	X	X	-
6	X	X	X	X	X	X

Table 8.1: Query languages used by network traffic analysis tools

8.3.3 Query Language Specification

In order to describe a Blaster worm infection in our IP flow filtering framework we use the definitions below. We define one branch for each Blaster stage as specified in the tflow fingerprint.

```
splitter S{}

filter f_scan {
   dstport = 135
```

```
   proto = tcp
}

grouper g_scan {
   module g1 {
      srcip = srcip
      dstip = dstip relative-delta 1
      stime = stime relative-delta 5ms
      stime = stime absolute-delta 5s
   }
   aggregate srcip, union(dstip), min(stime) as stime,
             max(etime) as etime, count
}

group-filter gf_scan {
   count > 20
}

filter f_victim {
   srcport = 135 OR dstport = 135
   proto = TCP
}

grouper g_group_tcp {
   module g1 {
      srcip = dstip
      dstip = srcip
      srcport = dstport
      dstport = srcport
      stime = stime relative-delta 5ms
   }
   module g2 {
      srcip = srcip
      dstip = dstip
      srcport = srcport
      dstport = dstport
      stime = stime relative-delta 5ms
   }
   aggregate g1.srcip as srcip, g1.dstip as dstip,
             min(stime) as stime, max(etime) as etime
}

filter f_control {
   srcport = 4444 OR dstport = 4444
```

```
   proto = tcp
}

filter f_tftp {
   srcport = 69 OR dstport = 69
   proto = udp
}

grouper g_tftp {
   module g1 {
      srcip = dstip
      dstip = srcip
      srcport = dstport
      dstport = srcport
      stime = stime relative-delta 5ms
   }
   module g2 {
      srcip = srcip
      dstip = dstip
      srcport = srcport
      dstport = dstport
      stime = stime relative-delta 5ms
   }
   aggregate g1.srcip as srcip, g1.dstip as dstip,
             min(stime) as stime, max(etime) as etime,
             g2.sum(bytes) as bytes
}

group-filter gf_tftp {
   bytes > 6K
}

merger M {
   module m1 {
      branches A,B,C,D
      A.srcip = B.srcip
      A.srcip = C.srcip
      A.srcip = D.dstip
      B.dstip = C.dstip
      B.dstip = D.srcip
      B.dstip in union(A.dstip)
      A < B OR A m B OR A o B
      B o C
      D d C
```

```
   }
   export m1
}

ungrouper U{}

input -> S
S branch A -> f_scan -> g_scan -> gf_scan -> M
S branch B -> f_victim -> g_group_tcp -> M
S branch C -> f_control -> g_group_tcp -> M
S branch D -> f_tftp -> g_tftp -> gf_tftp -> M
M -> U -> output
```

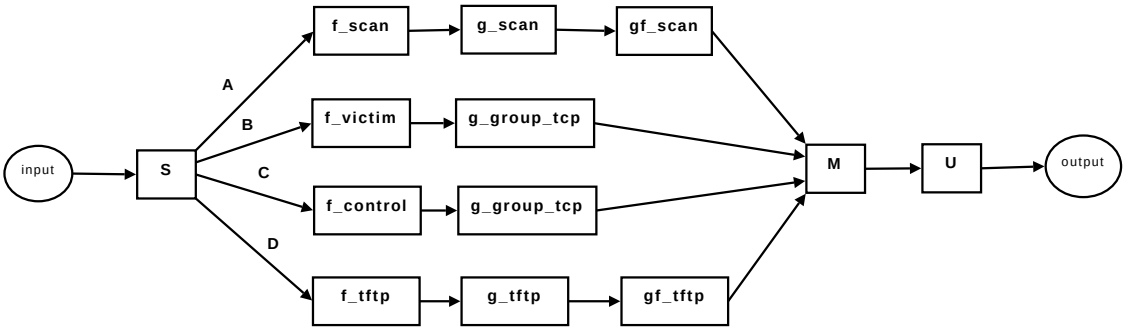

Figure 8.9: Capturing Blaster Worm Infections with the IP Flow Filtering Framework

- The first branch detects the scanning activity performed by the attacker. Initially, the `f_scan` filter picks out the flow records that have a destination port 135/TCP. Then the grouper `g_scan`, which consists of a single group module $g1$, partitions the flow records into groups, which have the same source IP address and the destination IP addresses consist of a block of subsequent IP addresses. Additional constraints are put so that we consider that a scanning attack lasts at most 5s and each scanning attack should not be interrupted for more than 5ms (since attackers usually generate the small SYN packets at a very high rate). Finally, for each group the grouper operator retains the source IP address, the set of destination IP addresses, the start and end time of the attack as well as the number of flows in the group. Each such flow group now contains information about the scan attack performed by a single host. The newly created group records are passed to the group-filter operator, which filters only these group records that refer to flow groups containing more than 20 flows. That is, we consider the traffic from a flow group a scanning attack only if the attacker has scanned more than 20 hosts. If the attacker has

scanned less than 20 hosts we consider the flow group normal traffic activity and drop it.

- The second branch, which consists of the filter `f_victim` and the grouper `g_group_tcp` aims at capturing the TCP connection that the attacker established with the victim i.e a TCP connection between the attacker and port 135 of the victim. The filter picks out flows with a source or destination port 135. The grouper `g_group_tcp` then aggregates all flow records that correspond to the same TCP connection in one group. The group module $g2$ adds to the flow group all flow records that have the same source and destination transport endpoints as the flow record that generated the flow group. The group module $g1$ adds the flow records that correspond to the opposite direction of the TCP connection i.e it adds those flow records for which the destination transport endpoint is the same as the source transport endpoint of the flow record that generated the group and vice versa (the source transport endpoint is the same as the destination transport endpoint of the flow record that generated the group). For each TCP connection then `g_group_tcp` retains the source and destination IP addresses as well as the start and end times for each group.

- The third branch aims at identifying the control connection between the attacker and port 4444/TCP of the victim. The filter `f_control` picks out the flow records with a source or destination port number 4444/TCP and the grouper `g_group_tcp` partitions them into groups as already explained.

- The last branch aims at capturing the TFTP download, which gets initiated by the victim host. The filter `f_tftp` picks out the flow records, which belong to UDP traffic to or from port 69 (tftp). Then the grouper `g_tftp` is very much like `g_group_tcp` in terms of the partitioning that it is performing. Both groupers contain two group modules, which aim at detecting the forward and backward direction of each TCPconnection /UDP transfer. `g_group_tcp` partitions the incoming stream of TCP flow records into groups so that each group corresponds to a separate TCP connection and `g_tftp` partitions the incoming stream of UDP flow records into groups so that each group corresponds to a separate TFTP/UDP transfer (to the extent to which we are able to distinguish different UDP/TFTP transfers by using the `delta` and `relative-delta` parameters). In general the grouper is not aware of what the incoming stream of flow records contains, thus it is not aware if it is grouping TCP or UDP flow records. Therefore, `g_group_tcp` can already do the job of splitting the incoming flow records into groups where each group represents a separate UDP/TFTP transfer. In this case however, we are also interested to retain

the amount of data exchanged in each UDP/TFTP transfer in order to do some filtering later in the `group-filter`. Therefore, the specification and the interpretation of `g_tftp` is the same as the one of `g_group_tcp` i.e the group module $g2$ adds to the flow group all flow records that belong to the forward direction of the UDP/TFTP transfer (as compared to the first flow records that generated the group) and the group module $g1$ adds all flow records that correspond to the backward direction of the UDP/TFTP transfer. The only addition of `g_tftp` as compared to `g_group_tcp` is that the former also retains the amount of data exchanged within each group (that is within each TFTP transfer).The resulting group records are then passed to a group-filter which retains only these group records, which represent a TFTP exchange of at least 6K since the netflow fingerprint of this stage specifies that the virus is approx. 6176 bytes.

- The next step consists of defining the merger M and the merging conditions. M contains a single merging rule module $m1$, which takes the four already defined branches A, B, C and D as an input. The first three rules from the merging rule module definition specify that the source IP address of the attacker should be the same during the different stages of the Blaster infection (since we want to retrieve a single attack from a single attacker host to a single victim host). The next three rules specify that the IP address of the victim should also stay the same and that the victim should be one of the scanned hosts from the first stage of the Blaster infection. The last three rules express the time and concurrency constraints among the four branches using Allen's time interval algebra. We assume that the scanning phase takes place completely `before`, `meets` or `overlaps` with the stage of successful TCP connection establishment on port 135 with the victim. Furthermore, the connection on port 135/TCP `overlaps` with the control connection on port 444/TCP. Finally from Figure 8.8 one notices that the TFTP transfer happens `during` the control connection (i.e the TFTP transfer is entirely contained within the control connection).

- The last part of our definition consists of defining the ungrouper and linking the already defined elements using pipes to build a model for our IP flow filtering framework.

8.4 Welchia/Nachi Worm

The Nachi worm [50], also known as Welchia worm, was an attempt to use a worm against a worm infection. Nachi exploited the same vulnerability as the original Blaster worm, namely a DCOM RPC vulnerability on port 135/TCP of hosts running Windows XP. In addition, Nachi also exploited a vulnerability in WebDAV on port 80/TCP found in Microsoft IIS 5.0. Therefore, the flow fingerprint for Blaster specified in Section 8.3 can be used to a large extent for identifying Nachi. In [51] however, Dübendorfer et. al. propose another approach for identifying a Nachi/Welchia infection.

8.4.1 Packet-Level Pattern

The worm uses an ICMP echo request to determine whether a specific IP address is in use and this causes massive ICMP activity. A characteristic pattern in the ICMP activity of a Nachi infected host is that while it scans a huge number of IP addresses it does not scan IP addresses which contain the octet 197.

8.4.2 Flow-Level Pattern

We can derive the netflow fingerprint that corresponds to the described packet-level pattern.

- Presence of a big number of flows carrying ICMP echo requests, originating from the same source IP address
- The flows are directed to a sequential IP address space in which addresses containing the octet 197 are not scanned. This means that there are no ICMP flows directed to IP addresses of the form 197.x.x.x, x.197.x.x, x.x.197.x and x.x.x.197

8.4.3 Query Language Specification

The following is the definition of a Nachi worm infection expressed in our IP flow filtering framework. This example demonstrates the flexibility in terms of linking the various components of our framework. Unlike the previous examples, the splitter here is only used after the grouper. Each branch then consists of a group filter. The definition is further illustrated in Figure 8.10.

```
filter f_icmp {
   proto = icmp
}
```

```
grouper g_seq {
   module g1 {
      srcip = srcip
      dstip = dstip relative-delta 1
      stime = stime relative-delta 0.5ms
      stime = stime absolute-delta 30s
   }
   aggregate srcip, union(dstip) as dstip, min(stime) as stime,
             max(etime) as etime, count
}

splitter S {}

group-filter gf_octet4_max {
   count > 10
   mask(max(dstip), 0.0.0.255) = 196
}

group-filter gf_octet4_min {
   count > 10
   mask(min(dstip), 0.0.0.255) = 198
}

group-filter gf_octet3_max {
   count > 10
   mask(max(dstip), 0.0.255.0) = 196
}

group-filter gf_octet3_min {
   count > 10
   mask(min(dstip), 0.0.255.0) = 198
}

group-filter gf_octet2_max {
   count > 10
   mask(max(dstip), 0.255.0.0) = 196
}

group-filter gf_octet2_min {
   count > 10
   mask(min(dstip), 0.255.0.0) = 198
}
```

```
group-filter gf_octet1_max {
   count > 10
   mask(max(dstip), 255.0.0.0) = 196
}

group-filter gf_octet1_min {
   count > 10
   mask(min(dstip), 255.0.0.0) = 198
}

merger M1 {
   module m1 {
      branches A,B
      A.srcip = B.srcip
      B < A
   }
   export m1
}

ungrouper U1 {}

merger M2 {
   module m2 {
      branches C,D
      C.srcip = D.srcip
      C.mask(min(dstip), 255.0.0.0) = D.mask(max(dstip), 255.0.0.0)
      D < C
   }
   export m2
}

ungrouper U2 {}

merger M3 {
   module m3 {
      branches E,F
      E.srcip = F.srcip
      E.mask(min(dstip), 255.255.0.0) = F.mask(max(dstip), 255.255.0.0)
      F < E
   }
   export m3
}

ungrouper U3 {}
```

```
merger M4 {
   module m4 {
      branches G,H
      G.srcip = H.srcip
      G.mask(min(dstip), 255.255.255.0) = H.mask(max(dstip), 255.255.255.0)
      H < G
   }
   export m4
}

ungrouper U4 {}

"input" -> f_icmp -> g_seq -> S
S branch A -> gf_octet1_min -> M1
S branch B -> gf_octet1_max -> M1
M1 -> U1 -> "output1"
S branch C -> gf_octet2_min -> M2
S branch D -> gf_octet2_max -> M2
M2 -> U2 -> "output2"
S branch E -> gf_octet3_min -> M3
S branch F -> gf_octet3_max -> M3
M3 -> U3 -> "output3"
S branch G -> gf_octet4_min -> M4
S branch H -> gf_octet4_max -> M4
M4 -> U4 -> "output4"
```

- The input stream of flow records is directly fed into the filter `f_icmp`, which picks out only the flow records that correspond to ICMP traffic. The filtered stream is then passed to the grouper `g_seq` and gets partitioned into groups in such a way that each group contains the flow records corresponding to the scanning activity originating from a single attacker. In order to achieve this goal, the definition of the group module in the grouper states that the source IP address stays the same within a group and that a new flow record gets added to the group only if its destination IP address differs from the destination IP address of the last added flow record by at most 1. This way we ensure that the set of destination IP addresses forms a sequential IP address space. In addition, we add some time constraints in the specification of the grouper so that there should not be a pause of more than 0.5ms among the start times of the flow records in a group. Last but not least, a group should not continue for more than 30 seconds. For each group of flow records the grouper creates a group record containing the source IP address (the attacker), the set of destination IP addresses

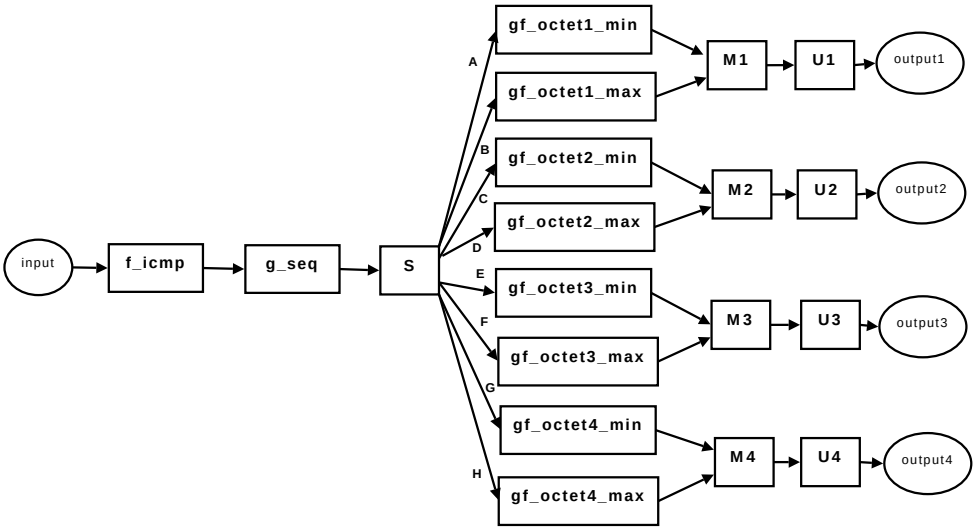

Figure 8.10: Capturing Nachi Worm Infections with the IP Flow Filtering Framework

(the victims), the start and end times of the group (that is the start of the first flow record and the end of the last one) and the number of flow records inside.

- The resulting group records created by the `g_seq` grouper are passed to the splitter S, which replicates the stream 8 times. The grouper has already ensured that each group record corresponds to a group of flow records with a sequential destination IP address space. The next goal would be to check whether this sequential IP address space *terminates* just before an IP address containing 197 as one of its octets or *starts* right after an IP address containing 197 as one of its octets. We perform these two checks for each of the four octets of the destination IP address field - for each of the four octets we check if its value for the biggest destination IP address in the sequential space is 196 or if its value for the the smallest destination IP address in the sequential destination IP address space is 198. This is how we end up with eight separate branches.

  For example, the group filter `gf_octet4_max` picks out only those group records where the biggest value of the fourth octet of the sequential destination IP address space is 196. The group filter `gf_octet4_min`, on the other hand, picks out those group records for which the smallest value of the fourth octet of the sequential destination IP address space is 198. Then these two branches are merged at `M4` and if a group record from the first branch occurs just before a group record from the second branch and both group records have the same source IP address (i.e

the same attacker) we may well assume that these two group records belong to a Nachi worm attack, in which the attacker has just skipped to scan the IP address with a fourth octet of 197. Thus we can pass these 2 particular group records to the ungrouper, where the respective flow records get extracted and given as one of the result streams.

- As described above, the stream of group records coming from the `g_seq` grouper is replicated 8 times and generates 8 branches. Each branch contains a group filter with the following purpose:
  - `gf_octet1_max` picks out group records that correspond to a sequential destination IP address space with the biggest IP address having a 1st octet of 196
  - `gf_octet1_min` picks out group records that correspond to a sequential destination IP address space with the smallest IP address having a 1st octet of 198
  - `gf_octet2_max` picks out group records that correspond to a sequential destination IP address space with the biggest IP address having a 2nd octet of 196
  - `gf_octet2_min` picks out group records that correspond to a sequential destination IP address space with the smallest IP address having a 2nd octet of 198
  - `gf_octet3_max` picks out group records that correspond to a sequential destination IP address space with the biggest IP address having a 3rd octet of 196
  - `gf_octet3_min` picks out group records that correspond to a sequential destination IP address space with the smallest IP address having a 3rd octet of 198
  - `gf_octet4_max` picks out group records that correspond to a sequential destination IP address space with the biggest IP address having a 4th octet of 196
  - `gf_octet4_min` picks out group records that correspond to a sequential destination IP address space with the smallest IP address having a 4th octet of 198
- The resulting streams of group records for each octet are merged at `M1`, `M2`, `M3` and `M4` respectively to form a pair of group records that correspond to an ICMP scanning activity that has just skipped an IP address with a value of 197 for the respective octet. Several merging requirements are defined in the merging rule module:

- The attacker should be the same for both group records that we are merging.
- The group record for the destination IP addresses ending at 196 should happen before the group record for the destination IP addresses starting from 198 (according to the notion of Allen's time interval algebra)
- The prefix for the destination IP address space should stay the same. For example, in the case of `M4` we will only merge a group record corresponding to a destination IP address space ending at `a.b.c.196` with a group record corresponding to a destination IP address space starting at `a.b.c.198` i.e the first three octets should stay the same to ensure that the only skipped IP address in this case is `a.b.c.197`.

8.5 STUN

Simple Traversal of UDP Through Network Address Translators (NATs) (STUN)[52] is a lightweight protocol that allows applications to discover the presence and types of NATs and firewalls between them and the public Internet. It also provides the ability for applications to determine the public IP addresses allocated to them by the NAT. In order to discover the type of NAT/firewall on the way a STUN client carries out a communication with a STUN server on its two available public IP addresses and two different port numbers. The STUN client sends messages requesting the STUN server to respond from a different IP address or a different port number. Depending on which response messages are received the STUN client is able to figure out behind which type of NAT and firewall it is located. The complete description of how the protocol works as well as the definition of the various NAT types is contained in RFC3489 [52].

We have chosen to analyze the STUN flow fingerprint since we believe that STUN is widely used by many VoIP applications today and identification of STUN activity in the network might indicate the use of a specific VoIP application. Furthermore, the traffic pattern of STUN allows us to show the capabilities of our new flow query language.

8.5.1 Packet-Level Pattern

Figure 8.11 shows the communication of a STUN client located behind a port-restricted NAT with a STUN server. As an external host, the STUN server can send a packet, with source IP address X and source port P, to the STUN client, which is an internal host, only if the STUN client has previously sent a packet to IP address X and port P. Figure 8.11 shows which response messages are received by the STUN client and which not due to the limitation of the NAT.

8.5.2 Flow-Level Pattern

The flow-level breakdown of the same communication is shown in Figure 8.12. For the flow diagram we assume that the NetFlow data has been collected on the internal network. Therefore, the flows from the public Internet that do not pass through the NAT/firewall are not visible to the NetFlow collector. It will be very beneficial if the NAT/firewall device can collect and export netflow records as well since then we would also have access to flows that come from the external Internet and cannot go through the NAT. This will add more precision to our netflow fingerprints. However,

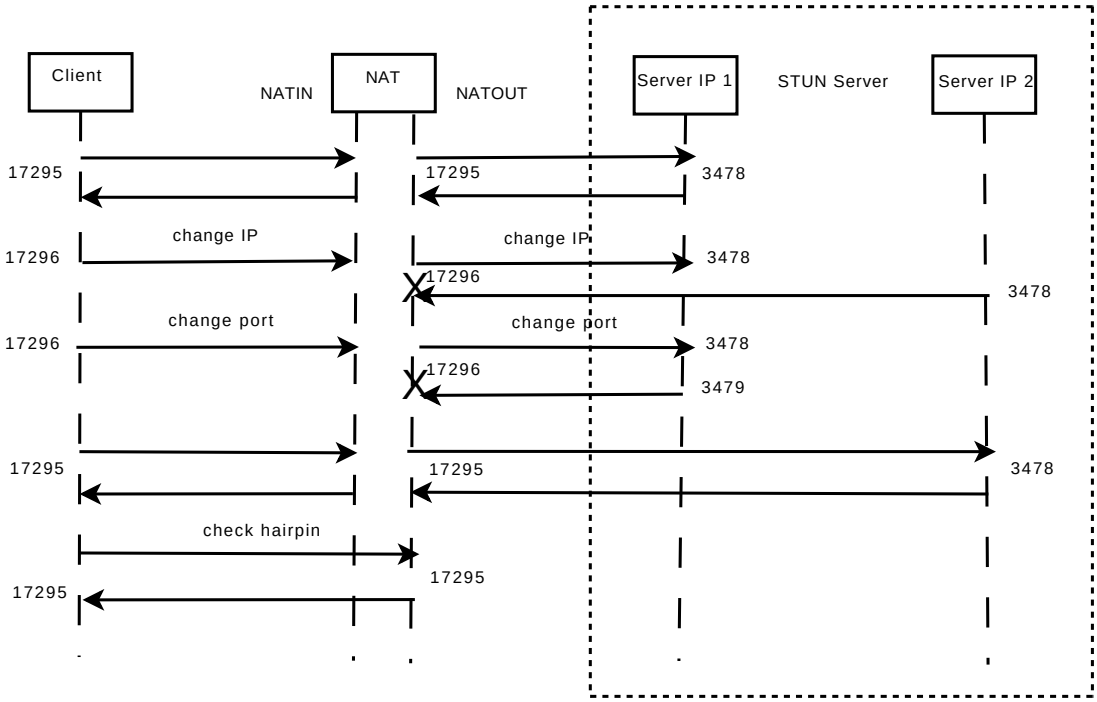

Figure 8.11: Packet Level Breakdown of a STUN Activity

in the general case the device running the NAT does not collect netflow records and thus we assume that the netflow data is only collected by routers on the inside of the network.

From the two figures we can conclude that STUN activity can be inferred by the presence or absence of the response flows coming from the STUN server. In the case of a port-restricted NAT it is obvious that response flows are only present when the request flows are also present. In this case it is also notable that the number of request messages is much bigger than the number of response messages, which is an indication that some of the responses are not received. Finally, all the flows in the STUN communication have transport endpoints that belong to the following groups:

- The Client IP address and one of the 2 client ports used by the STUN server. In many cases the port numbers are consecutive.
- The 2 public IP addresses of the server and one of the two ports used by the server. A STUN server usually uses ports 3478 and 3479.

The last two messages in the communication (the internal IP as a source and the public IP of the NAT as a destination) constitute a test if the NAT supports hairpin (described in RFC3489 [52]). Based on the above observations the following flow fingerprint that belongs to a STUN activity

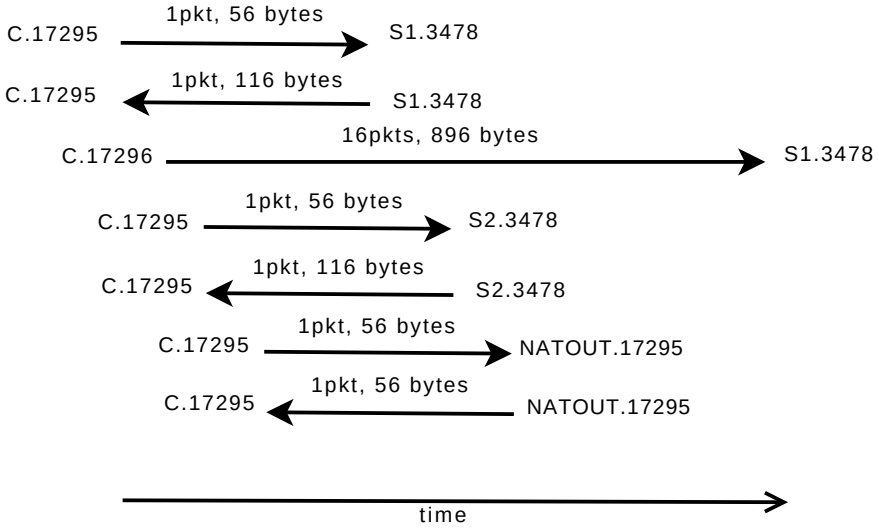

Figure 8.12: Flow Level Breakdown of a STUN Activity

behind a port-restricted NAT can be inferred:

- A set of flows that connects the following 2 groups
  - An internal IP address with 2 different (probably consecutive) port numbers.
  - Two public IPs with 2 different port numbers (probably 3478 and 3479).
- A response flow (i.e a public IP address X and port A as a source and a private IP address Y and port B as a destination) is present only if the request flow (the private IP address Y and port B as a source and the public IP address X and port A as destination) is present.
- The request flows carry more packets than the response flows (thus indicating missing response flows).

8.5.3 Query Language Specification

The following definition illustrates how the IP flow filtering framework can be used to detect STUN activity in the presence of a port restricted NAT. This example demonstrates the usage of more than one rule module in the merger definition in order to define some "missing" traffic patterns. The definition is further illustrated in Figure 8.13.

```
splitter S {}
```

```
filter f_stun {
   proto = udp
   srcport = 3478 OR srcport = 3479  \
   OR dstport = 3478 OR dstport = 3479
}

grouper g_udp {
   module g1 {
      srcip = dstip
      dstip = srcip
      srcport = dstport
      dstport = srcport
      stime = stime absolute-delta 5s
   }
   module g2 {
      srcip = srcip
      dstip = dstip
      srcport = srcport
      dstport = dstport
      stime = stime absolute-delta 5s
   }
   aggregate g1.srcip as srcip, g1.dstip as dstip,
             g1.srcport as srcport, g1.dstport as dstport,
             sum(packets) as packets, min(stime) as stime,
             max(etime) as etime
}

group-filter gf_udp_stun {
   packets = 2
   srcip in 192.168.0.0/16
   dstip notin 192.168.0.0/16
}

filter f_hairpin {
   proto = udp
   srcport = dstport
}

grouper g_udp_unidir {
   module g1 {
      srcip = srcip
      dstip = dstip
      srcport = srcport
      dstport = dstport
```

```
      stime = stime absolute-delta 5s
   }
   aggregate srcip, dstip, srcport, dstport,
             min(stime) as stime, max(etime) as etime
}

group-filter gf_udp_out {
   srcip in 192.168.0.0/16
   dstip notin 192.168.0.0/16
}

group-filter gf_udp_in {
   srcip notin 192.168.0.0/16
   dstip in 192.168.0.0/16
}

merger M {
   module m1 {
      branches A,B,C,D
      A.srcip = B.srcip
      A.srcip = C.srcip
      A.srcip = D.srcip
      A.dstip = D.dstip
      A.dstport = B.dstport
      A.dstport = D.dstport
      A.srcport = C.srcport
      A.srcport = B.srcport
      A.srcport = D.srcport delta 1
      A < D OR A o D
      B d D
      C d D
      B < C OR B o C
   }

   module m2 {
      branches E,D
      E.srcip = D.dstip
      E.dstip = D.srcip
      E.srcport = D.dstport
      E.dstport = D.srcport
      E < D OR E m D OR E o D
   }

   export m1 if m2=0
```

```
}

ungrouper U {}

input -> S
S branch A -> f_stun -> g_udp -> gf_udp_stun -> M
S branch B -> f_stun -> g_udp -> gf_udp_stun -> M
S branch C -> f_hairpin -> g_udp -> gf_udp_stun -> M
S branch D -> f_stun -> g_udp_unidir -> gf_udp_out -> M
S branch E -> f_stun -> g_udp_unidir -> gf_udp_in -> M
M -> U -> output
```

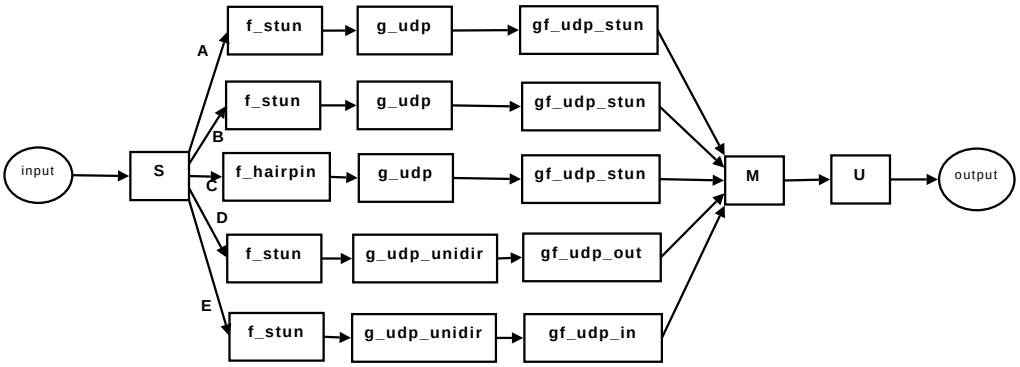

Figure 8.13: Capturing STUN traffic with the IP Flow Filtering Framework

- The first two branches (A and B) are identical and aim at detecting the UDP communication between the STUN client (located on the Intranet) and the two different IP addresses of the STUN server located on the Internet. As seen from Figure 8.12 each of these two UDP exchanges contains 2 packets, one request and one response packet. Both branches consist of a filter `f_stun`, a grouper `g_udp` and a group filter `gf_udp_stun`. The filter picks out those flow records which carry UDP traffic and have the well-known STUN port numbers (3478 and 3479) as either source or destination ports. The grouper `g_udp` then partitions the resulting stream of flow records into groups in such a way that each group contains the flow records that belong to the UDP communication between the same transport endpoints. As seen from previous examples such a grouper consists of two group modules, one group module matches the UDP traffic in the forward direction (compared to the flow record that generated the group) and one group module matches the UDP traffic in the opposite direction. In our example $g2$ matches the flow records in the forward direction compared to the first flow record that generated the flow group and $g1$ matches the flow records in the backward direction. Both group modules use a

rule which contains `relative-delta` in order to distinguish between different UDP exchanges between the same transport endpoints. That is if there is a pause of more than 5 seconds in the communication between the same transport endpoints we consider that these are two separate UDP exchanges and create one group for each of them. The grouper creates group records, which contain the source and destination IP addresses and port numbers (taken from the flow records that belong to $g1$) and the start and end times of the UDP exchange (by taking the min start time and max end time across the whole flow group). In addition, the group records also carry the number of packets contained in one UDP exchange. The group records are then passed to the group filter `gf_udp_stun`, which picks out only those flow groups consisting of two IP packets. This requirement is directly taken from Figure 8.12 where we observed that each of the two UDP exchanges that we want to capture with this branch consist of one request packet (56 bytes) and one response packet (116 bytes). In addition, the group filter retains only groups which have an IP address on the Intranet (private IP address space) as a source and a public IP address as a destination. We have considered only the class C private address space for a matter of simplicity.

- Branch C is very similar to A and B. It aims at detecting the hairpin test that a STUN client performs. This consists of a UDP exchange between the private IP address and port number that belongs to the STUN client and its corresponding public IP address and port assigned by the NAT. We observed in Figure 8.12 that the source and destination port numbers are the same and therefore the filter `f_hairpin` picks out UDP flow records which have identical values for a source and destination port number. The resulting flow records are then passed to the `g_udp` grouper, which partitions them into groups so that each groups contain the flow records that belong to a single UDP exchange between the same transport endpoints. The created group records are then fed into the group filter, which only allows groups that consist of two packets (one request and one response packet) to reach the merger. In addition, the group filter enforces that the hairpin test is performed between a host on the private network and a host on the Internet.

- Branch D aims at detecting the unidirectional UDP traffic between the STUN client and one of the hosts that belong to the STUN server as seen on Figure 8.12. Therefore, this branch starts with a filter, which allows only UDP flow records to pass (`f_stun`). It is then followed by the grouper `g_udp_unidir`, which partitions the flow records into groups so that each group contains a distinct unidirectional UDP exchange between two transport endpoints. Unlike the grouper `g_udp`, which creates one group for each bidirectional UDP exchange, `g_udp_unidir`

ensures that only flow records that belong to the same direction of a UDP exchange between the same transport endpoints are contained in the same group. `g_udp_unidir` creates the group records in the same way as `g_udp`, it retains the source and destination IP address and port numbers, the start and end times of the exchange as well as the number of packets contained in the exchange. The group records are then fed into the group filter `gf_udp_out`, which picks out only those groups with a source IP address in the private address space and a destination IP address in the public address space to reach the merger. Branch E has the same function as branch D with the only difference that the group records produced by the grouper are fed into a group filter that allows only groups with a source IP on the public Internet and a destination IP address on the private network to pass. Branch E will be used in the merger definition to enforce the condition that an incoming UDP flow record (source on the public Internet and destination on the private Intranet) can only be observed if it is preceded by a flow record between the same transport endpoints but in the opposite direction (source on the private Intranet and destination on the public Internet).

- The merger M consists of two merging rule modules $m1$ and $m2$. The output stream of M cosists of those group record tuples formed by the rules in $m1$ that produce an empty set as a result when merged with the branches specified in $m2$. $m1$ ensures certain dependencies between the group records from the four input branches A, B, C and D.
  - Branch A and B should have the same source transport endpoints (rules 1 and 8), that is the two UDP exchanges should originate from the same STUN client. Furthermore, the destination port numbers for the group records from A and B should have the same destination port number as observed on Figure 8.12. The last one is enforced by rule 5.
  - The hairpin test should originate from the same STUN client as in branches A, B and D. This is enforced by rules 2 and 7.
  - The unidirectional UDP exchange (branch D) should originate from the same IP address as in A, B and C and should be directed to the same IP address and port number as in A. This is enforced by rules 3, 4 and 6. In addition, this unidirectional UDP exchange has usually a source port number that differs from the source port number in A by one. This condition is enforced by rule 9.
  - The last four rules of $m1$ enforce the timing and concurrency constraints among the group records from the four branches. Follow-

ing Figure 8.12 we observe that the first UDP exchange (branch A) precedes or overlaps with the unidirectional UDP exchange branch D. In addition, the second UDP exchange and the hairpin test (branches B and C) are completely contained within the unidirectional UDP exchange (branch D). Finally, the second UDP exchange (branch B) precedes or overlaps with the hairpin test (branch C).

Once $m1$ has produced a set of resulting group record tuples the rules from the rule module $m2$ are taken into consideration. $m2$ takes its input from branches D and E. However branch D was also used during the merging in rule module $m1$ and therefore we only consider the group records from branch D that belong to the set of resulting group record tuples produced by $m1$. The rules in rule module $m2$ state that the source transport endpoint of the group record from branch D should be the same as the destination transport endpoint of the group record from E and vice versa. In addition, the group record from E should start before the group record from D. Each group record from branch D corresponds to an unidirectional UDP exchange originating from the local Intranet in the direction of the public Internet and each group record from branch E corresponds to an unidirectional UDP exchange in the opposite direction. Since a port-restricted NAT is present on the network path each group record from E must be preceded by a group record from D between the same transport endpoints. In order to enforce this condition for each group record from branch D we try to find a match (according to the rules in $m2$) among all group records from E. If for a particular group record from D we are able to find a matching group record from E, then there is a UDP exchange directed to the private Intranet that was originated from the public Internet and this violates the definition of a port-restricted NAT. In such a case the group record tuple with the "bad" group record from branch D is dropped and is not copied to the output stream of M. If for a particular group record from branch D we cannot find a matching group record from branch E then the respective UDP exchange was originated from the private Intranet and we copy the respective group record tuple to the output stream of the merger.

8.6 Skype

Skype is a popular peer-to-peer VoIP client developed by KaZaa in 2003. Skype encrypts all communications end-to-end and its protocol is not available to the public. In [53] Baset et. al. analyze the various functions of Skype such as login, call establishment, conferencing etc. from a network point of view and present a network traffic analysis at the packet level. While the paper provides detailed scenarios for login, call establishment and conferencing when the Skype Client (SC) is located behind various types of NAT we have chosen to discuss an unsuccessful Skype login attempt in our compilation of traffic patterns. The reason is that this allows us to fully explore the requirements of the flow record query language.

Skype is an overlay peer-to-peer network, which consists of ordinary hosts and super nodes (SN). An ordinary host is a Skype client that can be used to place voice calls and send text messages. A super node is an ordinary hosts end-point on the Skype network. A Skype client usually caches the IP addresses and port numbers of a number of super nodes in a Host Cache (HC).

8.6.1 Packet-Level Pattern

During login a Skype Client establishes a UDP exchange with at least one SN. If the UDP exchange fails the Skype client tries to establish a TCP connection with the same SN. If the TCP connection establishment on the cached port fails then Skype tries to connect to port 80 and port 443 in order to bypass a restrictive firewall. In [53] Baset et. al. claim that an unsuccessful Skype login attempt can be identified by a sequence of unsuccessful attempts to open a connection with a SN at the following ports:

- random port/UDP
- random port/TCP (same random port as in the UDP exchange attempt)
- 80/TCP
- 443/TCP

This sequence is repeated five times before login failure occurs. Each of the unsuccessful TCP connection establishments may trigger a RST TCP packet in the opposite direction. Furthermore, each time a Skype Client is started a TCP connection to port 80 with `www.skype.com` is established in order to check for an update. The packet level exchange caused by an unsuccessful Skype login attempt is shown in Figure 8.14.

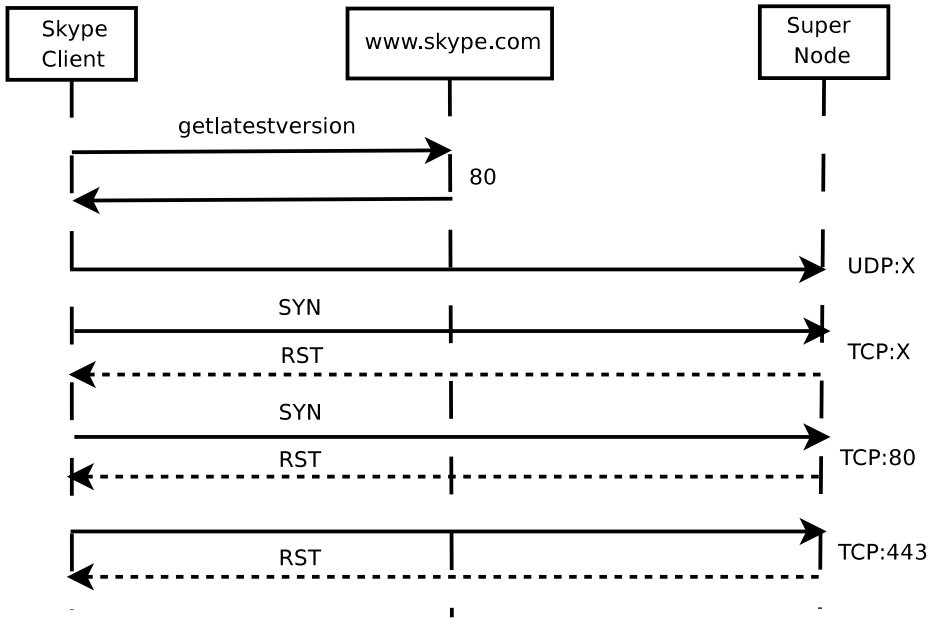

Figure 8.14: Packet Level Breakdown of an Unsuccessful Skype Login Attempt

8.6.2 Flow-Level Pattern

The corresponding flow-level breakdown is shown in Figure 8.15.

Based on our observations we identify the following flow fingerprint of an unsuccessful Skype login attempt, which consists of the following sequence of flows (order may be important):

- A pair of flows directed to/from port 80 of `www.skype.com`. These represent the TCP connection, which is used to check for Skype updates.
- An unidirectional UDP flow with a destination IP of a SN and a port X. The correspondent reverse (response) UDP flow with the source IP of the SN and port X will be missing. The unidirectional flow will probably consist of 5 packets since 5 attempts for UDP exchange are made.
- An unidirectional TCP flow with a destination IP of a SN and the same port X, carrying SYN packets only. The flow will probably consist of 5 packets since 5 connection attempts are made. The correspondent reverse (response) TCP flow with the source IP of the SN and port X will be missing or there will be a reverse flow carrying RST packets only.

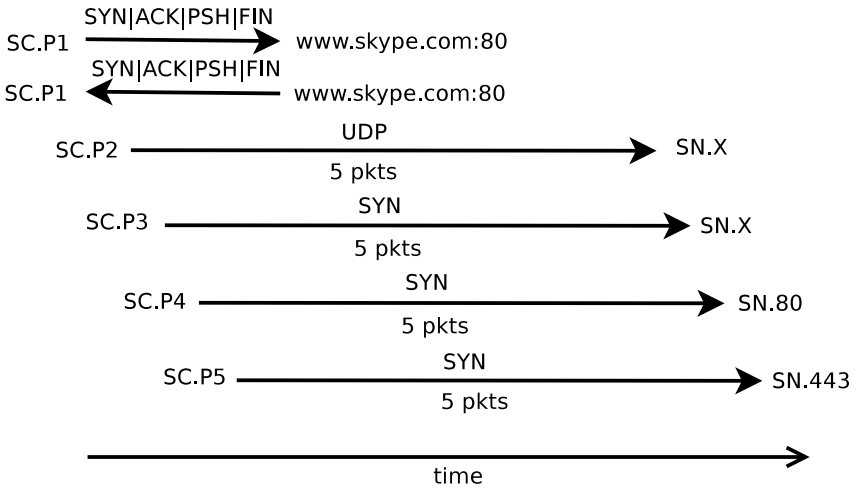

Figure 8.15: Flow Level Breakdown of an Unsuccessful Skype Login Attempt

- An unidirectional TCP flow with a destination IP of a SN and port 80, carrying SYN packets only. The flow will probably consist of 5 packets since 5 connection attempts are made. The correspondent reverse (response) TCP flow with the source IP of the SN and port 80 will be missing or there will be a reverse flow carrying RST packets only.
- An unidirectional TCP flow with a destination IP of a SN and port 443, carrying SYN packets only. The flow will probably consist of 5 packets since 5 connection attempts are made. The correspondent reverse (response) TCP flow with the source IP of the SN and port 443 will be missing or there will be a reverse flow carrying RST packets only.

8.6.3 Query Language Specification

The following definition illustrates how the IP flow filtering framework can be used to detect an unsuccessful Skype login attempt. The definition is further illustrated in Figure 8.16.

```
filter f_http {
   proto = tcp
   srcport = 80 OR dstport = 80
   srcip = www.skype.com OR dstip = www.skype.com
}

grouper g_http {
```

```
   module g1 {
      srcip = dstip
      dstip = srcip
      srcport = dstport
      dstport = srcport
      stime = stime absolute-delta 3s
   }
   module g2 {
      srcip = srcip
      dstip = dstip
      srcport = srcport
      dstport = dstport
      stime = stime absolute-delta 3s
   }
   aggregate g1.srcip as srcip, g1.dstip as dstip,
             min(stime) as stime, max(etime) as etime
}

filter f_udp {
   proto = udp
}

grouper g_unidir {
   module g1 {
      srcip = srcip
      dstip = dstip
      srcport = srcport
      dstport = dstport
      stime = stime absolute-delta 3s
   }
   aggregate srcip, dstip, srcport, dstport,
             sum(packets) as packets,
             min(stime) as stime, max(etime) as etime
}

group-filter gf_5pkts {
   packets = 5
}

filter f_skype_tcp_dst_80 {
   proto = tcp
   dstport = 80
   flags = S
}
```

```
filter f_skype_tcp_src_80 {
   proto = tcp
   srcport = 80
   flags = S
}

filter f_skype_tcp_dst_443 {
   proto = tcp
   dstport = 443
   flags = S
}

filter f_skype_tcp_src_443 {
   proto = tcp
   srcport = 443
   flags = S
}

merger M {
   module m1 {
      branches A,B,D,F
      A.srcip = B.srcip
      A.srcip = D.srcip
      A.srcip = F.srcip
      B.dstip = D.dstip
      B.dstip = F.dstip
      A o B
      B o D
      D o F
   }

   module m2 {
      branches B,C
      B.srcip = C.dstip
      B.srcport = C.dstport
      B.dstip = C.srcip
      B.dstport = C.srcport
      B o C
   }

   module m3 {
      branches D,E
      D.srcip = E.dstip
```

```
      D.dstip = E.srcip
      D.srcport = E.dstport
      D o E
   }

   module m4 {
      branches F,G
      F.srcip = G.dstip
      F.dstip = G.srcip
      F.srcport = G.dstport
      F o G
   }

   export m1 if m2=0 AND m3=0 AND m4=0
}

ungrouper U {}

input -> S
S branch A -> f_http -> g_http -> M
S branch B -> f_udp -> g_unidir -> gf_5pkts -> M
S branch C -> f_udp -> g_unidir -> M
S branch D -> f_skype_tcp_dst_80 -> g_unidir -> gf_5pkts -> M
S branch E -> f_skype_tcp_src_80 -> g_unidir -> M
S branch F -> f_skype_tcp_dst_443 -> g_unidir -> gf_5pkts -> M
S branch G -> f_skype_tcp_src_443 -> g_unidir -> M
M -> U -> output
```

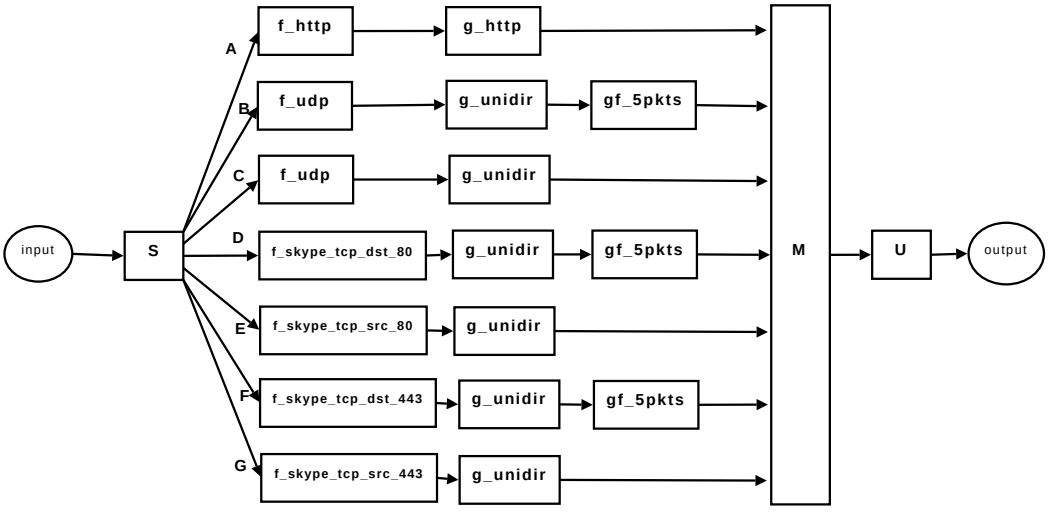

Figure 8.16: Capturing Skype traffic with the IP Flow Filtering Framework

- The first branch detects the flow records that belong to the HTTP connection with the server at `www.skype.com`. The filter `f_http` picks out only the flow records that belong to a TCP connection with either source or destination IP address `www.skype.com`, port 80. The grouper `g_http` then partitions the flow records into groups so that each group contains the flow records that belong to a single TCP connection between two transport endpoints. As described in previous examples this is easily achieved with two group modules where each group module is responsible for detecting flow records belonging to either the forward or backward direction of the TCP connection.

- The second branch (branch B) detects the unidirectional UDP exchange between the Skype client and one of the super nodes. As seen on Figure 8.15 this exchange consists of 5 packets. Initially all flow records are fed into a flow filter, which picks out only those flow records that belong to UDP traffic. The resulting stream of flow records is then passed to the grouper `g_unidir`, which partitions them into groups so that each group contains the flow records from a single unidirectional UDP exchange between two transport endpoints. The grouper consists of a single group module, which specifies that a flow record can be added to a group only if it has the same source and destination IP address and the same source and destination port number as the flow record that generated the group. For each group the grouper retains the source and destination IP addresses, the source and destination port numbers, the start and end times of the UDP exchange as well as the number of packets contained in that exchange. The resulting stream of group records is then passed to the group filter `gf_5pkts`, which only retains the group records with 5 packets. The latter is imposed by the observation from Figure 8.15 that the unidirectional UDP exchange between the Skype client and the server consists of exactly 5 packets. Branch C is identical to branch B, it consists of the same filter and grouper but does not contain a group filter. It aims at detecting all unidirectional UDP exchanges and its group records will be used at the merger for imposing the requirement that for all unidirectional UDP exchanges from branch B there should exist no response UDP exchanges i.e there should be no response from the super node (since we are considering a failed login attempt).

- Branches D and E are very similar in function to branches B and C. Branch D detects the attempt of the Skype client to establish a TCP connection with port 80 of the super node. As seen on Figure 8.15 the Skype client makes 5 attempts to establish such a connection i.e it sends 5 SYN packets. Therefore, we initially filter all flow records and retain only the flow records which are destined to port 80/TCP and have only the SYN flag set. The resulting stream is then passed to

the same grouper as in the UDP case, `g_unidir`, which partitions the stream into groups. Each group contains flow records that correspond to the unidirectional TCP communication between two transport endpoints. The newly created group records are again fed into the group filter `gf_5pkts`, which retains only the groups with 5 packets (for the same reason as in the UDP branch). Branch E, similarly to branch C, matches all unidirectional TCP communications and will be used at the merger to impose the requirement that for each unidirectional TCP communication from branch D there should be no response i.e there should be no reverse TCP flow from the super node.

- Branches F and G are identical to D and E, the only difference is that we try to match attempts to establish a TCP connection with port 443/TCP on the super node instead of port 80/TCP.
- The merger M contains four merging rule modules. $m1$ contains rules, which describe the relationships between the group records from branches A, B, D and F and provides the output stream of the merger, while $m2$, $m3$ and $m4$ contain rules to impose the conditions that there should be no response flow records corresponding to the attempts of the Skype client to establish a UDP exchange/TCP connection on port 80 and 443 with the super node.
  - The first five rules of $m1$ state that the Skype client (the source IP of the group records) should stay the same during the update with the Skype server and the attempts to establish a UDP or TCP connections with a super node. Furthermore, the rules impose the condition that all attempts for a UDP and TCP exchange are directed to the same super node i.e the super node should not change during the connection establishment attempt. Finally, the Allen's time interval rules impose a strict order on how the Skype client communicates with the update server and the super node. It first makes an update with the server at `www.skype.com`. Then it tries to establish a UDP exchange with the super node. If unsuccessful, the client makes an attempt to establish a TCP connection with port 80 of the super node. The last resort is to open a TCP connection with port 443 of the super node.
  - The merging rule module $m2$ takes the resulting stream of group record tuples and for each group record that comes from branch B checks if there is a matching group record from branch C. According to the merging rules of $m2$ if for a particular group record from branch B there exists a matching group record from branch C then there is evidence of a bidirectional UDP exchange between the Skype client and the super node. If this occurs the corresponding tuple of group records containing the "bad" group

record coming from branch B is dropped and not copied to the output stream of the merger.

- The merging rules of module $m3$ perform the same function as the ones in $m2$, however the attempt here is to detect a response from the super node to the attempt of the Skype client to establish a TCP connection with port 80. For each group record that comes from branch D (from the set of group record tuples produced by $m1$) $m3$ checks if there is a matching group record from branch E. According to the merging rules of $m3$ if for a particular group record from branch D there exists a matching group record from branch E then there is evidence of a bidirectional TCP exchange between the Skype client and the super node i.e there is a reponse from the super node to the attempt of the Skype client to establish a TCP connection with port 80. If this occurs the corresponding tuple of group records containing the "bad" group record coming from branch E is dropped and not copied to the output stream of the merger.
- The merging rules of module $m4$ are identical to those of module $m3$. However, here the input is taken from branches F and G and we try to detect a response from the super node to the attempt of the Skype client to establish a TCP connection with port 443.

The flow record tuples produced by $m1$ and not dropped by any of $m2$, $m3$ or $m4$ are copied to the output stream of the merger M.

Chapter 9

Conclusion and Future Work

Mention some words from the survey here first.....
During the first part of this report we conducted a survey of the existent query languages and network analysis tools for flow data. Some of the tools that we explored use SQL/stream languages to query flow traces, however this leads to poor query performance when flows are stored in a RDBMS. Another group of query languages are the filtering languages, which are easy to use but lack a time and concurrency dimension. The last group of query languages that we analyzed are the procedural languages such as scripts. The latter are very powerful in network flow analysis but they not trivial to understand for users.

After analyzing the pros and cons of the existent filtering and query languages we developed the design of an IP flow filtering framework, which contains its own language primitives. These primitives were chosen in such a way that the new IP flow record query language has the capability to describe aggregation and comparison based on the flow record attributes. In addition various dependencies such as flow correlation, timing and concurrency constraints, flow ordering and causal relationships can be expressed. The IP flow filtering framework has a limited number of operators, which can be defined and linked in a very flexible manner. This makes it relatively straightforward to use for the definition and detection of various traffic patterns.
In order to evaluate our new IP flow record query language we collected a set of traffic patterns that belong to some popular network applications and services. We analyzed HTTP and FTP transfers, the propagation of some well-known worms as well as Skype traffic. The flow fingerprint of each traffic pattern was derived and written down using the IP flow record query language. The definition of some rather complex traffic patterns in Section 8 allowed us to demonstrate the features of our new design.

The next step in this project will be to implement a prototype that understands the new IP flow record query language. It should consist of a parser, which reads the pattern definition and an engine, which implements the functions of the operators from the IP flow filtering framework. For some of the more complex operators such as the grouper and the merger there will be a need to do some research in order to decide which algorithms and heuristics should be used in order to optimize the performance. In addition, one should consider various possibilities for flow storage and choose the most efficient one. Once an implementation is available one can formulate queries to match patterns from a knowledge base against real flow traces. A user will select an Internet action (network service, attack, application) from a given list, the respective pattern will be retrieved from the knowledge base, and the prototypical pattern will be matched against real trace data. A good source of netflow traces to start with would be the PlanetLab and the EmanicsLab network [54, 55]. Later on, traces obtained from network operators can be used.

Bibliography

[1] Netflow, http://en.wikipedia.org/wiki/netflow.

[2] M. Kim, H. Kang, S. Hong, S. Chung and J. W. Hong. A Flow-based Method for Abnormal Network Traffic Detection. In *Proceedings of NOMS '04*, 2004.

[3] B. Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954, Cisco Systems, October 2004.

[4] B. Claise. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information. RFC 5101, Cisco Systems, January 2008.

[5] Wireshark, http://www.wireshark.org.

[6] tcpdump, http://www.tcpdump.org.

[7] B. Claise. Packet Sampling (PSAMP) Protocol Specifications. Internet Draft (work in progress) <draft-ietf-psamp-protocol-09.txt>, Cisco Systems, December 2007.

[8] P.Phaal, S. Panchen, N. McKee. InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. RFC 3176, InMon Corp., September 2001.

[9] N. Brownlee, C. Mills, G. Ruth. Traffic Flow Measurement: Architecture. RFC 2722, The University of Auckland, GTE Laboratories, GTE Internetworking, October 1999.

[10] S. Waldbusser. Remote Network Monitoring Management Information Base. RFC 2819, Lucent Technologies, May 2000.

[11] R. Sommer and A. Feldmann. NetFlow: Information loss or win? In *Proceedings of IMW'02*, pages 173–174, New York, NY, USA, 2002. ACM.

[12] Cisco Systems. *NetFlow Services Solution Guide*, 4th edition, January 2007.

[13] Flow-tools, http://www.splintered.net/sw/flow-tools/.

[14] B. Nickless. Combining Cisco NetFlow Exports with Relational Database Technology for Usage Statistics, Intrusion Detection, and Network Forensics. In *Proceedings of LISA '00*, pages 285–290, Berkeley, CA, USA, 2000. USENIX Association.

[15] B. Babcock, S. Babu, M. Datar, R. Motwani and J. Widom. Models and issues in Data Stream Systems. In *Proceedings of PODS '02*, pages 1–16, New York, NY, USA, 2002. ACM.

[16] C. Cranor, T. Johnson, O. Spataschek and V. Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *Proceedings of SIGMOD'03*, pages 647–651, New York, NY, USA, 2003. ACM.

[17] M. Sullivan and A. Heybey. Tribeca: a System for Managing Large Databases of Network Traffic. In *Proceedings of ATEC'98*, pages 13–24, Berkeley, CA, USA, 1998. USENIX Association.

[18] N. Brownlee. SRL: A Language for Describing Traffic Flows and Specifying Actions for Flow Groups. RFC 2723, The University of Auckland, October 1999.

[19] N. Brownlee. Using NeTraMet for Production Traffic Measurement. In *Proceedings of IM'01*, 2001.

[20] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of USENIX'93*, pages 259–270, Berkeley, CA, USA, 1993. USENIX Association.

[21] Nfdump, http://nfdump.sourceforge.net/.

[22] D. Moore, K. Keys, R. Koga, E. Lagache and KC. Claffy. The Coral Reef Software Suite as a Tool for System and Network Administration. In *Proceedings of LISA'01*, 2001.

[23] K. Keys, D. Moore, R. Koga, E. Lagache, M. Tesch and KC. Claffy. The Architecture of CoralReef: an Internet Traffic Monitoring Software Suite. In *Proceedings of PAM'01*. CAIDA, RIPE NCC, 2001.

[24] S. Kornexl, V. Paxson, H. Dreger, A. Feldmann and R. Sommer. Building a Time Machine for Efficient Recording and Retrieval of High-Volume Network Traffic. In *Proceedings of IMC'05*, Berkeley, CA, USA, 2005. USENIX Association.

[25] L. Deri and S. Suin. Ntop: Beyond ping and traceroute. In *Proceedings of DSOM '99*, pages 271–284, London, UK, 1999. Springer-Verlag.

[26] Ntop, http://www.ntop.org.

[27] S. Romig. The OSU Flow-tools Package and CISCO NetFlow Logs. In *Proceedings of LISA '00*, pages 291–304, Berkeley, CA, USA, 2000. USENIX Association.

[28] Cisco Systems. *Configuring IP Access Lists.*

[29] D. Plonka. FlowScan: A Network Traffic Flow Reporting and Visualization Tool. In *Proceedings of LISA '00*, pages 305–318, Berkeley, CA, USA, 2000. USENIX Association.

[30] Rrdtool, http://oss.oetiker.ch/rrdtool/.

[31] Cuflow, http://www.columbia.edu/acis/networks/advanced/cuflow/.

[32] A. Oslebo. Stager-A Web Based Application for Presenting Network Statistics. In *Proceedings of NOMS'06*, 2006.

[33] C. Estan, S. Savage and G. Varghese. Automatically Inferring Patterns of Resource Consumption in Network Traffic. In *Proceedings of SIGCOMM '03*, pages 137–148, New York, NY, USA, 2003. ACM.

[34] Michael Collins, Andrew Kompanek and Timothy Shimeall. *Analysts Handbook: Using SiLK for Network Traffic Analysis.* CERT, 0.10.3 edition, November 2006.

[35] D. Brauckhoff, M. May and B. Plattner. Flow-Level Anomaly Detection - Blessing or Curse? In *Proceedings of INFOCOM'07*, Anchorage, Alaska, USA.

[36] P. Barford and D. Plonka. Characteristics of Network Traffic Flow Anomalies. In *Proceedings of IMW '01*, pages 69–73, New York, NY, USA, 2001. ACM.

[37] S. Ha, M. Kim, H. Ju and J. W. Hong. The Architecture of NG-MON: A Passive Network Monitoring System for High-Speed IP Networks. In *Proceedings of DSOM '02*, pages 16–27, London, UK, 2002. Springer-Verlag.

[38] Symantec. *W32.Welchia.Worm.*

[39] T. Dubendorfer and B. Plattner. Host Behaviour Based Early Detection of Worm Outbreaks in Internet Backbones. In *Proceedings of WETICE '05*, pages 166–171, Washington, DC, USA, 2005. IEEE Computer Society.

[40] A. Wagner and B. Plattner. Entropy Based Worm and Anomaly Detection in Fast IP Networks. In *Proceedings of WETICE '05*, pages 172–177, Washington, DC, USA, 2005. IEEE Computer Society.

[41] T. Dubendorfer, A. Wagner and B. Plattner. A Framework for Real-Time Worm Attack Detection and Backbone Monitoring. In *Proceed-*

ings of IWCIP '05, pages 3–12, Washington, DC, USA, 2005. IEEE Computer Society.

[42] A. Wagner, T. Dubendorfer, L. Hammerle and B. Plattner. Flow-Based Identification of P2P Heavy-Hitters. In *Proceedings of ICISP '06*, Washington, DC, USA, 2006. IEEE Computer Society.

[43] M. Kim, H. Kang and J. Won-Ki Hong. Towards Peer-to-Peer Traffic Analysis Using Flows. In *Proceedings of DSOM'03*, pages 55–67, 2003.

[44] R. van de Meent and A. Pras. Assessing Unknown Network Traffic. Ctit technical report, University of Twente.

[45] A. Fin. A Genetic Approach to Qualitative Temporal Reasoning with Constraints. In *Proceedings of ICCIMA '99*, Washington, DC, USA, 1999. IEEE Computer Society.

[46] Iana, http://www.iana.org/assignments/port-numbers.

[47] Graffiti, http://www.graffiti.com/services.

[48] J. Quittek, S. Bryant, B. Claise, P. Aitken, J. Meyer. Information Model for IP Flow Information Export. RFC 5102, Cisco Systems, January 2008.

[49] T. Dübendorfer, A. Wagner, T. Hossmann and B. Plattner. Flow-level Traffic Analysis of the Blaster and Sobig Worm Outbreaks in an Internet Backbone. In *Proceedings of DIMVA'05*, Vienna, Austria, July 2005. Springer's Lecture Notes in Computer Science (LNCS 3548).

[50] CERT/CC. *CERT Advisory CA-2003-20 W32/Blaster worm.*

[51] T. Dübendorfer, A. Wagner and B. Plattner. A Framework for Real-Time Worm Attack Detection and Backbone Monitoring. In *Proceedings of IWCIP'05*, Darmstadt, Germany, November 2005. IEEE.

[52] J. Rosenberg, J. Weinberger, C. Huitema, R. Mahy. STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs). RFC 3489, Dynamicsoft, Microsoft, Cisco Systems, March 2003.

[53] S. Baset and H. Schulzrinne. An Analysis of the Skype Peer-to-Peer Internel Telephony Protocol. Dec 2004.

[54] Planetlab, http://www.planet-lab.org.

[55] Emanicslab, http://emanicslab.csg.uzh.ch.

Appendix A

FlowScan Reporting Module

```
sub Napster_wanted {
   if (ICMP != flow::protocol || not (TCP == flow::protocol
       and (1024 < flow::srcport and 1024 < flow::dstport)) {
return 0
}

   # flow is either ICMP or TCP on unprivileged ports
   if (inbound(flow)) {
direction = in;
outside_addr = flow::srcaddr;
inside_addr = flow::dstaddr
}

   elsif (outbound(flow)) {
direction = out;
outside_addr = flow::dstaddr;
inside_addr = flow::srcaddr
}

   if (TCP == flow::protocol and ACK & flow::tcp_flags and
       is_napserver(outside_addr)) {

# flow involves an ouside host that is
# a "publicly advertised"  napserver
remember_napster_server(outside_addr, flow::endtime);
remember_napster_user(inside_addr, flow::endtime)
}

   elsif (is_napuser(inside_addr)) {
# flow involves an inside host that has talked to
# a napserver recently
if ((TCP == flow::protcol and
    napster_ports(flow::srcport, flow::dstport)) or
            (ICMP == flow::protocol and 28 == flow::bytes/flow::pkts)) {
# Confidence is high that this is a Napster application
# flow because flow is either TCP on Napster "default"
# ports or  flow is ICMP using the "known" Napster ICMP
# packet size.
napster_total++;
return 1
}
else {
# Confidence is lower that this is really a Napster
# application flow  because the port numbers are not
# Napster defaults or the ICMP packet size isnt right.
# Well keep a count of these anyway.
maybe_napster_total++
}
   }
return 0
}
```

I hereby declare that this M.Sc. thesis is independent work and has not been submitted elsewhere for a conferral of a degree.