Design of a Stream-Based IP Flow Record Query Language

Vladislav Marinov and Jürgen Schönwälder

Computer Science, Jacobs University Bremen, Germany
{v.marinov,j.schoenwaelder}@jacobs-university.de

Abstract. Analyzing Internet traffic has become an 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 the filtering or query languages that these tools use have limited capabilities when it comes to describing more complex network activity. This paper proposes a framework and a new stream-based flow record query language, which allows certain types of traffic patterns to be defined and matched in a collection of flow records. The usage of the proposed new language is exemplified by constructing a query identifying the Blaster.A worm.

Keywords: Network measurement, NetFlow, IPFIX.

1 Introduction

The NetFlow protocol [1], originally designed by Cisco Systems, enables routers to export summary information about the traffic flows that traverse a router. Inspired by Cisco's early work, the IETF created a standard IP flow information export protocol called IPFIX [2] A network flow is defined as an unidirectional sequence of packets between given source and destination endpoints. Specifically, a flow is usually identified by the combination of the following seven key fields: source and destination IP address, source and destination port number, IP protocol type (TCP, UDP, etc.), ToS byte, and the input interface (ifIndex). In addition to the key fields, a flow record contains other accounting fields such as packet and byte counts, input and output interfaces, bit-wise logical or of TCP flags, timestamps, MPLS labels etc. Network elements (routers and switches) gather flow data and export it to collectors for analysis.

The flow records exported via NetFlow/IPFIX provide a summary about the traffic traversing a specific router. However, raw collections of flow records still contain too many details for network administrators and they are not useful unless processed by network analysis tools. Most of the existent flow record processing tools provide mechanisms for searching of specific flows through some simple operations like filtering by an IP address or port number or generating Top-N talkers reports. However, in order to match more complex flow patterns against collections of flow records, one needs a useful flow record query language.

C. Bartolini and L.P. Gaspary (Eds.): DSOM 2009, LNCS 5841, pp. 15–28, 2009.

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.

In this paper we propose a flow record query language, which allows to describe patterns in a declarative and easy to understand way. This paper is a followup of the early paper [5], where we discussed in some detail the motivation of this research. The proposed language is able to define filter expressions (needed to select relevant flows) and relationships (needed to relate selected flows). It allows to express causal dependencies between flows as well as timing and concurrency constraints. Existent query languages as discussed in Section 2 are not suitable for detecting complex traffic patterns because of either performance issues (SQL-based query languages) [3,4] or because they lack a time and concurrency dimension (BPF expressions and the other query languages we discuss). Furthermore, the new query language provides 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 query languages. Using the new query language, we built a knowledge base of flow fingerprints that belong to some common network services, applications and attacks. As an example, we describe the query detecting the flow fingerprint of the Blaster.A worm.

The rest of the paper is structured as follows. Section 2 provides a short survey of existing flow filtering and query languages. In Section 3 we present our stream-based flow query language and in Section 4 we show an application

example by using it to describe a common network traffic pattern. We conclude in Section 5 with a few remarks on ongoing work.

2 Related Work

According to [5] existing flow record query languages can be split into SQL-based query languages, filtering languages, and procedural languages.

2.1 SQL-Based Query Languages

Many of the early implementations of 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 [6] describes a system which uses standard MySQL and Oracle DBMS for storing the attributes of NetFlow records. Using powerful SQL queries, the tool was able to provide good support for basic intrusion detection and usage statistics. 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. The 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 relational DBMS. Based on these requirements B. Babcock et al. [4] propose the design of a Data Stream Management System (DSMS). Together with the model the authors also extend the SQL query language so that the DSMS can be queried over time and provide examples of network traffic reports that are generated based on flow data that is stored in such a DSMS. *Gigascope* [7] 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. *Tribeca* [3] 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-based query language which supports operations such as projection, selection, aggregation, multiplexing and demultiplexing of streams based on stream attributes. The query language also defines a windowing mechanism to select a timeframe for the analysis.

2.2 Filtering Languages

The *Berkeley Packet Filter (BPF)* [8] allows users to construct simple expressions for filtering network traces by IP address, port number, protocol etc. and translates these expressions into small programs executed by a generic packet

filtering engine. One popular use of the BPF is in the `tcpdump` utility. The BPF rules for constructing filter expressions are also used in `nfdump` [9], which is a powerful and fast filter engine used to analyze network flow records. `nfdump` is currently one of the *de facto* standard tools for analyzing NetFlow data and generating reports. BPF expressions are also used in the *CoralReef* network analysis tool described in [10,11] in order to generate traffic reports from collected trace files. The *Time Machine* tool described in [12] uses BPF expressions to define classes of traffic and BPF is also part of the query language used by the tool for retrieval of interesting traffic.

The `flow-tools` package [13] is another widely-used collection of applications for collecting and analyzing NetFlow data. Two of the flow-tools applications are responsible for filtering flows and generating reports: `flow-filter` and `flow-report`. The former application uses the Cisco Access Control List (ACL) format to specify a filter for IP addresses and command line arguments for specifying other filtering parameters such as port numbers, ASes etc. The latter uses a configuration file where reports can be defined by using a number of primitives.

2.3 Procedural Languages

FlowScan described in [14] 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 [15], and a visualization tool. FlowScan has the capability of generating powerful high-level traffic reports, which might help operators to detect interesting traffic patterns. However, reports must be specified as separate perl modules, which is not trivial and might involve some heavy scripting.

C.Estan et al. [16] proposes an approach for detecting high-level traffic patterns by aggregating NetFlow records in clusters based on flow record attributes. Aggregation on several flow attributes results in a multidimensional cluster. 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 [17] is another script-based collection of command-line tools for querying NetFlow data. It provides its own primitives for defining filtering expressions. Unlike other network analysis tools, *SiLK* contains two applications that allow an analyst to label a set of flows sharing 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 flows in a long lived session such as a FTP connection. `rwmatch` creates matched groups, which consist

of an initial record (a query) followed by one or more responses. Its most basic use is to group records into both sides of a bidirectional session, such as a HTTP request.

3 Stream-Based Flow Query Language

Our framework for IP flow filtering follows a stream-oriented approach — it consists of a number of processing elements or operators, which are connected with each other via pipes. Each element receives an input stream, performs some sort of operation on it (filtering, aggregation etc.) and the output stream is piped to the next element. Figure 1 shows the framework and in the following sections we describe each of its elements. A complete definition of the syntax and the semantics of the elements can be found in [19]. Section 4 provides an example illustrating the usage of the primitives of the stream-based flow query language. The names of the filtering primitives in our language are closely linked to the flow record attributes in RFC 5102 [18].

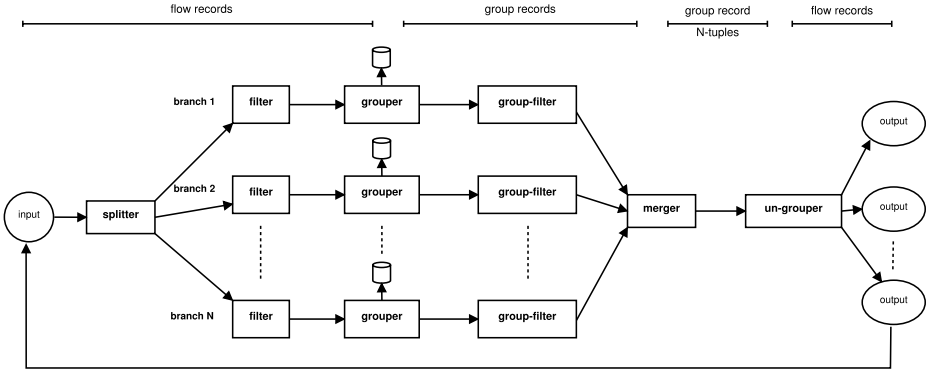


Fig. 1. IP flow filtering framework

3.1 Splitter

The `splitter` is the simplest operator in the IP flow filtering framework. It takes the input stream of flow 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`.

3.2 Filter

The `filter` operator takes a stream of flow records as 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.

3.3 Grouper

The `grouper` operator takes a stream of flow records as 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 logical and. The different rule modules on the other hand are logically linked by an implicit logical 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 [17] 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 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. The details of the algorithm are described in [19].

3.4 Group-Filter

The `group-filter` operator takes a stream of group records as 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.

3.5 Merger

The `merger` operator takes several streams of group records as 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 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. For example consider the following two queries:

Q1: 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.

Q2: 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 query *Q1* is relatively easy to define using a single rule module, for query *Q2* 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.

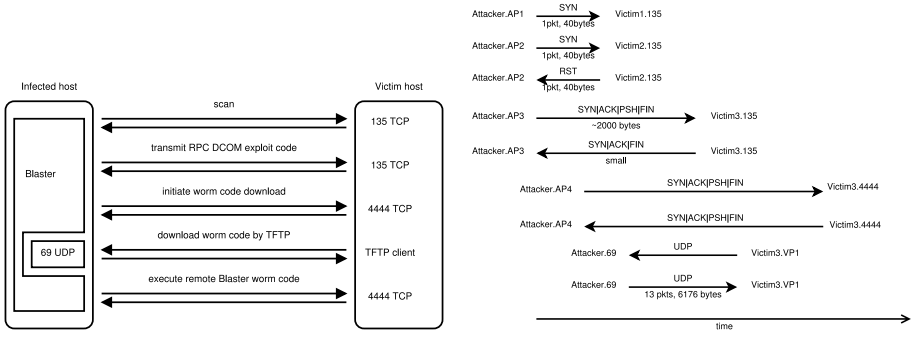
In general, the `merger` operator allows to perform grouping at a more general level compared to the `grouper` operator. Another powerful feature of the `merger` operator is its capability to express timing and concurrency constraints among various traffic groups by using Allen's time interval algebra [20].

3.6 Ungrouper

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 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.

4 Application

In this section we present the traffic pattern generated by a computer infected with the Blaster.A worm and define this pattern using our stream-based flow



(a) Packet level breakdown (b) Flow level breakdown

Fig. 2. Packet level and flow level breakdown of a Blaster infection

record query language. The Blaster.A worm [21] 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. Dübendorfer et al. [22] describe the various stages of the Blaster worm infection and perform an in-depth packet and flow level analysis.

The characteristic network activity (an infected attacker trying to infect other hosts on the network) associated with such an attack consists of the steps described in Figure 2(a). The flow-level breakdown of the Blaster attack is shown in Figure 2(b). The flow record 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

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 flow fingerprint.

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.

```
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
```

Additional constraints ensure 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

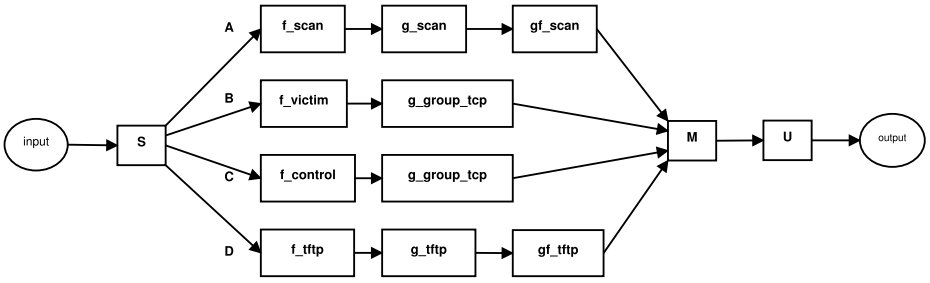


Fig. 3. Capturing Blaster worm infections with the IP flow filtering framework

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 TCP connection / 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 4444/TCP. Finally from Figure 2(b) 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.

5 Conclusions

After a careful analysis of the pros and cons of the existing filtering and query languages [5], we designed a new IP flow filtering framework and an associated

filtering language. The language 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 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 [19]. 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.

We are currently implementing a prototype consisting of a parser, which reads the flow pattern definition, and an execution engine, which implements the operators or our 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. In the future, we envision that protocol experts will, assisted by an interactive flow visualization system, develop queries for specific scenarios. Once our implementation is complete, we can test these queries and share them with non-experts through a knowledge base so that they can be easily matched against flow traces.

Acknowledgement

The work reported in this paper is supported by the EC IST-EMANICS Network of Excellence (#26854).

References

1. Claise, B.: Cisco Systems NetFlow Services Export Version 9. RFC 3954, Cisco Systems (October 2004)
2. Claise, B.: Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information. RFC 5101, Cisco Systems (January 2008)
3. Sullivan, M., Heybey, A.: Tribeca: a System for Managing Large Databases of Network Traffic. In: Proceedings of ATEC 1998, pp. 13–24. USENIX Association, Berkeley (1998)
4. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in Data Stream Systems. In: Proceedings of PODS 2002, pp. 1–16. ACM, New York (2002)
5. Marinov, V., Schönwälder, J.: Design of an IP Flow Record Query Language. In: Hausheer, D., Schönwälder, J. (eds.) AIMS 2008. LNCS, vol. 5127, pp. 205–210. Springer, Heidelberg (2008)

6. Nickless, B.: Combining Cisco NetFlow Exports with Relational Database Technology for Usage Statistics, Intrusion Detection, and Network Forensics. In: Proceedings of LISA 2000, pp. 285–290. USENIX Association, Berkeley (2000)
7. Cranor, C., Johnson, T., Spataschek, O., Shkapenyuk, V.: Gigascope: A Stream Database for Network Applications. In: Proceedings of SIGMOD 2003, pp. 647–651. ACM, New York (2003)
8. McCanne, S., Jacobson, V.: The BSD Packet Filter: A New Architecture for User-level Packet Capture. In: Proceedings of USENIX 1993, pp. 259–270. USENIX Association, Berkeley (1993)
9. Haag, P.: nfdump, `http://nfdump.sourceforge.net/`
10. Moore, D., Keys, K., Koga, R., Lagache, E., Claffy, K.: The Coral Reef Software Suite as a Tool for System and Network Administration. In: Proceedings of LISA 2001, pp. 133–144. USENIX Association, Berkeley (2001)
11. Keys, K., Moore, D., Koga, R., Lagache, E., Tesch, M., Claffy, K.: The Architecture of CoralReef: an Internet Traffic Monitoring Software Suite. In: Proceedings of PAM 2001, CAIDA, RIPE NCC (2001)
12. Kornexl, S., Paxson, V., Dreger, H., Feldmann, A., Sommer, R.: Building a Time Machine for Efficient Recording and Retrieval of High-Volume Network Traffic. In: Proceedings of IMC 2005. USENIX Association, Berkeley (2005)
13. Fullmer, M.: flow-tools, `http://www.splintered.net/sw/flow-tools/`
14. Plonka, D.: FlowScan: A Network Traffic Flow Reporting and Visualization Tool. In: Proceedings of LISA 2000, pp. 305–318. USENIX Association, Berkeley (2000)
15. Oetiker, T.: RRDTool, `http://oss.oetiker.ch/rrdtool/`
16. Estan, C., Savage, S., Varghese, G.: Automatically Inferring Patterns of Resource Consumption in Network Traffic. In: Proceedings of SIGCOMM 2003, pp. 137–148. ACM, New York (2003)
17. Collins, M., Kompanek, A., Shimeall, T.: Analysts' Handbook: Using SiLK for Network Traffic Analysis. CERT. 0.10.3 edn. (November 2006)
18. Quittek, J., Bryant, S., Claise, B., Aitken, P., Meyer, J.: Information Model for IP Flow Information Export. RFC 5102, Cisco Systems (January 2008)
19. Marinov, V.: Design of an IP Flow Record Query Language. Master's thesis, Jacobs University Bremen (May 2009)
20. Fin, A.: A Genetic Approach to Qualitative Temporal Reasoning with Constraints. In: Proceedings of ICCIMA 1999, Washington, DC, USA. IEEE Computer Society, Los Alamitos (1999)
21. Symantec: W32.Welchia.Worm (August 2003)
22. Dübendorfer, T., Wagner, A., Hossmann, T., Plattner, B.: Flow-Level Traffic Analysis of the Blaster and Sobig Worm Outbreaks in an Internet Backbone. In: Julisch, K., Krügel, C. (eds.) DIMVA 2005. LNCS, vol. 3548, pp. 103–122. Springer, Heidelberg (2005)