
# F ($\nu$2)

# A complete system integration of the network flow query language

by

Vaibhav Bajpai

A thesis for conferral of a Master of Science in Computer Science

Prof. Dr. Jürgen Schönwälder
Name and title of first reviewer

Dr. Heinrich Stamerjohanns
Name and title of second reviewer

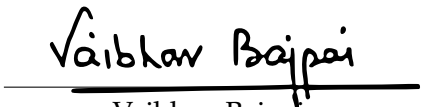
Date of Submission: August 23, 2012

Jacobs University — School of Engineering and Science

# DECLARATION

I hereby declare that this thesis is the result of my independent work, has not been previously accepted in substance for any degree and is not concurrently submitted for any degree. This thesis is being submitted in fulfillment of the requirements for the degree of Master of Science in Computer Science.

*Bremen, Germany, August 23, 2012*



Vaibhav Bajpai

Dedicated to the loving memory of my father Prof. Dr. S.K. Bajpai.

1945 – 2010

## ABSTRACT

Cisco's NetFlow protocol [1] and Internet Engineering Task Force (IETF)'s Internet Protocol Flow Information Export (IPFIX) [2] open standard have contributed heavily in pushing Internet Protocol (IP) flow export as the de-facto technique for sending traffic patterns. These patterns have the potential to be used for billing and mediation, bandwidth provisioning, detecting malicious attacks, network performance evaluation and overall improvement.

However, making sense of these patterns calls for sophisticated flow analysis tools that can mine them for such a usage. Unfortunately current tools fail to deliver owing to their poor language design and simplistic filtering methods. Our research group, by going clean slate has developed the Network Flow Query Language (NFQL) [3] that can cap flow-exports to full potential. It can process flow records, aggregate them into groups, apply absolute (or relative) filters and invoke Allen interval algebra rules [4] on them.

Flowy [5] is the prototype implementation of NFQL, which has undergone significant changes in the last few years. The core of this Python implementation is now being rewritten in C to make it comparable to the contemporary flow processing tools. The first major release in this effort, F($v1$) [6] can now read the flow-records in memory and apply absolute filters. The absolute filters, however are only one amongst the five stages of the complete NFQL processing pipeline. The rest of the stages either have a broken implementation or are completely non-existent. The flowquery used by the execution engine to mine the traces is also hardcoded in pipeline structs.

This thesis extends F($v1$) to provide a complete usable implementation of NFQL with *functional* pipeline stages. The engine is *robust* to work well with variety of NFQL queries that are now read and parsed at runtime using the `JSON` intermediate format. The filter and grouper implementations have also been reimagined to bring forth an orders of magnitude performance speedup. F($v2$) is *portable* with automated builds using `cmake` and is *verifiable* using a regression test-suite framework. An automated benchmarking suite enables the conducted performance evaluations to be repeatable. The conducted performance evaluations reveal that F($v2$) can now process flow traces with millions of records in matter of minutes, which used to take days in F($v1$). Such a performance boost makes F($v2$) comparable to contemporary flow-processing tools.

# PUBLICATIONS

[1] V. Bajpai, N. Melnikov, A. Sehgal, and J. Schönwälder, "Flow-Based Identification of Failures Caused by IPv6 Transition Mechanisms," in *Dependable Networks and Services*, vol. 7279 of *Lecture Notes in Computer Science*, pp. 139–150, Springer Berlin Heidelberg, 2012.

*Programs must be written for people to read,*
*and only incidentally for machines to execute.*

— H. Abelson and G. Sussman

## ACKNOWLEDGMENTS

I would like to express my gratitude to my supervisor Prof. Dr. Jürgen Schönwälder for providing me constant feedback and support throughout the entire duration of my masters thesis.

I would also like to thank my mother for her blessings and personal support and my brother Dr. Gaurav Bajpai for providing me continuous encouragement and for proof-reading my thesis.

# CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

# Part I

# INTRODUCTION

The network and user behavior traffic pattern analysis is creating a lot of traction owing to its wide applicability in accounting, resource provisioning and network monitoring purposes. This section is dedicated to perform an exhaustive study on the available techniques that can perform such traffic measurements and how they are being used today. In particular, we focus our attention to the currently favored flow-capture technique by examining the de-facto protocols that describe the semantics of this flow-export. The organization of the section is described below.

In chapter 1 we discuss the current state-of-the-art traffic measurement techniques, the protocols supporting them, their pros and cons and how they are being used to mine the behavioral patterns of the current network traffic.

In chapter 2 we discuss Cisco's proprietary and IETF's standardized protocol for IP flow export. We discuss their architecture, protocol operations, their message formats and the future they are heading towards as seen from today.

# 1 TRAFFIC MEASUREMENT APPROACHES

Researchers, service providers and security analysts have long been interested in network and user behavioral patterns of the traffic crossing the internet backbone. They want to use this information for the purpose of billing and mediation, bandwidth provisioning, detecting malicious attacks, network performance evaluation and overall improvement. Traffic measurement techniques that have been rapidly evolving in the last decade, have matured enough today to provide such an insight. In this chapter, we discuss some of these techniques and how they are being used to shape the future of the internet.

## 1.1 CAPTURING PACKETS

In this technique, raw packets traversing a monitoring point are captured for traffic measurement. The measurements can be done either live or the packets can be saved in a trace file for offline analysis. The trace files can range from containing mere headers to entire packets depending on the level of detailed analysis required.

```
1 $ tcpdump port 80 -w $FILE
2 $ tcpdump -r $FILE
```

Listing 1: `tcpdump`: Example

`tcpdump` and `wireshark` are the most popular tools used for packet capture and analysis. `tcpdump` [17] is a premier command-line utility that uses the `libpcap` [18] library for packet capture. A simple example to capture and read the Hypertext Transfer Protocol (HTTP) traffic is described in listing 1. The power of `tcpdump` comes from the richness of its expressions, the ability to combine them using logical connectives and extract specific portions of a packet using filters. `wireshark` [19] is a Graphical User Interface (GUI) application, aimed at both journeymen and packet analysis ninjas. It supports a large number of protocols, has a straightforward layout, excellent documentation, and can run on all major operating systems.

*tcpdump*

*wireshark*

Several studies have made use of this approach to analyze the network traffic patterns. The authors in [20], for instance use data mining methodologies to define clusters of behavior profiles by understanding the captured traffic of end hosts. These clusters are then fed into classifiers to automatically identify anomalous behavior patterns that are of interest to network operators. Similar profiling of end-hosts traffic

*applicability*

in performed in [21], but at the transport layer. This effort focusses on making the approach tunable to strike out a balance between the amount of traffic classified and the accuracy achieved by analyzing the traffic at multiple levels of details.

This approach benefits from the astounding level of detail it can provide. It allows deep packet inspection of the traces, thereby exposing even the application content being exchanged across the network.

*pros and cons*

This calls for privacy concerns and can even bring in legal repercussions to make this technique unattractive for traffic analyzers today. In addition, the actual usage of this method comes at a higher price of its storage overhead and its inability to scale to larger setups.

## 1.2 CAPTURING FLOWS

In this technique, packets traversing a monitoring point are not captured raw, instead they are aggregated together based on some common characteristics. The common characteristics are learnt by inspecting the packet headers as they cross the monitoring point. Flow-records resulting from such an aggregation are then exported to a collector for further analysis.

NetFlow and IPFIX are the two popular standards of IP flow information export. NetFlow [1] is a proprietary network protocol designed

*netflow*

by Cisco Systems. It allow routers to generate and export flow records to a designated collector. The latest version, NetFlow v9 provides flexibility of user-tailored export templates, Multiprotocol Label Switching (MPLS) and IPv6 support and a larger set of flow keys. IPFIX [2] on the other hand is an open standard by IETF deemed to be the logical

*ipfix*

successor of NetFlow v9 on which it is based. The novelty of the standard lies in its ability to describe record formats at runtime using templates based on an extensible and well-defined information model. The data transfer mechanism is also simplistic and extensible by being unidirectional and transport protocol agnostic.

The wide applicability of this approach is easily seen from the pervasive use of flow records for a vibrant set of network analysis applications. For instance, the authors in [22] use the flow characteristics in the traffic pattern to formalize a detection function that maps traffic

*applicability*

patterns to different Denial of Service (DoS) attacks, whereas in [23], the authors use the flow-record data to exploit timing characteristics of webmail clients to classify features that could identify webmail traffic from any other traffic running over HTTPS.

This is has been possible largely due to the hardware-assisted aggregation of the packets that has helped solve the storage overhead

*pros and cons*

and scalability limitation of packet capturing techniques. Overcoming these limitations have eventually allowed researchers to perform network analysis over a larger dataset passing across high-speed links. However, with the ever-increasing bandwidth demands, the speed of

the network links in the internet backbone is only slated to increase further, therefore the time is not too far when this issue might again scares us of its homecoming.

## 1.3 REMOTE MONITORING

In this technique, dedicated monitoring probes are deployed on network segments to continuously collect vital statistics and perform network diagnostic operations. The probes are configured to proactively monitor the network and automatically check for error conditions to later log and notify them to the management station.

The Remote Network Monitoring (RMON) Framework [24] for Simple Network Management Protocol (SNMP) [25] defines a number of Management Information Base (MIB) objects to be used by these monitoring probes. The RMON-1 standard [26] for instance, defines a MIB module to collect statistics, capture and filter packet contents at the logical link layer. The architecture in this standard has been further extended with a feature upgrade by the RMON-2 standard [27] to support similar analysis up to the application layer.

*rmon*

The novelty of this technique lies in the ability to immediately communicate important information to the managing station using events and alarms. The constructs are extremely flexible in giving full control over what conditions will cause an alarm and subsequently what event will be generated. The event-driven nature of such a monitoring platform however still does not satisfy the requirements of traffic analysis applications since the data that is pushed out is highly aggregated and lacks enough details to be useful.

*pros and cons*

## 1.4 REMOTE METERING

In this technique, meters are deployed at the network measurement points to capture flow data according to a predefined set of rules specified by the user. The model, as defined by the Realtime Traffic Flow Measurement (RTFM) working group [28] has been designed to be protocol agnostic and restrictive in the amount of flow data that can transmitted across the network and stored to reduce the processing time of network analysis applications.

The feature that sets this technique apart is the flexibility given to the user to specify their flow measurement requirements, thereby allowing them to filter out the traffic they do not care about. This calls for the users, to at the very outset analyze and freeze their requirements before they start off to capture the traffic. This is analogous to the flaws inherit in the waterfall model [29] of software design, whereby one need to design the design before one designs it.

*pros and cons*

# 2 FLOW EXPORT PROTOCOLS

Flow capture today, has emerged out to be one of the favored network measurement techniques. This has largely been due to the reduction in the monitoring traffic at the flow-level and the fine-grained control which was not previously possible using SNMP interface-level queries. As a result, each networking vendor has tried to come up with a standard protocol that defines the semantics of this flow export. In this pursuit, Cisco eventually managed to make their proprietary protocol so ubiquitously available, that the next-generation universal standard is based on it. In this chapter, we discuss Cisco's de-facto proprietary and the recently standardized IETF's open protocol for IP flow-export.

## 2.1 NETFLOW

NetFlow [1] by Cisco Systems is a protocol that allows network elements to export IP flow information to designated collectors from where they can be later retrieved for further analyses. The collected flow-records are flexible enough to be used for a variety of purposes such as billing and mediation, network and user monitoring, resource provisioning, security analysis and data mining research works.

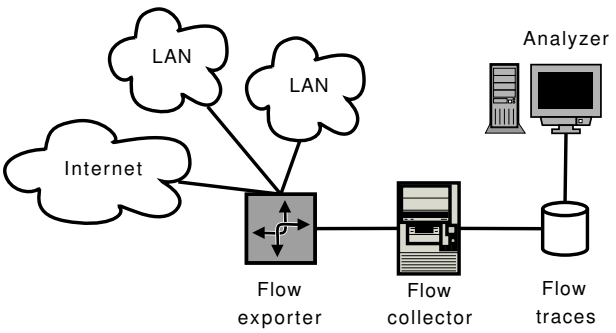



Figure 1: NetFlow: Overview [7]

A high-level abstracted functioning of the NetFlow protocol is shown in figure 1. The flow exporter reads the IP packets that cross its boundary to generate flow-records. The flow-records are exported based on some predefined expiration rules, such as a Transmission Control Protocol (TCP) `FIN` or `RST`, an inactivity timeout, a regular export timeout or crossing a low memory threshold. To achieve efficiency when handling large amounts of traffic, the flow-records are encapsulated in User Datagram Protocol (UDP) datagrams and are

*protocol operation*

deleted from the exporter once transmitted. On the other end, the collector on receiving these flow-records, decodes and stores them locally to be used for further processing.

```
1 (A) --> [SYN] ------>(B)
2 (A) <-- [SYN/ACK] <--(B)
3 (A) --> [ACK] ------>(B)
```

Listing 2: A Flow Example

*what is a flow?*

A flow is defined by a 7-tuple flow-key, namely: {`srcIP`, `dstIP`, `srcPort`, `dstPort`, `ipProto`, `ifIndex`, `ipTOS`}. IP packets with identical flow-keys become part of one flow. Two flows resulting from a three-way TCP handshake for example are shown in listing 2. In addition to the flow-key, flow-records can also contain additional accounting information such as flow start and end times, number of packets/octets in a flow, source/destination Autonomous Systems (AS) numbers, et al.

*version history*

The NetFlow version history is summarized in table 1. NetFlow v1 was introduced in the 90s, however it was only until v5 with the introduction of Classless Inter-Domain Routing (CIDR) and AS support that the technology got mainstream. Today, NetFlow v9 is the de-facto industry standard and is the bases for IETF's IPFIX effort to create a universal specification for IP flow-export.

| version | features |
|---|---|
| v1, {2, 3, 4} | original format with several internal releases |
| v5 | CIDR, AS support and flow sequence numbers |
| v{6, 7, 8} | router-based aggregation support |
| v9 | template-based with IPv6, and MPLS support |
| IPFIX | universal standard, transport-protocol agnostic |

Table 1: NetFlow Version History

*netflow v9*

NetFlow v9 introduces templates in its export format. With templates, the exporter only needs to send required fields to the flow collector thereby reducing the volume of flow-data exported. In addition, fields can be added/removed from the flows without changing the export format. The transmission of records encapsulated in UDP datagrams can lead to loss of flows when the link is congested and therefore the exporter and collector have usually been restricted to one-hop away dedicated links. To overcome this limitation, NetFlow v9 introduces transport support over congestion-aware Stream Control Transmission Protocol (SCTP). In addition, NetFlow v9 also provides support for MPLS and IPv6 addresses.

The ever increasing traffic volume crossing high-speed links, has been creating an enormous pressure on the routers that also engaged in NetFlow export. Sampled NetFlow was thus introduced by Cisco Systems as an extension to NetFlow v9 to tone down the gigantic computation, by allowing the routers to skip over to every $n^{th}$ packet for flow export. The sampling rate $(n)$ is indicated in the export header and is either configured or randomly selected.

*sampled netflow*

Though sampled NetFlow does a good job in reducing the exported traffic volume, the sampling rate is still static which either reduces accuracy at low traffic volumes or increases memory use at high traffic volumes. An adaptive algorithm introduced in [30] helps overcome this difficulty. The introduced renormalization technique helps guarantee that the sampling rate can not only adjust to variable traffic mixes but also to network congestion. It also ensures that the flow records do not span over measurement bins to be able to guarantee statistical accuracy. The authors claim, that these updates are easily deployable to any NetFlow v9 router through a software update. In addition they say, a simple hardware add-on (flow counting extension), can also add capability to accurately count non-TCP flows, a feature long waiting to be seen in NetFlow v9.

*adaptive netflow*

Flexible NetFlow is the newest version of NetFlow v9 that incorporates Packet Sampling (PSAMP) [31] ideas to be able to select individual packets and export them in a packet record. The packet selection can be either deterministic or random depending on the chosen filters and sampling mechanism [32]. The exported packet records can even be authenticated and encrypted using either Transport Layer Security (TLS) [33] or Datagram Transport Layer Security (DTLS) [34] to prevent data manipulation across the route. Since PSAMP is based on IPFIX [35], only its limited feature set is currently supported by Flexible NetFlow. Additional features include ability to custom define flow-keys and flow-expiration rules to drastically reduce the amount of content exported by restricting it to only the needed flow-fields, and additional flows with immediate and permanent caches to suit the export timings to specific needs.

*flexible netflow*

The challenge to identify relevant records in gigantic collected datasets have fumed recent independent studies to discover flow dependencies. For instance, the authors in [36], describe a model that uses flow timing information by extending the PageRank [37] algorithm to rank and thereby extract the most relevant flows. Their model is weighted using parameters like the amount of bandwidth consumed and the likehood of security threat a flow might result in.

*flowrank*

Today, as the industry is moving towards data center virtualization, it has become inherently critical to obtain insights into the data center network behavior for optimizations and resource provisioning. Since, Flexible NetFlow's visibility is limited to the IP protocol it currently cannot be used to monitor data-center traffic. NetFlow-lite was thus

introduced by Cisco Systems, to flows at the layer 2/3 level to increase data center visibility. NetFlow-lite uses similar packet sampling mechanisms as introduced in Sampled NetFlow along with the combined flexibility of Flexible NetFlow v9 at the switch level. NetFlow-lite captures the layer 2 traffic, encapsulates packet samples and pushes the NetFlow cache outside the switch into a element that can convert NetFlow-lite to Flexible NetFlow records. These flow-records are then later exported to legacy collectors from where they can be used for further processing. The authors in [38] provide the first implementation of NetFlow-Lite which works as an extension to nProbe [39] to seamlessly convert NetFlow-Lite records to NetFlow/IPFIX.

*netflow-lite*

## 2.2 IPFIX

IPFIX [2] by IETF is an interoperable protocol for IP flow export. It is deemed to be the logical successor of Flexible NetFlow v9. The working group defines IPFIX as, "a unidirectional, transport-independent protocol with flexible data representation and an information model covering most network management needs at layer 3 and 4". The PSAMP working group [35], that defines standards to individually sample packets in a flow export using statistical methods has adopted IPFIX as its underlying protocol for data transport.

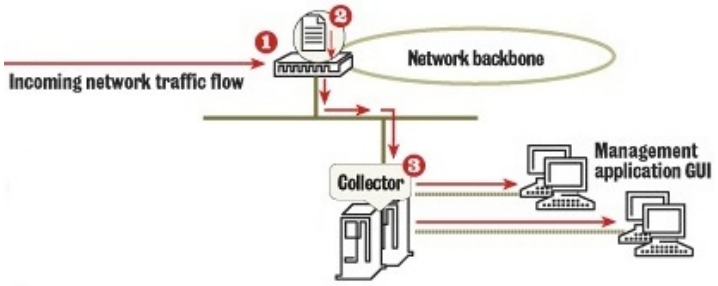



Figure 2: IPFIX: Overview [8]

The IPFIX architecture is described in [40] and is shown in figure 2. The architecture consists of three elements: a meter, which generates flows from IP packets, an exporter, which pushes these flows using IPFIX, and a collector, that collects and saves these flows for offline storage. All these elements have a one-to-many relationship among them. The group is also working to define an intermediary element, that might work to either aggregate or anonymize the flows.

*architecture*

A message is a fundamental unit of data exchange in IPFIX. Each such message consists of a 16-byte header along with a number of sets as shown in figure 3. A set can either be a template or a data set. Each such set in the message again contains a 16-bit header and a number of records associated with it. Each record within a template set is a template that refers to a data record. A template consists of a number

*messages and templates*

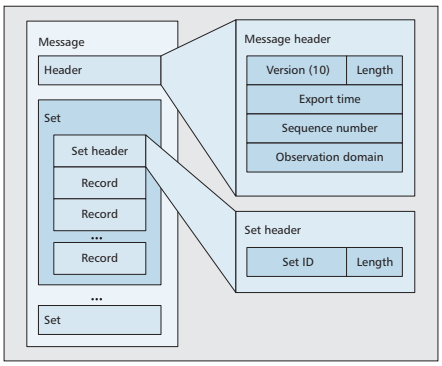



Figure 3: IPFIX: Messages [9]

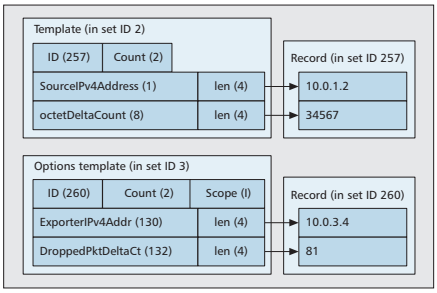



Figure 4: IPFIX: Templates [9]

of Information Elements (IE)s as shown in figure 4. These IEs are encoded using reduced-length encoding scheme. Internet Assigned Numbers Authority (IANA) keeps a registry [1] of all IEs with a 16-bit ID assigned to them. Templates can also contain enterprise-specific IEs that are scoped using Private Enterprise Numbers (PENs) [2].

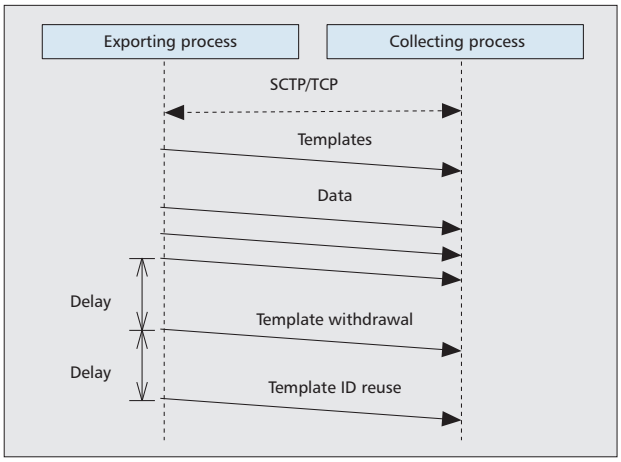



Figure 5: IPFIX: A Transport Session [9]

An IPFIX transport session is shown in figure 5. It starts off with the Exporter Process (EP) initiating a connection with the Collector Process (CP). Once the connection is established, the EP passes on the templates followed by the data that is described by them. These templates can later still be withdrawn by sending a control template of IE count zero. The transport session can use either SCTP, TCP or UDP as the underlying protocol, although SCTP is usually the preferred method given it allows selective reliability and congestion control. TCP is supported to allow secure transport over TLS, since DTLS is only supported over UDP and SCTP. The connection-less behavior of UDP calls for the template retransmission delay and template lifetime parame-

*transport and security*

1 http://www.iana.org/assignments/ipfix/ipfix.xml
2 http://www.iana.org/assignments/enterprise-numbers

ters to be exchanged between EP and CP. These transport sessions can also be stored in IPFIX files and sent on top application layer protocols.

*management, extensions and future of ipfix*

A MIB to monitor IPFIX devices using SNMP is defined in [41]. A similar configuration model to be used by NETCONF and YANG is being worked upon. In addition, several extensions have been defined to expand upon the protocol's functionality. For instance, [42] defines optional templates to allow bidirectional flows in a single IPFIX export whereas [31] supports aggregating common properties of multiple flows in a single record. IPFIX is even being looked upon as the future application-layer logging protocol as well as the underlying protocol for RESTful architectures. As a result, efforts to support structured data export over IPFIX are also under way.

## 2.3 SFLOW

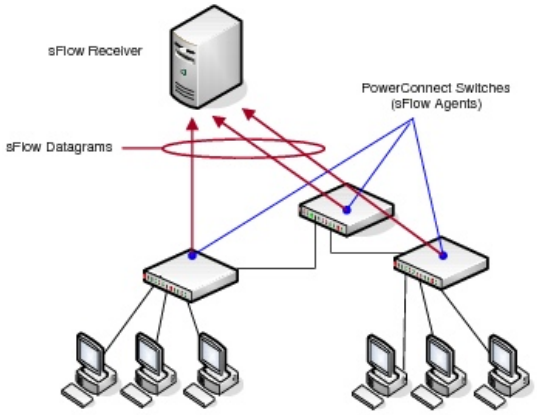



Figure 6: sFlow: Overview [10]

sFlow [43] by InMon Corporation is a competing technology used to capture traffic from switches and routers. It consists of an sFlow agent that captures traffic statistics and sends them across to a central data collector, called the sFlow analyzer as shown in figure 6. In order to be able to accurately monitor traffic at line speeds, the sFlow agent is built on a dedicated ASIC alongside the switching gear. In addition, the captured traffic is sampled before being encapsulated in sFlow datagrams and sent to the analyzer to provide scalability.

*sampling mechanisms*

A flow in sFlow is defined as all the packets that enter a source interface, are processed through a switching module, and eventually exit through a destination interface. Packet-based and Time-based are the two sampling methods supported by sFlow agents. Statistical packet-based sampling of switched flows uses a counter that is decremented whenever a packet crosses the switching gear. A sample is taken whenever the counter hits zero and is then reset. A sample involves copying the packet header or a packet feature extraction.

Time-based sampling of network interface statistics on the other hand involves the sFlow agent which is responsible for periodically polling each switching gear for feature extraction.

sFlow provides a standard interface to configure and monitor the sFlow agents using SNMP. This subverts the need to telnet to every switch of the network infrastructure and use its Command Line Interface (CLI) to make subtle changes which can turn out to be overly complex and time-consuming. A MIB module to remotely control the sFlow agents is defined in [43]. The MIB module is Structure of Managed Information (SMI) v2 compliant and can be translated back to SMI v1 without incurring any semantic differences.

*sflow and snmp*

sFlow uses a standard format to send sampled data from the sFlow agent to the sFlow analyzer. The data format is specified using External Data Representation (XDR) [44]. XDR allows compact representation and efficient encoding (or decoding) of the sampled data. The XDR specified sampled data is sent using UDP to a well-known host and port combination specified in the sFlow MIB. UDP is used as a transport mechanism owing to its less stringent buffer requirements and its robustness in delivering traffic information in a timely fashion.

*data format*

sFlow does not provide any security measures to protect the sampled data being transferred to the sFlow analyzer and is therefore at the risk of being eavesdropped. The sFlow analyzer in itself also does not verify the source addresses of the sampled data; as such the sFlow datagrams can easily be spoofed and identified as coming from one of the participating sFlow agents. In essence, now with Flexible NetFlow and IPFIX both providing PSAMP support, the packet sampling novelty of sFlow is losing significance. At one point, the capability of sFlow to monitor traffic at the layer 2 level was seen as an advantage as well, but that is also deemed to lose ground with the frequent adoption of NetFlow-lite.

*limitations and future*

# Part II

# STATE OF THE ART

The semantics and implementation of NFQL has undergone significant changes in the last few years. This section is dedicated to perform an inside-out study of the language and its prototypes, examining all their major (and minor) changes to allow us to better make a pragmatic stand towards its overall improvement. The organization of the section is described below.

In chapter 3 we look into the structure of NFQL by discussing each stage of the processing pipeline with their implementation details. The basic structures of the Flowy python framework are also discussed to understand its current limitations.

In chapter 4 we investigate the possibility of making NFQL map/reduce aware. The chapter starts off with a discussion of current map/reduce frameworks and looks into the ways to help parallelize the Flowy.

In chapter 5 we look into the first attempt to make NFQL implementation comparable with the state-of-the-art flow-analysis tools. After drilling down the performance hit sections of the python code, we witness how getting away with `pytables` and rethinking the rewrite of processing pipeline in C may help make it practically usable.

We conclude this discussion in chapter 6 by introducing a number of real-life application scenarios where NFQL has proved useful. We also look into a few current bleeding edge research projects where we believe NFQL could play a vital role in the near future.

# 3 NFQL AND FLOWY

Flowy [45, 5] is the first prototype implementation of the NFQL [3, 11, 46]. The query language allows to describe patterns in flow-records in a declarative and orthogonal fashion, making it easy to read and flexible enough to describe complex relationships among a given set of flows.

## 3.1 FLOWY PYTHON FRAMEWORK

Flowy is written in Python. The framework is subdivided into two main modules: the validator module and the execution module. The validator module is used for syntax checking and interconnecting of all the stages of the processing pipeline and the execution module is used to perform actions at each stage of the runtime operation.

*pytables and ply*

Flowy uses PyTables [47] to store the flow-records. PyTables is built on top of the Hierarchical Data Format (HDF) library and can exploit the hierarchical nature of the flow-records to efficiently handle large amounts of flow data. The `pytables` module provides methods to read/write to PyTables files. The `FlowRecordsTable` class instance within the module exposes an iterator interface over the records stored in the HDF file. The `GroupsExpander` class instance within the same module on the other hand exposes an iterator interface over the group records and facilitates ungrouping to flow records. In addition, Flowy uses Python Lex-Yacc (PLY) for generating a Look-Ahead LR Parser (LALR) parser and providing extensive input validation, error reporting and validation on the execution modules.

*records*

Flow-records are the principal unit of data exchange throughout Flowy's processing pipeline. The prototype implementation allows the `Record` class (defined in the `record` module) to be dynamically generated using `get_record_class(...)` allowing future implementations to easily plug in support for IPFIX or even newer versions of NetFlow [1] exports. The `FlowToolsReader` class instance (defined in `ftreader` module) provides an iterator over the records defined in `flow-tools` format. This can be plugged into the `RecordReader` class instance (defined in `record` module) to instantly get `Record` class instances.

*parsers and statements*

The `parser` module holds definitions for the lexer and parser. The statements when parsed are implicitly converted into instances of classes defined in the `statement` module. The instances contain meta-information about the parsed statement such as the values, line numbers and sub-statements (if any).

## 3.2 NFQL PROCESSING PIPELINE

The pipeline consists of a number of independent processing elements that are connected to one another using UNIX-based pipes. Each element receives the content from the previous pipe, performs an operation and pushes it to the next element in the pipeline. Figure 7 shows an overview of the processing pipeline. The flow record attributes used in this pipeline exactly correlate with the attributes defines in the IPFIX Information Model specified in RFC 5102 [48]. A complete description on the semantics of each element in the pipeline can be found in [3]

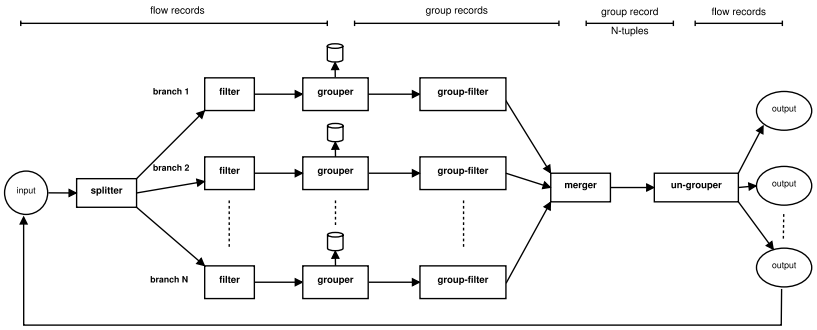



Figure 7: Flowy: Processing Pipeline [11]

*splitter*

The splitter takes the flow-records data as input in the `flow-tools` compatible format. It is responsible to duplicate the input data out to several branches without any processing whatsoever. This allows each of the branches to have an identical copy of the flow data to process it independently.

*splitter implementation*

The `splitter` module handles the duplication of the `Record` instances to separate branches. Instead of duplicating each flow-record to every branch (as specified in the specification), the implementation follows a pragmatic approach by filtering the records beforehand against all the defined filter rules to determine which branches a flow-record might end up in and saves this information in a record-mask tuple of boolean flags. The `go(...)` method in the `Splitter` class then iterates over all the (record, record-mask) pairs to dispatch the records to corresponding branches marked by their masks using the `split(...)` method. The class uses branch names to branch objects mapping to achieve the dispatch.

*splitter validator*

The `splitter_validator` module handles the splitter processing stage. The `SplitterValidator` class within the module uses the `Parser` and `FilterValidator` instances passed to it to create a `Splitter` instance and its child `Branch` instances.

*filter*

The filter performs *absolute* filtering on the input flow-records data. The flow-records that pass the filtering criterion are forwarded to the grouper, the rest of the flow-records are dropped. The filter compares separate fields of a flow-record against either a constant value or a value on a different field of the *same* flow-record. The filter cannot *relatively* compare two different incoming flow-records

*filter implementation*

The `filter` module handles the filtering stage of the pipeline. Since in the implementation the filtering stage occurs before the splitting stage, a single `Filter` class instance suffices for all the branches. Within the `filter` module, each filtering statement is converted into a `Rule` class instance, against which the flow-records are matched. The `Rule` instances are constructed using the (branch mask, logical operator, arguments) tuple. After matching the records against the rules, the record's branch mask is set and is then used by the splitter to dispatch the records to the filtered branches.

*filter validator*

The `filter_validator` module handles the filter processing stage. The `FilterValidator` class within the module uses the `Parser` instance passed to it to create a `Filter` instance once the check on semantical constraints have passed. The constraints involve checking whether records fields referenced in the filter definition exist, whether filters references in composite filter definitions exist and whether duplicate filter definitions are defined.

*grouper*

The grouper performs aggregation of the input flow-records data. It consists of a number of rule modules that correspond to a specific subgroup. A flow-record in order to be a part of the group should be a part of at-least one subgroup. A flow-record can be a part of multiple subgroups within a group. A flow-record cannot be part of multiple groups. The grouping rules can be either absolute or relative. The newly formed groups which are passed on to the group filter can also contain meta-information about the flow-records contained within the group using the aggregate clause defined as part of the grouper query.

*grouper implementation*

The `grouper` module handles the grouping of flow-records data. The `Group` class instance contains group-record's field information required for absolute filtering. It also contains the first and last records of the group required for relative filtering of the group-records. The `AggrOp` class instance handles the aggregation of group-records. The allowed aggregation operations are defined in `aggr_operators` module. Custom-defined aggregation operations are also supported using `-aggr-import` command line argument.

*grouper validator*

The `grouper_validator` module handles the grouper processing stage. The `GrouperValidator` class within the module uses the `Parser` and `SplitterValidator` instances passed to it to create a `Grouper` instance once the check on semantical constraints such as the presence of referenced names and non-duplicate names have passed. Three aggregation operations: `union(rec_id)`, `min(stime)`, `max(etime)` are added by default to each `Grouper` instance.

*group filter*

The group-filter performs *absolute* filtering on the input group-records data. The group-records that pass the filtering criterion are forwarded to the merger, the rest of the group-records are dropped. The group-filter compares separate fields (or aggregated fields) of a flow-record against either a constant value or a value on a different field of the *same* flow-record. The group-filter cannot *relatively* compare two different incoming group-records

*group filter implementation*

The `groupfilter` module handles the filtering of group-records. The `GroupFilter` class within the module iterates over the flow-records within the group and applies filtering rules across them. The filtering rules reuse the `Rule` class from the `filter` module. The flow-records are then added to the time index and stored in a pytables file for further processing. For groups that do *not* have a group-filter defined for them, run through a `AcceptGroupFilter` class instance. The `timeindex` module handles the mapping of the time intervals to the flow-records. The time index is used by the merger stage to learn about the records that satisfy the Allen relations. The `add(...)` method in the `TimeIndex` class is used to add new records to the time index. The `get_interval_records(...)` method on the other hand is used to retrieve records within a particular time interval.

*groupfilter validator*

The `groupfilter_validator` module handles the group-filter processing stage. The `GroupFilterValidator` class within the module uses the `Parser` and `Grouper` instances passed to it to create an instance of `GroupFilter`. The check for the referenced fields is performed against the a ggregate clause defined in grouper statements. The class instance uses the `AcceptGroupFilter` instance in case a branch does *not* have a group filter defined for it.

*merger*

The merger performs relative filtering on the N-tuples of groups formed from the N stream of groups passed on from the group-filter as input. The merger rule module consists of a number of a submodules, where the output of the merger is the set difference of the output of the first submodule with the union of the output of the rest of the submodules. The relative filtering on the groups are applied to express timing and concurrency constraints using Allen interval algebra [4]

*merger implementation*

The `merger` module handles the merging of stream of groups passed as input. It is implemented as a nested branch loop organized in an alphabetical order where every branch is a separate `for-loop` over its records. During iteration, each branch loop executes the rules that matches the arguments defined in the group record tuple and subsequently passes them to the lower level for further processing. The `Merger` class represents the highest level branch loop and as such it must iterate over all of its records since it does not have any rules to impose restrictions on the possible records. The `MergerBranch` on the other hand represents an ordinary branch loop with rules.

The `merger_validator` module handles the merger processing stage. The `MergerValidator` class within the module uses the `Parser` and

`GroupFilterValidator` instances passed to it to create a `Merger` instance once the check on referenced fields and branch names has passed. In addition, the validator also ensures semantic checks on Allen algebra such as whether the Allen relation arguments are correctly ordered, whether the Allen rules with the same set of arguments are connected by an OR and whether each branch loop is reachable by an Allen relation (or a chain of Allen relations) from the top level branch.

*merger validator*

The ungrouper unwraps the tuples of group-records into individual flow-records, ordered by their timestamps. The duplicate flow-records appearing from several group-records are eliminated and are sent as output only once.

*ungrouper*

The `ungrouper` module handles the unwrapping of the group-records. The generation of flow-records can also be suppressed using the `-no-records-ungroup` command line option. The `Ungrouper` class instance is initialized using a merger file and an explicit export order.

*ungrouper implementation*

The `ungrouper_validator` module handles the ungrouper processing stage. The `UngrouperValidator` class within the module uses the `Parser` and `MergerValidator` instances passed to it to create a `Ungrouper` instance. This processing stage does *not* require any validation.

*ungrouper validator*

# 4 FLOWY IMPROVEMENTS USING MAP/REDUCE

Flowy, although clearly setting itself apart with its additional functionality to query intricate patterns in the flows demonstrates relatively high execution times when compared to contemporary flow-processing tools. A recent study [12] revealed that a sample query run on small record set (around 250 MiB) took 19 minutes on Flowy as compared to 45 seconds on `flow-tools`. It, therefore is imperative that the application will benefit from distributed and parallel processing. To this end, recent efforts were made to investigate possibility of making Flowy Map/Reduce aware [12]

## 4.1 MAP/REDUCE FRAMEWORKS

Map/Reduce is a programming model for processing large data sets by automatically parallelizing the computation across large-scale clusters of machines [49]. It defines an abstraction scheme where the users specify the computation in terms of a `map` and `reduce` function and the underlying systems hides away the intricate details of parallelization, fault tolerance, data distribution and load balancing behind an Application Programming Interface (API).

Apache Hadoop [50] is a Map/Reduce Framework written in Java that exposes a simple programming API to distribute large scale processing across clusters of computers. However in order to make Flowy play well with the framework, the implementation either has to use a Python wrapper around the Java API or translate the complete implementation to Java through Jython. Even more since Flowy uses HDF files for it's I/O processing, staging the HDF files properly in the Hadoop Distributed File System (HDFS) [51] and then later streaming them using Hadoop Streaming utility would still be an issue as suggested in [12]

*apache hadoop*

Disco [52] is a distributed computing platform using the Map/Reduce framework for large-scale data intensive applications. The core of the platform is written in Erlang and the standard library to interface with the core is written in Python. Since the `map` and `reduce` jobs can be easily written as Python functions and dispatched to the worker threads in a pre-packaged format, it is less difficult to setup Disco to utilize Flowy as a `map` function. In addition, the usage of HDF files for I/O processing pose no additional modifications whatsoever since the input data files can be anywhere and supplied to the worker threads in absolute paths.

*the disco project*

## 4.2 PARALLELIZING FLOWY

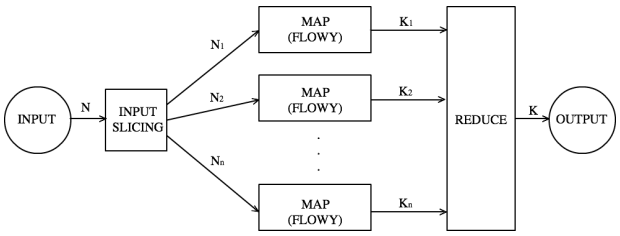



Figure 8: Parallelizing Flowy using Map/Reduce [12]

In an attempt to parallelize Flowy, it was run as a map function on a successful single node Disco installation as shown in 8. Although the setup on a multiple node cluster would be theoretically almost equivalent, Flowy has not yet been tested in such a scenario.

When running several instances of Flowy, it is imperative to effectively slice the input flow-records data in such a way so as to minimize the redundancy in distribution of input. To achieve this, the semantics of the flow-query needs to be examined from the simplest to the most complex cases. However, it is also important to realize that as of now it is not possible to *leave* out any stage in the Flowy's processing pipeline and the following examination was based on such an assumption.

*slicing inputs using only filters*

A flow query that involves only the filtering stage of the processing pipeline can slice its input flow data by either adding explicit export timestamps to allow each branch to skip records or separate out the input flow data into multiple input files for each branch.

*slicing inputs using groupers*

A flow query that also involves groupers and group-filters cannot use static slice boundaries since the grouping rules can be either absolute or relative. As a result, Flowy needs to be made aware of slice boundaries by passing the timestamps as command line parameters. In such a scenario, each branch will skip the pre-slices, whereby the actual slices and the post-slices will be processed to create relevant groups as shown in figure 9. It is advisable to slice the flow-records at low traffic spots to avoid the risk of cutting the records belonging to the same group.

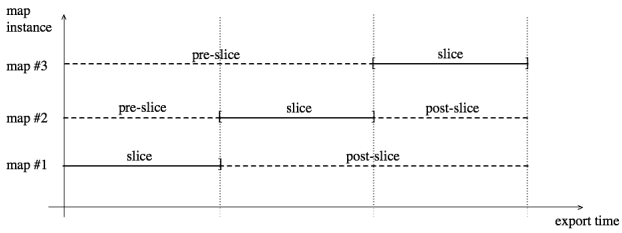



Figure 9: Slice Boundaries Aware Flowy [12]

The idea of skipping pre-slices and sweeping across post-slices can result in many fragmented redundant groups. These can be identified by the `reduce` function by removing the groups that are a proper subset of the previous group in the slice at the cost of additional complexity as shown in figure 10

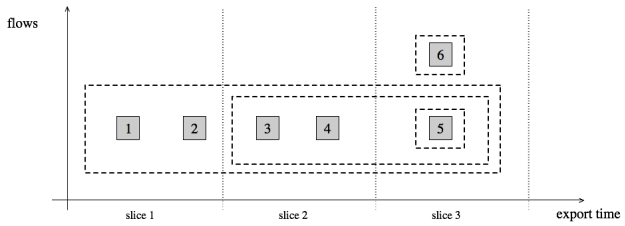



Figure 10: Flowy: Redundant Groups [12]

The relative dependency in the merger stage of the pipeline is even worse, since the comparison needs to take place between groups resulting from the output of separate `map` functions. This calls for inhibiting parallelism up to and including the group-filter stage. As a result each worker thread would return back its filtered groups to the master node, which then would apply the rules of the merger stage to all the received groups at once in a `reduce` function. In such a scenario, although the branch with the longest runtime complexity will become the bottleneck for the merger, the overall runtime would still be dramatically reduced when the number of branches are large.

*slicing inputs using mergers*

A Disco job function is created that contains the `map/reduce` function definitions and a location of an input file of flow-records data. A `sliceIt(...)` function within a newly defined `sliceFileCreator` module is used to create the input file. The function takes a HDF file and number of worker threads as input and writes out the slices in the input file by equally dividing HDF timespan by the number of worker threads.

*flowy as a map function*

In this way, the input file gets slice times for each worker thread in a separate line, which the Disco job function eventually reads to spawn a new `map` function with the slice times passed as arguments. The `map` function then starts an instance of Flowy and passes the slice times and the HDF file as command line parameters for processing.

This required modification to the `flowy_exec` module to add support for extra parameters. The filter stage of the pipeline was modified to allow for skipping of the pre-slices in the flow-records data. The grouper stage was modified as well to restrict creation of new groups that do *not* fall within the passed slice boundaries. However, the modification of the `reduce` function to work with the files pushed out by each Flowy instance of the `map` function to merge groups from each branch and eliminate duplicate records is left open.

# 5

F (V1)

In an attempt to develop a prototype implementation of NFQL which will be comparable with the contemporary flow-record analysis tools, the substitution of the performance hit sections of the python code was thought out. Flowy 2.0 [6] is the first attempt to try rewriting the core of the prototype implementation in C.

## 5.1 FLOWY PERFORMANCE ISSUES

| no. of records | overall | filter | grouper | merger |
|---|---|---|---|---|
| 103k | 1177s | 28s(2%) | 240s(20%) | 909s(77%) |
| 337k | 20875s | 110s(1%) | 2777s(13%) | 17988s(86%) |
| 656k | 70035s | 202s(0%) | 8499s(12%) | 61334s(87%) |
| 868k | 131578s | 274s(0%) | 15913s(12%) | 115391s(87%) |
| 1161k | 234714s | 1212s(1%) | 25480s(11%) | 208022s(88%) |

Table 2: Runtime Breakup of Individual Stages [6]

The runtime breakup of individual stages of the processing pipeline as shown in table 2 reveal that the grouper and merger incur a massive performance hit. A quick investigation hints towards usage of large deep nested loops in the merger with a worst-case $O(n^3)$ runtime complexity.

*deep nested loops*

In addition, pushing the flow-records data from one stage of the pipeline to another involved deep copying of the whole flow data whereby a mere passing across of a reference across a pipeline in a branch would have sufficed. Similar behavior is visible when the grouper when passing group records saved the individual flow-records in a temporary location tagged with the groups and/or subgroups they belonged to.

*deep copy of flow records*

The decision to use PyTables to read and write flow-records in HDF format also added to the complexity. Since, the input flow-records were most of the time in either `flow-tools` or `nfdump` file-formats, each time they had to be converted into HDF file formats prior to Flowy's execution which was unnecessary.

*pytables and hdf*

## 5.2 PRELIMINARY IMPROVEMENTS

The python parser written in PLY and the validators written for each stage of the processing pipeline that check for semantics correctness were left unmodified, since their execution time was invariant of the size of the input data and slightly varied on the query complexity.

Thread affinity masks were set for each new thread created to delegate the thread to a separate processor core. `try/except` blocks were narrowed down to only code that needed to be exception handled. A test-suite was developed with few sample queries and input traces to validate Flowy's results for regression analysis. A `setup.py` script was written to facilitate installation of Flowy and its dependencies and `options.py` was replaced with `flowy.conf` configuration file with the standard human-readable key-value pairs. The command line option handling was switched from `optparse` to `argparse` module and a switch was added for easy profiling. The profiling output was modified as well to allow standard tab delimiters which can be easily parsed by other tools. The flow query was also extended to allow file contents to be supplied using `stdin`. Variable names that are now part of Python identifiers were renamed.

*affinity masks, easier installation and configuration, better profiling and testing, extended command line switches*

A custom C library was written to directly read/write data in the `flow-tools` format to provide a drop-in replacement for PyTables and overcome the overhead of format conversions. The library sequentially reads the complete flow-records into memory to support random access required for relative filtering. Each flow-record is stored in a `char` array and the offsets to each field are stored in a separate `struct`. The array of such records are indexed allowing fast retrieval in $O(1)$ time. The C library was connected to the Python prototype using Cython [53][54]. This allowed the flow-records to be easily referenced by an identifier, thereby giving away the need to every time copy all the flow-records when moving ahead in the processing pipeline. Cython was used since it allowed to write C extensions in a Pythonic way by strong-typing variables, calling native C libraries and allowing usage of pointers and structs, thereby providing the best of both worlds [55].

*a custom c library to replace pytables, cython to connect the library to python*

## 5.3 RULE INTERFACES

A design decision was made to attempt to rewrite the entire processing pipeline in C. However, currently the core can only run absolute filters and cannot parse the flowquery file. Therefore the execution is triggered by a tedious manual filling of the `struct` by the contents of the query. A filter stage `struct` is shown in listing 3. The field to be filtered is indicated using a `field_offset` and `field_length` in the `char` array of a records. The value to be compared against with is also supplied which can be either a static value or another field of a record. `func` is a function pointer to the operation that is to be carried out on

*filter stage struct*

a record whose record identifier is passed to it. The filter runs in $O(n)$ time as it needs to traverse through all the records of the `char` array.

```
struct filter_rule {
  size_t field_offset;
  uint64_t value;
  uint64_t delta;
  bool (*func)(
    char *record,
    size_t field_offset,
    uint64_t value,
    uint64_t delta);
};
```

Listing 3: Filter Rule Struct [6]

## 5.4 EFFICIENT RULE PROCESSING

The comparison operations, previously were required to make costly checks on the length of the field type passed to them, to be able to make appropriate casts. Such checks are now no longer needed. F ($v1$) now allows filtering of records via two methods: using specialized comparison functions or using one main fall through `switch` statement. The implementation defaults to using specialized comparison functions to encourage modularity in source code.

```
bool filter_eq_uint8_t(...);
bool filter_eq_uint16_t(...);
...
```

Listing 4: Auto Generated Comparison Functions [6]

In the default method, there is a comparison function defined for every possible field length (33) and comparison operations (19). These functions are generated using a Python script [1] and are declared/defined in `auto_comps.{h,c}` as shown in listing 4. The rule definitions are now able to make calls using a function name derived from the combination of field length, delta type and operation. This subverts the need to define complex branching statements and reduces complexity. In the other method, the logic is executed by comparing the field length and the operation by falling through a huge switch statement. Such a huge switch statement is again generated using the same Python script and is defined in `auto_switch.c` as shown in listing 5.

*using function pointers and switch cases*

```
switch (group_modules[k].op) {
  case RULE_EQ | RULE_S1_8 | RULE_S2_8 | RULE_ABS:
  case RULE_EQ | RULE_S1_8 | RULE_S2_8 | RULE_REL:
  ...
```

Listing 5: Auto Generated Switch Statement [6]

1 `scripts/generate-functions.py`

## 5.5 FASTER GROUPING APPROACH

*possible grouping approaches*

A typical grouper module is shown listing 6. In order to be able to make comparisons on field offsets, the grouper initially creates a copy of the pointers in the filtered recordset. A näive approach is to linearly walk through all the pointers against each pointer in the copy leading to a complexity of $O(n^2)$. A smarter approach is to put the copy in a hash table and then try to map each pointer while walking down the recordset once, leading to a complexity of $O(n)$. The hash table approach, although will work on this specific example, will fail badly on other relative comparisons.

```
1 grouper g1 {
2   srcIP = srcIP
3   dstIP = dstIP
4 }
```

Listing 6: A Grouper Module

*using quick sort and binary search*

It is clear that a middle ground compromise was needed. As a result, using a binary search after a quick sort on the filtered recordset was thought out. To achieve this, the array of pointers to the copy are sorted according to the offset on the right side of the comparsion of the first grouping rule. Such a sorted array of pointers is then traversed linearly to find unique values. This helps the grouper perform binary searches to find records that will eventually group together. The preprocessing step takes $O(n * lg(n)) + O(n)$ in the worst case, with a $O(n * lg(k))$ for binary search on the entire recordset. The implementation of this idea is currently broken and segfaults at multiple stages. In essence, this approach is not currently used in practise and quite some effort is needed to bring it to life.

## 5.6 BENCHMARKS

| Number of records | Flowy | F (v1) |
|---|---|---|
| 103k | 1177s | 0.3s |
| 337k | 20875s | 3.4s |
| 656k | 70035s | 13s |
| 868k | 131578s | 23s |
| 1161k | 234714s | 86s |

Table 3: Flowy vs F(v1) [6]

The benchmarks in table 3 show the relative comparison of Flowy [5] with F (v1) [6]. As the author appropriately points out, an accurate

comparison of Flowy with F $v1$ cannot be done accurately, since, the the C execution engine lacks many feature-set and pipeline stages. However, running a common flow query that can be run on both the implementations was used and it was not surprising that F $(v1)$ was in orders of magnitude faster in comparison.

*vs flowy*

```
src port 80
src port 80 or dst port 25
src port 443 or (src port 80 and dst port 25)
```

Listing 7: F$(v1)$ vs `flow-tools, nfdump` [6]

In order to evaluate how well F now performs with these added improvements, the authors decided to compare it with the state-of-the-art flow-processing tools: `flow-tools` [56] and `nfdump` [57]. A set of 3 queries involving only absolute filters was defined as shown in listing 7 and evaluated on a set of $500K - 10M$ flow-records. It turned out that F $(v1)$ performed as well if not better than the other flow tools.

*vs flow-tools and nfdump*

# 6 NFQL: APPLICATIONS

NFQL helped to underpin a number of recent research efforts to solve real-world application problems that were deemed difficult before. This was possible due to the power and flexibility of the flow-query language to suit itself from generic to specific needs thereby opening doors of innovation. This section documents such efforts that use the in-house flow query language as well as a few others that exploit the flow level characteristics of the traffic patterns in general.

## 6.1 APPLICATION IDENTIFICATION USING FLOW SIGNATURES

The idea behind this study was to identify applications using flow traces on a network by analyzing potential left-behind signatures that describe them [58, 16]. This was based on the hypothesis that each application type generates unique flow signatures that might work as a fingerprint feature. To achieve this, a collection of network traces were recorded from several users and subsequently analyzed. The identified signatures were formalized by writing flow queries that were executed on Flowy [45]. Several separate instances of the network traces were queried to evaluate the approach and come to a conclusion.

```
splitter S {}

...

merger M {
  module m1 {
    branches A, B
    A.srcip = B.srcip
    A o B OR B o A
  }
  export m1
}

ungrouper U {}

"input" -> S
S branch A -> F_SSDP -> G_SSDP -> GF_SSDP -> M
S branch B -> F_NAT_PMP -> G_NAT_PMP -> GF_NAT_PMP -> M
M -> U -> "output"
```

Listing 8: Skype Application Signature [16]

A formalized Flowy query to identify Skype from the flow traces for an instance is described in listing 8. The filter, grouper and group-filter sections of each branch are shown separately in listings 10 and 9. Additional queries identifying variety of web browsers, mail clients, IM clients and media players can be found in [16].

```
filter F_SSDP {
  dstport = 1900
  port = protocol("UDP")
  dstip = 239.255.255.250
}

grouper G_SSDP {
  module g1 {
    srcip = scrip
    dstip = dstip
    srcport = srcport
  }
  aggregate srcip, sum(bytes) as B
}

groupfilter GF_SSDP {
  B = 321
}
```

Listing 9: Branch A [16]

```
filter F_NAT_PMP {
  dstport = 5351
  port = protocol("UDP")
}

grouper G_NAT_PMP {
  module g1 {
    srcip = scrip
    dstip = dstip
  }
  aggregate srcip, sum(bytes) as B
}

groupfilter GF_NAT_PMP {
  B = 160
}
```

Listing 10: Branch B [16]

*skype application signature*

The filter `F_SSDP` is used to identify the four identical UDP multicast messages the client sends out using Simple Service Discovery Protocol (SSDP) [59]. Similarly `F_NAT_PMP` filter is used to identify four Network Address Translation Port Mapping Protocol (NAT-PMP) [60] messages sent over UDP. The groupers `G_SSDP` and `G_NAT_PMP` group together flow records with the same source and destination IP address and the aggregate clauses describe the meta information with unique source IP addresses for each group records along with the total bytes carried within each group. The meta information is used to further filter the group-records in `GF_SSDP` and `GF_NAT_PMP` modules.

| UserID | Skype | Opera | Amarok | Chrome | Live |
|---|---|---|---|---|---|
| u0 | ✔ | ❍ | ✖ | ❍ | ❍ |
| u1 | ✔ | ❍ | ❍ | ❍ | ❍ |
| u2 | ❍ | ❍ | ❍ | ❍ | ❍ |
| u3 | ✔ | ❍ | ✖ | ❍ | ❍ |
| u4 | ❍ | ❍ | ❍ | ❍ | ❍ |
| u5 | ✔ | ❍ | ✔ | ✔ | ❍ |
| u6 | ❍ | ❍ | ❍ | ❍ | ❍ |
| u7 | ❍ | ✔ | ✔ | ❍ | ❍ |
| u8 | ❍ | ❍ | ❍ | ❍ | ❍ |
| u9 | ✔ | ✔ | ✔ | ✔ | ❍ |

Table 4: Application Flow Signatures: Results [16]

*success rate*

The identification results obtained from the analysis of flow-traces from ten unique users are compiled together in table 4. The results demonstrate a success rate of 96% for the five applications tested. This study reveals that it is possible to identify applications from their network flow fingerprints and is a first step towards automating the complete process whereby machine learning techniques would be used to automatically generate flow-queries and identify new applications and even more so newer versions of the same application.

## 6.2 CYBERMETRICS: USER IDENTIFICATION

The idea of identification of users based on biometric patterns such as keystroke dynamics [61], mouse interactions [62] or activity cycles in online games [63] has been long known. This study takes the idea even further by using flow-record patterns as a characteristic (cybermetrics) to identify a user on a network [7, 64]. Such a cybermetric user identification can be used for the purpose of providing secure access, system administration and network management. The feature extraction module of the analyzer as shown in figure 11 uses three distinct feature sets that could possibly be used to identify a user from a flow-record trace.

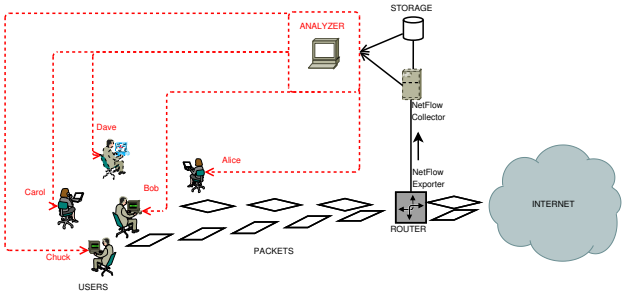



Figure 11: Cybermetrics: Overview [7]

Initial research efforts started with identifying application signatures in flow-records in [58, 16] and became relevant because different people have different preferences in the applications they use and as such a set of applications in flow-records is a characteristic feature of a user. Flowy queries were formalized for four different set of applications and tested against a known set of users. The evaluation results of the derived queries as shown in table 4 demonstrated a strong evidence of presence (or absence) of applications and thereby provided an eventual marker for user identification.

*application signatures*

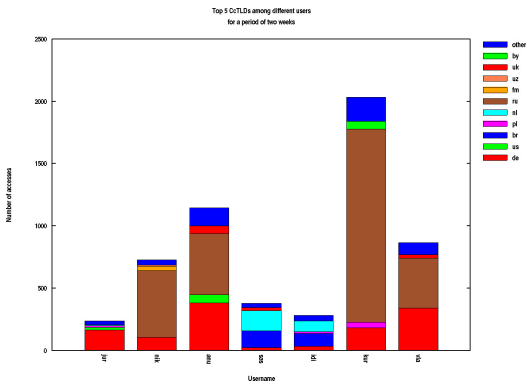



Figure 12: Geographical Preferences [7]

*geographical preferences*

The authors also looked into the geographical affiliations of different users by analyzing the Country Code Top-Level Domain (ccTLD) of the browsed websites. They proposed a hypothesis that a user's origins strongly influences their browsing activity. The analysis of the results established that the top five visited ccTLDs constituted more than 85% of the overall number of a user's visits as shown in figure 12.

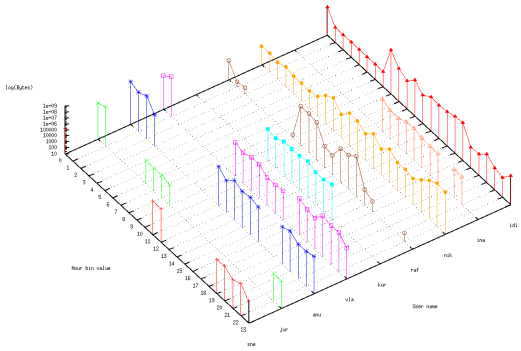


Figure 13: Daily Distributions for HTTP Traffic [7]

*flow-record statistics*

In the end, the authors introduced a proof-of-concept method of user differentiation based on statistical features. These features considered daily distributions of parameters that were based on different port numbers. For instance, figure 13 shows the daily distribution of different users based on their HTTP traffic usage. It was also witnessed that the time duration also played a key role in the process of feature formation, whereby the number of longer flows increased with the duration and consequently resulted in higher cross-correlations as shown in figure 14

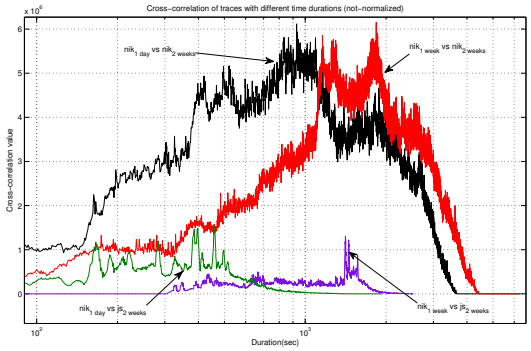



Figure 14: Cross Correlation of Traces [7]

This research is a first attempt to identify users based on their network flow fingerprints and the on-going effort is focussing on sophisticated machine learning techniques to learn behavioral patterns of known users to identify them in the future from their current network-flow traces.

## 6.3 IPV6 TRANSITION FAILURE IDENTIFICATION

The IPv4 address space depletion is upon us and has become more imminent in the last few years. While IPv6 can readily expand the extent of the Internet, deploying it alone is clearly not a solution today and hence there are a continuum of transitioning solutions that would help in this migration. In this study [13] we evaluated the compatibility of popular applications with such transitioning solutions: NAT64 [65] and Dual-Stack Lite [66]. The goal was to find potential failures by identifying application failure signatures left behind in the flow-record traces using Flowy. These failure signatures could later be used by service providers to automate the detection and eventually shorten the deployment verification cycle.

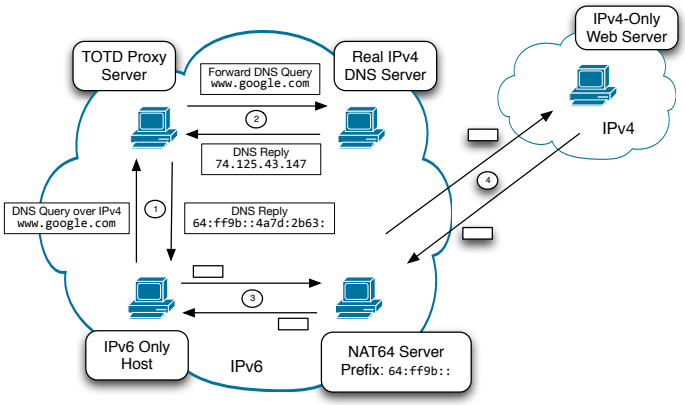



Figure 15: NAT64 Setup [13]

In the NAT64 deployment testbed as shown in figure 15, the authors witnessed failure in 3 applications: Skype, OpenVPN and Transmission. Flowy queries were defined to establish failure signatures for each application. A formalized Flowy query to identify Skype failure signature for an instance is described in listing 13. The filter sections of each branch are shown separately in listings 11 and 12.

*application operation under NAT64*

```
filter f-mDNS {
  dstport = 5353
  srcport = 5353
  dstip = 224.0.0.251
  duration > 1 sec
  duration < 5 sec
}
```

Listing 11: Branch A [13]

```
filter f-login1 {
  dstport = 443
  duration > 55 sec
  duration < 59 sec
}
```

Listing 12: Branch B-C-D [13]

Filter `f-mDNS` is used to filter multicast messages used by Skype to discover clients in the link-local network sent to the destination IP address-port combination (224.0.0.251 : 5353). Filter `f-login1` is used

*skype failure signature*

to filter 3 unsuccessful attempts to contact the login server each in a separate branch. The source port and the duration increases with decreasing number of packets for each subsequent flow.

```
splitter S {}

...

grouper g-login1 {
  module g1 {
    srcport = srcport
    dstip = dstip
    dstport = dstport
  }
  aggregate srcip, dstip, srcport, td,
  sum(packets) as pckt-sum, count(rec_id) as n
}

merger M {
  branches mDNS, LOGIN1, LOGIN2, LOGIN3

  LOGIN1.srcip = LOGIN2.srcip
  LOGIN2.srcip = LOGIN3.srcip
  LOGIN1.dstip = LOGIN2.dstip
  LOGIN2.dstip = LOGIN3.dstip

  LOGIN1.srcport = LOGIN2.srcport rdelta 1
  LOGIN2.srcport = LOGIN3.srcport rdelta 1

  LOGIN1.pckt-sum > LOGIN2.pckt-sum
  LOGIN2.pckt-sum > LOGIN3.pckt-sum

  mDNS.td < LOGIN1.td
  mDNS.td < LOGIN2.td
  mDNS.td < LOGIN3.td

  mDNS < LOGIN1
  mDNS < LOGIN2
  mDNS < LOGIN3
}

"input"-> S
S br mDNS -> f-mDNS -> g-mDNS -> gf-mDNS -> M
S br LOGIN1 -> f-login1 -> g-login1 -> gf-login1 -> M
S br LOGIN2 -> f-login2 -> g-login2 -> gf-login2 -> M
S br LOGIN3 -> f-login3 -> g-login3 -> gf-login3 -> M
M -> U -> "output"
```

Listing 13: Skype Failure Signature [13]

The groupers count the number of packets in each flow-records using `pckt-sum` which is later utilized by the merger stage to distinguish the branches. The group-filter stage finally is used to filter out groups with more than one record.

*failure when using IPv4 literals*

The NAT64 translation works when the applications running on the IPv6-only host explicitly make DNS requests to allow DNS64 to capture and masquerade them as fake IPv6 addresses that are eventually sent to the NAT64 box. If the applications use IPv4 literals to contact the servers, DNS64 is skipped and therefore NAT64 cannot perform the translation. This was reason behind the failure of the other two applications (OpenVPN and Transmission).

This study sets across a baseline to automate the failure detection by formalizing queries against flow-records. While a more exhaustive study encompassing wider set of applications still needs to be carried out, it is imperative that this unique approach is not just limited to IPv6 transition technologies, but can be utilized to identify failures in more generic cases.

## 6.4 OPENFLOW

OpenFlow [67] is an open standards protocol that runs between an Ethernet switch and an OpenFlow controller (a software designed to run on a x86 server) to securely manage the forwarding plane of the switch over the network as shown in figure 16. This enables the controller to push out policies that dictate how to process flow-records crossing the networking infrastructure to eventually improve bandwidth, reduce latency and save power.

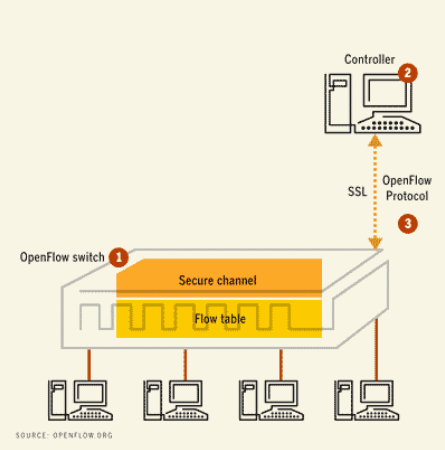



Figure 16: OpenFlow Architecture [14]

OpenFlow initially started as a way to allow researchers to experiment with new ideas in sufficiently realistic settings by allowing the live production networking gear to open a narrow programmable external interface to it whereby at the same time keeping the inner workings of the gear hidden and proprietary. The idea took off outside the academic setting in recent years with the need of data centers requiring to run large-scale map/reduce jobs with full cross-sectional bandwidth. Such a requirement called for flexible forwarding and programmable networks to meet the application-specific needs. Today, the commercial underpinning of OpenFlow are driven by the Infrastructure as a Service (IaaS) providers trying to virtualize their network architecture to solve the issue of multi-tenancy to implement Network as a Service (NaaS) architectures [68].

*motivation*

An OpenFlow switch manages a flow table to keep record of the flows crossing it. A flow table contains a packet header, an action and some statistical information about the flow. OpenFlow defines a common set of methods to program such flow tables irrespective of the way different vendors internally defined them. This allows a network administrator to partition the incoming traffic into numerous Virtual Local Area Networks (vLANs) thereby isolating the production and several experimental networks at the layer 2 level. Now, with the a complete suite of OpenFlow software stack defined on top of the

*programming flows*

controller, such a power is also available at the hands of the developers that gives them the ability to control the flow tables themselves and even decide the routes for their flow.

The OpenFlow protocol in itself is like an x86 instruction-set by itself. However, there is a lot of innovation possible at the software stack layer that can be built on top the controller that exposes the API and pushes this low-level instruction-set to the networking gear. For instance, the stack can deploy network-wide policies and administer Access Control Lists (ACLs) for each incoming flow or allow seamless handover of mobile hosts by rerouting requests making the networking gear location-aware in itself. As such, it is conspicuous that the possibilities are endless and is the beginning of a kick-start of a new internet evolution.

*software stack*

It is not difficult to anticipate that Flowy could be of much use for OpenFlow. It could be envisaged that the controller would define Flowy queries to get to a specific flow-entry in the flow table before sending action level instructions to the networking gear. In addition, Flowy could be extended to allow flow manipulation constructs to define the action instructions themselves which can be sent out by the controller. In a future outlook, Flowy can even be envisioned to allow procedural constructs (variables, functions, loops, conditions) around the declarative query to add power to what can be retrieved or sent back to the switches.

*flowy and openflow*

## 6.5 FLOW LEVEL SPAM DETECTION

| Feature | Description |
|---|---|
| Pkts | Packets |
| Rxmits | Retransmissions |
| RSTs | Packets with RST bit set |
| FINs | Packets with FIN bit set |
| Cwndo | Times o-window advertised |
| CwndMin | Minimum window advertised |
| MaxIdle | Maximum idle time between packets |
| RTT | Initial round trip time estimate |
| JitterVar | Variance of inter-packet delay |

Table 5: Features in Spam Flow [15]

Classical methods to mitigate spam such as content filtering and reputation analysis utilize the the weakness of spam messages and the places from where they originate from. Though currently effective, it's only a matter of time when spammers find a way to subvert around these vantage points. In this study [15, 69], the authors analyze the transport level characteristics of the email flows to differentiate between spam and legitimate email. These characteristics exploit the fundamental weakness of each spam: the requirements to send large amounts of the same email on resource constrained links owned by

compromised botnets which is unlikely to change in the near future. They reason that a spammer's traffic is more likely to experience TCP timeouts, retransmissions, resets and variable Round Trip Time (RTT) estimates. Based on this hypothesis they extract 13 learning features as shown in table 5 to formalize a machine learning problem.

*spamflow features*

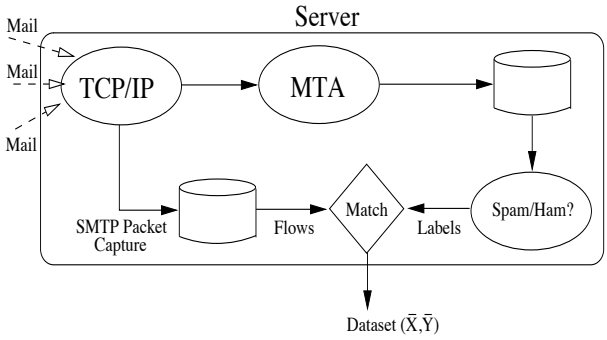



Figure 17: Spam Flow Classifier [15]

The data collection methodology is depicted in figure 17 where TCP packets corresponding to email messages are extracted and examined on a per-email flow basis. The packets in an email flow are coalesced together by using TCP port numbers in the email headers. Using machine learning feature selection, a spam classifier is built that matches each flow to a binary spam/ham ground-truth label. Support Vector Machines (SVMs) [70] are used for classification and Greedy Forward Fitting (FF) [71] is used for feature selection to find a set of features that provide the least training error. It turns out the classifier achieves 90% accuracy with 78% detectability of false-negatives from a particular content filter.

*a spam classifier*

One possible limitation of this approach is the inability to distinguish between botnets sending large quantities of spam and innocent busy hosts that happen to be on a congested network. This is most probably because of the näive Simple Mail Transfer Protocol (SMTP) flow aggregation and filtering. We believe, that Flowy can help overcome this shortcoming by automated flow-queries generated by another trained classifier that filters out these innocent hosts before passing them to the spam classifier thereby reducing the number of false negatives.

*flowy and spamflow*

This study presented a content and IP reputation agnostic scheme based on SMTP flow-level analysis of traffic stream. It is imperative, augmented with Flowy capabilities, this approach can be extended to identify any botnet generated traffic. Such a novel approach could then be used to also identify phishing attacks, scam infrastructure hosting, Distributed Denial of Service (DDoS), dictionary attacks and Completely Automated Public Turing Test to Tell Computers and Humans Apart (CAPTCHA) solvers.

*extending spamflow*

# Part III

# IMPLEMENTATION AND EVALUATION

This section dives deep into the implementation of the next iteration F($v2$) of the NFQL execution engine. This iteration provides a functional and robust implementation of the complete processing pipeline. It is flexible to allow runtime evaluation of NFQL flowqueries and is backed up with automated builds, regression and benchmarking suites. The organization of the section is described below.

In chapter 7 we begin by analyzing the current implementation snapshot. The analysis involves reverse-engineering the parser and the execution engine. This is followed by a discussion on an early visualization of a complete engine refactor using abstract objects and reasonings on an envisaged high-level execution workflow.

In chapter 8 we introduce the inner workings of the code. It begins by an explanation of how each stage is brought to life to result in a robust pipeline exeuction. The grouper and merger internals, being non-trivial are explained in more details. This is followed by an explanation on how the engine is engineered to make runtime query evaluation a possibility. The chapter concludes with a discussion on automated builds using `cmake` and a full-fledged regression-test suite using `python` scripts.

In chapter 9 we begin by a discussion of the execution engine profiling results. This is followed by how `python` scripts are used to completely automate the process of benchmarking the engine against SiLK. A number of queries are run a collection of varied-sized flow traces. The chapter concludes with a discussion of the evaluation graphs. We conclude the discussion in chapter 10 by documenting the future work by segregating it into major goals and minor issues that need to be resolved to carry the implementation work forward.

# 7 DESIGN

With a software that has undergone such significant iterative lifecycles over the past few years, it is imperative to understand and analyze the inner working of the application before diving in to add more functionality. Reverse engineering the current snapshot not only helped identify glitches to give a head start with preliminary improvements, but also enabled understanding the design of the eventual execution workflow. It also helped in early visualization of a complete engine refactor to introduce abstract objects that made it possible to evaluate the flow query at runtime. This chapter starts off with the analysis to set a platform for reasoning out the design patterns and the user functionality envisaged from the finished product.

## 7.1 FLOWY PARSER AND F(v1) ENGINE ANALYSIS

*compilation and runtime issues*

Since both the parser and the engine were developed in an isolated sandboxed environments, an extensive validation of how their functionality (or errors) would regress was always needed. In this pursuit, the first challenge was to get the F(v1) engine to compile smoothely. Since, the engine was using linux-specific integer types to read the flow record offsets, its compilation was an issue on other Unix flavors. As such moving to C99 [72] fixed-width integer types increased portability. In addition a number of extraneous files that were not part of the build plagued the source directory and were removed after thorough inspection. Boolean enums were replaced by C99 `bool` types and include guards were added in the headers to remove circular dependencies. These changes led to succesful compilation of the engine and an initial run iteration is shown in listing 14. It appears that the execution engine can read the flow records in memory and successfully filter records in each thread. However, it segfaults at the grouper stage, thereby ending the execution.

```
$ ./flowy2 < trace.ft
number of filtered records: 556
number of filtered records: 166
segmentation fault  ./flowy2 < trace.ft

(gdb) backtrace
...
#1  0x00000001000134fa in build_record_trees
#2  0x00000001000138a0 in grouper
#3  0x0000000100011eb9 in branch_start
...
```

Listing 14: F(v1): Segmentation Fault

*missing pipeline stages, hardcoded rules, assumptions*

In addition, the implementations for group filter, merger and ungrouper are missing. A major issue is that the complete flow query is hardcoded in pipeline structs as shown in listing 15. Similar rules are hardcoded for each branch. In addition the functions that evaluate the filter and the grouper rule also assume offsets of a specific integer type and result in undefined behavior once the parameters in the flow query are altered.

```
struct filter_rule filter_rules_branch1[1] = {
  { data->offsets.dstport, 80, filter_eq_uint16_t },
};

struct grouper_rule group_module_branch1[2] = {
  { data->offsets.srcaddr, data->offsets.srcaddr, grouper_eq_uint32_t_uint32_t_rel },
  { data->offsets.dstaddr, data->offsets.dstaddr, grouper_eq_uint32_t_uint32_t_rel }
};

struct grouper_aggr group_aggr_branch1[2] = {
  { data->offsets.srcaddr, aggr_static_uint32_t },
  { data->offsets.dstaddr, aggr_static_uint32_t },
};

binfos[0].branch_id = 0;
binfos[0].filter_rules = filter_rules_branch1;
binfos[0].num_filter_rules = 0;
binfos[0].group_modules = group_module_branch1;
binfos[0].num_group_modules = 2;
binfos[0].aggr = group_aggr_branch1;
binfos[0].num_aggr = 2;
```

Listing 15: F(*v*1): Flow Query Hardcoded in Pipeline Structs

*reverse-engineering*

To analyze the call flow and data structure collaboration and dependency, the execution engine was reverse engineered to generate Unified Modeling Language (UML) using `doxygen`. A similar technique was used to generate UML for the parser using `pylint` and `pyreverse`. Makefile targets were added to ease documentation generation for future developers as shown in listing 16.

```
[engine] $ make doc
[parser] $ make doc
```

Listing 16: F(*v*2): High Level Documentation

*arguments parsing in parser*

Flowy parser tools assumed correct number and format of command line arguments and poorly exited out of execution with `IndexError` exceptions. The python `argparse` module is now used to exit gracefully with usage instructions on bad input as shown in listings 17.

```
[parser] $ python src/flowy.py
usage: flowy.py [options] query.flw

[parser] $ python src/ft2hdf.py
usage: ft2hdf.py [options] input_path1 [input_path2 [...]] output_file.h5

[parser] $ python src/printhdf.py
usage: printhdf.py trace.h5

[parser] $ python print_hdf_in_step.py
usage: print_hdf_in_step [options] input_files
```

Listing 17: Flowy Interfaces

It was clear from the generated UMLs that the current snapshot required multiple stages of refactoring before it can be deemed maintainable. As such forward declarations were removed and thus arising circular dependencies were resolved by reorganizing the code in multiple files. For instance, a `base` header was added to include common library headers as shown in figure 18. An `error_functions` module was added to avoid plaguing error handlers everywhere. Each stage of the pipeline was moved into its separate module, while utility functions were moved to the `utils` module. All the pipeline structs were also moved to a specific `pipeline` header to increase readability.

*minor refactor*

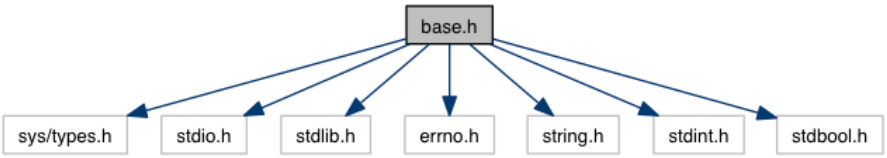



Figure 18: F(*v2*): Base Header

## 7.2 EXECUTION WORKFLOW AND ABSTRACT OBJECTS

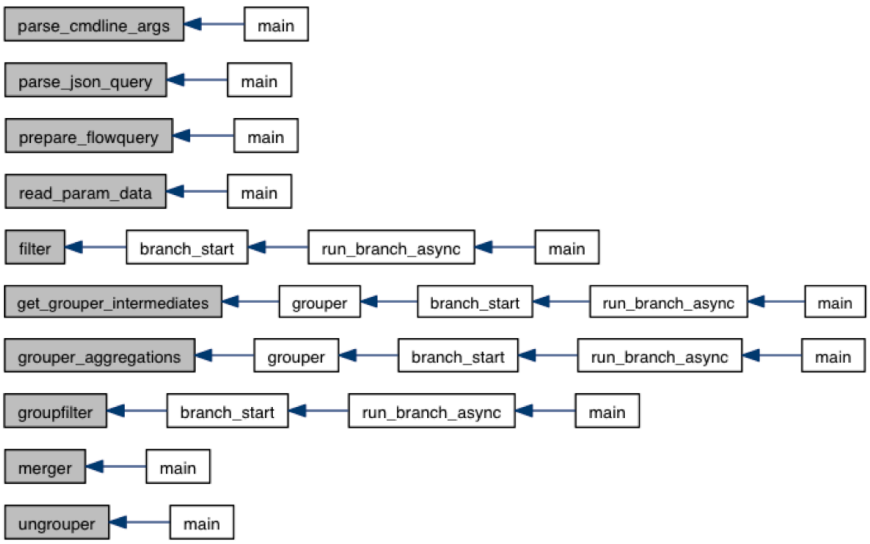



Figure 19: F(*v2*): Execution Engine Workflow

In order to keep the codebase maintainable, it was essential to design the execution engine workflow in such a way so as to naturally map it to the original pipeline model specification [11] as shown in figure 19. Each stage of the pipeline is a separate independent module blackboxed into one public interface function. Each stage is

also wrapped around conditional compilation macros to allow them to be easily enabled/disabled during development if desired so.

```
{
  "branchset": [
    { "filter": {
        "dnf-expr": [
          { "clause": [
              { "term": { ... } },
              { "term": { ... } }
            ]
          }
        ]
      },
      "grouper": {
        "dnf-expr": [
          { "clause": [
              { "term": { ... } },
              { "term": { ... } }
            ]
          }
        ],
        "aggregation": {
          "clause": [
            { "term": { ... } },
            { "term": { ... } }
          ]
        }
      },
      "groupfilter": {
        "dnf-expr": [
          { "clause": [
              { "term": { ... } },
              { "term": { ... } }
            ]
          }
        ]
      }
    },
  ],
  "merger": {
    "dnf-expr": [
      { "clause": [
          { "term": { ... } },
          { "term": { ... } }
        ]
      }
    ]
  },
  "ungrouper": {}
}
```

Listing 18: F($v2$): Flow Query Composition using DNF Expressions

*flowquery composition using dnf expressions*

A DNF is a disjunction of conjunctive clauses. The elements of the conjuctive clauses are terms. Each stage of the pipeline is represented in the flowquery as a DNF expression as shown in listing 18. The clauses in the DNF are OR'd together, while the terms in each clauses themselves are AND'd. This terminology using clauses and terms is more useful and intuitive over modules, submodules and rulesets used previously by NFQL and its implementations.

```
struct flowquery {
  size_t                                num_branches;
  size_t                                num_merger_clauses;

  struct branch**                       branchset;
  struct merger_clause**                merger_clauseset;
  struct merger_result*                 merger_result;
  struct ungrouper_result*              ungrouper_result;
};
```

Listing 19: F($v2$): Flow Query Struct

The abstract objects that store the JSON query and the results that incubate from each stage are designed to be self-descriptive and hierarchically chainable. The complete JSON query information for instance, is held within the `flowquery` struct as shown in listing 19. Each individual branch of the flowquery itself is described in a `branch` struct. A collection of these `branch` structs are referenced in the parent `flowquery` struct. All the clauses of the DNF expression are clubbed into `X_clauseset`, where `X` can be any stage as shown in listing 20.

*flowquery and branch struct*

```
struct branch {

/* ------------------------------------------------------------------------*/
/*                                inputs                                   */
/* ------------------------------------------------------------------------*/

  int                                       branch_id;
  struct ftio*                              ftio_out;
  struct ft_data*                           data;

  size_t                                    num_filter_clauses;
  size_t                                    num_grouper_clauses;
  size_t                                    num_aggr_clause_terms;
  size_t                                    num_groupfilter_clauses;

  struct filter_clause**                    filter_clauseset;
  struct grouper_clause**                   grouper_clauseset;
  struct aggr_term**                        aggr_clause_termset;
  struct groupfilter_clause**               groupfilter_clauseset;

/* ------------------------------------------------------------------------*/




/* ------------------------------------------------------------------------*/
/*                                output                                   */
/* ------------------------------------------------------------------------*/

  struct filter_result*                     filter_result;
  struct grouper_result*                    grouper_result;
  struct groupfilter_result*                gfilter_result;

/* ------------------------------------------------------------------------*/
};
```

Listing 20: F(*v*2): Branch Struct

A call to the public interface function of each stage returns a `X_result` struct object as shown in listing 21. The `X_result` objects encapsulate all elements of the stage into one single entity as shown in listing 21 to easily allow them to be passed around and for easy maintainbility of in-memory object stores.

*public interfaces*

```
struct grouper_result*
grouper(...) {...}

struct groupfilter_result*
groupfilter(...) {...}

struct merger_result*
merger(...) {...}

struct ungrouper_result*
ungrouper(...) {...}
```

Listing 21: F(*v*2): Public Interfaces

*result structs*

Each result struct holds information about the number of flow records that passed the stage and pointers to each such flow records. Since the group filter and merger stages do not work on the individual flows but on a collection; they take the `group` struct that encapsulates a collection of similar flows as input arguments. It is important to realize that the flow records themselves are never carried forward from each stage to its subsequents, but only offset pointers to the original flow trace are.

```
struct filter_result {
  uint32_t                                  num_filtered_records;
  char**                                    filtered_recordset;
};

struct grouper_result {
  char**                                    sorted_recordset;
  uint32_t                                  num_groups;
  struct group**                            groupset;
};

struct groupfilter_result {
  uint32_t                                  num_filtered_groups;
  struct group**                            filtered_groupset;
};

struct merger_result {
  uint32_t                                  num_group_tuples;
  size_t                                    total_num_group_tuples;
  struct group***                           group_tuples;
};

struct ungrouper_result {
  size_t                                    num_streams;
  struct stream**                           streamset;
};
```

Listing 22: F(*v*2): Result Structs

*greedy clauseset deallocation*

The query fragment structs (`X_clauseset`) used to get the result is greedily deallocated soon after the stage returns to keep the in-memory usage to the minimum. The `filter_clauseset` although is kept until the end of the grouper stage since it helps the grouper aggregation stage make decisions on whether a linear pass through the flow trace is required to aggregate a column that may have been already a criteron for the filter stage.

```
branch->grouper_result = grouper(...);
if (branch->grouper_result == NULL)  ...
else {
  /* free filter clauses */
  /* free grouper clauses */
  /* free grouper aggregation clause */
}

branch->gfilter_result = groupfilter(...);
if (branch->gfilter_result == NULL) ...
else {
  /* free group filter clauses */
}

fquery->merger_result = merger(...);
if (fquery->merger_result == NULL) ...
else {
    /* free merger clauses */
}
```

Listing 23: F(*v*2): Greedy Ruleset Deallocation

## 7.3 USER INTERFACE DESIGN

It is essential to allow the interface to be intuitive to any new user who is interested in using the tool for network analysis. In essence, this is achieved using the standard `getopt_long` call to allow both short and long option arguments. The execution engine appropriately displays the usage help when insufficient number of arguments are provided as shown in listing 24. The engine is also interactive to help one choose the right switches with required options.

*pretty usage help, tracking invalid options*

```
$ bin/engine
usage: bin/engine [OPTIONS] queryfile tracefile      query the specified trace
   or: bin/engine [OPTIONS] queryfile -              read the trace from stdin

OPTIONS:
-z, --zlevel             change the compression level (default:5)
-d, --dirpath            save the results as flow-tools files in given dirpath
-D, --debug              enable debugging mode
-v, --verbose            increase the verbosity level
-V, --version            output version information and exit
-h, --help               display this help and exit

$ bin/engine queryfile tracefile --foo
bin/engine: invalid option --foo

$ bin/engine queryfile tracefile --verbose
bin/engine: option --verbose requires an argument

$ bin/engine queryfile tracefile --verbose=5
ERROR: valid verbosity levels: (1-3)
```

Listing 24: F(*v2*): User Interface

Since the execution engine largely depends on the sanity of the query and trace files passed to it as arguments, it is essential to let the input files pass through a level of consistency check before going forward with the processing pipeline to avoid any undefined behavior as shown in listing 25.

*consistency checks*

```
$ bin/engine README.md tracefile
ERROR: json_tokener_parse_ex(...)

$ bin/engine queryfile README.md
ERROR: ftio_init(...)
```

Listing 25: F(*v2*): Consistency Checks

With a software undergoing such a rapid pace of development, it's helpful to be able to see the inner workings of each stage of the pipeline during a debugging lifecycle. As such, the engine echoes a backtrace whenever it fails gracefully as shown in listing 26.

*backtrace on graceful exits*

```
$ bin/engine foo bar
ERROR: open(...)
BACKTRACE:
0   engine              0x000000010bc891c2 print_trace + 34
1   engine              0x000000010bc893cb errExit + 395
2   engine              0x000000010bc898de read_param_data + 174
3   engine              0x000000010bc8c600 main + 80
4   engine              0x000000010bc6e054 start + 52
```

Listing 26: F(*v2*): Backtraces

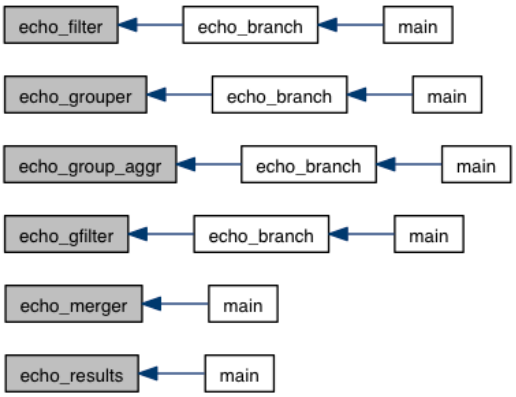



Figure 20: F(*v2*): Verbosity Levels Workflow

*debug and verbosity levels*

The engine also allows to increase the amount of output generated using a number of verbosity levels. A specific function is designed to handle the output generation of each stage of the pipeline as shown in figure 20. In its default state, the engine only outputs the resultant streams [1][2] of flow records. A debug (or `-verbose=3`) level engine execution is shown in listing 27. In addition to echoing the flow (or group) records resulting from each stage, it also echoes the results of each intermediate stage alongwith the original trace that was passed to it. With `-verbose=2`, the echo of the original trace is pruned, while intermediate results get pruned with `-verbose=1`.

```
$ bin/engine tracefile queryfile --debug

# capture hostname:     ihp.jacobs.jacobs-university.de ...

No. of Filtered Records: ...
No. of Sorted Records: ...
No. of Unique Records: ...
No. of Groups: (Verbose Output): ...

... 0     216.137.61.203  80    0     192.168.0.135 ...
... 0     216.137.61.203  80    0     192.168.0.135 ...

... 0     8.12.214.126    80    0     192.168.0.135 ...
... 0     8.12.214.126    80    0     192.168.0.135 ...

No. of Groups: 32 (Aggregations): ...

... 0     216.137.61.203  80    0     192.168.0.135 ...
... 0     8.12.214.126    80    0     192.168.0.135 ...

No. of Filtered Groups: (Aggregations): ...
No. of (to be) Matched Groups: ...

... 0     192.168.0.135   0     0     204.160.123.126 80    ...
... 0     87.238.86.121   80    0     192.168.0.135   0     ...

No. of Merged Groups: 3 (Tuples): ...

... 0     192.168.0.135   0     0     216.46.94.66    80    ...
... 0     216.46.94.66    80    0     192.168.0.135   0     ...

No. of Streams: ...
```

Listing 27: F(*v2*): Debugging

1 A `stream` is a collection of flow-records of a group unfolded by the ungrouper
2 A `streamset` is collection of all the streams unfolded by the ungrouper

The flow-records echoed to the standard output can also be written to a Netflow *v*5 flow-tools file. The `-dirpath` switch allows one to provide a directory path where the results can be stored. Each stream is stored as its own file with an ID to disambiguate it. Results from each stage of the pipeline can also be written to separate files with the increase in the verbosity level. In fact, `-dirpath` and `-verbose` work well together to adjust the level at which the writes are to be made. Such a feature allows one to blackbox each stage's functionality and analyze it independently, which can be useful not only for unit testing but also for performance evaluations purposes.

*writing results as flow-tools files*

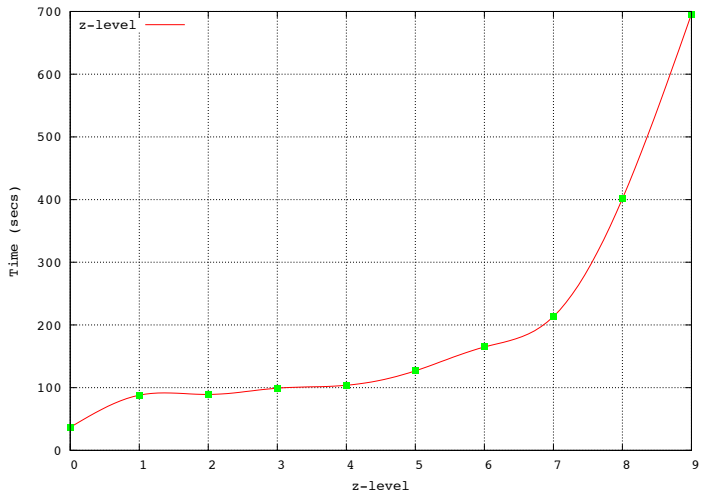



Figure 21: F(*v*2): `z-level` Effect on Performance

The engine uses the `zlib` [73] software library to compress the results written to the flow-tools files. `zlib` supports 9 compression levels with 9 being the highest. The NFQL engine supports `-zlevel` switch to allow the user to supply its desired choice of the of the compression level. A default level of 5 is used for writes if the switch is not supplied during runtime. Figure 21 shows the time taken to write a sample of records passing the filter stage for each `z-level`. It goes to show that each level adds its own performance overhead and must be used with discretion.

*adaptable compression levels*

# 8 IMPLEMENTATION

The effort to provide a clean usable implementation of the language was from the initial outset backed up by three goals. The first goal was to allow the implementation to flawlessly walk through all stages of the pipeline without incurring major performance overhead. The second goal was to abstract out the engine functionality in such a way so as to allow runtime evaluation of the flow query. The third goal was to provide a clean layout of the working code with a seamless build process to allow future developers to quickly get started on top of the current snapshot. This was supplemented by a thorough regression and benchmarking suite to make the code verifiable. This chapter introduces the inner workings of the code to explains how these goals were set into practise and brought to life.

## 8.1 FASTER FILTER

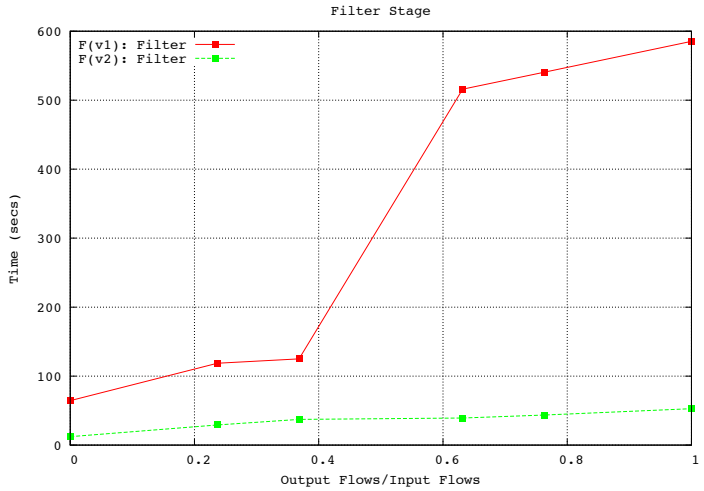



Figure 22: Filter Stage: F($v1$) vs F($v2$)

The execution engine in F($v1$) used to read all the flow records of a supplied trace into memory before starting the processing pipeline. Since, the filter stage uses the supplied set of absolute rules to make a decision on whether or not to keep a flow record; it had to pass through the whole in-memory recordset *again* to fill in the filter results. This technique involves multiple linear runs on the trace and therefore slows down when the ratio of number of filtered records to the total number of flow-records is high as shown in figure 22.

*faster filter*

The filter stage in F($v2$), therefore has now been merged with in-memory read of the trace. This means, a decision on whether or not to make room for a record in memory and eventually hold a pointer for it in filter results is done upfront as soon as the record is read from the trace. In addition, if a request to write the filter stage results to a flow-tools file has been made, the writes are also made as soon the filter stage decision is available, thereby allowing reading-filtering-writing to happen in $O(n)$ time, where $n$ is the number of records in the trace. It is important to note that the filtered records are saved in common location from where they are referenced by each branch. This helps keep the memory costs at a minimum when multiple branches are involved. A publically available Netflow $v5$ trace [1] was used to compare the performance of the new filter in F($v2$) with that from F($v1$) as shown in figure 22. The trace has around 20M flow-records. The queries executed are available in source repository [2] and have been omitted from here for brevity reasons.

## 8.2 GROUPER INTERNALS

*functional grouping using qsort and bsearch*

In order to be able to make comparisons on field offsets, a näive approach is to linearly walk through each filtered record against the filtered recordset leading to a complexity of $O(n^2)$, where $n$ is the number of filtered records. A smarter approach is to put the copy in a hash table and then try to map each pointer while walking down the filtered recordset once, leading to a complexity of $O(n)$. The hash table approach, although will work on this specific example, will fail badly on other relative comparisons. The F($v1$) execution engine [6] formulates an approach to grouping records using a binary search after a quick sort on the field offset of the first grouping rule of each record in the filtered recordset. This helps the grouper perform faster search lookups to find records that must group together in $O(n * lg(k))$ time with a preprocessing step taking $O(n * lg(n)) + O(n)$ in the average case, where $n$ is the number of filtered records and $k$ is the number of unique filtered records. The implementation of this idea was broken and used to segfault at multiple steps within the grouping stage. F($v2$) introduces a completely functional implementation of this approach that does not incorporate any assumptions on the type of field offsets and is robust enough to work with different types of grouping queries.

*cross platform qsort_r and bsearch_r*

The reetrant `qsort_r` was used, since it can pass an additional argument `thunk` to the comparator, which in our case is the field offset used for comparing two flow records. Since the order of arguments of `qsort_r` are different for `glibc` and `BSD`, the function invocation had to be wrapped around platform specific macros as shown in listing 28. More Surprisingly, there is currently no equivalent `bsearch_r` to

1 http://traces.simpleweb.org/traces/netflow/netflow1/netflow000.tar.bz2
2 http://github.com/vbajpai/mthesis-src/

complement `qsort_r`. As such, the contemporary `bsearch` function from the `glibc` library was adapted to accommodate the void and is defined in `utils` module.

```
struct grouper_type {
#if defined (__APPLE__) || defined (__FreeBSD__)
  int (*qsort_comp)(
                     void*            thunk,
                     const void*      e1,
                     const void*      e2
                   );
#elif defined (__linux)
  int (*qsort_comp)(
                     const void*      e1,
                     const void*      e2,
                     void*            thunk
                   );
#endif
...

#if defined(__APPLE__) || defined(__FreeBSD__)
  qsort_r(
           sorted_recordset_ref,
           num_filtered_records,
           get_grouper_intermediates(
           (void*)&grouper_ruleset[0]->field_offset2,
           gtype->qsort_comp
         );
#elif defined(__linux)
  qsort_r(
           sorted_recordset_ref,
           num_filtered_records,
           sizeof(char **),
           gtype->qsort_comp,
           (void*)&grouper_ruleset[0]->field_offset2
         );
#endif
```

Listing 28: F(*v*2): qsort_r Invocation

*groups as cooked netflow v5 records*

Group records are a conglomeration of several flow records with some common characteristics defined by the flow query. Some of the non-common characteristics can also be aggregated into a single value using group aggregations as shown in listing 30. Since, the execution engine supports multiple verbosity levels, it is useful if a single group record can be again mapped into a NetFlow *v*5 record template, so that it can be echoed as the representative of all its members. This was achieved using a struct `group` as shown in listing 29.

```
struct group {
  uint32_t                          num_members;
  char**                            members;
  struct aggr_result*               aggr_result;
};

struct aggr_result {
  char*                             aggr_record;
  struct aggr**                     aggrset;
};
```

Listing 29: F(*v*2): Group Struct

*ignoring redundant aggregation requests*

There can be a situtation where the query designer might incorrectly ask for aggregation on a field already specified in a grouper (or filter) module. If the relative operator is an equality comparison, the aggregation on such a field becomes less useful, since the members of the grouped record will always have the same value for that field.

The engine is now smart to realize this redundant request and ignores such aggregations as shown in listing 30.

```
grouper g_www_res {
   g1 {
      srcip = srcip
      dstip = dstip
   }
   aggregate srcip, dstip, sum(bytes) as bytes, bitOR(tcp_flags) as flags,
}

$ bin/engine queryfile tracefile --verbose=1
...
No. of Groups: ...

...    SrcIPaddress     ...      DstIPaddress      OR(Fl)    Sum(Octets)
...    4.23.48.126      ...      192.168.0.135     3         81034
...    8.12.214.126     ...      192.168.1.138     2         5065
```

Listing 30: F(*v*2): Aggregations Example

*clubbing records with no grouper rules*

Records are clubbed together into one group if no group modules are defined. Previously such a query used to form groups for each individual filtered record. That was less useful since then it was not possible to calculate meaningful aggregations on all the records that passed the filter stage. Now, when the group modules are empty, all the filtered records are clubbed into one group to allow aggregations as shown in listing 31.

```
grouper g_www_res {
   g1 {}
   aggregate sum(bytes) as bytes
}

$ bin/engine queryfile tracefile --verbose=1
...
No. of Groups: 1 (Aggregations)

...    Sum(Octets)
...    2356654
```

Listing 31: F(*v*2): Clubbing Records with No Grouper Rules

## 8.3 FASTER GROUPER

*faster grouping in a generic case*

The `qsort` and `bsearch` approach as suggested in F(*v*1), applies the sorting and searching only on the first grouping rule of the flow query. This results in a linear pass from the index retrieved by the `bsearch` till either the end of the filtered recordset or until the first rule fails. The linear pass is processed for each potential group. Such an approach works well when the ratio of the number of filtered records to the number of groups is low. However for higher ratios, the time increases exponentially as can be seen in figure 23. A better approach, as implemented in F(*v*2) is to sort the filtered recordset on all the requested grouping rules. A comparator that performs nested `qsort_r` comparisons to sort on each grouping rule is shown in listing 32. This helps the execution engine perform a nested `bsearch` to reduce the

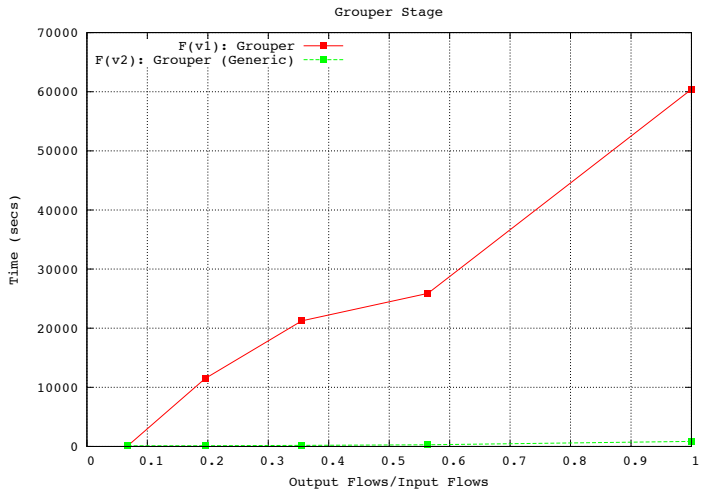



Figure 23: Grouper Stage: F($v1$) vs F($v2$)

linear pass to a fairly small filtered recordset. However, the number of elements upon which the search has to be performed needs to be known at each level of the binary lookup. In order to avoid a linear run to count the number of elements, the parent level `bsearch` invocation returns a boundary with the first and last element to enable such a calculation. The algorithm to find the first and last element using `bsearch` is shown in listing 33 and 34 respectively.

```
#if defined(__APPLE__) || defined(__FreeBSD__)
  int qsort_comp(void *thunk, const void *e1, const void *e2) {
#elif defined(__linux)
  int qsort_comp(const void *e1, const void *e2, void *thunk) {
#endif
    ...
    for (int i = 0; i < clause->num_terms; i++) {
        ...
    #if defined(__APPLE__) || defined(__FreeBSD__)
        result = gtype->qsort_comp((void*)term->field_offset2, e1, e2);
    #elif defined(__linux)
        result = gtype->qsort_comp(e1, e2, (void*)term->field_offset2);
    #endif
        if (result != 0) break;
    }
    return result;
  }
```

Listing 32: F($v2$): Nested qsort_r Comparator

The same trace as used in section 8.1 is used to compare the performance of the F($v1$) and F$v(2)$ grouper stage. The queries are are available in the source code repository. The evaluation is shown in figure 23. It is clear that when the ratio of the number of groups formed to the input filtered records is low, the performance is comparable. However on higher ratios, the times taken are far apart. It is because, the frequency of linear passes made in F($v1$) increases with the number of potential groups. However with the nested `bsearch` in F($v2$) the linear pass is small enough to not make a huge difference.

```
1  mid = (size_t)floor((low+high)/2.0);
2
3  if (comparison > 0)
4    low = mid + 1;
5
6  else if (comparison < 0)
7    high = mid - 1;
8
9  else if (low != mid)
10   high = mid;
11
12 else return mid;
```

Listing 33: `bsearch_first_item`

```
1  mid = (size_t)ceil((low+high)/2.0);
2
3  if (comparison > 0)
4    low = mid + 1;
5
6  else if (comparison < 0)
7    high = mid - 1;
8
9  else if (high != mid)
10   low = mid;
11
12 else return mid;
```

Listing 34: `bsearch_last_item`

*faster grouping in a specific case*

The grouping approach has further been optimized when the filtered records are grouped for equality. In such a scenario, the need to search for unique records and a subsequent binary search goes away. The groups can now be formed in $O(n)$ time with a more involved preprocessing step taking $O(p * n * lg(n)$ where is $n$ is the number of filtered records, and $p$ is the number of grouping rules. The performance evaluation of the grouper handling this special case against its behavior when handling generic cases is shown in figure 24. Profiling the performance of the grouper on queries that produce higher ratios reveal that a significant amount of time is taken in `bsearch`. The special case of equality comparisons eliminate these calls and further reduce the time in the long tail as shown in figure 24.

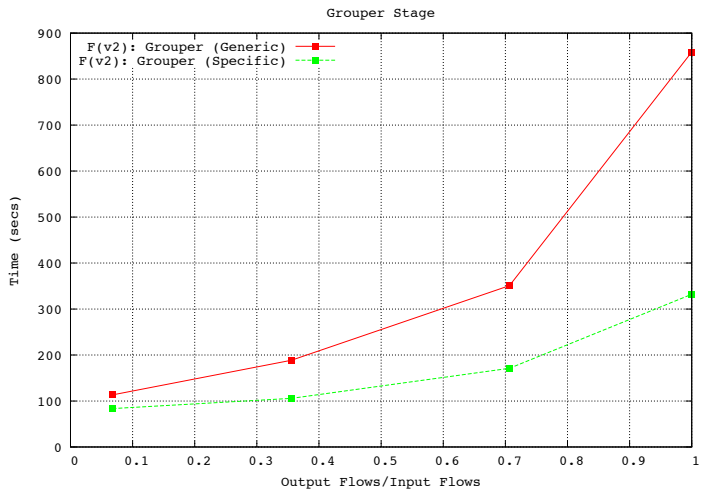



Figure 24: Grouper Stage: F(v2) (Generic) vs F(v2) (Specific)

## 8.4 ROBUST PIPELINE EXECUTION AND RUNTIME COMPLEXITY

The groupfilter is used to filter the groups produced by the grouper based on some absolute rules defined in a DNF expression. The `struct` rule holds information about the flow record offset, the value be-

ing compared to and the operator which maps to a unique `enum` value. This enum value is later used to map the operation to a specific group-filter function using a `switch` case as shown in listing 35. The group-filter functions are auto-generated using a python script `scripts/generate-functions.py`.

*group filter implementation*

```
/* assign group filter func for each term */
for (int k = 0; k < num_groupfilter_clauses; k++) {

  struct groupfilter_clause* gclause = groupfilter_clauseset[k];

  for (int j = 0; j < gclause->num_terms; j++) {

    /* assign a uintX_t specific function depending on grule->op */
    struct groupfilter_term* term = gclause->termset[j];
    assign_groupfilter_func(term);
  }
}

void
assign_groupfilter_func(struct groupfilter_term* const term) {

  switch (
          term->op->op |
          term->op->field_type
         ) {

    case RULE_EQ | RULE_S1_8:
      term->func = gfilter_eq_uint8_t;
      break;
    ...
  }
}
```

Listing 35: F(*v2*): Group Filter Implementation

The merger is used to relate the groups from different branches according to a criterean. Similar to the group filter, a unique `enum` value is used to map the operator to a specific specific merger function using a `switch` case as shown in listing 36. The merger functions are again auto-generated using `scipts/generate-functions.py`.

*merger implementation*

```
/* assign merger func for each term */
for (int k = 0; k < num_merger_clauses; k++) {

  struct merger_clause* mclause = merger_clauseset[k];

  for (int j = 0; j < mclause->num_terms; j++) {

    /* assign a uintX_t specific function depending on grule->op */
    struct merger_term* term = mclause->termset[j];
    assign_merger_func(term);
  }
}

void
assign_merger_func(struct merger_term* const term) {

  switch (
           term->op->op |
           term->op->field1_type |
           term->op->field2_type
         ) {

    case RULE_EQ | RULE_S1_8 | RULE_S2_8:
      term->func = merger_eq_uint8_t_uint8_t;
      break;
      ...
  }
}
```

Listing 36: F(*v2*): Merger Implementation

*ungrouper implementation*

Given the group tuples, the ungrouper returns a streamset of flow records [3] [4] . The matched group tuples are generated by the merger. The ungrouper returns as many streams as there are number of matched group tuples. An example output is shown in listing 37.

```
$ bin/engine tracefile queryfile
No. of Streams: 2
No. of Records in Stream (1): 24

... Sif   SrcIPaddress    SrcP  DIf   DstIPaddress    DstP  ...

... 0     192.168.0.135   56225 0     216.46.94.66    80    ...
... 0     216.46.94.66    80    0     192.168.0.135   56228 ...
```

Listing 37: F(*v*2): Ungrouper Result Echo

There are dedicated comparator functions for each `uintX_t` type of the field offset. Up until now, the choice for the function was made using a single function, `assign_fptr(...)`, which was called before the start of the pipeline to ensure all function pointers point to the right functions for each stage as shown in listing 38.

```
assign_fptr(struct flowquery *fquery) {
  for (int i = 0; i < fquery->num_branches; i++) {
    /* for loop for the filter */
    for (int j = 0; j < branch->num_filter_rules; j++) {...}
    /* for loop for the grouper */
    for (int j = 0; j < branch->num_grouper_rules; j++) {...}
    /* for loop for the group-aggregation */
    for (int j = 0; j < branch->num_aggr_rules; j++) {...}
    /* for loop for the group-filter */
    for (int j = 0; j < branch->num_gfilter_rules; j++) {...}
  }
}
```

Listing 38: F(*v*1): Early Comparator Assignments

*lazy comparator assignments*

This function is computationally expensive, since it falls through a *huge* `switch` statement to determine the function of right type. It is not guaranteed that given the type of the query and trace, the program will eventually go through each stage of the pipeline. It is also possible that the program exits before, because there is nothing more for the next stage to compute. The function pointers should therefore be set as late as possible as shown in listing 39. Each of these functions are called from their respective stages just before the comparison. As a result, we save the computation time wasted in setting the function pointer for stage X if X is never executed.

```
void assign_filter_func(struct filter_term* const fterm);
void assign_grouper_func(struct grouper_term* const gterm);
void assign_aggr_func(struct aggr_term* const aterm);
void assign_groupfilter_func(struct groupfilter_term* const term);
void assign_merger_func(struct merger_term* const term);
```

Listing 39: F(*v*2): Lazy Comparator Assignments

3 A `stream` is a collection of flow-records of a group unfolded by the ungrouper
4 A `streamset` is collection of all the streams unfolded by the ungrouper

The public interface function `grouper(...)` call was plagued with hardcoded `uint32_t` type assumptions on the field offset. These field offsets are used to make grouper rule comparisons. The function now internally calls `get_gtype(...)` to fall through a switch case to determine the type of the field offset at runtime as shown in listing 40.

*flexible grouper with no type assumptions*

```
struct grouper_type* get_gtype(uint64_t op) {
  switch (op) {
    case RULE_S2_8:
      gtype->qsort_comp = comp_uint8_t;
      gtype->bsearch = bsearch_uint8_t;
      gtype->alloc_uniqresult = alloc_uniqresult_uint8_t;
      gtype->get_uniq_record = get_uniq_record_uint8_t;
      gtype->dealloc_uniqresult = dealloc_uniqresult_uint8_t;
      break;
    case RULE_S2_16: ...
    case RULE_S2_32: ...
    case RULE_S2_64: ...
  }
  return gtype;
}
```

Listing 40: F(*v*2): Flexible Grouper

The group aggregation functions were hardcoded in `group_aggr` struct. The functions are now replaced with rules that map to a specific aggregation function. The mapping of the rule to the function is done using a `switch` case as shown in listing 41. The aggregation functions are auto-generated using `scripts/generate-functions.py`.

*flexible group aggregations*

```
/* assign a aggr func for each aggr clause term */
if (groupaggregations_enabled) {
  for (int j = 0; j < num_aggr_clause_terms; j++){

    struct aggr_term* term = aggr_clause_termset[j];
    assign_aggr_func(term);
  }
}

void
assign_aggr_func(struct aggr_term* const aterm) {

  switch (
          aterm->op->op |
          aterm->op->field_type
         ) {
    case RULE_STATIC | RULE_S1_8:
      aterm->func = aggr_static_uint8_t;
      break;
    ...
  }
}
```

Listing 41: F(*v*2): Flexible Group Aggregations

The aggregation function also needs to know the type of the offsets used previously in the filter and grouper rules to be able to fill in common fields in its cooked netflow *v*5 group aggregation record. As a result a `get_aggr_fptr(...)` function was defined that accepts those previous rules to fall through a switch to return a function pointer to an aggregation function of the correct type as shown in listing 42. This aggregation function is then later used to fill in the common fields. A similar call is made for the grouper rules as well.

*flexible group aggregation redundancy checks*

```
struct aggr*
(*get_aggr_fptr(
                  bool ifgrouper,
                  uint64_t op
               )) (
                     char** records,
                     char* group_aggregation,
                     size_t num_records,
                     size_t field_offset,
                     bool if_aggr_common
                   ) {
  ...
  switch (op) {
+   case RULE_EQ | RULE_S1_8:
+   case RULE_NE | RULE_S1_8:
+   case RULE_GT | RULE_S1_8:
    ...
+     aggr_function = aggr_static_uint8_t;
+     break;
    ...
  }
...
term->aggr_function = get_aggr_fptr(term->op->field_type);
(*aggr_function)(
                  group->members,
                  group_aggregation,
                  group->num_members,
                  field_offset,
                  TRUE
                );
```

Listing 42: F($v2$): Flexible Group Aggregation Redundancy Checks

*early filtering*

The execution engine in F($v1$) had a separate filter stage that was executed in the branch thread. As a result, the branch was not able to free the records that failed the filter rule, since they could have passed in some other branch. The records that failed in all branches were only free'd once all threads joined `main(...)` i.e., before calling the `merger(...)`. The execution engine in F($v2$) pushes the filter stage out of the branch and closer to where the trace is originally read. This enables the memory allocation of only records that passed the filter stage thereby reducing the runtime memory footprint of the engine as shown in listing 43.

```
struct ft_data *
ft_read (
          struct ft_data* data,
          struct flowquery* fquery
        ) {
  /* process each flow record */
  while ((record = ftio_read(&data->io)) != NULL) {
    /* process each branch */
    for (int i = 0, j; i < fquery->num_branches; i++) {
      /* process each filter clause (clauses are OR'd) */
      for (int k = 0; k < branch->num_filter_clauses; k++) {
        /* process each filter term (terms within a clause are AND'd) */
        for (j = 0; j < fclause->num_terms; j++) {
          /* run the comparator function of the filter rule on the record */
          ...
        }
      }
      /* if rules are satisfied then save this record */
      if (satisfied) {
        /* save the pointer in the filtered recordset */
        ...
        /* write to the output stream, if requested */
        ...
      }
    }
  }
}
```

Listing 43: F($v2$): Early Filtering

Each branch runs in its own thread. If any of the stages of the branch return a `NULL` when returning from their public interface function, there is no reason to continue the thread. The subsequent stages of the branch cannot do much with a `NULL` result. Therefore, the branch thread returns with a `EXIT_FAILURE` if either stage returns `NULL`, and with `EXIT_SUCCESS` on normal exit as shown in listing 44.

*early thread exits*

```
void * branch_start(void *arg) {
  ...
  if (branch->filter_result == NULL)
    pthread_exit((void*)EXIT_FAILURE);

  branch->grouper_result = grouper(...);
  if (branch->grouper_result == NULL)
    pthread_exit((void*)EXIT_FAILURE);

  branch->gfilter_result = groupfilter(...);
  if (branch->gfilter_result == NULL)
    pthread_exit((void*)EXIT_FAILURE);
  ...

  pthread_exit((void*)EXIT_SUCCESS);
}
```

Listing 44: F(*v*2): Early Thread Exits

Each stage of the processing pipeline is dependent on the result of the previous one. As a result, the stages should only proceed and process, when the previous returned results. Implementing such a response was straightforward for the grouper and group filter as shown in listing 45, the merger although was a little trickier. The merger stage proceeds only when every branch has non-zero filtered groups. The iterator initializer `iter_init(...)` deallocates and returns `NULL` if any one branch has 0 filtered groups. Consequently a check is performed in the merger to make sure `iter` is *not* `NULL`.

*context-aware pipeline stages*

```
/* grouper */
struct grouper_result*
grouper(...) {

  /* go ahead if there is something to group */
  if (fresult->num_filtered_records > 0) {...}
}

/* group filter */
struct groupfilter_result*
groupfilter(...) {

  /* go ahead if there is something to group filter */
  for (int i = 0, j = 0; i < gresult->num_groups; i++) {...}
}

/* merger */
struct merger_result*
merger(...) {

  /* initialize the iterator */
  struct permut_iter* iter = iter_init(num_branches, branchset);
  if (iter == NULL)
    return mresult;
  ...
}
```

Listing 45: F(*v*2): Context-Aware Pipeline Stages

*inline writes to file*

The results from each stage of the pipeline are echoed to the standard output just before the engine exits. This leads to an additional loop to echo the results, however echoes to the standard output are only used for debugging purposes. The writes to a file can however be legitimately requested in a real network analysis task. As a result, it is essential to avoid additional loops when performing writes to a file. The execution engine, therefore writes each result record to a file as soon as it is seen by the pipeline stage. Such a behavior occurs for each pipeline stage and and for all verbosity levels. The writes to file can be requested using `-dirpath` switch in combination with `-verbose`. A sample code from the group filter stage is shown in listing 46.

```
struct groupfilter_result* groupfilter( ... ) {

  /* initialize an output stream if file write is requested */
  ...
  /* iterate over each group */
  for (int i = 0, j = 0; i < gresult->num_groups; i++) {
    /* process each group filter clause (clauses are OR'd) */
    for (int k = 0; k < num_groupfilter_clauses; k++) {
      /* process each group filter term (terms within a clause are AND'd) */
      for (j = 0; j < gfclause->num_terms; j++) {

        /* run the comparator function of the filter term on the record */
        ...
      }
    }

    /* if rules are satisfied then save this record */
    if (satisfied) {
      /* write the record to the output stream */
      ...
    }
  }
}
```

Listing 46: F(*v2*): Inline Writes to File

*runtime complexity*

A rundown of the runtime complexity of each stage of the processing pipeline is shown in table 6. In the table, $n$ is the total number of flow records in the trace, while $k$ is the number of unique flow records that passed the filter stage, and $p$ is the number of grouping terms. The number of branches (or threads) spawned by the execution engine is $m$. It is clear that the merger is currently the bottleneck of the pipeline and needs further optimizations.

| Pipeline Stage | Runtime Complexity |
|---|---|
| Filter | $O(n)$ |
| Grouper | $O(p * n * lg(n)) + O(n) + O(n * lg(k))$ |
| Group Aggregation | $O(n)$ |
| Group Filter | $O(n)$ |
| Merger | $O(n^m)$ |
| Ungrouper | $O(n)$ |

Table 6: F(*v2*): Pipeline Runtime Complexity

## 8.5 MERGER INTERNALS

```
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;
}
```

Listing 47: Merger Pseudocode [11]

The merger pseudocode as defined in the NFQL specification [11] is shown in listing 47. Implementing this pseudocode in C is not straightforward. The level of nesting depends on the number of branches, and is therefore not known at compile time. The information on the number of branches comes from the query which is passed to the execution engine at runtime.

```
/* initialize the iterator */
struct permut_iter *iter = iter_init(binfo_set, num_branches);

/* iterate over all permutations */
unsigned int index = 0;
while(iter_next(iter)) {
  index++
  for (int j = 0; j < num_branches; j++) {
    /* first item */
    if(j == 0)
      printf("\n%d: (%zu ", index, iter->filtered_group_tuple[j]);
    /* last item */
    else if(j == num_branches -1)
      printf("%zu)", iter->filtered_group_tuple[j]);
    else
      printf("%zu ", iter->filtered_group_tuple[j]);
  }
}

/* free the iterator */
iter_destroy(iter);
```

Listing 48: F(*v2*): Merger Iterator Utility

As a result, an iterator that can provide all possible permutations of $m-$tuple (where $m$ is the number of branches) group record IDs was needed. The result of the iterator can then be used to make a match. The merger stage, begins by initializing this iterator passing it the number of branches, and information about each branch. Then, it loops over to get a new $m-$tuple of group record IDs on each iteration until the iterator returns `false`. A sample to print all possible group ID permutation is shown in listing 48, with the output in listing 49

*merger iterator utility*

```
  1: (1 1 1)
  2: (1 1 2)
  ...
  12: (3 2 2)
```

Listing 49: F(*v2*): Merger Iterator Utility Output

## 8.6 FASTER MERGER

The merger as required by the NFQL specification suggests matching each group record from one branch with every other record of each branch. This leads to a complexity of $O(n^m)$ where $n$ is the number of filtered group records and $m$ is the number of branches. The possible number of tries when matching group records however can be reduced by sorting the group records on the field offsets used for a match. The merger, is now smart enough to skip over iterator permutations when a state of a current field offset value may not allow any further match beyond the index in the current branch. For such an optimization to work, the filtered group records must be sorted in the order of field offsets specified in the merger clause. Specifying the filtered group records in any other order may lead to undefined behavior. This means, that the terms in the group clause must be arranged in such a way so to align with the order of terms in the merger clause.

*skipping iterator permutations on sorted group records*

The NFQL specification bases the merger matches on the notion of matched tuples. This means that a filtered group record can be written to a file multiple times if it is part of multiple matched tuples. This situation is very common and it worsens when different branches have similar filtered groups records. Since, the function of the merger is to find a match of groups records across branches based on a predefined condition, all the group records across branches that satisfy the condition can be clubbed into one collection instead of separate tuples. All the group records within a collection can then be written to the file. This eliminates the inherent redundancy and significantly improves the merger performance. The implementation of this approach incurs a reimplementation of the ungrouper. The ungrouper, as a result now accepts a collection of matched filtered group records as input. It then iterates over each collection to unfold it groups and write their flow record members to files.

*matched collections over matched tuples*

The performance comparison of this approach against the one suggested by the specification is tricky. The merger implementation of the original specification is slow. It is so slow that it keeps churning the CPU for days without results. The newer approach takes less than an hour. The performance evaluation of this newer approach is discussed in more details in the next chapter. The newer approach however is not currently pushed to `master` git branch[5]. This is because the newer merger implementation currently does not adapt itself to sort the filtered group records on the field offset hit by the terms defined in the merger clause, that do not exist in the grouper clause. In addition, more test cases need to be defined to ensure that the newer merger works well with a variety of NFQL queries before pushing it to the mainstream branch.

*performance comparison and present issues*

5 The newer merger implementation is available in the `fastmerger` git branch

## 8.7 RUNTIME QUERY EVALUATION

The complete JSON query is now read in at *runtime*. The branchsets and each DNF expression of the stage is a `JSON array` as shown in listing 50. A DNF expression is a disjunction of conjunctive clauses. Each clause in the expression is OR'd while each term within the clause is AND'd together. This terminology is more intuitive to modules, submodules and rulesets as used previously by F($v1$).

```
{
  "branchset": [
    { "filter": {
        "dnf-expr": [
          { "clause": [
              { "term": { ... } },
              { "term": { ... } }
            ]
          }
        ]
      },
      ...
    },
  ],
  "merger": {
    "dnf-expr": [
      { "clause": [
          { "term": { ... } },
          { "term": { ... } }
        ]
      }
    ]
  },
  "ungrouper": {}
}
```

Listing 50: F($v2$): Flow Query in JSON

`json-c`[6] is used to parse such a query file read into memory by calling `parse_json_query(...)`. The `json_query` is then used to prepare the `struct flowquery` used by the pipeline stages as shown in listing 51. The `json_query` struct is just an intermediate.

*parsing using json-c*

```
struct json {
  size_t                                num_branches;
  size_t                                num_merger_clauses;

  struct json_branch**                  branchset;
  struct json_merger_clause**           merger_clauseset;
};

struct json_branch {
  size_t                                num_filter_clauses;
  size_t                                num_grouper_clauses;
  size_t                                num_aggr_clause_terms;
  size_t                                num_groupfilter_clauses;

  struct json_filter_clause**           filter_clauseset;
  struct json_grouper_clause**          grouper_clauseset;
  struct json_aggr_term**               aggr_clause_termset;
  struct json_groupfilter_clause**      groupfilter_clauseset;
};

struct json*
json_query = parse_json_query(param_data->query_mmap);

struct flowquery*
fquery = prepare_flowquery(param_data->trace, json_query);
```

Listing 51: F($v2$): Parsing JSON query using `json-c`

6 http://oss.metaparadigm.com/json-c/

*generating json queries using python*

The JSON query is verbose and cumbersome to write manually. The python parser will eventually emit this intermediate format, so the next logical step is to generate the query from python. A python module (`scripts/queries/pipeline.py`) that encapsulates each pipeline stage as a separate class is shown in listing 52. Scripts that generate JSON queries can `import` this module to reduce code redundancy.

```
def protocol(name):
  return socket.getprotobyname(name)

class FilterRule: ...
class GrouperRule: ...
class AggregationRule: ...
class GroupFilterRule: ...
class MergerRule: ...
```

Listing 52: F(*v2*): Python Pipeline Module

*sample scripts*

A sample script to generate a query is shown in listing 53. Each `clause` is a list of python objects of a specific class of the `pipeline` module. At this point, the python parser just needs to create each stage term objects and the script will take care to emit the JSON. Example scripts to generate different queries are provided in `scripts/queries/`.

```
import json
from pipeline import FilterRule, GrouperRule, AggregationRule
from pipeline import GroupFilterRule, MergerRule
from pipeline import protocol

if __name__ == '__main__':

  # filter stage
  term1 = {'term': vars(FilterRule(...))}
  clause1 = {'clause': [term1] + ...}
  filter1 = {'dnf_expr': [clause1] + ...}

  # grouper stage
  term1 = {'term': vars(GrouperRule(...))}
  clause1 = {'clause': [term1] + ...}
  term1 = {'term': vars(AggregationRule(...))}
  aggregation1 = {'clause': [term1] + ...}
  grouper1 = {'dnf_expr': [clause1] + ..., 'aggregation': aggregation1}i

  # group filter stage
  term1 = {'term': vars(GroupFilterRule(...))}
  clause1 = {'clause': [term1] + ...}
  gfilter1 = {'dnf_expr': [clause1] + ...}

  branchset = []
  branchset.append(
                    {
                      'filter': filter1,
                      'grouper': grouper1,
                      'groupfilter': gfilter1,
                    }, ...
                  )

  # merger stage
  term1 = {'term': vars(MergerRule(...))}
  clause1 = {'clause': [term1] + ...}
  merger = {'dnf_expr': [clause1] + ...}

  # ungrouper stage
  ungrouper = {}

  query = {
            'branchset': branchset,
            'merger': merger,
            'ungrouper': ungrouper
          }
```

Listing 53: F(*v2*): Python Scripts to Generate JSON queries

The mapping of the JSON query to the structs defined in the execution engine is trickier than it looks. When reading the JSON query at runtime, the field offsets of the NetFlow v5 record struct are read in as strings from the JSON query. A utility function `get_offset(...)` was thus introduced that maps the read names to struct offsets as shown in listing 54. In addition, the type of each offset and the operations are also read from the JSON query as strings. This information is saved and thus used by the engine using an `enum` defined in `pipeline.h`. Therefore, another utility function `get_enum(...)` was defined to map this information to the unique `enum` members as shown in listing 54.

*runtime query internals*

```
size_t
get_offset(
           const char * const name,
           const struct fts3rec_offsets* const offsets
         ) {

  #define CASEOFF(memb)                       \
  if (strcmp(name, #memb) == 0)               \
    return offsets->memb

    CASEOFF(unix_secs);
    CASEOFF(unix_nsecs);
    ...

  return -1;
}

uint64_t
get_enum(const char * const name) {

  #define CASEENUM(memb)                      \
  if (strcmp(name, #memb) == 0)               \
    return memb

  CASEENUM(RULE_S1_8);
  CASEENUM(RULE_S1_16);
  ...
  CASEENUM(RULE_S2_8);
  CASEENUM(RULE_S2_16);
  ...
  CASEENUM(RULE_ABS);
  CASEENUM(RULE_REL);
  CASEENUM(RULE_NO);
  ...
  CASEENUM(RULE_EQ);
  CASEENUM(RULE_NE);
  ...
  CASEENUM(RULE_STATIC);
  CASEENUM(RULE_COUNT);
  ...
  CASEENUM(RULE_ALLEN_BF);
  CASEENUM(RULE_ALLEN_AF);
  ...
  return -1;
}
```

Listing 54: F(v2): JSON Parsing Utilities

The JSON query can also trigger and disable the stages at runtime. This means that one only has to supply the constructs that one wishes to use. The constructs that are not defined in the JSON query are inferred by the engine as a disable request. The execution engine uses disable flags that are turned on when the JSON query is parsed as shown in listing 55. These flags are used throughout the engine to only enable the requested functionality. It is important to note though that the engine is currently not smart to understand the interdependency

*disabling pipeline stages at runtime*

amongst the stages. For instance, disabling the merger when the ungrouper is kept enabled will lead to undefined behavior.

```
struct json*
parse_json_query(const char* const query_mmap) {

  ...
  /* filter */
  struct json_object* filter = json_object_object_get(branch_json, "filter");
  if (filter != NULL) filter_enabled = true;
  if (filter_enabled) ...

  /* grouper */
  struct json_object* grouper = json_object_object_get(branch_json, "grouper");
  if (grouper != NULL) grouper_enabled = true;
  if (grouper_enabled) ...

  /* grouper aggregation */
  if (grouper_enabled) {
    struct json_object* aggr = json_object_object_get(grouper, "aggregation");
    if (aggr != NULL) groupaggregations_enabled = true;
    if (groupaggregations_enabled) ...
  }

  /* group filter */
  struct json_object* gfilter = json_object_object_get(branch_json, "groupfilter");
  if (gfilter != NULL) groupfilter_enabled = true;
  if (groupfilter_enabled) ...

  /* merger */
  struct json_object* merger = json_object_object_get(query, "merger");
  if (merger != NULL) merger_enabled = true;
  if (merger_enabled) ...

  /* ungrouper */
  struct json_object* ungrouper = json_object_object_get(query, "ungrouper");
  if (ungrouper != NULL) ungrouper_enabled = true;
  ...
}
```

Listing 55: F($v2$): Disabling Pipeline Stages at Runtime

## 8.8 AUTOMATED BUILDS

*feature test macros, compiler flags*

The execution engine uses GNU99 extensions such as anonymous unions. In addition, since the engine depends on the `flow-tools` library that uses BSD extensions, it proved useful to include the `GNU_SOURCE` feature test macro. `GNU_SOURCE` allows to request and let the compiler enable a larger class of features. The execution engine is compiled using a `C99` compiler with `{-Wall, -Werror}` warning flags and `-O2` optimization level.

*cmake custom commands*

CMake [74] was used to ensure a compiler and platform independent build process [7]. Since the execution engine requires some headers/sources that are auto-generated by a python script, a custom command was added to run the script on compilation to add the generated files in `.build/` as shown in listing 56. These files are automatically included during the compilation and linked to the final binary. CMake also runs the build query scripts defined in `scripts/queries/` to generate some example `JSON` queries and moves them to the `examples/` folder ready to be used by the binary as shown in listing 56.

7 Detailed engine installation instructions are available in the Appendix.

```
# custom command to prepare auto-generated sources
add_custom_command (
  OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/auto-assign.h
         ${CMAKE_CURRENT_BINARY_DIR}/auto-assign.c
         ${CMAKE_CURRENT_BINARY_DIR}/auto-comps.h
         ${CMAKE_CURRENT_BINARY_DIR}/auto-comps.c
  COMMAND python ${CMAKE_SOURCE_DIR}/scripts/generate-functions.py
  COMMENT "Generating: auto-comps{h,c} and auto-assign.{h,c}"
)

# custom command to generate examples
file(GLOB pyFILES ${CMAKE_SOURCE_DIR}/scripts/queries/*.py)
foreach(pyFILE ${pyFILES})
  set(query "${pyFILE}_query")
  add_custom_command (
    OUTPUT ${query}
    WORKING_DIRECTORY ${CMAKE_SOURCE_DIR}/examples/
    COMMAND python ${pyFILE}
    COMMENT "Generating: JSON example query using ${pyFILE}"
  )
  list(APPEND queryFILES ${query})
endforeach(pyFILE)
```

Listing 56: F(*v2*): CMake Custom Commands

The CMake build process requires one to invoke quite a number of bash commands as shown in listing 57. In essence, a user does not need to know the CMake *way* to working around the build to use the execution engine. As such a `Makefile` is included that can make CMake calls to automate this operation. Additional targets are used to clean and generate `doxygen` documentation. The generated documentation goes in `doc/` and is subsequently deleted by a cleanup.

*makefile to automate cmake*

```
[engine] $ mkdir .build
[engine] $ cd .build
[.build] $ cmake ..
[.build] $ make
[.build] $ cd ..
[engine] $ rm -r .build

[engine] $ cat Makefile

make: build/Makefile
    (cd .build; make)
build/Makefile: build
    (cd .build; cmake -D CMAKE_PREFIX_PATH=$(CMAKE_PREFIX_PATH) ..)
build:
    mkdir -p .build
doc: Doxyfile
    (mkdir -p doc; doxygen Doxyfile)
clean:
    rm -f -r .build/ bin/ doc/
    rm -f -r examples/*.json
```

Listing 57: F(*v2*): Automating CMake Invocations

The Makefile can also take `CMAKE_PREFIX_PATH` as an argument and pass it on to CMake. `CMAKE_PREFIX_PATH` is used to supply arbitrary location of external libraries and include `PATH`. This can be useful since the `flow-tools` installation from source dispatches the library and headers in `/usr/local/flow-tools`.

*cmake prefix path*

```
[engine] $ make CMAKE_PREFIX_PATH=/usr/local/flow-tools
```

Listing 58: F(*v2*): CMake Prefix Paths

*packaging the parser*

There has never been a clean seamless way to install python flowy. Since the parser in the flowy implementation is eventually going to converge with the new execution engine, it is essential to provide an easy way to install and manage the parser. The software tool used in the python ecosystem to manage packages is `pip`. It uses a *flat* `requirements.txt` file to install all the package dependencies in one go. However, it requires that all the (to be) installed dependencies do not import external packages in their `egg` files. This turned to be the case for `numexpr` which is required by the parser, thereby resulting in failed installation. To circumvent the issue, a custom `Makefile`[8] was created that virtually adds a preprocessing pass to install `numexpr` dependencies before going forward with installation from `requirements.txt` as shown in listing 59.

```
make: numexpr
        (pip install -r requirements.txt)
numexpr: numpy
        (pip install numexpr==2.0.1)
numpy: cython
        (pip install numpy==1.6.1)
cython:
        (pip install Cython==0.15.1)
clean:
        rm -f -r build/
        rm -f -r src/*.pyc
        rm -f -r flowy-run/
        rm -f -r parsetab.py parser.out
        rm -f -r examples/output.h5
```

Listing 59: F(*v*2): Automating Parser Installation

## 8.9 REGRESSION TEST SUITE

A regression test-suite has been added in `tests/`. The suite `asserts` the numbers of results in each stage for a query-trace combination. It also looks for any segmentation faults if they may have occured. Tests can be run either individually or as a complete suite as shown in listing 60. The suite can also run in a verbose mode to see the expected and achieved result combination for run each test case.

```
[engine] $ make
[engine] $ tests/regression.py [-v]
.....................................................
----------------------------------------------------------------------
Ran 60 tests in 32.533s

OK

[engine] $ tests/test-query-http-tcp-session.py [-v]
..........
----------------------------------------------------------------------
Ran 10 tests in 8.672s

OK
```

Listing 60: F(*v*2): Regression Test Suite

8 Detailed parser installation instructions are available in the Appendix

# 9 PERFORMANCE EVALUATION

## 9.1 EXECUTION ENGINE PROFILING

The F(*v*1) execution engine had chunks of memory leaks. The blocks of heap memory were still reachable when the engine exited. As such, it was essential to profile the engine to properly deallocate all blocks before exit. Listing 61 shows the valgrind profile output of both versions. The 20kB of created and still living blocks in the current snapshot are due two libraries. The `dyld` library makes 81 `malloc` invocations that are not free'd by the library as shown in figure 25. On GNU/Linux, `dyld` is replaced is by `dlopen` which does not have this issue. The other set of libraries, `libsystem_c, libsystem_notify, libdispatch` make 10 `malloc` invocations that are again not free'd as shown in figure 25. These `malloc` calls invoke `localtime(...)` which uses `tzset(...)` to initialize and return `struct tm*`. This structure is never free'd apparently due to a bug in these libraries.


Figure 25: F(*v*2): Backtrace of Living Blocks on Exit Blocks

```
$ git checkout v0.1; make
$ valgrind bin/engine queryfile tracefile

==19000== LEAK SUMMARY:
==19000==    definitely lost: 6,912 bytes in 472 blocks
==19000==    indirectly lost: 0 bytes in 0 blocks
==19000==      possibly lost: 0 bytes in 0 blocks
==19000==    still reachable: 124,607 bytes in 710 blocks
==19000==         suppressed: 0 bytes in 0 blocks

$ git checkout master; make
$ valgrind bin/engine queryfile tracefile

==19164== LEAK SUMMARY:
==19164==    definitely lost: 0 bytes in 0 blocks
==19164==    indirectly lost: 0 bytes in 0 blocks
==19164==      possibly lost: 0 bytes in 0 blocks
==19164==    still reachable: 20,228 bytes in 37 blocks
==19164==         suppressed: 0 bytes in 0 blocks
```

Listing 61: F(*v*2): Valgrind-based Engine Profiling

## 9.2 BENCHMARKING SUITE

```
[engine] $ make; sudo benchmarks/nfql.py bin/engine trace[s]/ querie[s]/
benchmarking nfql ...
executing: [engine tcp-session trace-2012]:  1 2 3 4 5 6 7 8 9 10 (3.315148 secs)
executing: [engine tcp-session trace-2009]:  1 2 3 4 5 6 7 8 9 10 (0.034624 secs)
...

[engine] $ sudo benchmarks/silk.py trace[s]/ querie[s]/
benchmarking silk ...
executing: [silk http-tcp-session trace-2009]:  1 2 3 4 5 6 7 8 9 10 (0.102465 secs)
executing: [silk http-tcp-session trace-2012]:  1 2 3 4 5 6 7 8 9 10 (0.279106 secs)
...
```

Listing 62: F(v2): Automated Benchmarking

To be able to run and reproduce the benchmarking results as and when required it was essential to automate the whole process. The target design was to be able to use one script to run all sets of query-trace combination in one go for each network analysis application as shown in listing 62. The directories containing the traces and the queries required by the script can be supplied as command line arguments. Few examples are provided in `examples/`. The benchmarking suite only runs on python 2.7 and above. Attention is given to clear `pagecaches`, `dentries` and `inodes` before each iteration invocation as shown in listing 63. The script, therefore needs to run with `sudo` privileges. The results are saved in `benchmarks/results/`. SiLK query files are simply bash commands separated by a delimiter and are further discussed in the next section.

```
...
# clear pagecache, dentries and inodes
os.system('sync')
try:
  with open('/proc/sys/vm/drop_caches', 'w') as stream:
    stream.write('3\n')
...
```

Listing 63: F(v2) Benchmarking: Clearing Kernel Caches

## 9.3 RELATIVE COMPARISON WITH SILK

The benchmarking suite was used to run a number of queries over a public flow trace containing 20M records. We used trace 7 [1], from Simpleweb [2]. The input trace was compressed at `ZLIB_LEVEL` 5 using the `zlib` suite. It was also converted to `nfdump` and SiLK[3] [75] compatible formats and compressed with `zlib` keeping the same compression level. The suite was run on a high-end machine[4] with 24 cores of 2.5

1 http://traces.simpleweb.org/traces/netflow/netflow1/netflow000.tar.bz2
2 Simpleweb is a data repository of traffic traces from University of Twente
3 Detailed conversion instructions are available in the appendix
4 crystal.eecs.jacobs-university.de

GHz clock speed and 18 GiB of memory. The results and graphs are available on the `benchmarks` branch of the `git` repository.

```
# ratio: 0.0
rm -f $OUTPUT; \
rwfilter --packets=0-  --compression-method=zlib --fail=$OUTPUT $INPUT;

# ratio: 0.2
rm -f $OUTPUT; \
rwfilter --packets=3- --compression-method=zlib --pass=$OUTPUT $INPUT;

# ratio: 0.4
rm -f $OUTPUT; \
rwfilter --packets=2- --compression-method=zlib --pass=$OUTPUT $INPUT;

# ratio: 0.6
rm -f $OUTPUT; \
rwfilter --packets=2- --compression-method=zlib --fail=$OUTPUT $INPUT;

# ratio: 0.8
rm -f $OUTPUT; \
rwfilter --packets=3- --compression-method=zlib --fail=$OUTPUT $INPUT;

# ratio: 1.0
rm -f $OUTPUT; \
rwfilter --packets=0- --compression-method=zlib --pass=$OUTPUT $INPUT;
```

Listing 64: Filter Stage: SiLK Queries

The first set of queries attempt to stress the filter stage. They use varying values on the packet field offset to determine the amount of flow records that are passed by the filter. The resultant filtered records are written to `flow-tools` compatible file format and compressed at at `ZLIB_LEVEL` 5 using `zlib` suite. The ratio of the number of filtered records in the output trace to the number of the flow records in the input trace is plotted against time. SiLK example queries using `rwfilter` are shown in listing 64. The queries for NFQL, `flowtools`, `nfdump` are similar and can be referenced from `benchmarks` branch[5]. The evaluation results are shown in figure 26.

*stressing the filter*

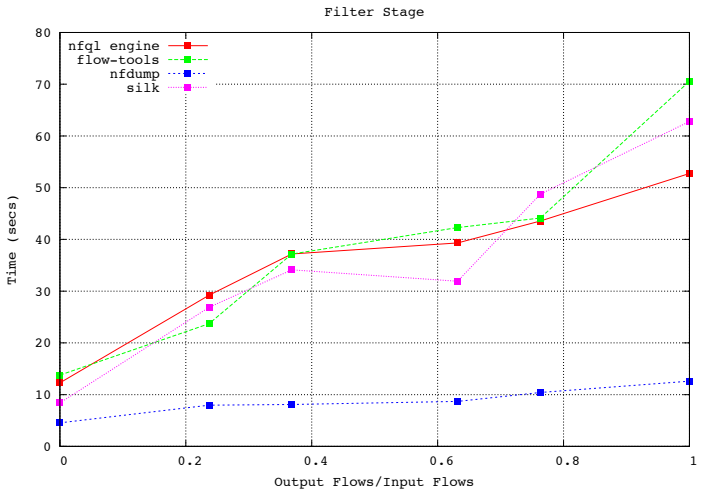



Figure 26: Filter Stage: NFQL vs SiLK, Flow-Tools, Nfdump

5 `benchmarks/august/filter/queries/{flowtools, nfql, nfdump, silk}`

*stressing the filter: discussion*

It can be seen that the performance of the filter stage in NFQL is comparable to that of `flowtools` and SiLK. SiLK takes less time on lower ratios, but then again SilK and `nfdump` also use their own proprietary format for trace files. As a result, the amount of data that needs to read (or written) may be different to what it is for NFQL and `flowtools`. On the other hand, `nfdump` appears to be significantly faster than the rest. This is because `nfdump` lacks `zlib` support, and as such the files that were read and written used `lzo` compression scheme which trades space for achieving faster compression and decompression. It is important to note, that all the tools were single-threaded in this evaluation, and did not completely utilize the 24 cores that were at their disposal. It comes as a realization, that filtering the input using multiple threads by memory mapping the trace and adding `lzo` compression will drastically improve NFQL's filter performance.

```
# ratio: 0.1
rm -f /tmp/filter.rwz $OUTPUT; \
rwfilter --packets=0- --compression-method=zlib --pass=/tmp/filter.rwz $INPUT; \
rwsort --fields=sIP /tmp/filter.rwz | \
rwgroup --id-fields=sIP --summarize \
        --compression-method=zlib --output-path=$OUTPUT;

# ratio: 0.4
rm -f /tmp/filter.rwz $OUTPUT; \
rwfilter --packets=0- --compression-method=zlib --pass=/tmp/filter.rwz $INPUT; \
rwsort --fields=sIP,dIP,sPort,dPort /tmp/filter.rwz | \
rwgroup --id-fields=sIP,dIP,sPort,dPort --summarize \
        --compression-method=zlib --output-path=$OUTPUT;

# ratio: 0.8
rm -f /tmp/filter.rwz $OUTPUT; \
rwfilter --packets=0- --compression-method=zlib --pass=/tmp/filter.rwz $INPUT; \
rwsort --fields=5-9 /tmp/filter.rwz | \
rwgroup --id-fields=5-9 --summarize \
        --compression-method=zlib --output-path=$OUTPUT;

# ratio: 1.0
rm -f /tmp/filter.rwz $OUTPUT; \
rwfilter --packets=0- --compression-method=zlib --pass=/tmp/filter.rwz $INPUT; \
rwsort --fields=1-9 /tmp/filter.rwz | \
rwgroup --id-fields=1-9 --summarize \
        --compression-method=zlib --output-path=$OUTPUT;
```

Listing 65: Grouper Stage: SiLK Queries

*stressing the grouper*

The second set of queries attempt to stress the grouper stage. They reuse the filter query that produces a 1.0 ratio to allow the grouper to receive the entire trace as a filtered recordset. The grouper part of the query then gradually increases the number of grouping terms in the DNF expression to increase the output/input ratio. The resultant groups are again written as `flowtools` files using the same `zlib` compression level. The ratio of the number of groups formed to the number of the input filtered records is plotted against time. SiLK example queries using a combination of `rwfilter-rwsort-rwgroup` are shown in listing 65. The queries for NFQL are similar and can be referenced from `benchmarks` branch[6]. `nfdump` and `flowtools` do not support grouping, and therefore are not considered in this evaluation. The evaluation results are shown in figure 27.

6 `benchmarks/august/grouper/queries/{nfql, silk}`

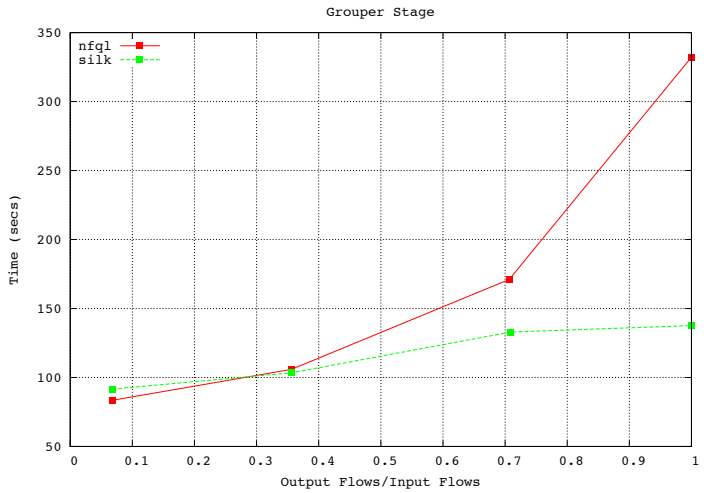



Figure 27: Grouper Stage: NFQL vs SiLK

The evaluation graph reveals that the performance of the NFQL grouper stage is close. The time taken by the tools are comparable on lower ratios, but on higher ratios, NFQL starts to drift apart. SiLK, however remains almost linear throughout the evaluation. Since most of the time is taken in writing the records to files, it is unclear whether SiLK's usage of a proprietary format which may reduce reads/writes is responsible for the drift on higher ratios. SiLK's `rwgroup` tool is also supplied a `-summarize` flag in all the queries. This gives SiLK the leverage to not store information about which members are part of the group. NFQL on the other hand needs to allocate resources (which may take time) to keep this information in its data structures, since the ungrouper later may request to write the members of a group while unfolding the tuples. The ungrouper although was disabled in this evaluation, the allocation of space for group members was not. It is also important to note that both the tools again remained single-threaded throughout the evaluation. SiLK took an advantage of an inherent concurency arising from how the query is structured as one single `bash` command using pipes. The pipe between `rwsort` and `rwgroup` makes the two process run concurrently, the effect of which gets more pronounced on higher ratios and can be a drift determining factor. The profiling results from GNU `gprof` [76] indicate that 60% of the time is taken in `qsort` comparator calls. As a result, it comes as no surprise, that bifurcating `qsort` invocation to multiple threads and later merging the results back using merge sort will help parallelize the grouper stage and maybe reduce the drift on higher ratios. In addition, since all of the evaluation queries had grouping terms using an equality comparator, NFQL can introspect such a grouping rule to dynamically optimize processing searches using a hashtable and turn to `qsort` based grouping only as a fallback.

*stressing the grouper: discussion*

```
# ratio: 0.0
rm -f /tmp/filter.rwz /tmp/grouper.rwz $OUTPUT; \
rwfilter --packets=0- --compression-method=zlib --pass=/tmp/filter.rwz $INPUT; \
rwsort --fields=1-9 /tmp/filter.rwz | \
rwgroup --id-fields=1-9 --summarize \
        --compression-method=zlib --output-path=/tmp/grouper.rwz; \
rwfilter --packets=0- --compression-method=zlib /tmp/grouper.rwz \
         --fail=$OUTPUT;

# ratio: 0.2
rm -f /tmp/filter.rwz /tmp/grouper.rwz $OUTPUT; \
rwfilter --packets=0- --compression-method=zlib --pass=/tmp/filter.rwz $INPUT; \
rwsort --fields=1-9 /tmp/filter.rwz | \
rwgroup --id-fields=1-9 --summarize \
        --compression-method=zlib --output-path=/tmp/grouper.rwz; \
rwfilter --packets=3- --compression-method=zlib /tmp/grouper.rwz \
         --pass=$OUTPUT;

# ratio: 0.4
rm -f /tmp/filter.rwz /tmp/grouper.rwz $OUTPUT; \
rwfilter --packets=0- --compression-method=zlib --pass=/tmp/filter.rwz $INPUT; \
rwsort --fields=1-9 /tmp/filter.rwz | \
rwgroup --id-fields=1-9 --summarize \
        --compression-method=zlib --output-path=/tmp/grouper.rwz; \
rwfilter --packets=2- --compression-method=zlib /tmp/grouper.rwz \
         --pass=$OUTPUT;

# ratio: 0.6
rm -f /tmp/filter.rwz /tmp/grouper.rwz $OUTPUT; \
rwfilter --packets=0- --compression-method=zlib --pass=/tmp/filter.rwz $INPUT; \
rwsort --fields=1-9 /tmp/filter.rwz | \
rwgroup --id-fields=1-9 --summarize \
        --compression-method=zlib --output-path=/tmp/grouper.rwz; \
rwfilter --packets=2- --compression-method=zlib /tmp/grouper.rwz \
         --fail=$OUTPUT;

# ratio: 0.8
rm -f /tmp/filter.rwz /tmp/grouper.rwz $OUTPUT; \
rwfilter --packets=0- --compression-method=zlib --pass=/tmp/filter.rwz $INPUT; \
rwsort --fields=1-9 /tmp/filter.rwz | \
rwgroup --id-fields=1-9 --summarize \
        --compression-method=zlib --output-path=/tmp/grouper.rwz; \
rwfilter --packets=3- --compression-method=zlib /tmp/grouper.rwz \
         --fail=$OUTPUT;

# ratio: 1.0
rm -f /tmp/filter.rwz /tmp/grouper.rwz $OUTPUT; \
rwfilter --packets=0- --compression-method=zlib --pass=/tmp/filter.rwz $INPUT; \
rwsort --fields=1-9 /tmp/filter.rwz | \
rwgroup --id-fields=1-9 --summarize \
        --compression-method=zlib --output-path=/tmp/grouper.rwz; \
rwfilter --packets=0- --compression-method=zlib /tmp/grouper.rwz \
         --pass=$OUTPUT;
```

Listing 66: Group Filter Stage: SiLK Queries

*stressing the group filter*

The third set of queries attempt to stress the group filter stage. They reuse the filter and grouper queries that produce a 1.0 ratio to allow the group filter to receive the entire trace as input. That means, each flow record of the original trace now becomes a group record for the group filter. The group filter then reuses the same varying values of the packet offset to determine the amount of groups that are filtered ahead. The filtered groups are again written as `flowtools` files using the same `zlib` compression level. The ratio of the number of filtered groups formed to the number of the input group records is plotted against time. SiLK example queries using a combination of `rwfilter-rwsort-rwgroup-rwfilter` are shown in listing 66. The queries for NFQL are similar and can be referenced from `benchmarks` branch[7]. The evaluation results are shown in figure 28.

7 `benchmarks/august/groupfilter/queries/{nfql, silk}`

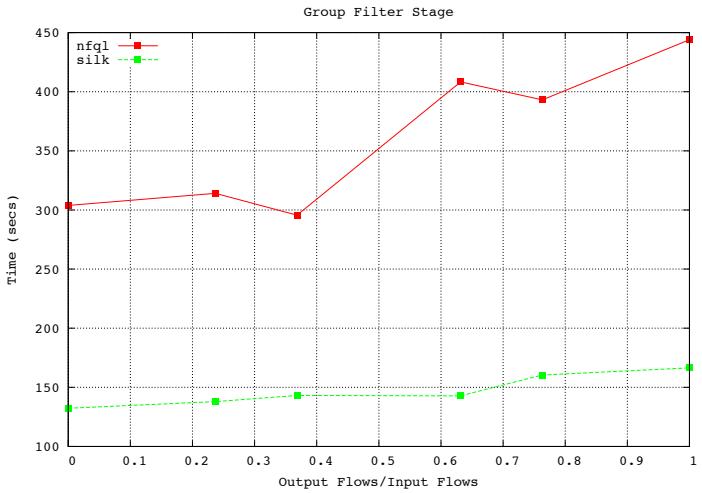



Figure 28: Group Filter Stage: NFQL vs SiLK

It can be seen that the timings of NFQL are far apart from that of SiLK. It is due to the drift already created by the grouper at 1.0 ratio in the previous stage. As a result, the group filter comes into play only after 300 seconds, whereas SiLK's group filtering already starts just below 150 seconds. Even if we normalize the graph, it can be observed that the group filter has a significantly higher slope. This is because it is only executed once the grouper returns, and therefore has to reiterate the groups to make a filtering decision.

*stressing the groupfilter: discussion*

```
# ratio: 0.0
"dnf-expr": [
  { "clause": [ {
      "term": { "op": { "type": "RULE_ABS", "name": "RULE_EQ" },
                "offset": { "f1_name": "dPkts", "f2_name": "dOctets", ... } }
    }]}
]

# ratio: 0.2
"dnf-expr": [
  { "clause": [ {
      "term": { "op": { "type": "RULE_ABS", "name": "RULE_EQ" },
                "offset": { "f1_name": "prot", "f2_name": "prot", ... } }
     }]},
  { "clause": [ {
      "term": { "op": { "type": "RULE_ABS", "name": "RULE_EQ" },
                "offset": { "f1_name": "tcp_flags", "f2_name": "tcp_flags", ... } }
    }]}
]

# ratio: 0.5
"dnf-expr": [
  { "clause": [ {
      "term": { "op": { "type": "RULE_ABS", "name": "RULE_EQ" },
                "offset": { "f1_name": "prot", "f2_name": "prot", ... } }
    }]}
]

# ratio: 1.0
"dnf-expr": [
  { "clause": [ {
      "term": { "op": { "type": "RULE_ABS", "name": "RULE_EQ" },
                "offset": { "f1_name": "srcaddr", "f2_name": "srcaddr", ... } }
    }]}
]
```

Listing 67: Merger Stage: NFQL Queries

*stressing the merger*

The fourth set of queries attempt to stress the merger stage. They reuse the filter, grouper and group filter queries that produce a 1.0 ratio. These queries are then run in two separate branches to produce identical filtered group records. The merger then applies the queries as listed 67 to produce different output to input ratios. The groups that are merged are again written as `flowtools` files using the same `zlib` compression level. The ratio of the number of merged groups to twice[8] the number of flow records in the original trace is plotted against time. The complete queries can be referenced from `benchmarks` branch[9]. The evaluation results are shown in figure 29. A data point for SiLK for the 0.2 ratio is not available since the NFQL query executed at that data point uses `OR` expressions which are not supported by SiLK. As a result, an equivalent SiLK query is not formulated.

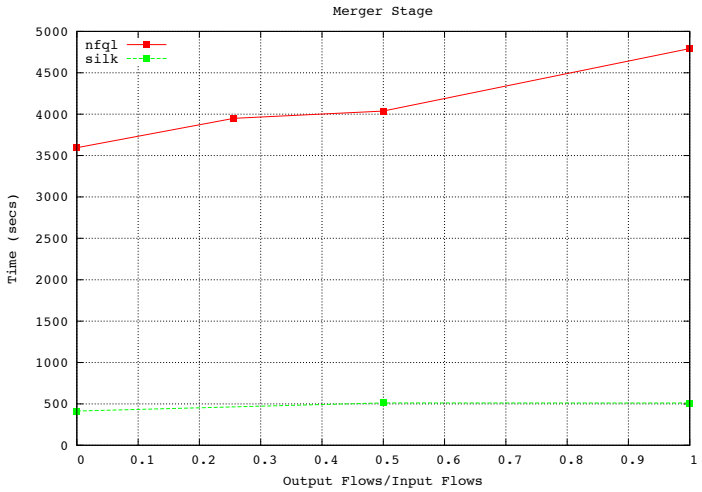



Figure 29: Merger Stage

*stressing the merger: discussion*

It can be seen that the merger is the most performance hit and time consuming stage of the NFQL pipeline thus far. It is due to the fact that the merger is working on twice the number of flow records than any other previous stage. In addition, each branch is writing the results of the filter, grouper and group filter stage to `flowtools` files. As a result, the amount of disk I/O involved is twice as much as well. Even though each branch is delegated to a separate core, most of time is taken in writing these results to the file. These results although look less promising, they are way better than the previous merger implementation. The newer merger takes advantage of sorted nature of filtered groups and therefore can signifantly reduce the number of merger matches. It also writes a merged group record to file only once despite the number of times it has matched. Without

8 Each branch pushes the entire trace as an input to the merger.

9 `benchmarks/august/merger/queries/{nfql, silk}`

these optimizations, running such queries on the merger would keep the CPU churning for days without results.

The last set of queries attempt to stress the ungrouper stage. They reuse the entire merger queries as is, but enable the ungrouper now as well. This means, that the ungrouper now attempts to unfold the merged groups returned by the merger to write their member flow records to `flowtools` files. However, since the merger receives each flow record as its own filtered group, each merged group has only one member. As a result, the ungrouper ends up executing its logic and rewriting the merged groups as `flowtools` files using the using the same `zlib` compression level. The ratio of the number of result flow records to twice the number of the flow records in the input trace is plotted against time. The queries for NFQL are similar to that of the merger and can be referenced from `benchmarks` branch[10]. The evaluation results are shown in figure 30.

*stressing the ungrouper*

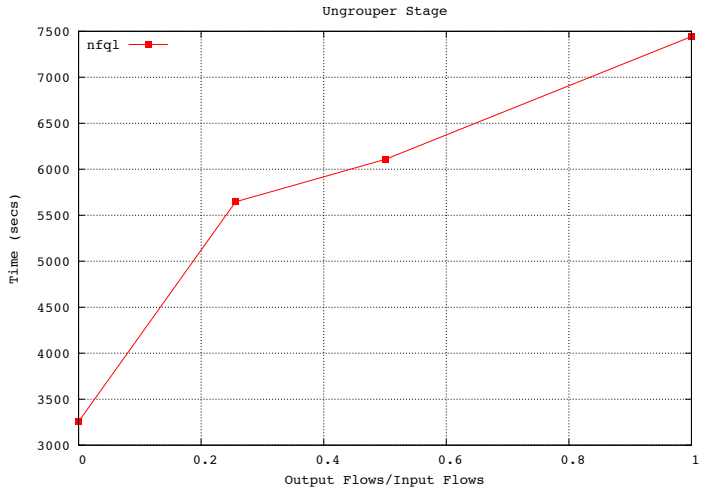



Figure 30: Ungrouper Stage

It can be seen that the evaluation behavior is similar to that shown by the merger, but is just more involved. This is because the ungrouper also has to write the entire merged groupset to file, as is done by merger. This is the reason for the execution engine in taking twice the amount of time. In fact, this evaluation is performed only to stress the functioning of the ungrouper and running queries that merge each filtered group filtered individually is less useful in practise. Ideally, the execution engine must shutdown the ungrouper if the merger merges every filtered group record in its output. This is because, such a behavior implicitly eliminates the need of the ungrouper.

*stressing the ungrouper: discussion*

10 `benchmarks/august/ungrouper/queries/nfql`

# 10 FUTURE WORK AND CONCLUSION

The C execution engine is a major leap forward to a fast and robust implementation of NFQL. It is the first time when a NFQL implementation can actually be used on practically sized flow traces. However it is far from being deemed complete. The fast-paced development is clear from the fact that the engine's issue tracker [1] has a 1 : 1 ratio of closed to open issues. As a result, the future outlook of the engine is divided into major goals and minor issues that still need to be addressed.

## 10.1 MAJOR GOALS

The execution engine uses the `flow-tools` API to read and parse the flow-records. The `flow-tools` API can only parse NetFlow *v5* records, thereby sandboxing the engine's functionality. It restricts the engine's understanding of a flow to a fixed NetFlow *v5* format and inhibits the capability to parse IP*v6* flows. Since NetFlow *v9* and now with IPFIX, the flow format is dynamically parsed using its accompanied template. It relaxes the definition of a flow and gives more power to the sender on how the data should be conglomerated together. There are three well-known implementations of IPFIX: CERT' `libfixbuf` [2], Fraunhofer FOKUS' `libipfix` [3] and WAND' `Maji` [4]. The former two APIs appear to be under heavy development and one of them is foreseen to be used by the engine to provide IPFIX capability in the future.

*ipfix support*

The python parser is currently unused. The idea at this stage is to allow the parser to parse and validate the flow query and generate an equivalent intermediate `JSON` format. This is supposed to be a preprocessing step. The `JSON` flowquery file can then later be supplied to the execution engine at runtime. It can also be foreseen to push parts of the `JSON` flowquery using a RESTful interface to multiple map/reduce jobs running the execution engine to completely distribute the workflow for faster processing. The first step to achieve such a convergence will be by removing the processing pipeline implementation code out of the python framework. The reverse engineered package and class diagrams generated using `pyreverse` are available in `parser/docs/` and will help one get started with this task. The installation and usage instructions available in the appendix will make the convergence head start a breeze. Alternatively, it is also possible to re-implement the frontend parser from scratch using `lex/YACC`.

*a frontend parser*

1 https://github.com/vbajpai/mthesis-src/issues
2 http://tools.netsa.cert.org/fixbuf/
3 http://libipfix.git.sourceforge.net/
4 http://research.wand.net.nz/software/maji.php

```
$ bin/engine examples/filter/filter-1.0.json examples/trace-2012.ftz

   % time     name
   66.67  ... ft_read
   33.33  ... ftio_read
   ...
```

Listing 68: F(v2) : GNU gprof on Filter

*memory mapping the tracefile, multithreaded filtering*

The execution engine currently memory maps the (usually small) `JSON` query file. However, the (usually large) trace file is not memory mapped. This is because the engine uses the `flowtools` library which uses `read` system calls to read the trace. Listing 68 shows the GNU `gprof` [76] profiling results when stressing only the filter stage. It is clear that the majority of the time is taken in reading the flow records from the trace file. Memory mapping the trace file to engine's virtual space will increase I/O performance. This is because the `mmap` region is a kernel's cache itself avoiding the need to create further copies in userspace. In addition, accessing the engine's virtual space is in orders of magnitude faster to making system calls and a linear walk through the file only requires disk access across page boundaries. Memory mapping will also allow to parallelize the filter stage by taking advantage of the available random file access.

*lzo compression support*

The execution engine uses the `flowtools` API to write the results as NetFlow v5 records. Writing the results to flat files is important since it allows one to not only blackbox a pipeline stage operation, but also help plug the results to another network analysis tool, thereby increasing interoperability. The `flowtools` API, however, currently only support `zlib` compression. Contemporary tools like `nfdump` and SiLK generally use `lzo`[5] compression to achieve faster compression and decompression rates when doing I/O operations. Adding `lzo` compression support in the `flowtools` API will further decrease the flow read/write times of the execution engine.

```
$ bin/engine examples/grouper/grouper-1.0.json examples/trace-2012.ftz

   % time     name
   40.00  ... comp_uint32_t
   12.73  ... qsort_comp
    7.27  ... comp_uint16_t
    5.45  ... comp_uint8_t
    ...
```

Listing 69: F(v2) : GNU gprof on Grouper

*parallelized grouper qsort*

The grouper in F(v2) sorts the filtered recordset on all the grouping rules. This implies that the comparator has to make nested calls using all the rules until a tie-breakup is resolved. This allows the grouper to make a nested `bsearch` invocation for the generic case to reduce a linear pass through the filtered recordset. However, the number of comparator calls increase and now account for almost 60% of the time

5 http://www.oberhumer.com/opensource/lzo/

taken in the grouper as can be seen from the GNU `gprof` results shown in listing 69. The `qsort` call is currently single-threaded. Since the size of the filtered recordset is known before the invocation, bifurcating the `qsort` invocation to multiple threads and merging the results back using merge sort will help parallelize the operation.

The current grouper implementation runs in $O(p * n * lg(n)) + O(n) + O(n * lg(k))$, while the merger runs in $O(n^m)$ time, where $n$ is the number of flow records, $m$ is the number of branches, $k$ is the number of unique filtered records and $p$ is the number of grouper terms. Therefore, a search tree lookup would help bring the runtime costs down, whereby one of the fields will be traversed sequentially in $O(n)$ time and for each field, a comparison will be performed by search tree lookups in $O(log(n))$ time bringing down the overall complexity of both merger and grouper to $O(nlog(n))$ respectively. In addition, letting the execution engine override the search tree lookups by hash table lookups for equality operators will further bring down the runtime to $O(n)$ for this specific case.

*search tree and hash table lookups*

The execution engine currently has limited multithreading. Each branch in the pipeline runs on a separate thread. However, this implies that the merger and ungrouper stages still remain single-threaded. It is possible to handle the merger's outermost branch loop using multiple threads in a non-blocking fashion to improve performance , or by writing a `pthreads` wrapper that auto detects the number of available cores, creates a appropriate size thread pool and equally divides the tasks among the threads. This would also lead to an increased complexity of managing mutual-exclusion of shared memory, but the performance gains will go a long way.

*multithreaded merger*

F(*v2*) introduces a regression test-suite framework that helps keep the quality of the execution engine at a certain minimum threshold. However, the execution engine is as robust as its test-suite. The tests defined in the regression test-suite are not exhaustive and do not touch every aspect of the engine's functionality. For instance, the queries executed by the suite only hit few of many comparator operations supported by the engine. Unless every aspect of the engine's offered functionality has a test case associated with it, it cannot be guaranteed that the engine will not fail in one or more corner cases not hit by the suite. As a result, it is imperative that addition of more test-cases to touch these corner cases will make the execution engine better

*exhaustive regression test-suite*

## 10.2 MINOR ISSUES

The execution engine in F(*v2*) introduces two approaches to boost up the speed of the grouper. One of the approaches uses `qsort` on each requested grouping rule and a nested `bsearch` to find a more specific filtered recordset pointer to signicantly reduce the linear pass when grouping members. This approach works well for a generic case and

spans different type of comparator operations. This approach is further optimized when the grouping is requested on equality comparison. In such a scenario, the need to perform a `bsearch` goes away, and groups can be formed in a single linear pass on the filtered recordset. The engine currently uses the second approach, while the first approach is in the source code history. Ideally, the engine must be smart to use the second approach when equality grouping is requested, and fallback to the first approach otherwise.

*dynamically choosing an appropriate grouper approach*

The group filter is currently a separate module in the execution engine as shown in listing 70. This means that it is invoked only after the grouper returns. As a consequence, the group filter has to iterate over the groups again in order to make decision on which ones to filter out. This decision can also be made as soon as the group is realized in the grouper. Such a refactor will phase out the distinct boundary between the grouper and group filter in the source code. However, it will eliminate a duplicate iteration of the groupset in the group filter thereby improving performance. The performance gains will be more visible when the ratio of the number of groups formed to the number of input filtered records is high.

*inline group filter*

```
if (groupfilter_enabled) {

  branch->gfilter_result = groupfilter(
                                       branch->num_groupfilter_clauses,
                                       branch->groupfilter_clauseset,

                                       branch->grouper_result,

                                       branch->data,
                                       branch->branch_id
                                      );
  ...
}
```

Listing 70: F(*v*2) : Group Filter as an Independent Module

The execution engine currently uses an additional data structure (`struct json`) to hold the parsed `JSON` query. This data structure is then read by `prepare_flowquery(...)` to generate `struct flowquery` which is eventually used by the pipeline stages as shown in listing 71. In essence, the intermediate `struct` is not needed, and is a redundant datastructure. There is no reason why `parse_json_query(...)` cannot directly read the query elements into `struct flowquery` and is a future refactor item.

*eliminating redundant structs*

```
struct json*
json_query = parse_json_query(param_data->query_mmap);
struct flowquery*
fquery = prepare_flowquery(param_data->trace, json_query);
```

Listing 71: F(*v*2): Redundant Structs

The ungrouper reads the tuples of merged group records and unfolds each one of them to create a stream. Each such stream is a

collection of flow-records that passed the whole pipeline. However, it is possible that a flow record is part of multiple groups in a single group tuple, and is therefore outputted multiple times. The engine currently does not eliminate such flow records repetitions. In the future, the engine can take an option if one desires to eliminate such repetitions. It is also does not order the flow records according to their timestamps as defined in the NFQL specification.

*eliminating redundant flows from a stream*

## 10.3 CONCLUSION

The NFQL execution engine has come a long way in a short time. It now consists of a robust implementation of the processing pipeline that adapts itself to the kind of query provided at runtime to dynamically decide the type of data and the type of operation to be performed. It is flexible to be able to read and parse an entire flowquery at runtime. It is fast to be able to process millions of flow traces in matter of seconds. It is portable and can seamlessly build on multiple Unix flavors and is verifiable using a regression test-suite that will allow future developers to work further to improve the engine with confidence.

# Part IV

# APPENDIX

The appendix is used to supplement the work done in the thesis with clear instructions on how to use the end-product. It also includes instructions on how to build and use other products that were used as reference points when conducting performance evaluation to allow the process to be easily repeatable by others. The organization of the appendix is described below.

In section A we discuss step-by-step instructions on how to install NFQL parser and the execution engine on Debian-based systems and OS X. This walkthrough is going to help not only the users, but also future developers to quickly get started with NFQL implementations.

In section B we discuss SiLK installation on Debian-based systems. We then investigate which SiLK tool can be used to achieve the desired effect of each stage of the NFQL processing pipeline. Additional SiLK analysis and capture tools are also discussed. We end the section, by enumerating the process of converting `flow-tools` compatible traces to SiLK proprietory format.

In section C we enlist the major iterations of the development lifecycle of the execution engine starting from $\mathrm{F}(v2.0)$ to $\mathrm{F}(v2.5)$. An explanation of the feature set in each iteration is followed by instructions on how `git tags` can be used to go back to a previous version. The appendix is concluded by a list of acronyms used in this work.

# A

# NFQL INSTALLATION AND USAGE

The NFQL Engine uses CMake to prepare the build. As a result, the installation process is quite seamless. The external dependencies that need to be installed are shown in listing 72. The build and usage instructions are also listed. The build environment was tested on 64-bit machines runing Debian Squeeze, Ubuntu 10.04 and 12.04.

*nfql engine on debian/ubuntu*

```
$ sudo apt-get install cmake
$ sudo apt-get install flow-tools-dev
$ sudo apt-get install zlib1g-dev
$ sudo apt-get install libjson0-dev
$ sudo apt-get install doxygen
$ sudo apt-get install graphviz

[engine] $ make
[engine] $ bin/engine querfile tracefile
[engine] $ make doc
[engine] $ make clean
```

Listing 72: NFQL Engine on Debian/Ubuntu

The NFQL parser uses the `pip` packaging and installation environment to set itself up. Besides some of the external dependencies required by the parser, it is highly recommended to use `virtualenv` and `virtualenvwrapper` to create a virtual environment where all the python libraries will be installed. This helps isolate the user's system-level python libraries and avoids any conflicts that may occur otherwise. The external dependencies and `pip` environment installation is shown in listing 73.

*nfql parser on debian/ubuntu*

```
$ sudo apt-get install libhdf5-serial-dev
$ sudo apt-get install liblzo2-dev

$ sudo apt-get install python-pip
$ sudo pip install pip --upgrade
$ sudo pip install virtualenv
$ sudo pip install virtualenvwrapper
```

Listing 73: NFQL Parser Dependencies on Debian/Ubuntu

A python virtual environment is setup from the parent directory of the parser by running `mkvirtualenv`. Once within the virtual environment, `make` takes care of installing all the python libraries required by the parser in one go. It installs `cython`, `numpy`, `numexpr` in a preprocessing step, and then lets `pip` handle the rest of the installation using a `requirements` file. The list of python libraries installed can be checked by running `pip freeze`. The parser usage instructions are provided in the next section. The virtual environment can be deactivated using `deactivate` and brought back again using `workon $NAME`. It can be destroyed using `rmvirtualenv` as shown in listing 74

```
[parser] $ mkvirtualenv parser
(parser)
[parser] $ make
(parser)
[parser] $ pip freeze

(parser)
[parser] $ python ft2hdf.py traces/ output.h5
(parser)
[parser] $ python printhdf.py output.h5
(parser)
[parser] $ python print_hdf_in_step.py output.h5
(parser)
[parser] $ python flowy.py queryfile

(parser)
[parser] $ make clean
(parser)
[parser] $ deactivate

[parser] $ rmvirtualenv parser
```

Listing 74: Platform-Agnostic NFQL Parser Build

*nfql engine on osx*

The NFQL engine installation on OS X is quite similar. The external dependencies can either be installed from source or using a package manager. It is recommended to use Homebrew[1] to install the external dependencies since it does not require and plague the install with `sudo` privileges and installs the packages at `/usr/local/`. The instructions are shown in listing 75. Alternatively, MacPorts[2] can also be used to build and install the tool as shown in listing 76. The build environment was tested on OS X Lion 10.7.

```
$ brew install cmake json-c
$ brew install doxygen graphviz

$ wget http://dl.dropbox.com/u/500389/flow-tools-0.68.4.tar.bz2
$ tar -xvf flow-tools-0.68.4.tar.bz2

[flow-tools-0.68.4] $ ./configure
[flow-tools-0.68.4] $ make
[flow-tools-0.68.4] $ make install

[engine] $ make CMAKE_PREFIX_PATH=/usr/local/flow-tools/
[engine] $ make doc

[engine] $ bin/engine queryfile tracefile
[engine] $ make clean
```

Listing 75: NFQL Engine on OS X using Homebrew

```
$ sudo port install cmake
$ sudo port install flow-tools
$ sudo port install json-c
$ sudo port install doxygen graphviz

[engine] $ make CMAKE_PREFIX_PATH=/opt/local
[engine] $ make doc

[engine] $ bin/engine queryfile tracefile
[engine] $ make clean
```

Listing 76: NFQL Engine on OS X using MacPorts

1 http://mxcl.github.com/homebrew/
2 http://www.macports.org

There is a ruby `formula` for `flow-tools` in the Homebrew package repository, however currently it fails to build. The *v*0.68 stable build at splintered.net[3] has a bug[4] that inhibits it from correctly parsing the trace timestamps on 64-bit machines. A forked flow-tools branch[5] resolves the 64-bit issues but then the latest stable release *v*0.68.5.1 does not build successfully. The previous release of the forked branch *v*0.68.4 works well and is used by the execution engine on OS X.

*flow-tools issues on osx*

```
$ brew install hdf5
$ brew install lzo

$ brew install python --framework
$ export PATH=/usr/local/share/python:$PATH

$ easy_install pip
$ pip install pip --upgrade
$ pip install virtualenv
$ pip install virtualenvwrapper
$ source /usr/local/bin/virtualenvwrapper.sh
```

Listing 77: NFQL Parser Dependencies on OS X

The NFQL parser installation on OS X only differs in the *way* how external libraries are installed. Homebrew is used to install the external packages. The default python framework on OS X is fairly old, it is in the best interest to also upgrade it using Homebrew. The older python packaging environment `easy_install` is used to install `pip`, and then `pip` is used to upgrade itself to the latest revision. The packaging from there is on handled by the `pip` environment. The external dependencies and `pip` environment installation is shown in listing 77, while the virtual environment setup and parser build process is exactly the same as shown in listing 74.

*nfql parser on osx*

3 http://www.splintered.net/sw/flow-tools/
4 http://ensight.eos.nasa.gov/FlowViewer/faq.html
5 https://code.google.com/p/flow-tools/

# B

# SILK INSTALLATION AND USAGE

SiLK[1] is a network traffic collection and analysis tool developed and maintained by the CERT Network Situational Awareness Team (CERT NetSA) at Carnegie Mellon University. SiLK was used as a reference point to compare the performance of the NFQL execution engine. This section illustrates the instructions on how to install and use SiLK. The installation was tried on Debian Squeeze and is pretty straight forward as shown in listing 78.

```
$ wget http://tools.netsa.cert.org/releases/silk-2.4.7.tar.gz
$ sha1sum silk-2.4.7.tar.gz | grep 2ff0cd1d00de70f667728830aa3e920292e99aec

$ ./configure
$ make
$ sudo make install
$ sudo ldconfig
```

Listing 78: SiLK Installation on Debian

The design and implementation of SiLK differs a lot from NFQL. SiLK believes in the philosophy of a tool performing a single task well. For instance, in SiLK there are separate tools to perform the task of each stage of the NFQL processing pipeline. The stage functionality is not full-fledged though. The grouping and merging operations can only be performed using an equality operator. This is assumed in the tool, thereby allowing it to perform optimization such as using hash tables to perform lookups. The usage instruction of tools that can perform (if not) an equivalent NFQL stage operation is given in listing 79.

*nfql equivalent silk tools*

```
# absolute filtering (dst port 80 or (src port 80 and dst port 25)
$ rwfilter --dport=80 --pass=out1.rwf.gz in.rwf.gz
$ rwfilter --sport=80 --dport=25 --pass=out2.rwf.gz in.rwf.gz

# concatenating flow records
$ rwcat --output=out.rwf.gz out1.rwf.gz out2.rwf.gz
$ rwcat out1.rwf.gz out2.rwf.gz >> out.rwf.gz
$ rwappend [--create] out.rwf.gz out1.rwf.gz out2.rwf.gz

# grouping flow records and setting thresholds
$ rwuniq --field=1 out.raw --bytes --packets=1000 --flows=200
$ rwgroup --id-field=1,2,3,4 --delta-field=9 --delta-value=3600 in.rwf.gz > out.rwf.gz

# merging flow records
$ rwmatch --relate=1,2 --relate=2,1 \
          --relate=3,4 --relate=4,3 \
          query.rwf.gz response.rwf.gz stdout
```

Listing 79: NFQL Equivalent SiLK Tools

1 http://tools.netsa.cert.org/silk/

*additional silk analysis and capture tools*

There are also stringent requirements to how flow-data needs to be organized before it can be piped into a tool. The grouping tool, for instance, assumes that the to-be supplied input flow data is already sorted on the field column. These requirements made it a little cumbersome to design a full-fledged NFQL query. The final query had over a dozen of SiLK tools piped together and saved as a bash script. These bash scripts were then called by the benchmarking suite for performance evaluation. Few additional tools available in the SiLK's repertoire are shown in listing 80.

```
# reading flow records
$ rwcut out.rwf.gz

# generating statistical summary
$ rwstats --overall-stats out.rwf.gz

# creating time series (10 minute interval)
$ rwcount --bin-size=600 out.rwf.gz

# sorting flow records (on srcIP)
$ rwsort --fields=1 --output=out-sort.rwf.gz out.rwf.gz

# remove duplicate flow fecords
$ rwdedupe --stime-delta=100 out1.rwf.gz out2.rwf.gz > out.rwf.gz

# splitting flow records
$ rwsplit out.rwf.gz --basename=splits --flow-limit=1000

# show silk file characteristics
$ rwfileinfo out.rwf.gz

# generate flows from text files
$ rwtuc --fields=1-9 out.txt > out.rwf.gz

# generate flows from tcpdump traces
$ rwptoflow out.pcap > out.rwf.gz
```

Listing 80: Additional SiLK Analysis and Capture Tools

*flow-tools to nfdump*

SiLK uses its own proprietary format for reading flow traces. In order to be able to perform the evaluation on the same flow traces, it was essential to convert the `flow-tools` [2] format trace files to SiLK proprietary format. The best way is to replay the original trace files on a host-port combination, and let SiLK's flow capturing daemon pick it up to save it in its proprietory format. Unfortunately, `flow-tools` does not have any such replay tool. Although, one is provided by the `nfdump` [3] package. Listing 81 shows how `flow-tools` traces were converted to `nfdump` format to allow them to be replayed later.

```
# install nfdump and ft2nfdump
$ sudo apt-get install nfdump
$ sudo apt-get install nfdump-flow-tools

# convert flow-tools traces to nfdump
$ flow-cat $INPUT | ft2nfdump | nfdump -w $OUTPUT
$ nfdump -r $OUTPUT
```

Listing 81: `flow-tools` to `nfdump`

2 http://www.splintered.net/sw/flow-tools/
3 http://nfdump.sourceforge.net/

The converted `nfdump` traces were then replayed using `nfreplay`. By default, the tools replays the traces to 127.0.0.1 at port 9995. A sensor configuration file was created on the other end for SiLK to collect the replayed data. The configuration file defines the NetFlow protocol used in the replay, host-port combination and definition of internal and external IP blocks to separate the flow-traces accordingly. SiLK's `rwflowpack` was then used to regenerate the traces in the proprietary format. The tool segregates the flows into multiple hierarchically arranged directories. Since, only the runtime analysis of SilK on the original query was of interest, a combination of `find` and `xargs` was used to flattern the directory and combine the flows into one file as shown in listing 82.

*nfdump to silk*

```
# replay the nfdump trace
$ nfreplay -r $TRACE

# configure a sensor to collect replayed data
$ cat sensors.conf
probe S1 netflow-v5
  listen-on-port 9995
  protocol udp
  accept-from-host 127.0.0.1
end probe
sensor S1
  netflow-v5-probes S1
  internal-ipblocks 10.0.0.0/8
  external-ipblocks remainder
end sensor

# collect flow data and save in binary silk files
$ rwflowpack \
  --site-config-file=/usr/local/share/silk/generic-silk.conf \
  --sensor-configuration=sensors.conf \
  --root-directory=/var/log/silk/ \
  --log-destination=both

rwflowpack[14830]: Forked child 14831.  Parent exiting
rwflowpack[14831]: Using packing logic from ...
rwflowpack[14831]: Creating stream cache
rwflowpack[14831]: Starting flow processor #1 for PDU Reader
rwflowpack[14831]: Creating PDU Reader Source Pool
rwflowpack[14831]: Creating PDU Reader for probe 'S1' on 0.0.0.0:9995
rwflowpack[14831]: Starting flush timer
rwflowpack[14831]: Started manager thread for PDU Reader

# flatten the silk root directory
$ find /var/log/silk -type f -exec cp {} /var/log/silkflat/ \;

# combine all silk files into a single archive
$ ls | rwcat --xargs --output-path=/var/log/silk.gz
```

Listing 82: `nfdump` to SiLK

# C

# NFQL RELEASE NOTES

The execution engine is now verifiable using a full-fledged regression suite, alongwith an automated benchmarking suite. The python parser implementation has been properly packaged to allow seamless single-step installation using `make` and `pip`. The installation and usage instructions for both Debian/Ubuntu and OS X are included. The workset of the release is shown in listing 83.

*it's verifiable*

```
$ git show v0.5

tag v0.5
Tagger: Vaibhav Bajpai <contact@vaibhavbajpai.com>
Date:   Wed Jul 11 10:33:58 2012 +0200
Commit 8d2f9b374a1e104e97398de47542cd5c0479a0dc

* better engine usage on run.
* evaluation of query ruleset lengths at RUNTIME.
* python pipeline module to encapsulate pipeline stage classes.
* painless parser installation using make.
* parser installation instructions on debian/ubuntu and osx.
* regression test-suite for the execution engine.
* silk installation and usage instructions.
* instructions to convert flow-tools traces to silk.
* automated benchmarking suite.
* resolved issues:
  * no segfault on srcIP = dstIP in a grouper rule.
  * no segfault when no grouper rules are defined.
```

Listing 83: Release Notes: v0.5

The execution engine is now portable to be able to seamlessly build on multiple Unix flavors, tagged as v0.4. CMake is used to orchestrate the build process. It uses custom commands to invoke scripts that can prepare the auto-generated sources and sample `JSON` queries. A Makefile is used to automate the CMake invocations. Feature-test and platform-specific macros allow the code to become portable. The snapshot contains installation instructions for both Debian/Ubuntu and OS X. A workset of the release is shown in listing 84.

*it's portable*

```
$ git show v0.4

tag v0.4
Tagger: Vaibhav Bajpai <contact@vaibhavbajpai.com>
Date:   Fri May 18 15:07:42 2012 +0200
Commit 00c17385e37dd944c9139205a5eb3660c707858a

*  _GNU_SOURCE feature test MACRO and -std=c99
*  (__FreeBSD, __APPLE__) and __linux MACROS around qsort_r(...)
*  reverted to a flat source structure for the CMake build process.
*  CMake custom command to call a script to create auto-generated sources and headers.
*  CMake custom command to call a scripts in queries/ to save sample JSON queries in
     examples/
*  Makefile to automate invocation of CMake commands.
*  installation instruction for Ubuntu.
*  installation instruction for Mac OS X.
```

Listing 84: Release Notes: v0.4

*it's flexible*

The execution engine is now flexible to read and parse the entire flowquery at *runtime*, tagged as *v*0.3. The flowquery is written in an intermediate `JSON` format and read using `json-c`. The `JSON` queries are themselves generated using python scripts. The engine is fail-safe using consistency checks that allow it to fail gracefully when either the trace or the flowquery cannot be read. Each branch thread can now exit early on failure and each stage of the pipeline only proceeds when the previous stage returned results. A workset of the release is shown in listing 85.

```
$ git show v0.3

tag v0.3
Tagger: Vaibhav Bajpai <contact@vaibhavbajpai.com>
Date:   Wed May 16 18:25:22 2012 +0200
Commit 1c323fa66b9aaaad56ad7c4127b8d187eaf4ec0c

* complete query is read at RUNTIME using JSON-C
* JSON queries are generated using python scripts
* glibc backtrace(...) to print the back trace on errExit(...)
* gracefully exiting when trace cannot be read
* gracefully exiting when JSON query cannot be parsed
* branch thread returns EXIT_FAILURE if either stage returns NULL
* branch thread returns EXIT_SUCCESS on normal exit
* each stage proceeds only when previous returned results
* flow-cat ... | flowy-engine $QUERY -
```

Listing 85: Release Notes: *v*0.3

*it's robust*

The execution engine is robust to work well with variety of queries, tagged as *v*0.2. The complete engine has been refactored with a maintainable design to allow better execution workflow and abstract objects. Each stage of the pipeline has one public interface function that takes a ruleset as input and returns a result abstract object. The engine has been profiled to eliminate any memory leaks and allow early deallocation of objects as soon as they are not required. The engine is now smart enough to realize and ignore redundant aggregation requests. All the hardcoded rules of the flowquery are clubbed together in one header file for easy maintainability. The dedicated rule function pointer assignments are lazy. A workset of the release state is shown in listing 86.

```
$ git show v0.2

tag v0.2
Tagger: Vaibhav Bajpai <contact@vaibhavbajpai.com>
Date:   Wed Apr 18 13:24:16 2012 +0200
Commit 2c571f80cd076172cbd00ef7f9976b88cb44b425

* complete engine refactor.
* complete engine profiling (no memory leaks).
* issues closed:
    - greedily deallocating non-filtered records in O(n) before merger(...).
    - resolved a grouper segfault when NO records got filtered.
    - all records are grouped into 1 group when no grouping rule specified.
    - aggregation on common fields touched by filter/grouper rules is ignored.
    - no uintX_t assumptions for field offsets.
    - rules are clubbed together and assigned using a loop.
    - function parameters are as minimum as required.
    - function parameters are safe using [const] ptr and ptr to [const].
    - lazy rule->func(...) assignment when the stage is entered.
```

Listing 86: Release Notes: *v*0.2

This tag starts off the F($v2$) branch. The complete pipeline now works for the first time, tagged as $v0.1$. The grouper segfaults have been resolved. The resulting group records are cooked as NetFlow $v5$ records with their field offsets representing group aggregations. This snapshot contains the first-ever group filter, merger and ungrouper implementation in C. The stages do not assume type of the field offsets that are not known until *runtime*. The engine now works with multiple verbosity levels increasing the amount of echo on each level. A workset of the release is shown in listing 87.

*it's functional*

```
$ git show v0.1

tag v0.1
Tagger: Vaibhav Bajpai <contact@vaibhavbajpai.com>
Date:   Fri Apr 6 19:07:49 2012 +0200
Commit a8a67a13aa07f671d21d062537a2ef17e58dcc07
...

* reverse engineered parser to generate UML.
* froze requirements to allow single step installation of the python parser.
* doxygen documentation of the engine.
* prelim JSON parsing framework for the parser and engine to spit and parse the JSON
    queries.
* replaced GNU99 extensions dependent code with c99.
* resolved numerous segfaults in the grouper.
* generated group aggregations as a separate (cooked) NetFlow v5 record.
* flexible group aggregations with no uintX_t assumptions on field offsets.
* first-ever group filter implementation.
* reorganized the src/ directory structure
* enabled multiple verbosity levels in the engine.
* first-ever merger implementation.
* flexible filters and group filters with no uintX_t assumptions on field offsets.
* first-ever ungrouper implementation.
```

Listing 87: Release Notes: $v0.1$

This tag represents an evolution of the core of the former Python implementation (Flowy) in C. It is also referred as F($v1$). It can efficiently read flow records into memory and filter them. Each branch of the pipeline runs in a separate thread. Each rule type has its own dedicated function pointer generated using a python script. The idea of using `qsort/bsearch` for efficient grouper processing is in place, but the implementation is broken with numerous segfaults. The engine has concerns as shown in listing 88.

*it's a start*

```
$ git show v0.0

tag v0.0
Tagger: Vaibhav Bajpai <contact@vaibhavbajpai.com>
Date:   Thu May 17 10:48:02 2012 +0200
Commit 8cb309c8a956c99e6b1494eddb601c8f6a520696

* read flow-records into memory
* rewrite of the execution pipeline in C (non functional)
* efficient rule processing with dedicated function pointers
* reduced grouper complexity using qsort(...) and bsearch(...)
* concerns
        - flow query is currently hardcoded in pipeline structs
        - functions assume specific uintX_t offsets
        - numerous grouper segfaults
        - no group filter, no ungrouper
        - commented out merger (segfaults when uncommented)
        - code dependent on GNU99 extensions
        - some headers are missing include guards
        - unused extraneous source files and headers
```

Listing 88: Release Notes: $v0.0$

# D

## ACRONYMS

IPFIX Internet Protocol Flow Information Export

HDF Hierarchical Data Format

LALR Look-Ahead LR Parser

PLY Python Lex-Yacc

HDFS Hadoop Distributed File System

API Application Programming Interface

SSDP Simple Service Discovery Protocol

IP Internet Protocol

UDP User Datagram Protocol

TCP Transmission Control Protocol

NAT-PMP Network Address Translation Port Mapping Protocol

ccTLD Country Code Top-Level Domain

HTTP Hypertext Transfer Protocol

IaaS Infrastructure as a Service

NaaS Network as a Service

vLANs Virtual Local Area Networks

ACLs Access Control Lists

MPLS Multiprotocol Label Switching

RTT Round Trip Time

SVMs Support Vector Machines

FF Greedy Forward Fitting

SMTP Simple Mail Transfer Protocol

DDoS Distributed Denial of Service

CAPTCHA Completely Automated Public Turing Test to Tell Computers and Humans Apart

RMON Remote Network Monitoring

MIB Management Information Base

SNMP Simple Network Management Protocol

RTFM Realtime Traffic Flow Measurement

GUI Graphical User Interface

IETF Internet Engineering Task Force

DoS Denial of Service

AS Autonomous Systems

CIDR Classless Inter-Domain Routing

SCTP Stream Control Transmission Protocol

PSAMP Packet Sampling

TLS Transport Layer Security

DTLS Datagram Transport Layer Security

IE Information Elements

IANA Internet Assigned Numbers Authority

PENs Private Enterprise Numbers

EP Exporter Process

CP Collector Process

SMI Structure of Managed Information

CLI Command Line Interface

XDR External Data Representation

UML Unified Modeling Language

NFQL Network Flow Query Language

DNF Disjunctive Normal Form

# BIBLIOGRAPHY


[1] B. Claise, "Cisco Systems NetFlow Services Export Version 9." RFC 3954 (Informational), Oct. 2004.

[2] B. Claise, "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information." RFC 5101 (Proposed Standard), Jan. 2008.

[3] V. Marinov, "Design of an IP Flow Record Query Language," Master's thesis, Jacobs University Bremen, Campus Ring 1, 28759 Bremen, Germany, August 2009.

[4] J. F. Allen, "Maintaining knowledge about temporal intervals," *Communications of the ACM*, vol. 26, pp. 832–843, November 1983.

[5] K. Kanev, "Flowy - Network Flow Analysis Application," Master's thesis, Jacobs University Bremen, Campus Ring 1, 28759 Bremen, Germany, August 2009.

[6] J. Schauer, "Flowy 2.0: Fast Execution of Stream based IP Flow Queries," bachelor's thesis, Jacobs University Bremen, Campus Ring 1, 28759 Bremen, Germany, May 2011.

[7] N. Melnikov, "Cybermetrics: Identification of Users through Network Flow Analysis," Master's thesis, Jacobs University Bremen, Campus Ring 1, 28759 Bremen, Germany, August 2010.

[8] P. Kohler and B. Claise, "IPFIX Fine-Tunes Traffic Analysis," *Network World*, Aug. 2003.

[9] B. Trammell and E. Boschi, "An Introduction to IP Flow Information Export (IPFIX)," *Communications Magazine, IEEE*, vol. 49, pp. 89 –95, april 2011.

[10] Dell, Texas, *Dell PowerConnect M6220, M6348, M8024, and M8024k Switch Users Configuration Guide*.

[11] V. Marinov and J. Schönwälder, "Design of a Stream-Based IP Flow Record Query Language," in *Proceedings of the 20th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management: Integrated Management of Systems, Services, Processes and People in IT*, DSOM '09, (Berlin, Heidelberg), pp. 15–28, Springer-Verlag, 2009.

[12] P. Nemeth, "Flowy Improvements using Map/Reduce," bachelor's thesis, Jacobs University Bremen, Campus Ring 1, 28759 Bremen, Germany, May 2010.

[13] V. Bajpai, N. Melnikov, A. Sehgal, and J. Schönwälder, "Flow-Based Identification of Failures Caused by IPv6 Transition Mechanisms," in *Dependable Networks and Services*, vol. 7279 of *Lecture Notes in Computer Science*, pp. 139–150, Springer Berlin Heidelberg, 2012.

[14] B. Daviss, "Building a Crash-Proof Internet," *New Scientist*, vol. 26, pp. 38–41, June 2009.

[15] R. Beverly and K. Sollins, "Exploiting Transport-Level Characteristics of Spam," in *Proceedings of the Fifth Conference on Email and Anti-Spam (CEAS)*, Aug. 2008.

[16] V. Perelman, "Flow signatures of Popular Applications," bachelor's thesis, Jacobs University Bremen, Campus Ring 1, 28759 Bremen, Germany, May 2010.

[17] V. Jacobson, C. Leres, and S. McCanne, *tcpdump - dump traffic on a network*. Lawrence Berkeley National Laboratory, University of California, Berkeley, CA.

[18] V. Jacobson, C. Leres, and S. McCanne, *pcap - Packet Capture library*. Lawrence Berkeley National Laboratory, University of California, Berkeley, CA.

[19] G. Combs, *wireshark - Interactively dump and analyze network traffic*. University of Missouri, Kansas City.

[20] K. Xu, Z.-L. Zhang, and S. Bhattacharyya, "Profiling Internet Backbone Traffic: Behavior Models and Applications," in *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '05, (New York, NY, USA), pp. 169–180, ACM, 2005.

[21] T. Karagiannis, K. Papagiannaki, and M. Faloutsos, "BLINC: Multilevel Traffic Classification in the Dark," in *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '05, (New York, NY, USA), pp. 229–240, ACM, 2005.

[22] M.-S. Kim, H.-J. Kong, S.-C. Hong, S.-H. Chung, and J. Hong, "A Flow-based Method for Abnormal Network Traffic Detection," in *Network Operations and Management Symposium, 2004. NOMS 2004. IEEE/IFIP*, vol. 1, pp. 599 –612 Vol.1, april 2004.

[23] D. Schatzmann, W. Mühlbauer, T. Spyropoulos, and X. Dimitropoulos, "Digging into HTTPS: Flow-based Classification of Webmail Traffic," in *Proceedings of the 10th annual conference on Internet measurement*, IMC '10, (New York, NY, USA), pp. 322–327, ACM, 2010.

[24] S. Waldbusser, R. Cole, C. Kalbfleisch, and D. Romascanu, "Introduction to the Remote Monitoring (RMON) Family of MIB Modules." RFC 3577 (Informational), Aug. 2003.

[25] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "Simple Network Management Protocol (SNMP)." RFC 1157 (Historic), May 1990.

[26] S. Waldbusser, "Remote Network Monitoring Management Information Base." RFC 2819 (Standard), May 2000.

[27] S. Waldbusser, "Remote Network Monitoring Management Information Base Version 2." RFC 4502 (Draft Standard), May 2006.

[28] N. Brownlee, C. Mills, and G. Ruth, "Traffic Flow Measurement: Architecture." RFC 2722 (Informational), Oct. 1999.

[29] W. W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques," in *Proceedings of the 9th International Conference on Software Engineering*, ICSE '87, (Los Alamitos, CA, USA), pp. 328–338, IEEE Computer Society Press, 1987.

[30] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a Better NetFlow," in *Proceedings of the 2004 conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM 2004, (New York, NY, USA), pp. 245–256, ACM, 2004.

[31] N. Duffield, D. Chiou, B. Claise, A. Greenberg, M. Grossglauser, and J. Rexford, "A Framework for Packet Selection and Reporting." RFC 5474 (Informational), Mar. 2009.

[32] T. Zseby, M. Molina, N. Duffield, S. Niccolini, and F. Raspall, "Sampling and Filtering Techniques for IP Packet Selection." RFC 5475 (Proposed Standard), Mar. 2009.

[33] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2." RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176.

[34] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security." RFC 4347 (Proposed Standard), Apr. 2006. Updated by RFC 5746.

[35] B. Claise, A. Johnson, and J. Quittek, "Packet Sampling (PSAMP) Protocol Specifications." RFC 5476 (Proposed Standard), Mar. 2009.

[36] S. Wang, R. State, M. Ourdane, and T. Engel, "FlowRank: Ranking NetFlow Records," in *Proceedings of the 6th International Wireless Communications and Mobile Computing Conference*, IWCMC '10, (New York, NY, USA), pp. 484–488, ACM, 2010.

[37] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[38] L. Deri, E. Chou, Z. Cherian, K. Karmarkar, and M. Patterson, "Increasing Data Center Network Visibility with Cisco NetFlow-Lite," in *Network and Service Management (CNSM), 2011 7th International Conference on*, pp. 1 –6, oct. 2011.

[39] L. Deri, "nprobe: an open source netflow probe for gigabit networks," in *In Proceedings of Terena TNC 2003*, 2003.

[40] G. Sadasivan, N. Brownlee, B. Claise, and J. Quittek, "Architecture for IP Flow Information Export." RFC 5470 (Informational), Mar. 2009. Updated by RFC 6183.

[41] T. Dietz, A. Kobayashi, B. Claise, and G. Muenz, "Definitions of Managed Objects for IP Flow Information Export." RFC 5815 (Proposed Standard), Apr. 2010.

[42] B. Trammell and E. Boschi, "Bidirectional Flow Export Using IP Flow Information Export (IPFIX)." RFC 5103 (Proposed Standard), Jan. 2008.

[43] P. Phaal, S. Panchen, and N. McKee, "InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks." RFC 3176 (Informational), Sept. 2001.

[44] S. Microsystems, "XDR: External Data Representation standard." RFC 1014, June 1987.

[45] K. Kanev, N. Melnikov, and J. Schönwälder, "Implementation of a stream-based IP flow record query language," in *Proceedings of the Mechanisms for autonomous management of networks and services, and 4th international conference on Autonomous infrastructure, management and security*, AIMS'10, (Berlin, Heidelberg), pp. 147–158, Springer-Verlag, 2010.

[46] V. Marinov and J. Schönwälder, "Design of an IP Flow Record Query Language," in *Proceedings of the 2nd international conference on Autonomous Infrastructure, Management and Security: Resilient Networks and Services*, AIMS '08, (Berlin, Heidelberg), pp. 205–210, Springer-Verlag, 2008.

[47] F. Alted and M. Fernández-Alonso, "PyTables: Processing And Analyzing Extremely Large Amounts Of Data In Python," 2003.

[48] J. Quittek, S. Bryant, B. Claise, P. Aitken, and J. Meyer, "Information Model for IP Flow Information Export." RFC 5102 (Proposed Standard), Jan. 2008. Updated by RFC 6313.

[49] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2004.

[50] T. White, *Hadoop: The Definitive Guide*. Definitive Guide Series, O'Reilly, 2010.

[51] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pp. 1 –10, May 2010.

[52] P. Mundkur, V. Tuulos, and J. Flatow, "Disco: A Computing Platform for Large-Scale Data Analytics," in *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, Erlang '11, (New York, NY, USA), pp. 84–89, ACM, 2011.

[53] D. S. Seljebotn, "Fast numerical computations with Cython," in *Proceedings of the 8th Python in Science Conference* (G. Varoquaux, S. van der Walt, and J. Millman, eds.), (Pasadena, CA USA), pp. 15 – 22, 2009.

[54] I. Wilbers, H. P. Langtangen, and Å. Ødegård, "Using Cython to Speed up Numerical Python Programs," in *Proceedings of MekIT'09* (B. Skallerud and H. I. Andersson, eds.), pp. 495–512, NTNU, Tapir, 2009.

[55] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. Seljebotn, and K. Smith, "Cython: The Best of Both Worlds," *Computing in Science Engineering*, vol. 13, pp. 31 –39, march-april 2011.

[56] S. Romig, "The OSU Flow-tools Package and CISCO NetFlow Logs," in *Proceedings of the 14th USENIX conference on System administration*, (Berkeley, CA, USA), pp. 291–304, USENIX Association, 2000.

[57] P. Haag, "Netflow Tools NfSen and NFDUMP," in *Proceedings of the 18th Annual FIRST conference*, 2006.

[58] V. Perelman, N. Melnikov, and J. Schönwälder, "Flow signatures of Popular Applications," in *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, pp. 9 –16, May 2011.

[59] M. Bodlaender, "UPnP 1.1 - Designing for Performance Compatibility," *Consumer Electronics, IEEE Transactions on*, vol. 51, pp. 69 – 75, feb. 2005.

[60] S. Cheshire, M. Krochmal, and K. Sekar, "NAT port mapping protocol (NAT-PMP)," Internet-Draft draft-cheshire-nat-pmp-03.txt, IETF Secretariat, Fremont, CA, USA, Apr. 2008.

[61] F. Bergadano, D. Gunetti, and C. Picardi, "User Authentication through Keystroke Dynamics," *ACM Trans. Inf. Syst. Secur.*, vol. 5, pp. 367–397, November 2002.

[62] A. Ahmed and I. Traore, "A New Biometric Technology Based on Mouse Dynamics," *IEEE Transactions on Dependable and Secure Computing*, vol. 4, pp. 165 –179, July-Sept 2007.

[63] K.-T. Chen and L.-W. Hong, "User identification based on Game-Play Activity Patterns," in *Proceedings of the 6th ACM SIGCOMM workshop on Network and System Support for Games*, NetGames '07, (New York, NY, USA), pp. 7–12, ACM, 2007.

[64] N. Melnikov and J. Schönwälder, "Cybermetrics: User Identification through Network Flow Analysis," in *Proceedings of the Mechanisms for autonomous management of networks and services, and 4th international conference on Autonomous infrastructure, management and security*, AIMS'10, (Berlin, Heidelberg), pp. 167–170, Springer-Verlag, 2010.

[65] M. Bagnulo, P. Matthews, and I. van Beijnum, "Stateful NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers." RFC 6146 (Proposed Standard), Apr. 2011.

[66] A. Durand, R. Droms, J. Woodyatt, and Y. Lee, "Dual-Stack Lite Broadband Deployments Following IPv4 Exhaustion." RFC 6333 (Proposed Standard), Aug. 2011.

[67] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Computer Communications Review*, vol. 38, pp. 69–74, March 2008.

[68] T. Benson, A. Akella, A. Shaikh, and S. Sahu, "CloudNaaS: A Cloud Networking Platform for Enterprise Applications," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, (New York, NY, USA), pp. 8:1–8:13, ACM, 2011.

[69] G. Kakavelakis, R. Beverly, and J. Young, "Auto-learning of SMTP TCP Transport-Layer Features for Spam and Abusive Message Detection," in *LISA 2011, 25th Large Installation System Administration Conference* (T. A. Limoncelli and D. Hughes, eds.), (Berkeley, CA, USA), USENIX, LOPSA, USENIX Association, Dec. 2011.

[70] C. Cortes and V. Vapnik, "Support-Vector Networks," *Machine Learning*, vol. 20, pp. 273–297, 1995. 10.1007/BF00994018.

[71] Y. Yang and J. O. Pedersen, "A Comparative Study on Feature Selection in Text Categorization," 1997.

[72] ISO, "The ANSI c standard (c99)," Tech. Rep. WG14 N1124, ISO/IEC, 1999.

[73] P. Deutsch and J.-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3." RFC 1950 (Informational), May 1996.

[74] K. Martin and B. Hoffman, *Mastering CMake 4th Edition*. USA: Kitware, Inc., 4th ed., 2008.

[75] CERT/NetSA at Carnegie Mellon University, "SiLK (System for Internet-Level Knowledge)." [Online]. Available: http://tools.netsa.cert.org/silk [Accessed: July 10, 2012].

[76] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *SIGPLAN Not.*, vol. 17, pp. 120–126, June 1982.

## COLOPHON

This thesis was typeset with LaTeX 2ε using Hermann Zapf's *Palatino* and *Euler* type faces. The listings are typeset in *Bera Mono*, originally developed by Bitstream, Inc. The typographic style was inspired by *The Elements of Typographic Style*.