Network Flow Query Language – Design, Implementation, Performance and Applications

Vaibhav Bajpai and Jürgen Schönwälder
Computer Science, Jacobs University Bremen, Germany
(v.bajpai | j.schoenwaelder)@jacobs-university.de

***Abstract*—Cisco's NetFlow protocol and IETF's IPFIX open standard are widely deployed protocols for collecting network flow statistics. Understanding intricate traffic patterns in these network statistics requires sophisticated flow analysis tools that can efficiently mine network flow records. We present NFQL, a network flow query language, which can be used to write expressive queries to process flow records, aggregate them into groups, apply absolute or relative filters, and invoke Allen interval algebra rules to merge group records. We demonstrate `nfql`, an implementation of the language that has comparable execution times to SiLK and `flow-tools` with absolute filters. However, it trades performance when grouping and merging flows in favour of more operational capabilities that help increase the expressiveness of NFQL. We present two applications to demonstrate richer capabilities of the language. We show queries to identify flow signatures of popular applications and behavioural signatures to identify SSH compromise detection attacks.**

***Index Terms*—NetFlow, IPFIX, flow-tools, nfdump, SiLK, application signatures, SSH compromise detection**

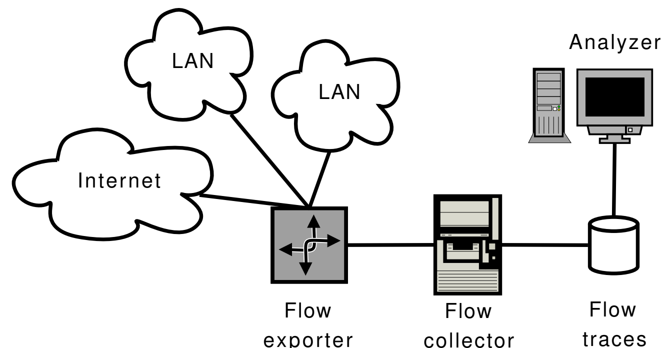

Fig. 1. *An abstract view of flow-export protocols such as NetFlow and IPFIX. A flow exporter reads IP packets that cross its boundary to generate flow-records. The flow-records are exported based on some predefined expiration rules. A flow collector on receiving these flow-records decodes and stores them locally to be used for analysis by the flow analyzer [17].*

I. Introduction

Researchers, service providers and security analysts are 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 and network performance evaluation. Traffic measurement techniques ranging from capturing raw packets [1], [2] and aggregating flow records [3], [4] to remote monitoring and metering provide such insights. NetFlow and Internet Protocol Flow Information Export (IPFIX) are the two popular protocols for collecting network flow records [5]. NetFlow [6] is a network protocol designed by Cisco Systems which allows routers to generate and export flow records to a designated collector for further analysis as shown in Fig. 1. IPFIX [7], [8] on the other hand is an open standard defined by the Internet Engineering Task Force (IETF), which is based on NetFlow version 9. The popularity of these protocols can be attributed to reduction in monitoring traffic volumes at the flow-level and the fine-grained control which was not previously possible using Simple Network Management Protocol (SNMP) interface-level queries. Their wide applicability [9] can also be seen from the pervasive use of flow records for a number of different network analysis applications. For instance, Myung-Sup Kim *et al.* in [3] use flow characteristics to formalize a detection function that maps traffic patterns to different Denial of Service (DoS) attacks, while Dominik Schatzmann *et al.* in [4] exploit timing characteristics of webmail clients to classify features in flow records that could identify webmail traffic from any other traffic running over HTTPS. Arthur Callado *et al.* in [10] provide a survey of such flow-based techniques that perform behaviour analysis and anomaly detection on Internet backbone traffic. Anna Sperotto *et al.* in [11] take this further and provide a survey of how network flow analysis can be used to detect intrusion attacks. Understanding such intricate traffic patterns requires sophisticated flow analysis tools that can mine flow records for such a usage. However, capabilities of these tools are limited by their language design. For instance, current tools (such as `flow-tools` [12], [13] and `nfdump` [14], [15]) either only allow absolute value comparison or limit relative comparisons to equality relationships (such as SiLK [16]) between flow record fields. As a consequence, applications cannot leverage these tools owing to their simplistic design.

We present NFQL, a network flow query language which can be used to write queries to process flow records. NFQL can filter flows (we use flows as a shorthand for flow records). It can combine filtered flows into groups (unlike `flow-tools` and `nfdump`) and can calculate statistics on the resultant groups. It can further merge grouped flows. It can apply absolute or relative filters when grouping or merging. It can also apply temporal relations between groups using Allen interval algebra rules [18]. Furthermore, (unlike SiLK) it can unfold grouped flows back into individual flows. `nfql` [19] is an implementation of NFQL that has comparable execution times to `flow-tools` and SiLK with absolute filters. We

utilise nfql to present two applications to demonstrate the power of NFQL. Overall, we provide 4 main **contributions** –

– NFQL, a network flow query language. It can filter flows, combine flows into groups, calculate temporal relationships and aggregated statistics on groups, merge grouped flows, (see § III) apply absolute or relative filters, and unfold grouped flows into individual flows. NFQL supports 6 absolute comparators, 7 interval operators, 10 aggregation operators and 3 bitwise operators.
– nfql [19], an implementation of NFQL. It can read and write flows in NetFlow v5 and IPFIX format. The query exclusively uses IPFIX (see § IV) entity names and datatypes to keep consistency across trace formats. nfql can disable each pipeline stage at runtime and write output flows to disk at different compression levels.
– A performance evaluation that shows that nfql has comparable execution times to SiLK and flow-tools (see § V) with absolute filters. The queries used for evaluation are released to support development of a general benchmarking suite for flow analysis tools.
– Two applications demonstrate the power and expressiveness of the query language. NFQL queries are presented to identify flow (see § VI) signatures of popular applications and behavioural signatures to identify SSH compromise detection attacks.

This paper builds on our work published previously in [20], [21], [22], [23], [24]. This paper not only provides a summary of our research but also extends it in several ways. We have implemented support for IPFIX flows in nfql. We have added a translation layer to allow the NFQL query to exclusively use IPFIX entities and datatypes. A front-end parser has also been implemented to validate queries written in the NFQL Domain Specific Language (DSL). We also revisited NFQL queries that identify application signatures for current versions of applications. Furthermore, we present a new application of NFQL to identify SSH compromise detection attacks.

II. Background and Related Work

Flow records [5] are typically identified by a seven tuple flow-key consisting of source and destination IP address, source and destination port, IP protocol, ingress interface and IP type of service. IP packets sharing this information belong to one flow. In addition to the flow-key, flow records can also contain additional accounting information such as flow start and end times, sum of bytes in a flow or source and destination Autonomous Systems (AS) numbers. A flow exporter reads IP packets (see Fig. 1) that cross its boundary to generate flow-records. The flow-records are exported based on some predefined expiration rules such as a 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, flow-records once transmitted to the collector are deleted from the exporter. A collector receives flow-records, decodes and stores them locally to be used for further processing by the flow analyzer.

NetFlow and IPFIX are two popular standards of IP flow information export. NetFlow [6] is a proprietary network protocol designed by Cisco Systems. Table I provides a summary of the NetFlow version history. 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 became mainstream. 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 [7] on the other hand is an open standard defined by the IETF, which is based on NetFlow v9. The novelty of IPFIX lies in its ability to describe record formats at runtime using templates based on an extensible information model [25]. The data transfer mechanism is unidirectional and transport protocol agnostic.

TABLE I
NetFlow Version History

VERSION	FEATURES
v1, v2, v3, v4	original format with several internal releases
v5	CIDR / AS support and flow sequence numbers
v6, v7, v8	router-based aggregation support
v9	template-based with IPv6 and MPLS support
IPFIX	universal standard, transport-protocol agnostic

TABLE II
Flow Query Languages

TOOL	QUERY LANGUAGE
Nickless *et al.* [26]	SQL
Babcock *et al.* [27]	extended SQL
Gigascope	GSQL
Tribeca	proprietary
tcpdump	BPF
nfdump	BPF
CoralFeef	BPF
Time Machine	BPF
flow-tools	proprietary
FlowScan	perl scripts
AutoFocus	proprietary
SiLK	proprietary

A number of languages have been developed to query the flows exported by these protocols. Table II provides a summary of query languages used by network traffic analysis tools today. As can be seen, a number of network analysis applications are based on SQL. For instance, Bill Nickless *et al.* in [26] present a system that uses a relational database to store attributes of NetFlow records. Brian Babcock *et al.* in [27] take this further and propose the design of a Data Stream Management System (DSMS) that extends SQL to model flows as transient data streams. Charles Cranor *et al.* in [28] propose Gigascope, a stream database for network monitoring applications. It uses GSQL, an adaptation of SQL that allows time window definitions inside a query. Mark Sullivan *et al.* in [29] introduce Tribeca, a stream-oriented DBMS that supports projection, selection, aggregation, multiplexing and demultiplexing of streams. Filtering languages on the other hand rely on the Berkeley Packet Filter (BPF) [30] to specify rules to filter a stream of packets. BPF can construct logical expressions for filtering network traces which can be translated into small programs that are executed by a generic packet fil-

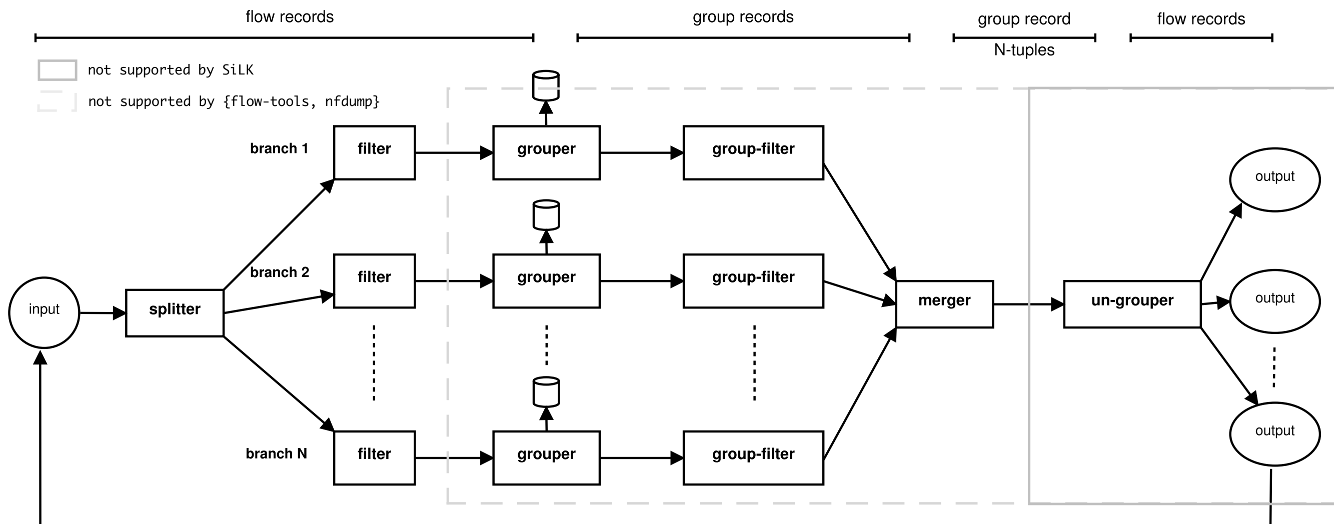

Fig. 2. *NFQL execution pipeline consisting of six stages: splitter, filter, grouper, group filter, merger and ungrouper. The filter, grouper and group filter stages can have multiple instances using branches. A branch is used to create a logical separation of disparate tasks.* `flow-tools` *and* `nfdump` *only support the filter stage while SiLK does not support an ungrouper stage of the NFQL processing pipeline.*

tering engine. BPF rules are used by `tcpdump`, `nfdump` [15], CoralReef [31] and Time Machine [32] network analysis tools. `flow-tools` [13] is backed up by a procedural language design. It uses the Cisco Access Control List (ACL) format to prepare filter expressions and proprietary primitives to define flow reports. FlowScan [33] uses a collection of perl scripts to glue together a flow-collector, a Round-robin Database (RRD) and a visualization tool to generate traffic reports.

A range of graphical utilities such as `ntop` [34], FlowScan [33], FlowViewer, NfSen [15] and Stager [35] can be used to perform simple network analysis. These tools understand the NetFlow format while `ntop` and Stager can also process IPFIX flow-records. We do not describe every related tool in this paper, but refer the reader to [15] for an exhaustive survey of open-source and commercial tools used for flow-export, collection and analysis. We instead further describe tools that we use in our performance evaluation. For instance, `flow-tools` and `nfdump` are among the most popular tools used for analyzing NetFlow data. `flow-tools` [13] is a suite of programs for capturing and processing NetFlow v5 flow records. It consists of 24 separate tools that work together by connecting them via UNIX pipes. It can capture, read, filter, and print flow records internally saved in a fixed-size format. `nfdump` [15] is a very similar tool that uses a different storage format. SiLK [16] provides a collection of command-line tools that can be used to write scripts for querying flow records. SiLK comes quite close to providing similar capabilities as NFQL. However, relative comparisons in SiLK can only be performed using an equality operator, while NFQL supports a richer set of comparison operations (such as greater than, less than, greater or equal, less or equal) when comparing flows. The design and implementation of SiLK, also differs from that of `nfql`. For instance, in SiLK there are separate tools to perform the task of each stage (see Fig. 2) of the NFQL processing pipeline. There are also stringent requirements on how the flow-data needs to be organized in SiLK before it can be piped into a tool. For instance, the grouping tool assumes that the input flow data is already sorted on the field column. These requirements can make it a little cumbersome to design an NFQL query in SiLK. For instance, trying to mimic a NFQL query in SiLK sometimes ends up as a shell script with over a dozen of SiLK tools piped together.

III. Language Design

The query language consists of a number of independent stages that are connected to one another to form a processing pipeline as shown in Fig. 2. The pipeline model is modular and it consists of six different stages – splitter, filter, grouper, group filter, merger and ungrouper. The filter, grouper and group filter stages can have multiple instances using branches. A branch serves to create a logical separation of disparate tasks. The input flow records enter through the splitter and the resultant flows that satisfy the pipeline conditions exit out of the ungrouper stage. `flow-tools` and `nfdump` are limited to absolute comparisons of flow attributes, which is simply one stage of the NFQL processing pipeline while SiLK does not support ungrouping of grouped flows.

A. *Processing Pipeline*

We use a sample query as a running example to not only present each pipeline stage but also to introduce the DSL used to express the query. Let's assume we want to find within our trace all flow pairs representing HTTP (or HTTPS) traffic over both IPv4 and IPv6 that have exchanged more than 200 packets in both directions. In order to do this we define two branches. Branch A will retrieve outgoing flows while branch B will retrieve incoming flows. A merger will later be used to correlate incoming and outgoing flows. Although IPFIX is able to export bidirectional flow records [36], we assume our input trace in this example consists of unidirectional flow records.

Splitter – is the first pipeline stage as shown in Fig. 2. It reads flow records from disk in NetFlow v5 or IPFIX format. The splitter duplicates the input data to several branches without any processing whatsoever. This allows each branch to receive an identical copy of the flow data to process it independently. A splitter is always executed and takes no additional arguments. It is implicitly specified and is therefore not needed to be described in the query.

Filter – is the second stage of the processing pipeline and the first stage in each branch. It performs absolute filtering of flow records. The filter compares fields of a flow-record against either a constant value or a value on a different field of the *same* flow-record. `flow-tools` and `nfdump` support only this stage of the pipeline. Listing 1 shows the DSL used to express the filter. It is used to select TCP packets destined (branch A) or sourced (branch B) over port 80 (or 443).

```
1  branch A {
2    filter F {
3      destinationTransportPort = 80 OR \
4        destinationTransportPort = 443
5      protocolIdentifier = TCP
6    }
7  }
8
9  branch B {
10   filter F {
11     sourceTransportPort = 80 OR \
12       sourceTransportPort = 443
13     protocolIdentifier = TCP
14   }
15 }
```

Listing 1. *Filter construct for HTTP (or HTTPS) requests and responses*

Each statement is called a *term*. A collection of terms wrapped in braces is called a *clause*. Terms within a clause are conjunctives while multiple clauses are disjunctives. In this way, each stage in our pipeline is a Disjunctive Normal Form (DNF) expression. For instance, Listing 1 has only one clause with two terms in each branch. As such, the filter lets flows pass which specify a destination port 80 or 443 *and* which were sent via TCP. Note that the DSL uses IPFIX entity names and datatypes (see § IV for details) [25] for both NetFlow v5 and IPFIX flows. The flows that pass the filter are forwarded to the next stage, while the rest of the flows are dropped.

Grouper – combines flow records together into groups and it optionally assigns attributes to those groups. Flows can be grouped using either an absolute or relative comparator. Listing 2 shows the DSL used to express the grouper stage. In our example, the grouper DSL happens to be identical for both branches. This grouper combines together all flows (lines 3 – 14) which share the same source and destination IPv4 (or IPv6 endpoint) and whose timestamp (lines 13 – 14) differs by a maximum of 500 ms. Furthermore, we use an aggregation block (lines 16 – 25) to assign attributes to each created group. For instance, each group is labeled with the shared source and destination IP addresses (lines 17 – 20), the overall amount of transmitted bytes (line 21) and packets (line 22) and flow start and end times (lines 23 and 24). The newly formed groups and their attributes are passed on to the next stage.

Group Filter – is the last processing stage of a branch. The group-filter performs absolute filtering over group attributes. The group-filter compares fields (or aggregated fields) of a group-record against either a constant value or a value on a different field of the *same* group-record. Listing 3 shows the DSL used to express the group filter stage. Note, in our example, the group filter DSL happens to be identical in both branches. This group filter passes groups that have exchanged more than 200 packets in both directions. The passed group-records are forwarded to the next stage, while the rest of the group-records are dropped.

```
1  branch ... {
2    grouper ... {
3      sourceIPv4Address = \
4      sourceIPv4Address OR \
5      sourceIPv6Address = \
6      sourceIPv6Address
7
8      destinationIPv4Address = \
9      destinationIPv4Address OR \
10     destinationIPv6Address = \
11     destinationIPv6Address
12
13     flowStartMilliseconds = \
14     flowStartMilliseconds delta 500
15
16     aggregation {
17       static(sourceIPv4Address)
18       static(sourceIPv6Address)
19       static(destinationIPv4Address)
20       static(destinationIPv6Address)
21       sum(octetDeltaCount)
22       sum(packetDeltaCount)
23       min(flowStartMilliseconds)
24       max(flowEndMilliseconds)
25     }
26   }
27 }
```

Listing 2. *Grouper construct for matching IP endpoints.*

```
1  branch ... {
2    groupfilter ... {
3      packetDeltaCount > 200
4    }
5  }
```

Listing 3. *Group filter construct for passing groups with > 200 packets*

Merger – merges group records from different branches to create streams. Listing 4 shows the DSL used to express the merger stage. In our example, the HTTP request flow is matched with the HTTP response flow to create an HTTP session. Groups from one branch whose source IP endpoint is equal to the destination IP endpoint of the other branch and the other way round (lines 6 – 14) are matched together. We also need to make sure (line 16) that groups from branch A must carry less data than the groups in branch B. This indicates that the flows matched by branch A represent an HTTP request while the flows matched by branch B represent an HTTP response. Furthermore, the time spent for matching groups must either overlap `(A o B)` or the request should finish with the response `(A f B)`. This is accomplished using Allen interval algebra notation used in Line 17. We further refer the reader to [18] that contains pictorial representation of each Allen interval operation.

Ungrouper – is the last processing step. It unfolds the stream of grouped flows into individual flows and saves them to disk. SiLK does not support this stage. As a consequence,

- **Supported Comparison Operations:**
  - `EQ, NE, GT, LT, LE, GE`
- **Supported Interval Operations:**
  - `X before Y (X < Y)`
  - `X is equal to Y (X = Y)`
  - `X meets Y (X m Y)`
  - `X overlaps with Y (X o Y)`
  - `X during Y (X d Y)`
  - `X starts Y (X s Y)`
  - `X finishes Y (X f Y)`
- **Supported Aggregations:**
  - `UNION, MIN, MAX, SUM, MEDIAN`
  - `COUNT, MEAN, STDDEV, XOR, PROD`
- **Supported Bitwise Operations:**
  - `AND, OR, IN`

Fig. 3. *A summary of all possible operations. NFQL supports 6 absolute comparators, 7 interval operators, 10 aggregation operators and 3 bitwise operators. Only underlined operations are supported by SiLK.*

once flows are grouped by SiLK, they cannot be unfolded back into individual flows. Listing 5 shows the DSL used to express the ungrouper. It does not take any arguments. Unlike the splitter, if the ungrouper statement is not specified, this step in the pipeline is not executed.

```
1  branch A { ... }
2
3  branch B { ... }
4
5  merger M {
6    A.sourceIPv4Address = \
7    B.destinationIPv4Address OR \
8    A.sourceIPv6Address = \
9    B.destinationIPv6Address
10
11   A.destinationIPv4Address = \
12   B.sourceIPv4Address OR \
13   A.destinationIPv6Address = \
14   B.sourceIPv6Address
15
16   A.octetDeltaCount < B.octetDeltaCount
17   A o B OR A f B
18 }
```

Listing 4. *Merger construct for matching HTTP (or HTTPS) sessions*

```
1  ungrouper U { }
```

Listing 5. *Ungrouper contruct to save individual flows to disk*

B. Operators and Functions

NFQL (unlike SiLK that only supports equality comparisons) allows several comparison operators as shown in Fig 3. It also supports temporal comparisons using Allen time interval algebra rules [18]. All operations can be appended with the `delta` keyword, followed by a value. This allows one to make inexact comparisons. For example, an Allen time interval comparison could be appended with `delta 1s` to allow a mismatch of at most 1 second. NFQL also allows

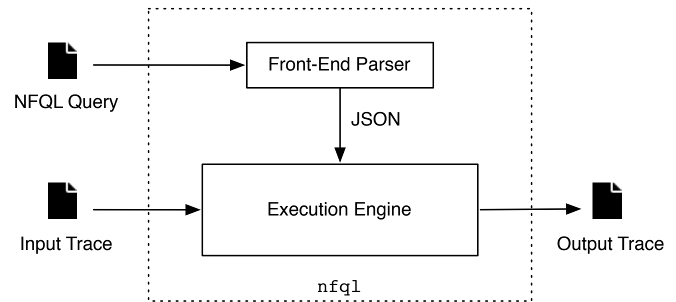

Fig. 4. *The nfql architecture consists of a front-end parser backed up by an execution engine. The front-end parser converts a user query written in the NFQL DSL into a JSON format. The execution engine reads an input trace from disk, executes the NFQL pipeline according to the JSON query and writes the output trace to disk. The execution engine is written in C and the front-end parser is written in Python.*

functions to be applied to certain values. These functions can be either bitwise operations or aggregations on values. Overall, NFQL supports 6 absolute comparators, 7 interval operators, 10 aggregation operators and 3 bitwise operators.

IV. Implementation

`nfql` [19] is a reference implementation of NFQL. The architecture is composed of an execution engine and a front-end parser as shown in Fig. 4. The front-end parser is used to validate the NFQL query and to generate its JSON (see Listing 6 for an example) intermediate representation. The execution engine reads the flow-query in this intermediate JSON format [37] along with flow traces that are read in memory for efficient processing. The execution engine is the brain of `nfql` where the complete processing pipeline (see § III) is executed to process the input flow trace to produce an output trace that is written to disk. The execution engine is written in C and the front-end parser is written in Python.

```
1  "filter": {
2    "dnf-expr": [{
3      "clause": [{
4        "term": {
5          "offset": {
6            "name": "destinationTransportPort",
7            "value": 80
8          },
9          "op": "RULE_EQ"
10       }
11     }]
12   }]
13 }
```

Listing 6. *JSON representation of a filter construct*

A. Front-End Parser

A front-end parser is used to validate a query written in the NFQL DSL and to convert it into an intermediate JSON representation. This intermediate format is helpful in completely decoupling the parser from the performance sensitive execution engine. As a result, the execution engine can now be deployed on a high-end machine, while the parsing can either be done locally or through a remote web service. The

intermediate JSON format also allows one to write additional frontends to the execution engine that can emulate other popular DSL formats such as `nfdump`. Listing 6 shows an example of a JSON representation of a filter construct. Each stage of the pipeline is expressed in the JSON query as a DNF expression as previously discussed in § III.

The NFQL query can also disable the pipeline stages at runtime. This means that one only has to supply the constructs that one wishes to use. The parser will not emit the disabled stages in the intermediate JSON representation.

B. JSON Intermediate Format

The execution engine uses `json-c` [38] to parse the JSON representation of the NFQL query. C structs are used to map the query fields. When reading the JSON query at runtime, the field names of a flow record are read in as strings. Utility functions are defined that map these field names to internal `struct` offsets and the field types / operations to internal `enum` members. Furthermore, each individual branch of the pipeline is described in a self-contained `branch` struct. In this way, the abstract objects that store the JSON query and the results that incubate from each pipeline stage become self-descriptive and hierarchically chainable as shown in Listing 7.

```
1  struct flowquery {
2   size_t                     num_branches;
3   size_t                     num_merger_clauses;
4
5   struct branch**            branchset;
6   struct merger_clause**     merger_clauseset;
7   struct merger_result*      merger_result;
8   struct ungrouper_result*   ungrouper_result;
9  };
10
11 struct branch {
12  int                        branch_id;
13  struct ftio*               ftio_out;
14  struct ft_data*            data;
15
16  size_t                     num_filter_clauses;
17  size_t                     num_grouper_clauses;
18  size_t                     num_aggr_clause_terms;
19  size_t                     num_groupfilter_clauses;
20
21  struct filter_clause**     filter_clauseset;
22  struct grouper_clause**    grouper_clauseset;
23  struct aggr_term**         aggr_clause_termset;
24  struct groupfilter_clause** groupfilter_clauseset;
25
26  struct filter_result*      filter_result;
27  struct grouper_result*     grouper_result;
28  struct groupfilter_result* gfilter_result;
29 };
```

Listing 7. *C structs that hold the JSON query and its results.*

The execution engine uses disable flags to enable/disable a pipeline stage. These flags get turned on for the pipeline stage constructs that are not written in the JSON query by the front-end parser.

C. I/O Processing

The execution engine supports reading and writing flows in NetFlow v5 and IPFIX [39] format. A C library has been written to read and write NetFlow v5 flows using `flow-tools` [13] and IPFIX flows using `libfixbuf` [40] which supports templates and information elements defined in the IPFIX information model [25]. The library sequentially reads flow-records into memory. These flows are indexed to support random access retrieval in $O(1)$ time. This allows NFQL grouper and merger stages to support relative filtering of flows, which is not possible with `flow-tools` and `nfdump`. Each flow-record is stored in a `char` array and the offsets to each field are stored in separate `structs` as shown in Listing 8. The engine also supports the capability to read multiple input traces from `stdin`. This allows users to either `flow-cat` (using `flow-tools` suite for NetFlow v5 traces) or `rwcat` (using SiLK suite for IPFIX traces) the input trace and pipe the results into `nfql` for processing. In this way, `nfql` can be easily plugged into a UNIX pipeline.

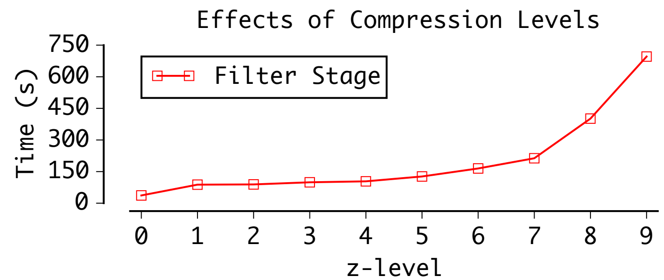

Fig. 5. *The effects of compression level on the performance of the filter stage. An increase in each level adds an overhead on the time to write flows to disk.*

```
1  struct ft_data {
2    int                        fd;
3    struct ftio                io;
4    struct fts3rec_offsets     offsets;
5    struct ftver               version;
6    u_int64_t                  xfield;
7    int                        rec_size;
8    char**                     recordset;
9    size_t                     num_records;
10 };
```

Listing 8. *C struct that holds the flow trace.*

Writing of flows can be requested at any stage of the processing pipeline. This not only allows the user to see intermediary results but is also useful for debugging purposes. Flows can be written in either binary format or printed on screen in a human-readable output. The execution engine uses the `zlib` [41] software library to compress results that are being written to disk. `zlib` supports 9 compression levels with 9 being the highest compression level. The execution engine allows the user to configure the desired compression level at runtime. A default level of 5 is used for writing to disk if a choice is not indicated. Fig. 5 shows the time taken to write a sample of filtered flows of an input trace (see § V for details on the trace and evaluation machine) depending on the requested level of compression. It can be seen that each level adds its own performance overhead. Furthermore, one must note that not every flow analyser uses `zlib` compression technique. For instance, `nfdump` uses `lzo` [42] compression algorithm to trade space for faster compression and decompression.

As mentioned before, the NFQL query uses IPFIX entity names and datatypes [25] for both NetFlow v5 and IPFIX

TABLE III
RUNTIME COMPLEXITY

PIPELINE STAGE	RUNTIME COMPLEXITY
Filter *(worst case)*	$O(n)$ where $n = num(flows)$
Grouper *(average case)*	$O(n * lg(k)) + O(p * n * lg(n))$ where $k = num(unique(flows)), p = num(terms)$
Group Aggregation *(worst case)*	$O(n)$
Group Filter *(worst case)*	$O(g)$ where $g = num(groups)$
Merger *(worst case)*	$O(g^m)$ where $m = num(branches)$
Ungrouper *(worst case)*	$O(g)$

TABLE IV
IPFIX ENTITIES → NETFLOW V5 FIELD NAMES

NETFLOW v5	IPFIX
`srcaddr`	`sourceIPv4Address`
`dstaddr`	`destionationIPv4Address`
`nexthop`	`ipNextHopIPv4Address`
`dPkts`	`packetDeltaCount`
`dOctets`	`octetDeltaCount`
`dFlows`	`deltaFlowCount`
`First`	`flowStartSysUpTime`
`Last`	`flowEndSysUpTime`
`srcport`	`sourceTransportPort`
`dstport`	`destinationTransportPort`
`tcp_flags`	`tcpControlBits`
`prot`	`protocolIdentifier`
`tos`	`ipClassOfService`
`src_as`	`bgpSourceAsNumber`
`dst_as`	`bgpDestinationAsNumber`
`src_mask`	`sourceIPv4PrefixLength`
`dst_mask`	`destinationIPv4PrefixLength`

flows. This allows the query to remain consistent across trace formats. In order to maintain backward compatbility with NetFlow v5, we devised a translation of IPFIX entities to NetFlow v5 field names as shown in Table IV.

D. Execution Workflow

In order to be able to make comparisons on field offsets of a term, the NFQL query supplies the type of the comparison and the length of the field offset. As such, this information is read by the execution engine only at runtime by when it needs a comparator function to perform this task. There are around 450 Information Elements (IE) [43] registered for the IPFIX protocol alone. In order to subvert the need to define complex branching statements with so many entities, a dedicated comparator is defined for every possible field length and comparison operation. A Python script generates C source code for these comparators at compile time conforming to the structure shown in Listing 9.

```
1  struct filter_term {
2    size_t                               field_offset;
3    uint64_t                             value;
4    uint64_t                             delta;
5    struct filter_op*                    op;
6    bool (*func)(
7                   const char* const     record,
8                   size_t                field_offset,
9                   uint64_t              value,
10                  uint64_t              delta
11                 );
12 };
```

Listing 9. *C struct that holds the filter construct of a JSON query.*

This allows term definitions to make runtime calls using a function name derived from the combination of operation type and field length. The execution engine runs each branch in a separate POSIX thread. Affinity masks are used to help delegate each thread to a separate processor core. This allows the engine to parallelize portions of the NFQL pipeline. Table III provides a summary of the runtime complexity of each stage. We discuss further optimizations performed within each stage of the pipeline.

Splitter – The execution engine uses pointers to reference a flow record in the `char` array of flows. This eliminates the need to copy flow-records when splitting across branches in the pipeline. As a result, there is no dedicated splitter stage in the execution engine. Each branch references the flow records from a common memory location. This helps keep memory costs at a minimum when multiple branches are involved.

Filter – The execution engine needs to read all flow records of the input trace into memory before starting the processing pipeline. Since the filter stage uses a set of absolute rules provided by the query to make a decision on whether or not to accept a flow record, it has to pass through the whole in-memory set of flows *again* to produce filtered results. This technique involves multiple linear runs on the trace and therefore slows down when the ratio of the number of filtered flows to the total number of input flows is high. We optimise this behavior by running the filter stage during the process of reading the trace. This means, a decision on whether or not to make room for a flow in memory and eventually hold a pointer for it in filtered results is done upfront as soon as the flow is read from the trace. In addition, if a request to write the filter stage results to disk has been made, the writes are also made as soon as the filter stage decision is available. This allows reading-filtering-writing to happen in $O(n)$ time, where n is the number of flows in the trace as shown in Table III. Using a publicly available input trace (see § V for details on the trace and evaluation machine) we compare the performance benefits of such an inline filter as shown in Fig. 6. 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 the time taken to process the trace. It can be seen that an inline filter stage implementation runs 10 times faster than a dedicated filter and its benefits are more pronounced when more flows are accepted by the filter in the processing pipeline.

Grouper – allows relative comparison between field offsets of two different flows. In order to do such comparisons, a simple approach is to linearly walk through each flow against the entire set leading to a complexity of $O(n^2)$, where n is

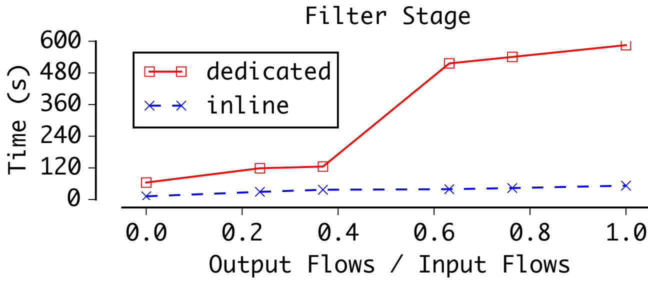

Fig. 6. *Performance comparison of an inline filter against a dedicated filter stage as defined by the processing pipeline. An inline filter runs 10 times faster than a dedicated filter and its benefits are more pronounced when more flows are accepted by the filter.*

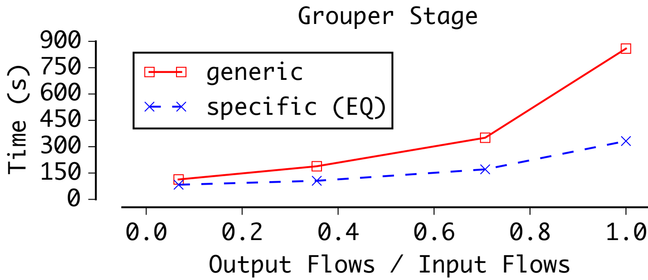

Fig. 7. *Performance benefits of special case handling of the equality operator when grouping flows against the generic case. With an equality operation, the need to search for unique records and a subsequent binary search goes away.*

the number of filtered flows. A better approach can be to use a hash table and then map each pointer while walking down the filtered flows once, leading to a complexity of $O(n)$. The hash table approach, however, only works on equality comparisons (such as SiLK), while NFQL supports more (see Fig. 3) operations. As such, the execution engine follows a hybrid approach. It sorts the filtered flows based on the field offsets indicated in the NFQL query. This helps the execution engine perform a nested binary search to reduce the linear pass to a fairly small set of filtered flows. As a result, the grouper can perform faster search lookups to find records that must group together in $O(n * lg(k))$ time with a preprocessing step taking $O(p * n * lg(n))$ in the average case, where n is the number of filtered records, p is the number of grouping terms in a clause and k is the number of unique filtered flows as shown in Table III. We further optimized the grouper for equality comparisons. With an equality comparison, the need to search for unique records and a subsequent binary search goes away. Fig. 7 shows the performance benefits of special case handling of the equality operator against the generic case. Groups with an equality operation can now be formed in $O(n)$ time with the same preprocessing step taking $O(p*n*lg(n))$.

Each resultant group record is a conglomeration of multiple flow records with some common characteristics. Some of the non-common characteristics (such as number of packets in each flow) can also be aggregated into a single value using group aggregations as defined in the query. Such an aggregated group record is again mapped to a flow record template of the input trace. This allows aggregated group records to be written to disk as a representative of all its members.

Group Filter – filters groups based on absolute rules defined by the query. The implementation is similar to that of a filter and the stage has a complexity of $O(g)$ where g is the number of grouped flows as shown in Table III.

Merger – is used to relate filtered groups from different branches to create streams. However, the number of branches that need to be spawned is not known until runtime. As a result, the execution engine uses an iterator that can provide all possible permutations of grouped flows. The result of the iterator is later used to make a match.

Furthermore, the merger needs to match a grouped flow from one branch with grouped flows of every other branch. This leads to a complexity of $O(g^m)$ where g is the number of filtered grouped flows and m is the number of branches as shown in Table III. The possible number of tries when matching grouped flows can be reduced by sorting grouped flows on the field offsets specified by the query. This allows us to optimize the merger to skip over iterator permutations when the state of a current field offset value may not allow any further match beyond the index in the current branch. This means, if same field offsets are used, the query designer can get performance benefits by keeping the same order of terms in both grouper and merger stages.

The query language also bases merger matches on the notion of matched tuples. This means that a filtered grouped flow can be written to disk multiple times if it is part of multiple matched tuples. This situation worsens when different branches result in similar filtered grouped flows. Since, the function of the merger is to find a match of grouped flows across branches, all grouped flows across branches that satisfy the condition can be clubbed into one collection instead of separate tuples. All grouped flows within a collection can then be written disk at once. This eliminates the inherent redundancy and significantly improves performance of the merger stage.

Ungrouper – accepts a collection of matched filtered grouped flows as input and iterates over each collection to unfold its groups and write their flow record members to disk.

E. Further Performance Optimizations

There can be a situation where the user writing the query may incorrectly ask for aggregation on a field already specified in a grouper (or filter) clause. If the relative operator is an equality comparison, the aggregation on such a field becomes less useful, since members of the grouped flow will always have the same value for that field. The execution engine detects this kind of redundant request and ignores such aggregations.

The execution engine has dedicated comparator functions for each type of operation and the type of the field offset it operates upon. It is not guaranteed that given the type of the query and the trace, the engine will eventually complete all stages of the pipeline. It is also possible that the engine exits early, because there is nothing more for the next stage to compute. The function pointers therefore are set as late as possible and are invoked from respective stages just before the comparison is requested.

Each stage of the processing pipeline is dependent on the result of the previous one. As a result, the next stage should only execute, when the previous stage returned results. Implementing such a response was straightforward for the grouper and group filter. However, the merger stage proceeds only when every branch has non-zero filtered groups. The iterator initializer deallocates and returns `NULL` if any one branch has 0 filtered groups. Consequently a check is performed in the merger to make sure that the initalizer is *not* `NULL`.

The flow-records echoed to the standard output can also be written to disk. In fact, results from each stage of the pipeline can be written to disk. This leads to additional loops over the records if the writes are made at the end of the processing pipeline. The execution engine therefore writes each record to a file as soon as it exits out of the pipeline stage.

V. Performance Evaluation

We evaluated the performance of `nfql` using a publicly available trace (#07, with ∼20M flows) from the SimpleWeb [44] repository. The input trace is in the `flow-tools` format and is compressed using the `zlib` suite using `ZLIB_LEVEL` 5. In order to perform comparisons, we also converted the input trace to `nfdump` and SiLK formats while keeping the same compression level. The performance evaluation was run on a machine with 24 cores of 2.5 GHz clock speed and 18 GiB of memory. We black box each stage of the pipeline and evaluate it against contemporary flow analysis tools.

Filter – We developed a set of queries to stress the filter stage of `flow-tools`, `nfdump`, `nfql` and SiLK. Each query increases the threshold on the `packetDeltaCount` field offset [25] to control the amount of flow records that are passed by the filter. The resultant filtered records are written to disk and compressed at `ZLIB_LEVEL` 5. 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 the time taken to process the trace as shown in Fig. 8. It can be seen that the performance of the filter stage in `nfql` is comparable to that of SiLK and `flow-tools`. SiLK takes less time on lower ratios, but SilK and `nfdump` also use their own file format. As a result, the amount of data that needs to read (or written) is 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` uses the `lzo` compression scheme which trades space for achieving faster compression and decompression. As such, adding `lzo` compression support in `nfql` will help further decrease the I/O times of the execution engine. Note that all the tools were single-threaded in this evaluation. `nfdump` and `flow-tools` only support the filter stage of the pipeline and therefore are not considered in further evaluations.

Grouper – The second set of queries stress the grouper stage of `nfql` and SiLK. We reuse the filter query that produces a 1.0 output/input ratio to allow the grouper to receive the entire trace as filtered flows. Similar to the filter stage evaluation, 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 written to disk using the same `zlib` compression level. The ratio of the number of groups formed to the number of the input filtered flows is plotted against time taken to process the trace as shown in Fig. 9. It can be seen that the time taken by the tools are comparable on lower ratios, but on higher ratios, `nfql` starts to drift apart. Since most of the time is taken in writing the records to files, it is unclear whether SiLK's usage of its own file format is responsible for the drift. SiLK's query also invokes the `rwgroup` tool with a `--summarize` flag to force it to write only the first record of each group to make both tools write the same number of records. Since SiLK does not support unfolding of grouped flows, it does 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 stream. It is also important to note that both the tools again remained single-threaded throughout the evaluation. SiLK took advantage of an inherent concurency arising from a pipe between `rwsort` and `rwgroup`, which makes the two processes run concurrently, the effect of which gets more pronounced on higher ratios. The profiling results from GNU `gprof` [45] indicate that 60% of the time is taken in `qsort` comparator calls. As a result,

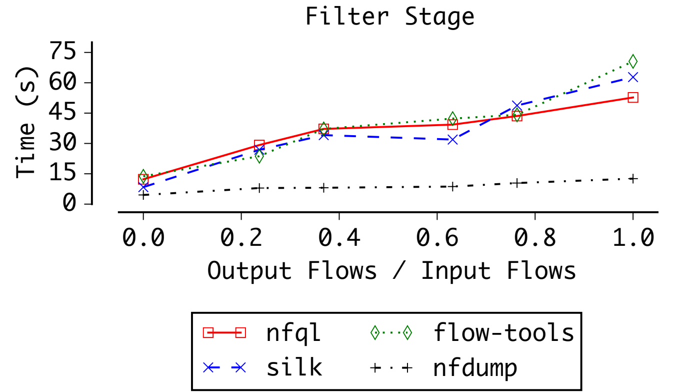

Fig. 8. *Performance comparison of the filter stage of four analysis tools – nfql, SiLK, flow-tools, nfdump. Performance of nfql is comparable to that of flow-tools and SiLK. nfdump appears significantly faster because it uses the lzo compression scheme to trade space for higher compression speeds.*

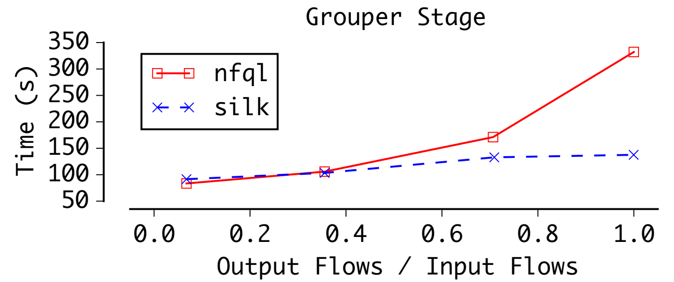

Fig. 9. *Performance comparison of the grouper stage of nfql and SiLK. SiLK saves time in higher ratios by not storing information of each flow within a group since it does not support unfolding groups into original flows.*

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 for other comparison operators.

Group Filter – The third set of queries stress the group filter stage of `nfql` and SiLK. We reuse the filter and grouper queries that produce a 1.0 output/input ratio to allow the group filter to receive the entire trace as input. This 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 `packetDeltaCount` field offset [25] to control the amount of groups that are filtered further. The filtered groups are written to disk using the same `zlib` compression level. The ratio of the number of output filtered groups to the number of the input groups is plotted against time taken to process the trace as shown in Fig. 10. 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 the 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 `nfql` group filter has a 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.

Merger – The fourth set of queries stress the merger stage of `nfql` and SiLK. We reuse the filter, grouper and group filter queries that produce a 1.0 output/input ratio. These queries are then run in two separate branches to produce identical filtered group records. The merger then applies match rules to produce different output to input ratios. The groups that are merged are written to disk using the same `zlib` compression level. The ratio of the number of merged groups to twice (since each branch pushes the entire trace as an input to the merger) the number of flow records in the original trace is plotted against time as shown in Fig. 11. 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. It can be seen that the merger is the most performance critical stage of the NFQL pipeline. 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 results of the filter, grouper and group filter stage to disk. As a result, the amount of disk I/O involved is twice as much too. Even though each branch is delegated to a separate core, most of the time is spent in writing flows to disk. The optimized merger takes advantage of the sorted nature of filtered groups (see § IV) and therefore can significantly reduce the number of merger matches. It also writes a merged group record to a file only once despite the number of times it has matched.

Ungrouper – The last set of queries stress the ungrouper stage of `nfql`. They reuse the entire merger queries as is, but enable the ungrouper as well. This means, that the ungrouper now attempts to unfold the merged groups returned by the merger to write individual flow records to disk using the same `zlib` compression level. 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 rewriting the merged groups to disk. This means that the execution engine ends up taking twice the amount of time than the merger. SiLK does not support unfolding of grouped flows and is therefore not considered in this final evaluation.

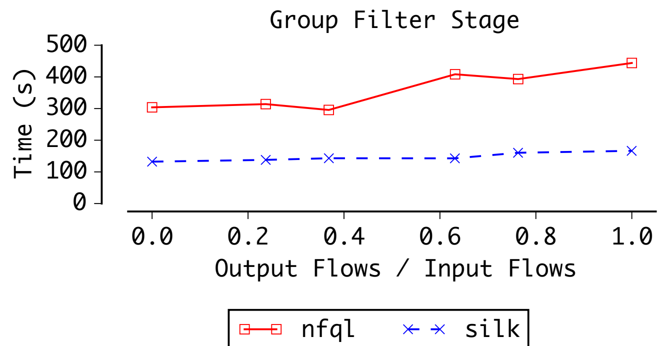

Fig. 10. *Performance comparison of the group filter stage of nfql and SilK. SiLK performs better by running an inline filter while grouping the flows.*

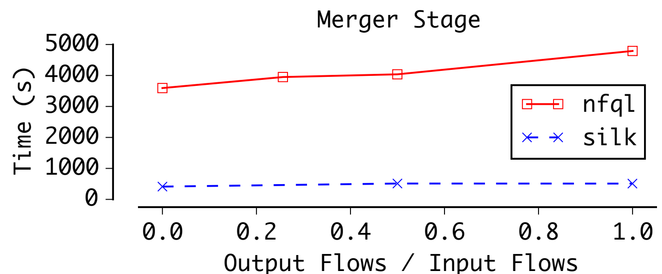

Fig. 11. *Performance comparison of the merger stage of nfql and SiLK. A merger is the most performance critical stage where SiLK performs better.*

Overall, we observe that `nfql` has comparable execution times in the filter stage to SiLK and `flow-tools`. SiLK performs better in the later stages because it can optimize its operations in favor of the limited set of equality operations (see Fig. 3) and its usage of a different file storage format. `nfql` trades performance in the grouping and merging stages to increase the expressiveness of NFQL (see § VI) by supporting more operations. We believe this to be an acceptable trade-off since the language expands the scope of current flow processing tools. The performance evaluation queries developed as a part of this research work are released [19] to the community to support development of a more general benchmark suite for flow analysis tools.

VI. Applications

We present two real-world applications of NFQL – using flow-signatures to identify applications and using behavioural signatures to identify SSH compromise detection attacks.

A. Application Identification using Flow Signatures

Techniques to identify classes (such as P2P traffic, or web traffic) of application mix from network flow records are available in literature. However, network operators not only are interested in application mix, but are also interested in identification of specific applications (such as Chrome or Skype) that generate traffic in their network. This information not only assists in security assessments to identify malicious software, but also can be used to profile network usage statistics broken down by specific applications. In this study, we propose a technique to identify applications by searching for their signatures in flow traces without using the Deep Packet Inspection (DPI) mechanism. This is based on the hypothesis that applications generate unique flow signatures that can be used as a fingerprint for their identification. In order to verify our hypothesis, we recorded flow traces of several applications and subsequently analyzed the traces to identify flow signatures of these applications. These flow signatures were formalized as NFQL queries which were executed on several flow traces to evaluate the approach.

Example – For the purpose of demonstration, we explain an example NFQL query which was used to identify Skype signatures from network flow records. Salman Baset *et al.* in [46] perform a thorough inspection of Skype application behavior during login, call establishment and media transfer. We not only build our query upon these observations, but also revisit and adapt the NFQL queries for current version of Skype. For instance, on start-up, a Skype client connects to `skype.com` and sends an HTTP GET request `/getlatestversion` in order to check whether updates are available. During initial login, the client sends four Simple Service Discovery Protocol (SSDP) messages over UDP to port 1900. Listing 10 shows the NFQL branch construct to identify these SSDP messages.

```
1   branch SSDP {
2     filter F_SSDP {
3       destinationTransportPort = 1900
4       protocolIdentifier = UDP
5       destinationIPv4Address = 239.255.255.250
6     }
7     grouper G_SSDP {
8       sourceIPv4Address = sourceIPv4Address
9       destinationIPv4Address = \
10      destinationIPv4Address
11
12      aggregation {
13        sourceIPv4Address
14        destinationIPv4Address
15        sum(octetDeltaCount)
16        sum(packetDeltaCount)
17      }
18    }
19    groupfilter GF_SSDP {
20      packetDeltaCount >= 4
21    }
22  }
```

Listing 10. *Branch construct to identify SSDP messages*

During login, a client also sends four NAT Port Mapping Protocol (NAT-PMP) [47] messages. The NAT-PMP new port mapping request is sent over UDP to port 5351 and is resent up to nine times by doubling the timeout (starting with 250 ms) in each successive interval. Listing 11 shows the NFQL branch construct to identify these NAT-PMP messages.

```
1   branch NAT_PMP {
2     filter F_NAT_PMP {
3       destinationTransportPort = 5351
4       protocolIdentifier = UDP
5     }
6     grouper G_NAT_PMP {
7       sourceIPv4Address = sourceIPv4Address
8       destinationIPv4Address = \
9       destinationIPv4Address
10      aggregation {
11        sourceIPv4Address
12        destinationIPv4Address
13        sum(packetDeltaCount)
14      }
15    }
16    groupfilter GF_NAT_PMP {
17      packetDeltaCount >= 4
18    }
19  }
```

Listing 11. *Branch construct to identify NAT-PMP messages*

At each login, a Skype client contacts a Skype login server. Afterwards, the client connects to a bootstrap supernode on port 33033. The signature for Skype has an undefined order of SSDP and NAT-PMP requests, but they are always intersecting. To reflect this behaviour, we use a logical OR (line 20) on two conditions: a group from either branch should overlap with each other. The entire NFQL query used for identification of Skype signatures is shown in Listing 12.

We further refer the reader to [23] for explanation of more example NFQL queries used to identify other applications.

```
1   branch supernode {
2     filter F_supernode {
3       destinationTransportPort = 33033
4       protocolIdentifier = UDP
5     }
6     grouper G_supernode {
7       sourceIPv4Address = sourceIPv4Address
8       aggregation {
9         sourceIPv4Address
10      }
11    }
12  }
13  merger {
14    SSDP.sourceIPv4Address = \
15    NAT_PMP.sourceIPv4Address
16
17    NAT_PMP.sourceIPv4Address = \
18    supernode.sourceIPv4Address
19
20    NAT_PMP o SSDP OR SSDP o NAT_PMP
21  }
22  ungrouper {}
```

Listing 12. *NFQL query to identify Skype application signature.*

Validation – of NFQL queries was performed using a controlled evaluation of a known data-set. The dataset was obtained by identifying a pool of ten users who collected flow data for a period of two weeks. Users were asked to report applications they used during this collection period. Five of those applications were chosen for identification. Table V shows the results of application identification using NFQL queries. A ✔ marker indicates true positives, ✖ indicates true negatives and ❍ indicates false positives. The two false positives resulted from incorrect classification of iTunes flow

TABLE V
APPLICATION FLOW SIGNATURES: RESULTS

USER	SKYPE	OPERA	AMAROK	CHROME	LIVE
#1	✔	✖	❍	✖	✖
#2	✔	✖	✖	✖	✖
#3	✖	✖	❍	✖	✖
#4	✔	✖	✖	✖	✖
#5	✖	✖	✖	✖	✖
#6	✔	✖	✔	✔	✖
#7	✖	✖	✖	✖	✖
#8	✖	✔	✔	✖	✖
#9	✖	✖	✖	✖	✖
#10	✔	✔	✔	✔	✖

records as those of the Amarok player because Amarok mimics the iTunes behaviour and therefore has signatures similar to iTunes. This reveals that 48 out of 50 application signatures were correctly classified leading to a success rate of 96% for these applications.

B. SSH Compromise Detection

Rick Hofstede *et al.* in [48] present a technique to detect SSH compromises using NetFlow records. They rely on behavioural signatures of SSH clients and brute force attack tools to characterise SSH attacks. We present NFQL queries to represent the three-phase transition model that can be used detect such SSH compromise attacks. The queries have been tested by simulating a SSH dictionary attack in a lab setting.

Scan Phase – In this phase, an attacker port scans all endpoints in an IP address block to detect hosts running a SSH daemon on port 22. According to [48], in this phase attacking flows have no more than 2 packets per flow, and there are at least 200 flow records per 1 minute. Listing 13 presents a NFQL branch construct to cover this phase.

```
1  branch SshScan {
2    filter ScanFilter {
3      destinationTransportPort = 22
4      protocolIdentifier = TCP
5      packetDeltaCount <= 2
6    }
7    grouper ScanGrouper {
8      sourceIPv4Address = sourceIPv4Address
9      flowStartSysUpTime = flowStartSysUpTime \
10     delta 60000
11     aggregation {
12       sourceIPv4Address
13       count(destinationIPv4Address)
14     }
15   }
16   groupfilter ScanGroupFilter {
17     destinationIPv4Address >= 200
18   }
19 }
```

Listing 13. *A branch construct to detect a SSHCure scan phase.*

The filter stage (lines 2–6) is used to filter flows that have no more than 2 packets per flow and where the packets were transmitted using TCP over port 22. In the grouper stage, we begin by grouping flows by their source address (line 8). We then restrict all flows in a group to start (line 9) within the same minute. We further use an aggregator (lines 12–13) to save source addresses and count the number of flows within each group. In the group filter stage (lines 16–19), we filter groups with at least 200 flows.

Brute-force Phase – In this phase, an attacker performs a brute-force attack with different user name and password combinations to the target hosts that responded in the first phase. As a result, the amount of traffic significantly increases in comparison to the previous stage. According to [48], this phase has 8–14 packets per flow in a minute with at least 20 flow records per minute per attacker. Listing 14 presents the NFQL branch construct to cover this phase.

```
1  branch BruteForce {
2    filter BFFilter {
3      destinationTransportPort = 22
4      protocolIdentifier = TCP
5      packetDeltaCount >= 8
6      packetDeltaCount <= 14
7    }
8    grouper BFGrouper {
9      sourceIPv4Address = sourceIPv4Address
10     flowStartSysUpTime = flowStartSysUpTime \
11     delta 60000
12     aggregation {
13       sourceIPv4Address
14       count(destinationIPv4Address)
15     }
16   }
17   groupfilter BFGroupFilter {
18     destinationIPv4Address >= 20
19   }
20 }
```

Listing 14. *A branch construct to detect a SSHCure bruteforce phase.*

The filter stage (lines 3–6) is used to identify flows where packets are transmitted using TCP over port 22, with each flow consisting of 8–14 number of packets. We reuse the grouper (lines 8–15) from the scanning phase. In the group filter stage we pass groups with at least 20 group records per attacker.

Compromise Phase – After a login to a compromised host is successful, there is still some traffic between the attacker and the host, which is less intensive when compared with the traffic of the brute-force phase. According to [48], this phase is identified as the phase which differs from the brute-force phase where each flow has less than 8 packets per minute or more than 14 packets per minute. Listing 15 presents the NFQL branch construct to cover this phase.

```
1  branch DieOff {
2    filter DOFilter {
3      destinationTransportPort = 22
4      protocolIdentifier = TCP
5      packetDeltaCount < 8 OR \
6      packetDeltaCount > 14
7    }
8    grouper DOGrouper {
9      sourceIPv4Address = sourceIPv4Address
10     destinationIPv4Address = \
11     destinationIPv4Address
12     aggregation {
13       sourceIPv4Address
14       sum(packetDeltaCount)
15     }
16   }
17   groupfilter DOGroupFilter {
18     packetDeltaCount > 2
19   }
20 }
```

Listing 15. *A branch construct to detect a SSHCure Dieoff Phase.*

The filter stage (lines 2−6) is used to identify flows where packets are transmitted using TCP over port 22 with number of packets in each flow either being less than 8 or greater than 14. The grouper stage (lines 8−16) is used to group flows according to their source and destination addresses. Since, flow records from the scanning phase can pass through these filters as well, we define a group filter (lines 17−19) which removes all flow records with less than 2 packets.

Listing 16 present the entire query where the merger finally merges brute-force and compromise phases by the same source address which is the assumed address of the attacker.

```
1  branch SshScan { ... }
2  branch BruteForce { ... }
3  branch DieOff { ... }
4
5  merger {
6    BruteForce.sourceIPv4Address = \
7    DieOff.sourceIPv4Address
8  }
9  ungrouper {}
```

Listing 16. *A NFQL query to perform SSH compromise detection.*

Overall, we demonstrated that NFQL can be used to identify applications from their network flow fingerprints and detect SSH compromise detection attacks.

VII. Conclusion

We presented NFQL, a stream-based flow query language. It can filter flows, combine flows into groups, calculate temporal relationships and aggregated statistics on groups. It can merge grouped flows, apply absolute or relative filters, and unfold grouped flows into individual flows. We demonstrated an implementation of NFQL [19] that has comparable execution times in the filter stage to SiLK and `flow-tools`. SiLK performs better in grouping and merging stages since it can optimize its operations in favor of the limited set of comparisons it supports. `nfql` trades performance in the grouping and merging stages to support more operations to expand the scope of current flow-processing tools. We presented applications to demonstrate this expressiveness of NFQL.

We believe it would be useful in the future to invest more effort [49] towards compiling a general benchmarking methodology for such flow-analysis tools. Towards this goal, we are releasing our performance evaluation queries [19] to the community to support and strengthen this effort.

VIII. Acknowledgements

Vladislav Marinov proposed the DSL specification. Kaloyan Kanev and Johannes Schauer wrote initial implementations. Vladislav Perelman wrote queries to identify application signatures. Durim Morina wrote the front-end parser. Corneliu-Claudiu Prodescu added IPFIX support. Steffie Jacob Eravuchira and Veranika Liaukevich revisited queries to identify application signatures for current versions of applications and also wrote queries to detect SSH compromises.

This work was funded by Flamingo, a Network of Excellence project (ICT-318488) supported by the European Commission under its Seventh Framework Programme.

References

[1] K. Xu, Z. Zhang, and S. Bhattacharyya, "Profiling Internet Backbone Traffic: Behavior Models and Applications," ser. ACM SIGCOMM, 2005, pp. 169–180. [Online]. Available: http://doi.acm.org/10.1145/1080091.1080112

[2] T. Karagiannis, K. Papagiannaki, and M. Faloutsos, "BLINC: Multilevel Traffic Classification in the Dark," ser. ACM SIGCOMM, 2005, pp. 229–240. [Online]. Available: http://doi.acm.org/10.1145/1080091.1080119

[3] M. Kim, H. Kang, S. Hong, S. Chung, and J. W. Hong, "A Flow-based Method for Abnormal Network Traffic Detection," ser. IEEE/IFIP Network Operations and Management Symposium, NOMS, 2004. [Online]. Available: http://dx.doi.org/10.1109/NOMS.2004.1317747

[4] D. Schatzmann, W. Mühlbauer, T. Spyropoulos, and X. A. Dimitropoulos, "Digging into HTTPS: Flow-Based Classification of Webmail Traffic," ser. ACM SIGCOMM Internet Measurement Conference IMC, 2010, pp. 322–327. [Online]. Available: http://doi.acm.org/10.1145/1879141.1879184

[5] J. Quittek, T. Zseby, B. Claise, and S. Zander, "Requirements for IP Flow Information Export (IPFIX)," RFC 3917 (Informational), Internet Engineering Task Force, Oct. 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3917.txt

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

[7] B. Claise, B. Trammell, and P. Aitken, "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information," RFC 7011 (Internet Standard), Internet Engineering Task Force, Sep. 2013. [Online]. Available: http://www.ietf.org/rfc/rfc7011.txt

[8] B. Trammell and E. Boschi, "An Introduction to IP Flow Information Export (IPFIX)," ser. IEEE Communications Magazine, vol. 49, no. 4, 2011, pp. 89–95. [Online]. Available: http://dx.doi.org/10.1109/MCOM.2011.5741152

[9] T. Zseby, E. Boschi, N. Brownlee, and B. Claise, "IP Flow Information Export (IPFIX) Applicability," RFC 5472 (Informational), Internet Engineering Task Force, Mar. 2009. [Online]. Available: http://www.ietf.org/rfc/rfc5472.txt

[10] A. C. Callado, C. A. Kamienski, G. Szabo, B. P. Gero, J. Kelner, S. F. L. Fernandes, and D. F. H. Sadok, "A Survey on Internet Traffic Identification," ser. IEEE Communications Surveys and Tutorials, vol. 11, no. 3, 2009, pp. 37–52. [Online]. Available: http://dx.doi.org/10.1109/SURV.2009.090304

[11] A. Sperotto, G. Schaffrath, R. Sadre, C. Morariu, A. Pras, and B. Stiller, "An Overview of IP Flow-Based Intrusion Detection," ser. IEEE Communications Surveys and Tutorials, vol. 12, no. 3, 2010. [Online]. Available: http://dx.doi.org/10.1109/SURV.2010.032210.00054

[12] "flow-tools - A set of programs for processing and managing NetFlow exports from Cisco and Juniper routers." http://freecode.com/projects/flow-tools, [Online; accessed 05-Aug-2016].

[13] S. Romig, M. Fullmer, and R. Luman, "The OSU Flow-tools Package and CISCO NetFlow Logs," ser. Conference on Systems Administration LISA, 2000, pp. 291–303. [Online]. Available: http://dl.acm.org/citation.cfm?id=1045502.1045521

[14] "nfdump - A tool to collect and process netflow data on the command line," http://nfdump.sourceforge.net, [Online; accessed 05-Aug-2016].

[15] R. Hofstede, P. Celeda, B. Trammell, I. Drago, R. Sadre, A. Sperotto, and A. Pras, "Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX," ser. IEEE Communications Surveys and Tutorials, vol. 16, no. 4, 2014, pp. 2037–2064. [Online]. Available: http://dx.doi.org/10.1109/COMST.2014.2321898

[16] M. Thomas, L. Metcalf, J. M. Spring, P. Krystosek, and K. Prevost, "SiLK: A Tool Suite for Unsampled Network Flow Analysis at Scale," ser. IEEE International Congress on Big Data, 2014. [Online]. Available: http://dx.doi.org/10.1109/BigData.Congress.2014.34

[17] G. Sadasivan, N. Brownlee, B. Claise, and J. Quittek, "Architecture for IP Flow Information Export," RFC 5470 (Informational), Internet Engineering Task Force, Mar. 2009, updated by RFC 6183. [Online]. Available: http://www.ietf.org/rfc/rfc5470.txt

[18] J. F. Allen, "Maintaining Knowledge about Temporal Intervals," ser. Communications of the ACM, vol. 26, no. 11, 1983, pp. 832–843. [Online]. Available: http://doi.acm.org/10.1145/182.358434

[19] "nfql - A C implementation of the network flow query language," http://nfql.vaibhavbajpai.com, [Online; accessed 05-Aug-2016].

[20] V. Marinov and J. Schönwälder, "Design of an IP Flow Record Query Language," ser. Conference on Autonomous Infrastructure, Management and Security, AIMS, 2008, pp. 205–210. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-70587-1_20

[21] ——, "Design of a Stream-Based IP Flow Record Query Language," ser. IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, DSOM, 2009, pp. 15–28. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04989-7_2

[22] K. Kanev, N. Melnikov, and J. Schönwälder, "Implementation of a Stream-Based IP Flow Record Query Language," ser. Conference on Autonomous Infrastructure, Management and Security, AIMS, 2010. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-13986-4_21

[23] V. Perelman, N. Melnikov, and J. Schönwälder, "Flow Signatures of Popular Applications," ser. IFIP/IEEE International Symposium on Integrated Network Management, IM, 2011, pp. 9–16. [Online]. Available: http://dx.doi.org/10.1109/INM.2011.5990668

[24] V. Bajpai, J. Schauer, and J. Schönwälder, "NFQL: A Tool for Querying Network Flow Records," ser. IFIP/IEEE International Symposium on Integrated Network Management IM, 2013. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6573045

[25] B. Claise and B. Trammell, "Information Model for IP Flow Information Export (IPFIX)," RFC 7012 (Proposed Standard), Internet Engineering Task Force, Sep. 2013. [Online]. Available: http://www.ietf.org/rfc/rfc7012.txt

[26] B. Nickless, J. Navarro, and L. Winkler, "Combining Cisco NetFlow Exports with Relational Database Technology for Usage Statistics, Intrusion Detection, and Network Forensics," ser. Conference on Systems Administration (LISA), 2000, pp. 285–290. [Online]. Available: http://dl.acm.org/citation.cfm?id=1045502.1045520

[27] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and Issues in Data Stream Systems," ser. ACM Symposium on Principles of Database Systems PODS, 2002, pp. 1–16. [Online]. Available: http://doi.acm.org/10.1145/543613.543615

[28] C. D. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk, "Gigascope: A Stream Database for Network Applications," ser. ACM SIGMOD Conference on Management of Data, 2003, pp. 647–651. [Online]. Available: http://doi.acm.org/10.1145/872757.872838

[29] M. Sullivan and A. Heybey, "Tribeca: A System for Managing Large Databases of Network Traffic," ser. USENIX Annual Technical Conference, 1998. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268256.1268258

[30] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," ser. USENIX Winter Technical Conference, 1993, pp. 259–270. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267303.1267305

[31] D. Moore, K. Keys, R. Koga, E. Lagache, and K. C. Claffy, "The CoralReef Software Suite as a Tool for System and Network Administrators," ser. Conference on Systems Administration LISA, 2001, pp. 133–144. [Online]. Available: http://dl.acm.org/citation.cfm?id=1047531.1047546

[32] S. Kornexl, V. Paxson, H. Dreger, A. Feldmann, and R. Sommer, "Building a Time Machine for Efficient Recording and Retrieval of High-Volume Network Traffic," ser. Internet Measurement Conference, IMC, 2005, pp. 267–272. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251086.1251109

[33] D. Plonka, "FlowScan: A Network Traffic Flow Reporting and Visualization Tool," ser. Conference on Systems Administration LISA, 2000, pp. 305–317. [Online]. Available: http://www.usenix.org/publications/library/proceedings/lisa2000/plonka.html

[34] L. Deri and S. Suin, "Effective Traffic Measurement Using ntop," ser. IEEE Communications Magazine, vol. 38, no. 5, May 2000, pp. 138–143. [Online]. Available: http://dx.doi.org/10.1109/35.841838

[35] A. Øslebø, "Stager A Web Based Application for Presenting Network Statistics," ser. IEEE/IFIP Network Operations and Management Symposium, NOMS, 2006. [Online]. Available: http://dx.doi.org/10.1109/NOMS.2006.1687613

[36] B. Trammell and E. Boschi, "Bidirectional Flow Export Using IP Flow Information Export (IPFIX)," RFC 5103 (Proposed Standard), Internet Engineering Task Force, Jan. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5103.txt

[37] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," RFC 7159 (Proposed Standard), Internet Engineering Task Force, 2014. [Online]. Available: http://www.ietf.org/rfc/rfc7159.txt

[38] "json-c - A JSON implementation in C." https://github.com/json-c/json-c, [Online; accessed 05-Aug-2016].

[39] B. Trammell, E. Boschi, L. Mark, T. Zseby, and A. Wagner, "Specification of the IP Flow Information Export (IPFIX) File Format," RFC 5655 (Proposed Standard), Internet Engineering Task Force, Oct. 2009. [Online]. Available: http://www.ietf.org/rfc/rfc5655.txt

[40] "libfixbuf - An implementation of the IPFIX Protocol as a C library, for building IPFIX Collecting and Exporting Processes." http://tools.netsa.cert.org/fixbuf, [Online; accessed 05-Aug-2016].

[41] P. Deutsch and J.-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3," RFC 1950 (Informational), Internet Engineering Task Force, May 1996. [Online]. Available: http://www.ietf.org/rfc/rfc1950.txt

[42] "Lempel–Ziv–Oberhumer (LZO) - A lossless data compression algorithm," http://www.oberhumer.com/opensource/lzo, [Online; accessed 05-Aug-2016].

[43] "IP Flow Information Export (IPFIX) Entities," http://www.iana.org/assignments/ipfix/ipfix.xml, [Online; accessed 05-Aug-2016].

[44] "SimpleWeb - Traces," https://www.simpleweb.org/wiki/index.php/Traces, [Online; accessed 05-Aug-2016].

[45] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: a Call Graph Execution Profiler," ser. Symposium on Compiler Construction, SIGPLAN, 1982, pp. 120–126. [Online]. Available: http://doi.acm.org/10.1145/800230.806987

[46] S. Baset and H. Schulzrinne, "An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol," ser. IEEE International Conference on Computer Communications, INFOCOM, 2006. [Online]. Available: http://dx.doi.org/10.1109/INFOCOM.2006.312

[47] S. Cheshire and M. Krochmal, "NAT Port Mapping Protocol (NAT-PMP)," RFC 6886 (Informational), Internet Engineering Task Force, Apr. 2013. [Online]. Available: http://www.ietf.org/rfc/rfc6886.txt

[48] R. Hofstede, L. Hendriks, A. Sperotto, and A. Pras, "SSH Compromise Detection using NetFlow/IPFIX," ser. Computer Communication Review, vol. 44, no. 5, 2014, pp. 20–26. [Online]. Available: http://doi.acm.org/10.1145/2677046.2677050

[49] P. Velan, "Practical experience with IPFIX flow collectors," ser. IFIP/IEEE International Symposium on Integrated Network Management (IM), 2013, pp. 1021–1026. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6573125

Vaibhav Bajpai is a Postdoctoral Researcher in the Computer Networks and Distributed Systems research group at Jacobs University Bremen, Germany. He works with Prof. Dr. Jürgen Schönwälder. His current research focuses on Internet performance measurements using large-scale measurement platforms such as SamKnows and RIPE Atlas. He is interested in measuring IPv6 and access network performance from end-user networks. He received his PhD (2016) and Masters (2012) degrees in Computer Science from Jacobs University Bremen, Germany and his Bachelors degree (2009) in Computer Science and Engineering from Uttar Pradesh Technical University, India. He worked as a systems engineer at Infosys Technologies Limited, India for a year before coming to Germany.

Jürgen Schönwälder is Professor of Computer Science at Jacobs University Bremen where he is leading the Computer Networks and Distributed Systems research group. His research interests include network management, distributed systems, network measurements, embedded networked systems, and network security. He is an active member of the Internet Engineering Task Force (IETF). He has edited more than 30 network management related specifications and standards. He has been principal investigator in several European research projects (Emanics, Flamingo, Leone). He currently serves on the editorial boards of the Springer Journal of Network and Systems Management and the Wiley International Journal of Network Management. He is co-editor of the Network and Service Management series of the IEEE Communications Magazine.