Bachelor Thesis
Computer Science

Flowy 2.0

fast execution of stream based IP flow queries

Johannes Schauer

Abstract. Network traffic analysis using flow records is used for gathering statistics, usage patterns, billing, network planning or intrusion and attack detection. Unfortunately current flow analysis tools suffer from poor query syntax and limited functionality for more complex analysis tasks. This project proposes the traffic analysis tool called Flowy which solves mentioned issues by using a powerful query language that goes beyond simple filtering of packets by some of their attributes. This bachelor thesis presents a fast C based implementation of the core of the former Python implementation of Flowy.

1 Introduction

With the growth of networks and their generated traffic there is an ever increasing need of network owners for better tools allowing to analyze the huge amounts of data transfered therein. Unfortunately bandwidths of network hardware seem to grow faster than space of permanent storage solutions which makes it unfeasible to capture all transfered traffic. Even only capturing the headers of forwarded packets without their payload would generate enormous amounts of data. The NetFlow format which was developed by Cisco Systems [6] allows for more efficient recording of transfered data by organizing traffic meta data into flow records. Each flow record is identified by a seven tuple of the following items:

- soure IP address
- source port number
- destination IP address
- destination port number
- IP protocol
- input Interface
- IP type of service

A flow record stores information like overall amount of data and number of packets transfered, start and end timestamps or TCP flags. Aggregation of the data from several packets that represent a single flow in such a way greatly reduces the amount of storage required but still allows for a sufficient amount of analysis of the generated traffic. The most widely used formats of NetFlow are v5 and v9 where v5 structures flow information in a fixed packet format, while v9 allows for an extensible format defined by templates. The IETF proposes the IPFIX format [7] as a standardized flow record format based on NetFlow v9. Devices that can collect traffic data, like routers and switches, will export the summaries of seen flows via NetFlow/IPFIX to a collector, which could either perform a real-time analysis or store those flow records for later use.

Current most popular analysis software like nfdump [10] and flow-tools [9] for identifying, filtering and aggregating flows only allow for very basic inspection of the gathered packets. They allow to filter flows by selecting only those flows that match a single or a set of attributes like source IP, port number or number

of packets, but they are unable to select flows based on relative comparisons between flows. This missing functionality in existing software does not make it fit for a range of analysis tasks. Such tasks include situations in which someone wants to detect portsscans where ports on a host are scanned in incremental fashion. Another use case is the detection of a specific kind of network traffic (like one produced by malware) that can only be accomplished by comparing multiple flows and filter them relative to each other. Or one might simply just want to filter for bidirectional connections where two flows with matching source and destination IP addresses and port numbers belong together. Using existing tools this kind of queries are hard to implement as they do not allow for relative comparisons of different flows but only apply absolute matching to a stream of flows.

Flowy improves on existing analysis tools by using a uniform easy to write syntax for it's filtering language and offers more powerful features in detecting traffic with given properties and complex relations among a given set of flows. While the first implementation in Python [11] was meant as a prototype for the developed query language [12] and focused on completeness and correctness it suffers from a number of performance issues that will be discussed further.

In the following chapter this document will shortly give a description of the two most popular flow analysis software nfdump and flow-tools. Section 3 will contain an overview of the query language used by Flowy. In section 4 will handle the former Python implementation of Flowy and its drawbacks while the section after that will in detail describe how those are addressed in the Flowy rewrite. Section 6 will then show how the changes improved the performance of Flowy by showing some benchmark results while the second to last section gives an outlook on future work that can or should be done on the topic.

2 Existing Tools for Flow Analysis

This section will give an overview over two very popular software pieces for analysis of NetFlow data.

2.1 flow-tools

flow-tools [9] is a very popular suite of 24 applications that work together by connecting them via pipes. The most important utilities are:

- `flow-capture` for capturing NetFlow data exported by a network device and storing it on the local disk in the flow-tools format
- `flow-cat` which reads collected flow records and passes them on to a next tool via *stdout*
- `flow-filter` to filter flows by simple absolute filtering rules
- `flow-report` to create summaries and aggregate and sort the flows passed to it.

Other tools it provides are `flow-export` to convert captured flows into MySQL, PostgreSQL or CSV and `flow-print` to produce human readable output. A usual analysis will concatenate captured datafiles together using `flow-cat`, pass them into `flow-filter` that filters by a filter file given to it and print the result using `flow-report`. The flow-tools suite excel in their ease of use and simplicity of concatenating different commands but it suffers from the relatively simple filter rules and overhead created by using pipes to chain different commands together.

2.2 nfdump

Also being very popular, nfdump [10] works similar to flow-tools but uses a different storage format and a smaller number of separate application but is also working by piping them together. NetFlow data is captured using `nfcapd` and then processed by `nfdump` which can filter as well as display the sorted and filtered result. nfdump's filter format is similar to the tcpdump filter syntax. It suffers from its limited possibilities to aggregate the results and from the filtering options being given as command line options as well as a filter file. The power of its filtering rules is similar to that of flow-tools and as such is mostly limited to absolute comparisons of flow attributes.

3 Query Language

Flow records are processed in a stream oriented approach. The query language [12] allows to assemble such streams and connect them together by pipes as shown in figure 1. A stream consists of filter, grouper, groupfilter, merger and ungrouper components which are described in a query file. The naming of filtering primitives of the query language is closely linked to flow record attributes listed in RFC 5102 [13].

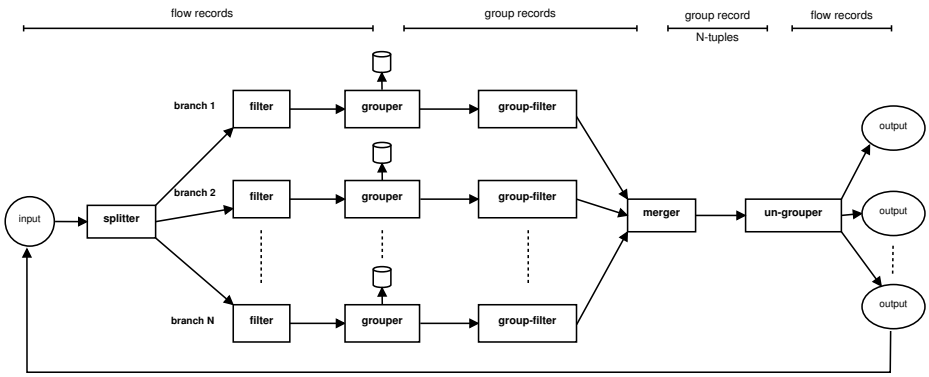

Fig. 1. Flowy processing pipeline from [12]

3.1 Processing Pipeline

The splitter is the most simple stage as it only reads flows from an input trace file and distributes them into the defined branches.

A filter does absolute filtering by only passing along those records that match a given rule. All other records are dropped. Filtering supports ranges and is also able to compare different fields of flow records with each other but it cannot do relative filtering and compare values to other flow records.

In the grouper stage flow records are grouped together by absolute or relative comparisons. That means that in this stage records can be compared with each other. Each grouper may contain several modules which are subgroups. Group membership is achieved by a record matching at least one of a group's subgroups. A record may belong to several subgroups but only belongs to one group. A group can also be associated with aggregated data of the subgroups which might be of use at a later stage. The grouper will create group records and pass those on to the next stage.

The groupfilter filters out groups based on absolute rules over a group's normal or aggregated attributes. It does not compare values between different group records and drops group records not matching the filtering rules.

The merger takes several streams of input group records and based on merging rules merges them into N-tuples of group records where N is the number of input streams. The merger does relative comparison between group attributes and is also able to make timing- and concurrency-based decisions using Allen's time interval algebra [1]. The merger can contain several modules and to form a group tuple, N groups have to meet the requirements of the first module but not the others.

In the ungrouper stage, the received group record tuples are expanded into separate streams of flow records that are output in the order of their timestamps. Each output stream is one result of the specified query and does not contain any duplicate flow records.

3.2 Sample flow query file

Figure 2 shows an example of an flow query file. The splitter does not take any options and is therefore always specified in the way shown in the example above. It is followed by two filter definitions which belong to branches A and B, respectively. The filter in branch A handles the HTTP requests by filtering for the destination port 80 of a flow record while the filter of branch B takes care of HTTP responses by filtering for the source port. The grouper at each branch will group flow records that share the same source and destination IP addresses, and also do not have a start time that exceeds the end time of the previous flow record by more than one second. The aggregation mechanism will attach meta-data information to each group. In the query example from Figure 2, the meta-data will consist of the following information: the source and destination IP, the sum of all bytes the member flow records in that group have, the number of contained flow records, a bitwise OR of the flow records' TCP flags as well

```
1  splitter S {}
2
3  filter www_req {
4          dstport = 80
5  }
6
7  filter www_res {
8          srcport = 80
9  }
10
11 grouper g_www_req {
12         module g1 {
13                 srcip = srcip
14                 dstip = dstip
15                 etime < stime delta 1s
16         }
17         aggregate srcip, dstip, sum(bytes) as bytes, count(rec_id) as n,
18                 bitOR(tcp_flags) as flags, union(srcport) as srcports
19 }
20
21 grouper g_www_res {
22         module g1 {
23                 srcip = srcip dstip = dstip etime < stime delta 1s }
                       aggregate
24 srcip, dstip, sum(bytes) as bytes, count(rec_id) as n, bitOR(tcp_flags)
       as
25 flags, union(dstport) as dstports }
26
27 groupfilter ggf {
28         bitAND(flags, 0x13) = 0x13
29 }
30
31 merger M {
32         module m1 {
33                 branches B, A
34                 A.srcip = B.dstip
35                 A.srcports = B.dstports
36                 A.bytes < B.bytes
37                 B oi A OR B d A
38         }
39         export m1
40 }
41
42 ungrouper U {}
43
44 "./netflow-trace.h5" -> S
45 S branch A -> www_req -> g_www_req -> ggf -> M
46 S branch B -> www_res -> g_www_res -> ggf -> M
47 M->U->"./ungrouped.h5"
```

Fig. 2. Sample flow query file matching HTTP downloads from [11]

as source and destination ports for branches A and B respectively. The group filter takes the formed groups and checks if the groups' TCP flags contain the SYN, ACK and FIN flags as a simple heuristic for them forming a complete TCP stream. The two branches are merged by the merger M, which puts group records into tuples according to the following rule: the source IP address and port number of the request branch A match the destination IP address and port number of the response branch B. Since the request is expected to be smaller than the response there is one rule that compares the overall amount of bytes in both group records. The last requirement comes from Allen's time interval algebra and states that the response should arrive after or during the request. The ungrouper in the end will expand the tuples of group records into individual streams of flow records.

The last few lines assemble the whole pipeline with its two branches. Data is read from "./netflow-trace.h5" and put into the splitter. The splitter then distributes flow records into the two branches A and B where they are filtered, grouped, groupfiltered and merged in the end. The merger then passes the group record tuples into the ungrouper, which outputs the flow records into "./ungrouped.h5".

4 Original Python Implementation

On program execution the flow query file is parsed using the Python PLY [3] module that provides a lexer and parser. After successful execution an instance of that parser class is then passed to all stages of the pipeline that is to be executed. Each of the stages (splitter, filter, grouper, groupfilter, merger, ungrouper) is implemented as Python class. Each of these classes consists of a validator and an execution module. The module that is executed by the main script flowy.py is the validator class that checks the result of the parser for integrity and only then executes the execution module. Each part of the pipeline in addition to the parser instance is also given an instance of the previous pipeline step. Every created branch is handled by a separate thread. Since the grouper passes on group records instead of flow records it will save the flow records it receives into a temporary storage and tag them with the group and subgroup they belong to. The grouped flow records are later on retrieved by the ungrouper. As the splitter's only task is to distribute flow records to the filters, the actual implementation applies the filters first and only then lets the splitter distribute the data into each branch that matches the filter rules. This interchange avoids a repetitive execution of similar filtering rules at each of the branches and also avoids needless copying of flow records from one branch to the next.

4.1 Performance Issues

One of the biggest inhibitors to performance is Python [14], the language Flowy was implemented in. Performance is wasted by the interpreted and weakly typed

nature of Python as well as aspects like: repeatedly creating and destroying dictionaries and lists, looking up and changing items in them, passing them around, repeatedly calling functions, non-local variable lookup, usage of immutable objects, using range instead of xrange and much more, which would go beyond the scope of this section. Most of these problems can be solved by switching from Python to C.

Another big overhead is created using PyTables [2] as it adds complexity where simplicity is needed. Pytables also requires a conversion of the input data prior to its processing, which adds to the overhead.

number of records	overall in s	filter in s (%)	grouper in s (%)	merger in s (%)
103k	1177 s	28 s (2%)	240 s (20%)	909 s (77%)
337k	20875 s	110 s (1%)	2777 s (13%)	17988 s (86%)
656k	70035 s	202 s (0%)	8499 s (12%)	61334 s (87%)
868k	131578 s	274 s (0%)	15913 s (12%)	115391 s (87%)
1161k	234714 s	1212 s (1%)	25480 s (11%)	208022 s (88%)

Table 1. Runtime for http-download.flw broken down to the individual stages

Profiling cases reveal that the bottlenecks are mainly concentrated in the grouper and merger (See table 1). Specifically they lie in the repeatedly called functions (creating function calling overhead) and either contain larger loops where a dictionary is queried or filled or where objects are passed around using deepcopy(). Testing showed that avoiding `deepcopy()` in different code sections did not change execution at all and was just a waste of memory operations. The worst impact however comes from a $O(n^3)$ behavior of the merger in the Python implementation of Flowy. This issue will be discussed in the following.

An obvious pitfall concerns the handling of flow and group records. In the current implementation a flow or group record traverses a branch by having its full data always copied, even though that data never changes. Passing a reference to the original flow or group record inside the branch would suffice. Even worse, complete flow records are copied into a temporary storage by the grouper which, in turn, increases the number of memory operations and the amount of memory used.

5 Flowy Improvements

During the course of the thesis work several improvements to the Python implementation of Flowy have been done. The following section describes those improvements as well as the implementation of the Flowy core algorithm in plain C.

5.1 Unmodified parts

While most of the code had to change to allow for any improvements, it was decided that the PLY based flow query parser and the validators of the flow query will stay the way they were. This is justified by the fact that the execution time those code parts take is negligible compared to the main execution and is also not dependent on the amount of input records and only slightly depends on the complexity of the input query. In those parts Python demonstrates its ability of achieving much with little code while not slowing down the overall processing, as there are no complex computations involved, compared to the core algorithm.

5.2 Early improvements

Several small changes have already been implemented in the process of Flowy's code investigation.

Some changes which already had small impacts on performance were, for example, changing the affinity mask of each of the created threads, so that each of them runs on a separate processor core (if available) instead of all threads running on the single core assigned to the parent process. Another modification concerned a removal of a *try/except* block that enclosed nearly the whole code which was unnecessary, since the except block was only to check for parsing errors of the flow query file.

Functional improvements include the addition of a small test suite. It already contains some sample flow query files and input traces that are given to Flowy by a script that also checks whether the md5sum of Flowy's output matches the expected hash. The suite will be used later when changing big parts of the code to verify that the result is still the same and to detect regressions. A setup.py file was written to facilitate easy installation of Flowy or make it easier for possible packaging for distributions. Since the configuration file options.py did not contain anything that made that file worth to exist as a Python script it was converted into flowy.conf using the usual key/value pair configuration file format that should be easier for users to edit. Command line option were not working in the original Flowy version so the parser was first switched from the deprecated *optparse* module to *argparse* and for every configuration option an argument was added that would override that configuration value. Additionally a profiling switch was added so that profiling could be done without changes in the source. Also, in order to improve batch processing the flow query format was extended to also accept "-" as a filename in which case the file's content was either given by a filename as a command line argument or using *stdin*.

The profiling output is now not given in the old format using brackets to separate values but using tabs as delimiters so that the result of a profiling run can be more easily parsed or displayed by other tools. Additionally, flowy.py and flow_exec.py as well as ft2hdf.py, print_hdf_in_step.py and printhdf.py were merged into one source file each. Finally ambiguous variable names that are otherwise used as Python built-ins, like filter, file, hash, id and map were renamed.

5.3 Cython

```
1  cdef extern from "include/ftreader.h":
2      struct ft_data:
3          int fd
4          ftio io
5          fts3rec_offsets offsets
6          ftver version
7          u_int64 xfield
8          int rec_size
9          char **records
10         int numrecords
11     ft_data *ft_open(char *filename)
12
13 cdef class FtReader:
14     cdef ft_data *data
15
16     def __init__(self, filename):
17         self.data = ft_open(filename)
18
19     def get_numrecords(self):
20         return self.data.numrecords
```

Fig. 3. A Cython snippet from ftreader.pyx

Cython [4] is a language that was written to make the development of C extensions for Python easier. Originally, writing a native extension or library that Python code could interface with was a very verbose process. The encapsulation of methods and objects in datatypes defined in python.h, required repeated usage of very similar code patterns for each interface. Cython allows to automate that process and even more so, develop a C extension in Python syntax.

At the most basic level, Cython code has the exact same syntax as Python and is even py3k compatible. There exist only some additional foreign constructs to allow for declaration of static types or tighter loops. There exist only few Python constructs like lambda statements, nested functions and generators (all requiring closures) that are not yet available in mainline Cython but are being developed in the Cython-closures branch [5]. Cython will translate its Python-like code into a normal C source file that can be compiled using gcc. Therefore, Cython is not only a language that is mostly compatible to Python syntax, but also a source code compiler. The C file that was generated from Cython code will be in a readable format containing comments that indicate what line of the Cython code was translated into what piece of C code. Naively compiling normal Python code with Cython will not cause any speed improvements yet, as the compiled code will still use Python objects in their dynamically-typed way, and hence will still suffer from their additional computation overhead.

A significant performance improvement can be achieved by making use of enhancing Cython specific syntax elements. These syntax elements allows to assign types to variables, call functions from C libraries by including a C header or use C constructs like structs and pointers. When Cython compiles a Cython

source file into C, sections containing those native C constructs will mostly be translated one-to-one into corresponding C code without making use of Python objects and thus avoid the connected overhead. Figure 3 shows a snippet from Flowy's Cython code, depicting the Python-like syntax and class definitions combined with static types from C, the inclusion of a C header and calling of C library functions.

In the case of Flowy, Cython was used to bridge between C libraries that were written and the surrounding Python code. In particular, a library was written that would provide a "pythonic", object oriented interface to C libraries. Those libraries provide support for parsing flow-tools traces, retrieving records or certain attributes and writing out flow-tools compatible records. With this Cython module, wrapping a flow record storage library written in C, it is possible to retrieve flow records by an id. Hence, during processing, only the id has to be passed around instead of the full flow record.

5.4 Data Format

The original Flowy implementation [11] uses PyTables to store and process flow records. PyTables is a library written to manage huge amounts of hierachically structured data. It does so by using HDF [8] as the storage format for which it provides various indexes for fast data retrieval. PyTables is a feature-rich and well-supported library but its complexity is unsuited for usage in Flowy. The advantages of using PyTables for data management come from its ease of use for storage and retrieval of data in an efficient manner. The downside is that existing data which will mostly exist in either the nfdump or flow-tools format has to be converted in to the HDF file format prior to execution of Flowy resulting in additional overhead. Another downside is that PyTables exhibits a "more-than-needed" complexity. That is, a replacement of a general purpose data storage solution like PyTables with a more specific custom solution, adjusted for Flowy, will certainly lead to performance improvements, since one can optimize it for the special usage patterns of Flowy. In addition a custom solution could be implemented in a way that nfdump or flow-tools data can be accessed directly without conversion to a data format specific to Flowy which removes the overhead of a needed conversion and also does not add yet another proprietary data storage format the user has to handle.

In contrast to flow-tools and nfdump, Flowy in most cases does not access data in a linear fashion. This is because flow-tools and nfdump only apply absolute filtering to each flow record they are fed while Flowy needs each flow record to be randomly accessibly to do relative filtering.

To address those issues a C library was written that implemented random access to records in flow-tools traces. The functionality is currently limited to flow-tools only as flow-tools provide an easy to use library to access flow-tools data from third party programs whereas no such thing exists for nfdump. But support for more flow trace formats can be included afterward. As the flow-tools library only allows for sequential access to records of a trace, the custom library that was written sequentially reads flow records into memory where they are

then randomly accessible. Like in flow-tools, storage is done via `char` arrays. This avoids having to manage 20 different kinds of NetFlow formats supported by flow-tools and also avoids wasting memory by putting record fields into a structure allowing for storage of all 33 possible fields. To access data from that array the offset of each field is stored into another `struct`. Since the length of each field is static it is globally stored as `#include`s. Each record is identified by an id. The id is simply their index in the array of records. This allows for retrieval of records in $O(1)$ time. The library allows for retrieval of a full records or certain attributes only. The latter functionality is heavily used by the comparisons in Flowy's filtering stages.

As a result of the replacement of PyTables with the new parser, big parts of the code where not needed anymore and other parts had to be rewritten.

5.5 Rewrite of Core Algorithms in C

During the removal of PyTables from the Python code it became apparent that, to significantly speedup Flowy's execution time, it would not be enough to replace each stage of the filtering pipeline with Cython or C by itself but the whole core of Flowy's record filtering had to be rewritten. Trying to tackle different parts separately would have introduced unnecessary complexity in handing data structures from one stage over to the next as well as unnecessary performance loss during these phases. Implementing the whole processing pipeline in C was the best way to assure that the process gets optimized as much as possible.

The rewrite that was done has certain limitations but is already powerful enough to process a query as displayed in figure 2. Queries are not yet possible to be read from a flow query file but are done by filling `struct`s with content that are currently hardcoded. Nevertheless they are only hardcoded in the sense that their content can also be programatically modified. Once this is implemented in the Python Flowy implementation, the Flowy Python wrapper would (after parsing validating the flow query file) create a Cython class with proper arguments which would in turn supply the core C implementation with properly filled `struct`s.

An example for those `struct`s can bee seen in figure 4 and 5 which represent branch A and the merger of the query shown in figure 2. As already explained above, this part is still hardcoded but as one can also see, it is easy to create those `struct` arrays on runtime to resemble any other query given by a parsed flow query file.

The overall idea is, that each processing stage is given an array of `struct`s that describe the filtering being done in that stage - be it absolute or relative filtering. For that, the `struct` arrays contain data about the field to be compared (field offset and length) and the value to compare it to. This can be another field for the grouper operation or a static value for the filter and groupfilter stages. Additionally every comparison operation takes a delta value to allow for checks with an epsilon margin. The last entry of each of the `struct`s is a function pointer to the operation that is to be carried out. These functions take the record ids to be compared and the fields, offsets and delta specified in the

```
1  struct absolute_filter_rule filter_rules_branchA[2] = {
2      { data->offsets.dstport, LEN_DSTPORT, 80, 0, filter_equal },
3      { 0, 0, 0, 0, NULL }
4  };
5
6  struct relative_group_filter_rule group_module_branchA[4] = {
7      { data->offsets.srcaddr, LEN_SRCADDR, data->offsets.srcaddr,
           LEN_SRCADDR, 0, gfilter_rel_equal },
8      { data->offsets.dstaddr, LEN_DSTADDR, data->offsets.dstaddr,
           LEN_DSTADDR, 0, gfilter_rel_equal },
9      { data->offsets.Last, LEN_LAST, data->offsets.First, LEN_FIRST, 1,
           gfilter_rel_lessthan },
10     { 0, 0, 0, 0, 0, NULL }
11 };
12
13 struct grouper_aggr group_aggr_branchA[6] = {
14     { data->offsets.srcaddr, LEN_SRCADDR, aggr_static },
15     { data->offsets.dstaddr, LEN_DSTADDR, aggr_static },
16     { data->offsets.dOctets, LEN_DOCTETS, aggr_sum },
17     { data->offsets.tcp_flags, LEN_TCP_FLAGS, aggr_or },
18     { data->offsets.dstport, LEN_DSTPORT, aggr_union },
19     { 0, 0, NULL }
20 };
21
22 struct absolute_group_filter_rule gfilter_branchA[2] = {
23     { 3, 0x13, 0x13, gfilter_and},
24     { 0, 0, 0, NULL }
25 };
```

Fig. 4. Sample C `struct` arrays, filled with content representing branch A of the flow query as shown in figure 2

```
1  struct merger_filter_rule mfilter[4] = {
2      { 0, 0, 1, 1, 0, mfilter_equal },
3      { 0, 4, 1, 4, 0, mfilter_list_equal }
4      { 0, 2, 1, 2, 0, mfilter_lessthan },
5      { 0, 0, 0, 0, 0, NULL }
6  };
```

Fig. 5. Sample C `struct` array, filled with content representing the merger of the flow query as shown in figure 2

`structs` as their arguments. The comparison functions retrieve the fields from the records according to their offset and field length and return the result of the requested comparison. Using function pointers as arguments avoids to check for the operation to be carried out in a `switch` statement on each comparison.

The group aggregation specifications of line 13-20 in figure 4 are similar but specify the field offset and according field length to be aggregated together with a function pointer to an aggregation operation. The aggregation function will then either assign the value of the first record (aggr_static) or traverse all records in that group and return results that are a sum, count, union or intersection of those.

The merger rules that are depicted in figure 5 show relative comparison rules of groups from each branch. The integers depict the branch the aggregated value comes from, the aggregated value index in order as they are specified in the aggregation rules of figure 4, a delta and a function pointer for the comparison operation to be carried out.

The implementation of the filtering stages is then straight forward. The filter and groupfilter just traverse through all flow records or flow groups respectively and check each of them for compliance with the supplied filtering rule by an appropriate call to the function the supplied function pointer points to. The grouper and merger on the other hand operate in $O(n^2)$ and $(O(n^k))$ respectively, where k is the number of branches, so $k = 2$ in this case by just comparing every flow record with any other or every group with any other, respectively. This is optimized in the way that a record can not be part of more than one group and if it is, it is not checked again in subsequent checks.

What can currently not be specified on the fly and is really hardcoded is the arrangements of the pipelines. While it is possible to create as many branches as necessary it is not possible to skip the filter or groupfilter stages yet. A fact connected to that is, that in contrast to the Python implementation, the C implementation strictly adheres to the overview given in figure 1 whereas the Python implementation was switching the splitter and filter around for performance issues. Since the C implementation passes flow record pointers around, this is no longer a bottleneck.

The C implementation not yet complete though. While it can already handle simple queries as in Figure 2, it can not yet execute queries with more than one module in the grouper or merger and can also not yet aggregate fields with the union or intersect operation since those would require the storage of arrays of integers instead of a single integer for all the other aggregation operations.

Despite these facts the current implementation seems to work correctly as the correct output was generated when testing it with on a set of records and a given query to produce an expected (known) output.

6 Benchmarks

A comparison of Python Flowy to the current C core of Flowy is very difficult and can not be done as accurately as one would like to. This is due to many

number of records	runtime old Python Flowy in s	runtime new C Flowy in s
103k	1177	0.3
337k	20875	3.4
656k	70035	13
868k	131578	23
1161k	234714 (2.7 days)	86

Table 2. runtime for http-download.flw

reasons. Firstly, the reimplementation of the Flowy core is only the core without the additional surroundings that make the Python Flowy implementation usable. Secondly the C implementation still lacks many features and thus only limited comparison is possible.

To benchmark two implementations, the flow query given in figure 2 was taken and the two things the C flowy implementation can not yet do were removed from it: union aggregation and the OR in the merger. Then the query was put into the C code, which basically equals what can be seen in figures 4 and 5. I ran the query on a trace that was taken from normal laptop usage over the course of 14 days which make up to about 1.1 million records. To find out each of the implementations dependency on input record number, I created four additional traces, each being a smaller subset of the original.

Due to the extensive runtime, the benchmarks with Python Flowy were only done once while for the C implementation there were done six runs where the first one was discarded due to possible influences of caching and the result of the last five was averaged.

As one can see in table 2 the values differ tremendously. The values achieved for the C implementation are not very surprising. If one considers that from the 1.1 million records, after the filtering only 150.000 are left and that the grouper operates in less than quadratic complexity because it does not check records that already belong to a group then this boils down to only a few million comparisons in the grouper stage and even fewer in the merger as the groups are also filtered. And doing a few million comparisons in C does not take much time. Most of the time is probably spend with `malloc` and `memcpy` when the datastructures are set up.

On the other hand the Python implementation is calculating for almost 3 days on the 1.1 million record set. While it is not the purpose of this paper to explain why that is, some parts of the code give the impression that the merger, that takes the biggest chunk of runtime is actually having a complexity of $O(n^3)$ which probably a bug (at least it could not be made out why the certain piece of code that leads to $O(n^3)$ would be necessary) but the issue was not investigated further since the code is to be replaced anyways.

The important fact is, that a query of how it is required by the Flowy idea can be implemented in a way such that the runtime is finally in an area that is manageable - even though the complexity is still nearly quadratic. It could be

argued that some parts of the original http-download.flw query were missing but it is highly unlikely that those additional filter conditions would lead to runtimes of a higher order of magnitude.

```
1  struct absolute_filter_rule filter_rules_branchA[2] = { {
2  data->offsets.dstport, LEN_DSTPORT, 80, 0, filter_equal }, { 0, 0, 0, 0,
       NULL }
3  };
```

Fig. 6. Simple filter for records with destination port 80

```
1  time sh -c "flow-cat traces | flow-filter -P80 > /dev/null"
2  time sh -c "flow-cat traces | ./flowy > /dev/null"
```

Fig. 7. Commands for benchmarking Flowy against flow-filter

Another requirement was to make Flowy's performance comparable to the one of other flow record filter tools. For this purpose, the C Flowy implementation was slightly rewritten to be reduced to the set of functionality flow-tools offers, which is absolute comparison of Flow records only. The filter setup of Flowy can be seen in figure 6. As above, for each of the benchmarks a first trial run was done and then the commands given in figure 7 where executed five times. The flow records used were from the same 1.1 million trace as used before. As expected Flowy was able to execute the query in 1.5 s on average which was only slightly more than the flow-filter average of 1.4 s. A slightly higher runtime of Flowy was expected since it wasts some time with allocating memory for all input records. This is not necessary for sequential traversal as done by flow-filter but essential for the random lookups of flow record attributes a normal Flowy query would require. A downside of using Flowy for such simple queries is of course that Flowy can only manage as many flow records as they fit into memory. To support bigger dumps it would be necessary to implement on disk caching (for example by using `mmap`).

7 Outlook

Despite the work already accomplished there is not only still much to be done, but also many ways to drastically improve Flowy's performance by even more orders of magnitude.

7.1 Work to be done to make Flowy fully functional again

As explained above, despite lots of work already being done on the python and C code, the remaining bit is to integrate those two. The python code will serve

to parse and validate a flow query file and build the datastructures that are then handed over to the C code that does all the hard work. The core will then give the filtered records back to the python wrapper that cares about their presentation to the user.

7.2 Further Improvements to Flowy's Core Algorithm

Since benchmarking was so far only done with up to 1.1 million flow records and only two branches in the query, the current complexity of the parsing algorithm of $O(n^2)$ for the grouper and $O(n^k)$ (where k is the number of branches, so $k = 2$) for the merger still does not result in unmanageable running times. But with more records and with queries involving more than two branches or with more than one module for the grouper and merger this complexity will have too big of an impact on the runtime.

Search with Trees To improve the situation one should build a search tree over the data that is to be traversed. Building the tree will be have a complexity of $O(n \log(n))$ and lookups will have a complexity of $O(\log(n))$. B+trees are especially suited for the desired purpose as they still allow to traverse the indexed data sequentially. This way it will not only be possible to retrieve records for a specific key but also continue traversing records starting with that key in ascending or descending order. Such trees have to be build for the grouper and merger execution. The filter and groupfilter both traverse data linearly and hence cannot be better than $O(n)$. As an example, before executing the grouper, Flowy would check what fields will be compared. For each field type that is to be compared, Flowy would then create an array of record pointers where the pointers are ordered by that field and also the B-tree with pointers to the proper records in the leafs. Suppose a grouping module that checks for equality of source address of one record to the destination address address of another record. The grouper would now traverse the records ($O(n)$) and for each record find records that are of the same destination address ($O(\log(n))$). Hence the whole grouping operation would be operation of $O(n \log(n))$. There have been done preliminary tests with that approach but as those did not get further than the example that was just explained the work is not yet fully tested or included here.

Reducing Amount of Work in Innermost Loops by Specialized Functions The individual fields of flow records are having different sizes: `unsigned char`, `unsigned short` and `unsigned int`. To save as much memory as possible for storing the flow records in memory, every flow records is stored as a `char` array of the exact required size. Access to the data is given by the offset of the data and knowing the size of a field. As comparisons in filters and modules are specified with function pointers, every comparison operation operation does not only take the fields that are to be compared as an argument but also their length and their offsets in a flow record. Thus, for every comparison operation there is currently a check for the length of the input fields to properly

cast the values from the character arrays. This check could be avoided if for every combination of flow records and for every operation there was one function taking care of explicitly that operation. This way one would not call the `filter_rel_equal` function, passing as arguments the flow records array, the records to be compared and their offsets and their lengths but a function like `filter_rel_equal_srcaddr_dstaddr`. Each such function would not need to do any additional checks for input field lengths and would be reduced to the bare comparison operation. Since such comparisons are the innermost loop element of the filter, grouper, groupfilter and merger their optimization might yield significant performance increases. The only downside to that would be, that to accommodate for all combinations of comparison functions and input values a total of 20691 functions would be needed. It would be 19 operations (6 integer comparisons, 13 Allen's interval operations) times 33 possible flow record fields squared when relative comparison is done in the group filter. For an overview of all possible comparison operations, see figure 3. What sounds unmanageable in numbers could be managed by just auto generating the C code for all the required functions before compilation. Python code that does that, is already written for testing purposes and shows that the overhead of memory consumption to store all of them would be just about 3.0 MB which is thought to be small enough to be worth further investigations.

symbol	operation name
=	equal as integer comparison
!=	not equal
¡	less than as integer comparison
¿	greater than as integer comparison
¡=	less than or equal
¿=	greater than or equal
=	equal as Allen's interval relation
¡	before
¿	after
m	meets
mi	meets inverse
o	overlaps
oi	overlaps inverse
s	starts
si	starts inverse
d	during
di	during inverse
f	finishes
fi	finishes inverse

Table 3. Integer comparisons and Allen's interval relations. Integer comparisons are done between individual fields of records, Allen comparisons are done between start and end times of two records.

More Efficient Multithreading Currently, multithreading is limited to exactly one thread for each branch. This means that only the filter, grouper and groupfilter are running on more than one core and also that the number of threads does depend on the query being executed. It would make more sense to auto-detect the amount of available cores and also have a customizable configuration option that distributes work to an appropriate number of threads. This should be done for tasks that are currently single threaded (merger, ungrouper) but also for the individual branches if enough cores are available. For example, executing a query using two branches on a quad core would potentially be more efficient if each branch could make use of two of the cores and if the merger would not run on only one core but on all four of them.

Pitfalls of this approach would certainly be the much increased complexity of sharing memory between different threads. The current multithreaded implementation does not have such issues as the data used by both branch threads, the records are not being changed but only read by the threads. Additionally the result of both threads is only used once both are joined. In a setup as described above, one would have to distribute tasks over a number of N threads for the filter, grouper and groupfilter and find an efficient algorithm that does the same for the merger. As a very simple approach to solve that problem a small pthread library was written that for N threads distributes every Nth task to each of the dynamically created thread pools such that each thread handles $\frac{1}{N}$th of the total amount of tasks. This is assuming that such a simple splitting of the work will still lead to all threads taking approximately the same amount of time to finish their tasks. As work on that was also not extended further than the implementation of the aformended library it also did not become part of this thesis. Due to octo core consumer processors this topic remains worthwhile for future investigation.

More Functionality Additional functionality that is currently missing is the support for nfdump based flow traces, more comparison operations (like the << and >> operators: "much less than" and "much more than"), an intersect aggregation operation as the companion of the union operation and the possibility to specify all filters in conjunctive normal form. This would mean that for all filters or modules one would give a list of comparison criteria which would all be AND'ed but one could also always use the OR keyword in a line.

8 Conclusion

Flowy was suffering from great slowness due to performance critical parts being implemented in Python. The improvements being done on the Python code and the rewrite of Flowy's core in C though show, that tremendous improvements can be achieved by just using C and more efficient data structures and memory.

The Python Flowy implementation is still more powerful than the C implementation but adding the missing parts to the C code is not a problem and only a matter of time.

It seems, that in contrast to the day-long running times that made the Python Flowy experience very disappointing, it is possible to combine a powerful query language with fast execution time.

References

1. J. F. Allen. Maintaining Knowledge About Temporal Intervals. *Communications of the ACM*, 26(11):832–843, 1983.
2. F. Alted, I. Vilata, et al. PyTables: Hierarchical datasets in Python. *http://www.pytables.org/*, 2002–.
3. D. M. Beazley. Ply (python lex-yacc). *http://www.dabeaz.com/ply/*, 2001–.
4. S. Behnel, R. Bradshaw, and D. S. Seljebotn. cython. *http://cython.org/*, 2008–.
5. S. Behnel, R. Bradshaw, and D. S. Seljebotn. cython closures branch. *http://hg.cython.org/cython-closures/*, 2008–.
6. B. Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954, Cisco Systems, Oct. 2004.
7. B. Claise. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information. RFC 5101, Cisco Systems, Jan. 2008.
8. M. Folk, R. E. McGrath, and K. Yang. Mapping HDF4 Objects to HDF5 Objects. Technical report, National center for supercomputing applications, University of Illinois, 2002.
9. M. Fullmer. flow-tools. *http://www.splintered.net/sw/flow-tools/*, 2001–.
10. P. Haag. nfdump. *http://nfdump.sourceforge.net/*, 2004–.
11. K. Kanev, N. Melnikov, and J. Schönwälder. Implementation of a stream-based ip flow record query language. In *AIMS*, pages 147–158, 2010.
12. V. Marinov and J. Schönwälder. Design of a Stream-Based IP Flow Record Query Language. In *DSOM '09*, pages 15–28, Berlin, Heidelberg, 2009. Springer-Verlag.
13. J. Quittek, B. Claise, P. Aitken, and J. Meyer. Information Model for IP Flow Information Export. RFC 5102, Cisco Systems, Jan. 2008.
14. G. V. Rossum. *The Python Language Reference Manual*. Network Theory Ltd., Sept. 2003.