F (ν2)

A complete system integration of stream-based IP flow-record querier

Vaibhav Bajpai

Masters Thesis

School of Engineering and Science
Jacobs University Bremen
Bremen, Germany

June 2012

[January 20, 2012 at 18:12]

ABSTRACT

Short summary of the contents in English...

CONTENTS

I INTRODUCTION 1

1 TRAFFIC MEASUREMENT APPROACHES 3
- 1.1 Capturing Packets . . . 3
- 1.2 Capturing Flows . . . 3
- 1.3 Remote Monitoring . . . 3
- 1.4 Remote Metering . . . 3

2 FLOW EXPORT PROTOCOLS 5
- 2.1 NetFlow . . . 5
- 2.2 IPFIX . . . 5
- 2.3 sFlow . . . 5

3 LANGUAGES AND TOOLS 7
- 3.1 SQL-based Query Languages . . . 7
  - 3.1.1 NetFlow exports as relational DBMS . . . 7
  - 3.1.2 Data Stream Management System . . . 7
  - 3.1.3 Gigascope . . . 7
  - 3.1.4 Tribeca . . . 7
- 3.2 Filtering Languages . . . 7
  - 3.2.1 flow-tools . . . 7
  - 3.2.2 nfdump . . . 7
- 3.3 Procedural Languages . . . 7
  - 3.3.1 FlowScan . . . 7
  - 3.3.2 Clustering NetFlow Exports . . . 7
  - 3.3.3 SiLK Analysis Suite . . . 7

II STATE OF THE ART 9

4 FLOWY 11
- 4.1 Python Framework . . . 11
  - 4.1.1 PyTables and PLY . . . 11
  - 4.1.2 Records . . . 11
  - 4.1.3 Parsers and Statements . . . 12
- 4.2 Processing Pipeline . . . 12
  - 4.2.1 Splitter . . . 12
  - 4.2.2 Filter . . . 13
  - 4.2.3 Grouper . . . 14
  - 4.2.4 Group-Filter . . . 14
  - 4.2.5 Merger . . . 15
  - 4.2.6 Ungrouper . . . 16
- 4.3 Future Outlook . . . 16
  - 4.3.1 Reduced Copying . . . 16
  - 4.3.2 Using PyTables in-kernel searches . . . 17
  - 4.3.3 Multithreaded Merger . . . 17

5 FLOWY IMPROVEMENTS USING MAP/REDUCE 19

5.1 Map/Reduce Frameworks . . . 19
5.1.1 Apache Hadoop . . . 19
5.1.2 The Disco Project . . . 19
5.2 Parallelizing Flowy . . . 20
5.2.1 Slicing Inputs . . . 20
5.2.2 Flowy as a Map Function . . . 22
6 FLOWY 2.0 23
6.1 Performance Issues . . . 23
6.2 Flowy Improvements . . . 24
6.2.1 Early Improvements . . . 24
6.2.2 Data Format . . . 24
6.2.3 Rewrite of Core Algorithms in C . . . 25
6.3 Benchmarks . . . 26
6.4 Future Outlook . . . 26
6.4.1 System Integration . . . 27
6.4.2 Searching with Trees . . . 27
6.4.3 Specialized Functions in Inner Loops . . . 27
6.4.4 Efficient Multithreading . . . 27
6.4.5 Additional Functionality . . . 28
7 F 29
7.1 Rule Interfaces . . . 29
7.2 Flowy 2.0 Improvements . . . 31
7.2.1 Efficient Rule Processing . . . 31
7.2.2 Divide and Conquer for Fast Relative Comparisons 32
7.3 Benchmarks . . . 32
8 FLOWY: APPLICATIONS 33
8.1 Application Identification using Flow Signatures . . . 33
8.2 Cybermetrics: User Identification . . . 35
8.3 IPv6 Transition Failure Identification . . . 37
8.4 OpenFlow . . . 39
8.5 Flow Level Spam Detection . . . 40

III MOTIVATION 43

IV WORK PLAN 45

V IMPLEMENTATION AND EVALUATION 47
9 DESIGN 49
10 IMPLEMENTATION 51
11 PERFORMANCE EVALUATION 53
12 FUTURE WORK 55
13 CONCLUSION 57

VI APPENDIX 59
A APPENDIX 61

BIBLIOGRAPHY 62

LIST OF FIGURES

Figure 1 Flowy: Processing Pipeline [1] . . . 12
Figure 2 Parallelizing Flowy using Map/Reduce [2] . . . 20
Figure 3 Slice Boundaries Aware Flowy [2] . . . 21
Figure 4 Flowy: Redundant Groups [2] . . . 21
Figure 5 Cybermetrics: Overview [3] . . . 35
Figure 6 Geographical Preferences [3] . . . 35
Figure 7 Daily Distributions for HTTP Traffic [3] . . . 36
Figure 8 Cross Correlation of Traces with Varying Times [3] 36
Figure 9 NAT64 Setup [4] . . . 37
Figure 10 OpenFlow Architecture [5] . . . 39
Figure 11 Spam Flow Classifier [6] . . . 41

LIST OF TABLES

Table 1 Runtime Breakup of Individual Stages [7] . . . 23
Table 2 Flowy vs Flowy2 [7] . . . 26
Table 3 Application Flow Signatures: Results [8] . . . 34
Table 4 Features in Spam Flow [6] . . . 40

LISTINGS

Listing 1 Filter Rule Struct [7] . . . 25
Listing 2 Merger Rule Struct [7] . . . 25
Listing 3 Flowy2 vs `flow-tools` [7] . . . 26
Listing 4 Flow Query Struct [9] . . . 29
Listing 5 Branch Info Struct [9] . . . 29
Listing 6 Grouper Struct [8] . . . 30
Listing 7 Group Struct [8] . . . 30
Listing 8 Grouper Aggregation Struct [9] . . . 30
Listing 9 Auto Generated Comparison Functions [9] . . . 31
Listing 10 Auto Generated Switch Statement [9] . . . 31
Listing 11 Queries to Benchmark F [9] . . . 32
Listing 12 Skype Application Signature [8] . . . 33
Listing 13 Branch A [8] . . . 34

Listing 14 Branch B [8] 34
Listing 15 Branch A [4] 37
Listing 16 Branch B-C-D [4] 37
Listing 17 Skype Failure Signature [4] 38

ACRONYMS

IPFIX Internet Protocol Flow Information Export

HDF Hierarchical Data Format

LALR Look-Ahead LR Parser

PLY Python Lex-Yacc

HDFS Hadoop Distributed File System

API Application Programming Interface

CNF Conjunctive Normal Form

SSDP Simple Service Discovery Protocol

IP Internet Protocol

UDP User Datagram Protocol

TCP Transmission Control Protocol

NAT-PMP Network Address Translation Port Mapping Protocol

ccTLD Country Code Top-Level Domain

HTTP Hypertext Transfer Protocol

IaaS Infrastructure as a Service

NaaS Network as a Service

vLANs Virtual Local Area Networks

ACLs Access Control Lists

RTT Round Trip Time

SVMs Support Vector Machines

FF Greedy Forward Fitting

SMTP Simple Mail Transfer Protocol

DDoS Distributed Denial of Service

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

Part I

INTRODUCTION

You can put some informational part preamble text here

1 TRAFFIC MEASUREMENT APPROACHES

1.1 CAPTURING PACKETS

1.2 CAPTURING FLOWS

1.3 REMOTE MONITORING

1.4 REMOTE METERING

2 FLOW EXPORT PROTOCOLS

2.1 NETFLOW

2.2 IPFIX

2.3 SFLOW

3 LANGUAGES AND TOOLS

3.1 SQL-BASED QUERY LANGUAGES

3.1.1 *NetFlow exports as relational DBMS*

3.1.2 *Data Stream Management System*

3.1.3 *Gigascope*

3.1.4 *Tribeca*

3.2 FILTERING LANGUAGES

3.2.1 *flow-tools*

3.2.2 *nfdump*

3.3 PROCEDURAL LANGUAGES

3.3.1 *FlowScan*

3.3.2 *Clustering NetFlow Exports*

3.3.3 *SiLK Analysis Suite*

Part II

STATE OF THE ART

The semantics and implementation of our in-house stream-based flow-record querier has underwent significant changes in the last few years. This section is dedicated to perform an inside-out study of the querier, examining all its major (and minor) changes to allow us to better make a pragmatic stand towards its overall packaging and improvement. The organization of the section is described below.

In chapter 4 we look into the structure of the flow query language by discussing each stage of the processing pipeline with their implementation details. The basic structures of the framework that underpin the implementation are also discussed. In the end, we ponder over the current prototype limitation and its suggestive improvements.

In chapter 5 we investigate the possibility of making Flowy Map/Reduce aware. The chapter starts off with a discussion of current Map/Reduce frameworks and looks into the ways to help parallelize Flowy.

In chapter 6 and 7 we look into the first attempt to make Flowy comparable with the state-of-the-art flow-analysis tools. After drilling down the performance hit sections of the code, we witness how getting away with PyTables and rewriting the complete core implementation in C helped make the tool eventually usable. We end by examining the recommended approach to glue the two implementations together to bring the best of both worlds.

We conclude this discussion in chapter 8 by introducing a number of real-life application scenarios where Flowy has proved useful. We also looked into a few current bleeding edge research projects where we believe Flowy could play a vital role in the near future.

4 FLOWY

Flowy [10, 11] is the first prototype implementation of a stream-based flow record query language [12, 1, 13]. The query language allows to describe patterns in flow-records in a declarative and orthogonal fashion, making it easy to read and flexible enough to describe complex relationships among a given set of flows.

4.1 PYTHON FRAMEWORK

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

4.1.1 *PyTables and PLY*

Flowy uses PyTables [14] to store the flow-records. PyTables is built on top of the Hierarchical Data Format (HDF) library and can exploit the hierarchical nature of the flow-records to efficiently handle large amounts of flow data. The `pytables` module provides methods to read/write to PyTables files. The `FlowRecordsTable` class instance within the module exposes an iterator interface over the records stored in the HDF file. The `GroupsExpander` class instance within the same module on the other hand exposes an iterator interface over the group records and facilitates ungrouping to flow records.

In addition, Flowy uses Python Lex-Yacc (PLY) for generating a Look-Ahead LR Parser (LALR) parser and providing extensive input validation, error reporting and validation on the execution modules.

4.1.2 *Records*

Flow-records are the principal unit of data exchange throughout Flowy's processing pipeline. The prototype implementation allows the `Record` class (defined in the `record` module) to be dynamically generated using `get_record_class(...)` allowing future implementations to easily plug in support for Internet Protocol Flow Information Export (IPFIX) or even newer versions of NetFlow [15] exports. The `FlowToolsReader` class instance (defined in `ftreader` module) provides an iterator over the records defined in `flow-tools` format.

This can be plugged into the `RecordReader` class instance (defined in `record` module) to instantly get `Record` class instances.

4.1.3 *Parsers and Statements*

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

4.2 PROCESSING PIPELINE

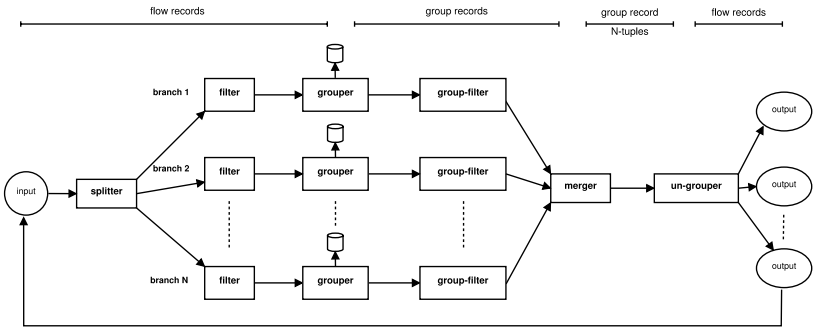

Figure 1: Flowy: Processing Pipeline [1]

The pipeline consists of a number of independent processing elements that are connected to one another using UNIX-based pipes. Each element receives the content from the previous pipe, performs an operation and pushes it to the next element in the pipeline. Figure 1 shows an overview of the processing pipeline. The flow record attributes used in this pipeline exactly correlate with the attributes defines in the IPFIX Information Model specified in RFC 5102 [16]. A complete description on the semantics of each element in the pipeline can be found in [12]

4.2.1 *Splitter*

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

4.2.1.1 *Splitter Implementation*

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

4.2.1.2 *Splitter Validator*

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

4.2.2 *Filter*

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

4.2.2.1 *Filter Implementation*

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

4.2.2.2 *Filter Validator*

The `filter_validator` module handles the filter processing stage. The `FilterValidator` class within the module uses the `Parser` instance passed to it to create a `Filter` instance once the check on semantical constraints have passed. The constraints involve checking whether

records fields referenced in the filter definition exist, whether filters references in composite filter definitions exist and whether duplicate filter definitions are defined.

4.2.3 *Grouper*

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

4.2.3.1 *Grouper Implementation*

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

4.2.3.2 *Grouper Validator*

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

4.2.4 *Group-Filter*

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

4.2.4.1 *Group-Filter Implementation*

The `groupfilter` module handles the filtering of group-records. The `GroupFilter` class within the module iterates over the flow-records within the group and applies filtering rules across them. The filtering rules reuse the `Rule` class from the `filter` module. The flow-records are then added to the time index and stored in a pytables file for further processing. For groups that do *not* have a group-filter defined for them, run through a `AcceptGroupFilter` class instance.

The `timeindex` module handles the mapping of the time intervals to the flow-records. The time index is used by the merger stage to learn about the records that satisfy the Allen relations. The `add(...)` method in the `TimeIndex` class is used to add new records to the time index. The `get_interval_records(...)` method on the other hand is used to retrieve records within a particular time interval.

4.2.4.2 *Group-Filter Validator*

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

4.2.5 *Merger*

The merger performs relative filtering on the N-tuples of groups formed from the N stream of groups passed on from the group-filter as input. The merger rule module consists of a number of a submodules, where the output of the merger is the set difference of the output of the first submodule with the union of the output of the rest of the submodules. The relative filtering on the groups are applied to express timing and concurrency constraints using Allen interval algebra [17]

4.2.5.1 *Merger Implementation*

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

impose restrictions on the possible records. The `MergerBranch` on the other hand represents an ordinary branch loop with rules.

4.2.5.2 *Merger Validator*

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

4.2.6 *Ungrouper*

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

4.2.6.1 *Ungrouper Implementation*

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

4.2.6.2 *Ungrouper Validator*

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

4.3 FUTURE OUTLOOK

4.3.1 *Reduced Copying*

The `reset(...)` method of the `BranchMask` class performs a deepcopy on objects which significantly lowers performance. The invocation of this method can be inhibited by either removing the branch mask mechanism for simpler queries or removing it entirely. In addition

avoiding usage of immutable containers (tuples) can also reduce internal copying during mutation.

4.3.2 *Using PyTables in-kernel searches*

PyTables can accelerate flow-records selection using a `where` iterator. The `where` clause is passed to the PyTables kernel which is written in C, therefore the selection can occur at C speed and only the filtered flow-records reach the Python space. This would require PyTables in-kernel search query support in the filtering rules and the `pytables` module would have to be extended to read from PyTables filtered flow-records.

4.3.3 *Multithreaded Merger*

The merger stage in the processing pipeline is currently the most computation intensive operation and is unfortunately single-threaded. As suggested in [11] it should be possible to handle the outermost branch loop using multiple threads in a non-blocking fashion to improve performance.

5 FLOWY IMPROVEMENTS USING MAP/REDUCE

Flowy, although clearly setting itself apart with its additional functionality to query intricate patterns in the flows demonstrates relatively high execution times when compared to contemporary flow-processing tools. A recent study [2] revealed that a sample query run on small record set (around 250MB) took 19 minutes on Flowy as compared to 45 seconds on `flow-tools`. It, therefore is imperative that the application will benefit from distributed and parallel processing. To this end, recent efforts were made to investigate possibility of making Flowy Map/Reduce aware [2]

5.1 MAP/REDUCE FRAMEWORKS

Map/Reduce is a programming model for processing large data sets by automatically parallelizing the computation across large-scale clusters of machines [18]. It defines an abstraction scheme where the users specify the computation in terms of a `map` and `reduce` function and the underlying systems hides away the intricate details of parallelization, fault tolerance, data distribution and load balancing behind an Application Programming Interface (API).

5.1.1 *Apache Hadoop*

Apache Hadoop is a Map/Reduce Framework written in Java that exposes a simple programming API to distribute large scale processing across clusters of computers [19]. However in order to make Flowy play well with the framework, the implementation either has to use a Python wrapper around the Java API or translate the complete implementation to Java through Jython. Even more since Flowy uses HDF files for it's I/O processing, staging the HDF files properly in the Hadoop Distributed File System (HDFS) [20] and then later streaming them using Hadoop Streaming utility would still be an issue as suggested in [2]

5.1.2 *The Disco Project*

Disco is a distributed computing platform using the Map/Reduce framework for large-scale data intensive applications [21]. The core of the platform is written in Erlang and the standard library to interface with the core is written in Python. Since the `map` and `reduce` jobs can be easily written as Python functions and dispatched to the worker

threads in a pre-packaged format, it is less difficult to setup Disco to utilize Flowy as a `map` function. In addition, the usage of HDF files for I/O processing pose no additional modifications whatsoever since the input data files can be anywhere and supplied to the worker threads in absolute paths.

5.2 PARALLELIZING FLOWY

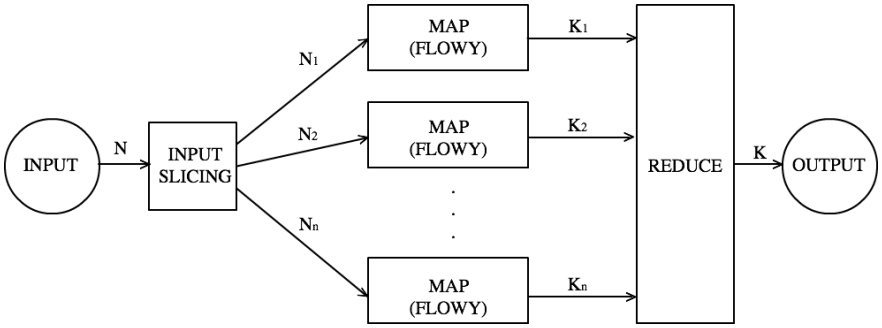

Figure 2: Parallelizing Flowy using Map/Reduce [2]

In an attempt to parallelize Flowy, it was run as a map function on a successful single node Disco installation as shown in 2. Although the setup on a multiple node cluster would be theoretically almost equivalent, Flowy has not yet been tested in such a scenario.

5.2.1 *Slicing Inputs*

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

5.2.1.1 *Using* only *Filters*

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

5.2.1.2 *Using Groupers*

A flow query that also involves groupers and group-filters cannot use static slice boundaries since the grouping rules can be either absolute or relative. As a result, Flowy needs to be made aware of slice boundaries by passing the timestamps as command line parameters. In such a scenario, each branch will skip the pre-slices, whereby the actual slices and the post-slices will be processed to create relevant groups as shown in figure 3. It is advisable to slice the flow-records at low traffic spots to avoid the risk of cutting the records belonging to the same group. The idea of skipping pre-slices and sweeping across

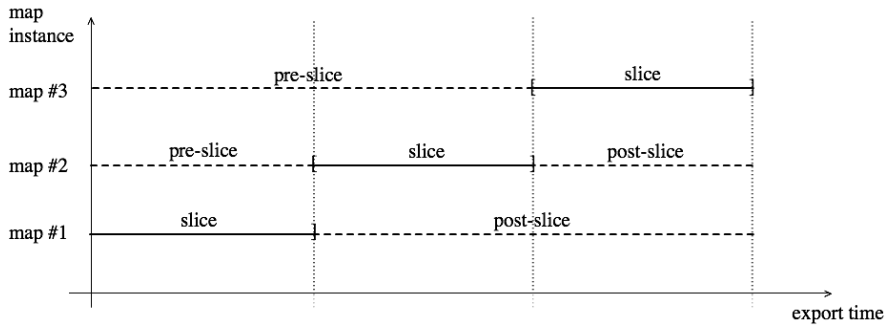

Figure 3: Slice Boundaries Aware Flowy [2]

post-slices can result in many fragmented redundant groups. These can be identified by the `reduce` function by removing the groups that are a proper subset of the previous group in the slice at the cost of additional complexity as shown in figure 4

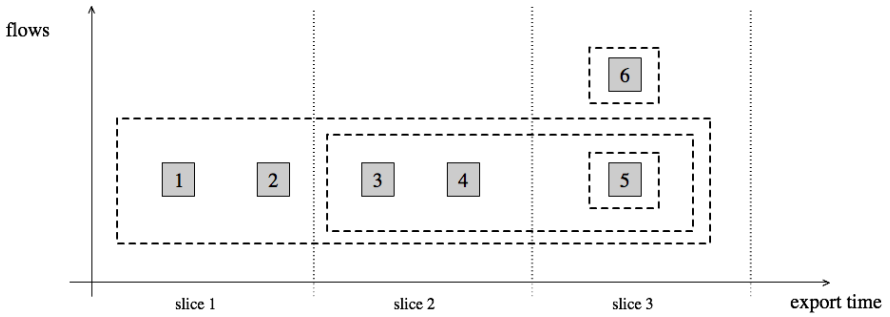

Figure 4: Flowy: Redundant Groups [2]

5.2.1.3 *Using Mergers*

The relative dependency in the merger stage of the pipeline is even worse, since the comparison needs to take place between groups resulting from the output of separate `map` functions. This calls for inhibiting parallelism up to and including the group-filter stage. As

a result each worker thread would return back its filtered groups to the master node, which then would apply the rules of the merger stage to all the received groups at once in a `reduce` function. In such a scenario, although the branch with the longest runtime complexity will become the bottleneck for the merger, the overall runtime would still be dramatically reduced when the number of branches are large as suggested in [9]

5.2.2 *Flowy as a Map Function*

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

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

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

6 FLOWY 2.0

In an attempt to make the first prototype implementation of Flowy comparable with the contemporary flow-record analysis tools, the substitution of the performance hit sections of the Python code was thought out. Flowy 2.0 [7] is the outcome of a complete rewrite of the core of the prototype implementation in C making it relatively faster in orders of magnitude.

6.1 PERFORMANCE ISSUES

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

Table 1: Runtime Breakup of Individual Stages [7]

The runtime breakup of individual stages of the processing pipeline as shown in 1 reveal that the grouper and merger incur a massive performance hit. A quick investigation hints towards usage of large deep nested loops in the merger with a worst-case $O(n^3)$ runtime complexity.

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

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

6.2 FLOWY IMPROVEMENTS

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

6.2.1 *Early Improvements*

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

A C library was written to parse and read/write flow-records in `flow-tools` compatible format. The C library was connected to the Python prototype using Cython [22][23]. This allowed the flow-records to be easily referenced by an identifier, thereby giving away the need to every time copy all the flow-records when moving ahead in the processing pipeline. Cython was used since it allowed to write C extensions in a Pythonic way by strong-typing variables, calling native C libraries and allowing usage of pointers and structs, thereby providing the best of both worlds [24].

6.2.2 *Data Format*

A custom C library was written to directly read/write data in the `flow-tools` format to provide a drop-in replacement for PyTables and overcome the overhead of format conversions. The library sequentially reads the complete flow-records into memory to support random access required for relative filtering. Each flow-record is stored in a `char` array and the offsets to each field are stored in a separate `struct`. The array of such records are indexed allowing fast retrieval in $O(1)$ time. The C library is currently limited to support *only* `flow-tools` formats; `nfdump` file formats are yet to be supported.

6.2.3 *Rewrite of Core Algorithms in C*

A design decision was made to rewrite the entire processing pipeline in C. However, currently the core cannot parse the flow-query file, therefore the execution is triggered by a tedious manual filling of the `structs` by the contents of the query.

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

Listing 1: Filter Rule Struct [7]

A filter stage `struct` is shown in listing 1. The field to be filtered is indicated using a `field_offset` and `field_length` in the `char` array of a records. The value to be compared against with is also supplied which can be either a static value or another field of a record. `func` is a function pointer to the operation that is to be carried out on a record whose record identifier is passed to it. The filter runs in $O(n)$ time as it needs to traverse through all the records of the `char` array.

```
1  struct merger_rule {
2    size_t branch1;
3    size_t field1;
4    size_t branch2;
5    size_t field2;
6    uint64_t delta;
7    bool (*func)(struct group *group1,
8      size_t field1,
9      struct group *group2,
10     size_t field2,
11     uint64_t delta);
12 };
```

Listing 2: Merger Rule Struct [7]

Similarly, a merger stage `struct` is shown in listing 2. `branch{1,2}` are branch identifiers and `field{1,2}` are the aggregated field identifers in the order of aggregation. `func` is a function pointer pointing to the operation to be carried out. The merger runs in $O(n^k)$ time where k is the number of branches. The `char` arrays in each branch are disjoint since a record cannot be part of more than one group.

The current core implementation also strictly adheres to the processing pipeline shown in figure 1. As such, it is not currently possible to skip stages. In addition it is not currently possible to have more than one merger or grouper in the flow-query or aggregate fields in the grouper module since `char` array storage is not possible.

6.3 BENCHMARKS

Number of records	Flowy	Flowy 2.0
103k	1177s	0.3s
337k	20875s	3.4s
656k	70035s	13s
868k	131578s	23s
1161k	234714s	86s

Table 2: Flowy vs Flowy2 [7]

A flow query with the union aggregations stripped off was used as a sample to compare the runtime performance of Flowy [11] with Flowy2 [7] . The benchmarks are shown in figure 2. It is conspicuous how well the replacement of the core algorithms from Python to C turned out to be.

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

Listing 3: Flowy2 vs `flow-tools` [7]

In another test, Flowy2's functionality was reduced to absolute filtering to compare its performance with a state-of-the-art `flow-tools` analysis tool using 3. It turned out Flowy2 performed just as comparable if not better on an average.

6.4 FUTURE OUTLOOK

In a follow up to a commendable effort in making the Flowy prototype drastically improve by orders of magnitude, the author in [7] has suggested numerous areas of improvement to make the software fully functional again.

6.4.1 *System Integration*

The Python prototype is currently left unused. The idea is at this stage is to allow the Python prototype to parse and validate the flow query file which in turn would pass the contents to a Cython wrapper which on the fly will forward them to the core to properly fill in the structs. At this point, the C core will process the query pipeline and eventually forward back the results to the Python prototype which it can use to display the results in a human friendly format.

6.4.2 *Searching with Trees*

The benchmarks performed in [7] had a complexity of $O(n^2)$ for the grouper and merger. This was when the number of branches in the pipeline was reduced to maximum of 2 and the flow-query had a single module for both the merger and grouper. With the current implementation, this complexity is deemed to increase exponentially as the number of records, branches and the grouper, merger modules in the flow-query increase. Therefore, having a search tree lookup for the grouper and merger stage would help bring the runtime costs down, whereby one of the fields will be traversed sequentially in $O(n)$ time and for each field comparison will be performed by search tree lookups in $O(\log(n))$ time bringing down the complexity to $O(n\log(n))$. B+trees would essentially work in this case, since records can still be traversed sequentially along a list after a search tree lookup.

6.4.3 *Specialized Functions in Inner Loops*

The comparison operations are currently passed an offset and the length of the field type to be compared as shown in listings 1, 2. The length needs to be checked before making a cast to an appropriate type inside these functions. Such checks can be avoided by writing specialized functions for each combination of the field type (33) and supported operations (19) totaling to 20K functions. Such functions can be dynamically generated from the Python code and would take around 3MiB of space in memory as suggested in [7] which looks like worth the effort considering these functions are invoked from the innermost loops in each stage of the pipeline, and therefore squeezing such optimizations would go a long way in improving the C core.

6.4.4 *Efficient Multithreading*

The core C implementation currently has limited multithreading. Each branch in the pipeline runs on a separate thread and uses affinity masks to delegate the thread to a separate processor core. However, this implies that merger and ungrouper stages still remain single-

threaded and the multithreaded utilization largely depends on the query being executed. The situation can be improved by writing a `pthreads` wrapper that auto detects the number of available cores, creates a appropriate size thread pool and equally divides the tasks among the threads in the pool. This would also lead to increased complexity of managing mutual exclusion of shared memory and needs to be investigated.

6.4.5 *Additional Functionality*

The core C implementation currently can only parse flow-records in `flow-tools` and support for `nfdump` file formats is left out. The comparison (» and «) and aggregation (intersect) operations are not full blown and can be extended. The possibility to write the filters in Conjunctive Normal Form (CNF) form still needs to investigated.

7

F

In lieu of the significant leaps made by Flowy 2.0 in making the initial prototype usable, additional efforts were made by the same author to work upon the enlisted areas of improvements mentioned in 6.4. To mark this evolution of initial prototype to the current bleeding edge state, it was decided to rename the implementation to F [9] with an exhaustive performance evaluation against the state-of-the-art flow processing tools [25, 26] that operate on absolute filters.

7.1 RULE INTERFACES

The design of the rule interfaces for a flow-query was rethought. An object-oriented approach was followed to abstract out details into multiple levels of inheritance. The `flowquery` struct for instance, is the parent of all the rule interfaces as shown in listing 4.

```
1  struct flowquery {
2    size_t num_branches;
3    struct branch_info *branches;
4    struct merger_rule **mrules;
5  };
```

Listing 4: Flow Query Struct [9]

`branch_info` struct defines rules for each branch. It conglomerates filter, grouper and group-filter stages as shown in listing 5.

```
1  struct branch_info {
2    int branch_id;
3    struct ft_data *data;
4    struct filter_rule *filter_rules;
5    size_t num_filter_rules;
6    struct grouper_rule *group_modules;
7    size_t num_group_modules;
8    struct grouper_aggr *aggr;
9    size_t num_aggr;
10   struct gfilter_rule *gfilter_rules;
11   size_t num_gfilter_rules;
12   struct group **filtered_groups;
13   size_t num_filtered_groups;
14 };
```

Listing 5: Branch Info Struct [9]

```
struct grouper_rule {
  size_t field_offset1;
  size_t field_offset2;
  uint64_t delta;
  uint16_t op;
  bool (*func)(
    struct group *group,
    size_t field_offset1,
    char *record2,
    size_t field_offset2,
    uint64_t delta);
};
```

Listing 6: Grouper Struct [8]

```
struct group {
  char        **members;
  size_t      num_members;
  struct aggr *aggr;
  uint32_t    start;
  uint32_t    end;
};

struct aggr {
  size_t   num_values;
  uint64_t *values;
};
```

Listing 7: Group Struct [8]

The group-filter struct is similar to the filter struct previously shown in listing 1. The grouper struct is shown in listing 6 and is used to perform relative comparison on the flow-records. It takes in offsets of the fields to be grouped, their lengths and a comparison function. Possible comparison functions are `eq`, `ne`, `lt`, `gt`, `le` and `ge`. The comparison function creates a `group` instance, a pointer to which is passed to it. The group struct is shown in listing 7 which apart from the information about the members, also points to a grouper aggregation struct that contains meta-information resulting from calling an aggregation function.

```
struct grouper_aggr {
  int module;
  size_t field_offset;
  struct aggr (*func)(
    char **group_records,
    size_t num_records,
    size_t field_offset);
};
```

Listing 8: Grouper Aggregation Struct [9]

The grouper aggregation struct is shown in listing 8 and consists of the module to aggregate over, the field offset and the aggregation function. Possible aggregation functions are `static`, `count`, `union`, `min/max`, `mean/median`, `stddev`, `sum/prod`, `and/or/xor`. The merger stage struct is the same as was previously shown in listing 2 and allows relative comparison between groups from different branches.

The rules are now possible to be written in CNF. CNF allow the flexibility to define every possible logical expression with the available comparison operations. The comparison (» and «) and the `intersect` aggregation operations still need to be implemented though as was previously mentioned in section 6.4.5.

7.2 FLOWY 2.0 IMPROVEMENTS

This study focusses on optimizing deep nested loops in each processing stage and improving the overall complexity of the grouper and merger as previous enlisted in sections 6.4.3 and 6.4.2.

7.2.1 *Efficient Rule Processing*

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

```
1 bool filter_eq_uint8_t(...);
2 bool filter_eq_uint16_t(...);
3 ...
```

Listing 9: Auto Generated Comparison Functions [9]

using function pointers

In the default method, there is a comparison function defined for every possible field length (33) and comparison operations (19). These functions are generated using a Python script [1] and are declared/defined in `auto_comps.{h,c}` as shown in listing 9. The rule definitions are now able to make calls using a function name derived from the combination of field length, delta type and operation. This subverts the need to define complex branching statements and reduces complexity.

```
1 switch (group_modules[k].op) {
2   case RULE_EQ | RULE_S1_8 | RULE_S2_8 | RULE_ABS:
3   case RULE_EQ | RULE_S1_8 | RULE_S2_8 | RULE_REL:
4   ...
```

Listing 10: Auto Generated Switch Statement [9]

using switch statement

In the other method, the logic is executed by comparing the field length and the operation by falling through a huge switch statement. Such a huge switch statement is again generated using the same Python script and is defined in `auto_switch.c` as shown in listing 10.

1 `fun_gen.py`

7.2.2 *Divide and Conquer for Fast Relative Comparisons*

The grouper and the merger have always been the performance hit stages of the processing pipeline. In the previous implementation, the grouper had a complexity of $O(n^2)$ whereas the merger had a complexity of $O(n^m)$ where n is the number of groups and m is the number of branches.

In order to reduce the number of comparisons in these stages, using a binary search after a quick sort on the flow (or group) records was thought out. To achieve this, the array of pointers to flow (or group) records were sorted according to the first grouping (or merging) rule. Such a sorted array of pointers was then traversed linearly to find unique values and point to them using another array of pointers to records. This helped the grouper (and the merger) perform binary searches to find records that would group together, by using the knowledge of records that satisfied the first rule. This eventually reduced the complexity to $O(n * k)$ for the grouper, and $O(n^{m-1} * k)$ for the merger, where $k << n$.

using quick sort and binary search

However, it still looks like having an actual search tree would benefit the grouper and merger, whereby one of the fields will be traversed sequentially in $O(n)$ time and for each field, the comparison will be performed by search tree lookups in $O(\log(n))$ time bringing down the complexity to $O(n\log(n))$ and is a future action item.

using search trees

7.3 BENCHMARKS

In order to evaluate how well F now performs with these added improvements, the authors decided to compare it with the state-of-the-art flow-processing tools: `flow-tools` [25] and `nfdump` [26]. Since these tools do not currently support relative filtering of flow-records, a set of 3 queries involving only absolute filters was defined as shown in listing 11 and evaluated on a set of 500K − 10M flow-records.

```
1 src port 80
2 src port 80 or dst port 25
3 src port 443 or (src port 80 and dst port 25)
```

Listing 11: Queries to Benchmark F [9]

It turned out that F performed as well if not better than the other flow-processing tools. F's complexity linearly increased with the increase in flow-records, thereby demonstrating a complexity of $O(n)$.

8

FLOWY: APPLICATIONS

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

8.1 APPLICATION IDENTIFICATION USING FLOW SIGNATURES

The idea behind this study was to identify applications using flow traces on a network by analyzing potential left-behind signatures that describe them [27, 8]. This was based on the hypothesis that each application type generates unique flow signatures that might work as a fingerprint feature. To achieve this, a collection of network traces were recorded from several users and subsequently analyzed. The identified signatures were formalized by writing flow queries that were executed on Flowy [10]. Several separate instances of the network traces were queried to evaluate the approach and come to a conclusion.

```
splitter S {}

...

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

ungrouper U {}

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

Listing 12: Skype Application Signature [8]

A formalized Flowy query to identify Skype from the flow traces for an instance is described in listing 12. The filter, grouper and group-filter sections of each branch are shown separately in listings 14 and 13. Additional queries identifying variety of web browsers, mail clients, IM clients and media players can be found in [8].

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

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

groupfilter GF_SSDP {
  B = 321
}
```

Listing 13: Branch A [8]

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

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

groupfilter GF_NAT_PMP {
  B = 160
}
```

Listing 14: Branch B [8]

The filter `F_SSDP` is used to identify the four identical User Datagram Protocol (UDP) multicast messages the client sends out using Simple Service Discovery Protocol (SSDP) [28]. Similarly `F_NAT_PMP` filter is used to identify four Network Address Translation Port Mapping Protocol (NAT-PMP) [29] messages sent over UDP. The groupers `G_SSDP` and `G_NAT_PMP` group together flow records with the same source and destination IP address and the aggregate clauses describe the meta information with unique source IP addresses for each group records along with the total bytes carried within each group. The meta information is used to further filter the group-records in `GF_SSDP` and `GF_NAT_PMP` modules.

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

Table 3: Application Flow Signatures: Results [8]

The identification results obtained from the analysis of flow-traces from ten unique users are compiled together in table 3. The results demonstrate a success rate of 96% for the five applications tested. This study reveals that it is possible to identify applications from their network flow fingerprints and is a first step towards automating the complete process whereby machine learning techniques would be used to automatically generate flow-queries and identify new applications and even more so newer versions of the same application.

8.2 CYBERMETRICS: USER IDENTIFICATION

The idea of identification of users based on biometric patterns such as keystroke dynamics [30], mouse interactions [31] or activity cycles in online games [32] has been long known. This study takes the idea even further by using flow-record patterns as a characteristic (cybermetrics) to identify a user on a network [3, 33]. Such a cybermetric user identification can be used for the purpose of providing secure access, system administration and network management. The feature extraction module of the analyzer as shown in figure 5 uses three distinct feature sets that could possibly be used to identify a user from a flow-record trace.

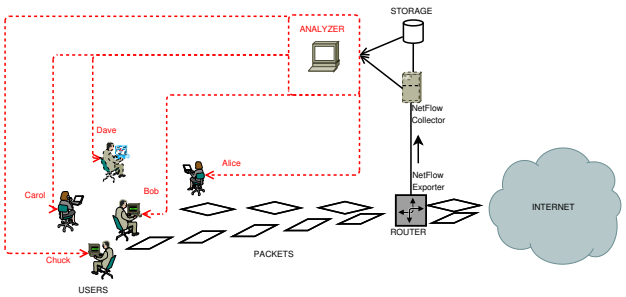

Figure 5: Cybermetrics: Overview [3]

Initial research efforts started with identifying application signatures in flow-records in [27, 8] and became relevant because different people have different preferences in the applications they use and as such a set of applications in flow-records is a characteristic feature of a user. Flowy queries were formalized for four different set of applications and tested against a known set of users. The evaluation results of the derived queries as shown in table 3 demonstrated a strong evidence of presence (or absence) of applications and thereby provided an eventual marker for user identification.

application signatures

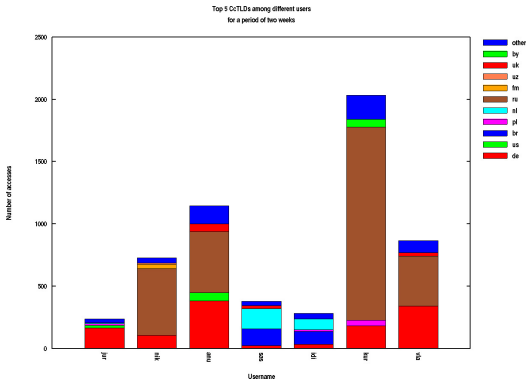

Figure 6: Geographical Preferences [3]

geographical preferences

The authors also looked into the geographical affiliations of different users by analyzing the Country Code Top-Level Domain (ccTLD) of the browsed websites. They proposed a hypothesis that a user's origins strongly influences their browsing activity. The analysis of the results established that the top five visited ccTLDs constituted more than 85% of the overall number of a user's visits as shown in figure 6.

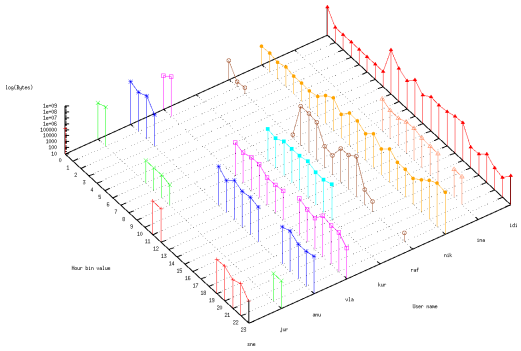

Figure 7: Daily Distributions for HTTP Traffic [3]

flow-record statistics

In the end, the authors introduced a proof-of-concept method of user differentiation based on statistical features. These features considered daily distributions of parameters that were based on different port numbers. For instance, figure 7 shows the daily distribution of different users based on their Hypertext Transfer Protocol (HTTP) traffic usage. It was also witnessed that the time duration also played a key role in the process of feature formation, whereby the number of longer flows increased with the duration and consequently resulted in higher cross-correlations as shown in figure 8

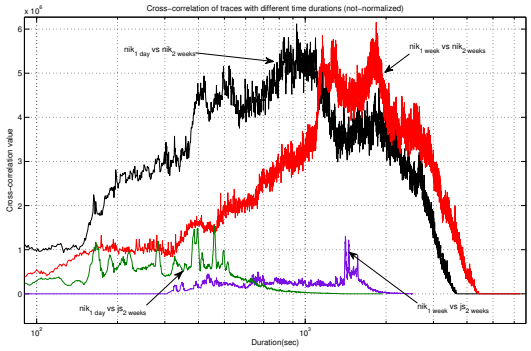

Figure 8: Cross Correlation of Traces with Varying Times [3]

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

8.3 IPV6 TRANSITION FAILURE IDENTIFICATION

The IPv4 address space depletion is upon us and has become more imminent in the last few years. While IPv6 can readily expand the extent of the Internet, deploying it alone is clearly not a solution today and hence there are a continuum of transitioning solutions that would help in this migration. In this study [4] we evaluated the compatibility of popular applications with such transitioning solutions: NAT64 [34] and Dual-Stack Lite [35]. The goal was to find potential failures by identifying application failure signatures left behind in the flow-record traces using Flowy. These failure signatures could later be used by service providers to automate the detection and eventually shorten the deployment verification cycle.

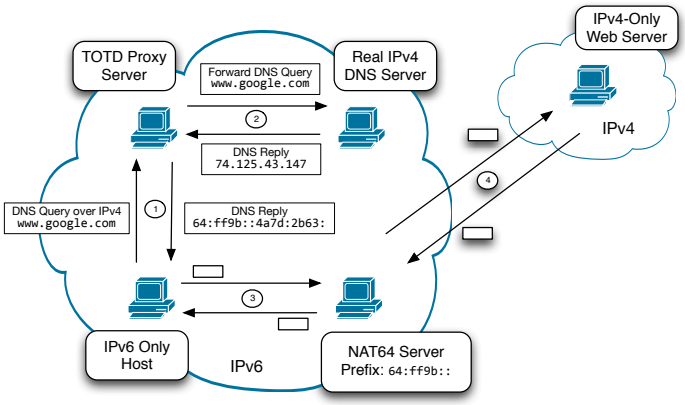

Figure 9: NAT64 Setup [4]

In the NAT64 deployment testbed as shown in figure 9, the authors witnessed failure in 3 applications: Skype, OpenVPN and Transmission. Flowy queries were defined to establish failure signatures for each application. A formalized Flowy query to identify Skype failure signature for an instance is described in listing 17. The filter sections of each branch are shown separately in listings 15 and 16.

application operation under NAT64

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

Listing 15: Branch A [4]

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

Listing 16: Branch B-C-D [4]

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

skype failure signature

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

```
splitter S {}

...

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

merger M {
  branches mDNS, LOGIN1, LOGIN2, LOGIN3

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

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

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

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

  mDNS < LOGIN1
  mDNS < LOGIN2
  mDNS < LOGIN3
}

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

Listing 17: Skype Failure Signature [4]

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

failure when using IPv4 literals

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

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

8.4 OPENFLOW

OpenFlow [36] is an open standards protocol that runs between an Ethernet switch and an OpenFlow controller (a software designed to run on a x86 server) to securely manage the forwarding plane of the switch over the network as shown in figure 10. This enables the controller to push out policies that dictate how to process flow-records crossing the networking infrastructure to eventually improve bandwidth, reduce latency and save power.

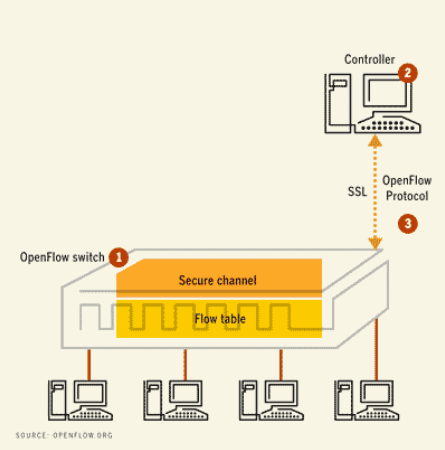

Figure 10: OpenFlow Architecture [5]

OpenFlow initially started as a way to allow researchers to experiment with new ideas in sufficiently realistic settings by allowing the live production networking gear to open a narrow programmable external interface to it whereby at the same time keeping the inner workings of the gear hidden and proprietary. The idea took off outside the academic setting in recent years with the need of data centers requiring to run large-scale map/reduce jobs with full cross-sectional bandwidth. Such a requirement called for flexible forwarding and programmable networks to meet the application-specific needs. Today, the commercial underpinning of OpenFlow are driven by the Infrastructure as a Service (IaaS) providers trying to virtualize their network architecture to solve the issue of multi-tenancy to implement Network as a Service (NaaS) architectures [37].

motivation

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

programming flows

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

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

software stack

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

flowy and openflow

8.5 FLOW LEVEL SPAM DETECTION

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

Table 4: Features in Spam Flow [6]

Classical methods to mitigate spam such as content filtering and reputation analysis utilize the the weakness of spam messages and the places from where they originate from. Though currently effective, it's only a matter of time when spammers find a way to subvert around these vantage points. In this study [6, 38], the authors analyze the transport level characteristics of the email flows to differentiate between spam and legitimate email. These characteristics exploit the fundamental weakness of each spam: the requirements to send large amounts of the same email on resource constrained links owned by

compromised botnets which is unlikely to change in the near future. They reason that a spammer's traffic is more likely to experience Transmission Control Protocol (TCP) timeouts, retransmissions, resets and variable Round Trip Time (RTT) estimates. Based on this hypothesis they extract 13 learning features as shown in table 4 to formalize a machine learning problem.

spamflow features

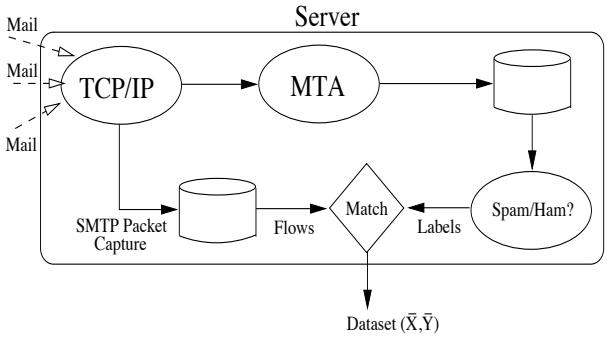

Figure 11: Spam Flow Classifier [6]

The data collection methodology is depicted in figure 11 where TCP packets corresponding to email messages are extracted and examined on a per-email flow basis. The packets in an email flow are coalesced together by using TCP port numbers in the email headers. Using machine learning feature selection, a spam classifier is built that matches each flow to a binary spam/ham ground-truth label. Support Vector Machines (SVMs) [39] are used for classification and Greedy Forward Fitting (FF) [40] is used for feature selection to find a set of features that provide the least training error. It turns out the classifier achieves 90% accuracy with 78% detectability of false-negatives from a particular content filter.

a spam classifier

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

flowy and spamflow

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

extending spamflow

Part III

MOTIVATION

You can put some informational part preamble text here

Part IV

WORK PLAN

You can put some informational part preamble text here

Part V

IMPLEMENTATION AND EVALUATION

You can put some informational part preamble text here

9 DESIGN

10 IMPLEMENTATION

11

PERFORMANCE EVALUATION

12 FUTURE WORK

13 CONCLUSION

Part VI

APPENDIX

A

APPENDIX

Put your appendix here.

BIBLIOGRAPHY

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

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

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

[4] V. Bajpai, N. Melnikov, and J. Schönwälder, "Automated Failure Identification under IPv6 Transition Mechanisms." 2012.

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

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

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

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

[9] J. Schauer, N. Melnikov, and J. Schönwälder, "F." 2012.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

DECLARATION

Put your declaration here.

Bremen, Germany, June 2012

Vaibhav Bajpai