F $(\nu 2)$

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

Vaibhav Bajpai

Masters Thesis Proposal

supervised by
Prof. Dr. Jürgen Schönwälder

Computer Networks and Distributed Systems
School of Engineering and Science
Jacobs University Bremen
Bremen, Germany

February 2012

ABSTRACT

With the long dominance of Cisco's NetFlow [1] protocol and now with the emergence of Internet Engineering Task Force (IETF)'s Internet Protocol Flow Information Export (IPFIX) [2] open standard, traffic measurement practitioners have finally settled down with using Internet Protocol (IP) flow export as the de-facto technique for sending traffic patterns. These patterns have the potential to be used for billing and mediation, bandwidth provisioning, detecting malicious attacks, network performance evaluation and overall improvement.

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

F [5] is the prototype implementation of our in-house flow query language which has underwent significant changes in the last few years. The core of the former Python implementation [6] has now been rewritten in C [7] to make it comparable to the contemporary flow processing tools. However, this has disconnected the flow-query parser present in the former implementation. The two implementations have now branched off so much that both currently live in their own parallel universe. This thesis takes up the challenge to glue the better parts of both of these implementations together to create a complete package that has the full-blown functionality and exploits the best of both worlds. In the process, it is also planned to bring the implementation up to speed with bleeding edge IPFIX support, parallelize it by making it MapReduce [8] aware and recover it from limitations learnt from the wealth of experience gained after managing the two branches for the last few years.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

Part I

INTRODUCTION

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

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

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

1 TRAFFIC MEASUREMENT APPROACHES

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

1.1 CAPTURING PACKETS

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

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

Listing 1: `tcpdump`: Example

tcpdump

`tcpdump` and `wireshark` are the most popular tools used for packet capture and analysis. `tcpdump` [19] is a premier command-line utility that uses the `libpcap` [20] library for packet capture. A simple example to capture and read the Hypertext Transfer Protocol (HTTP) traffic is described in listing 1. The power of `tcpdump` comes from the richness of its expressions, the ability to combine them using logical connectives and extract specific portions of a packet using filters. `wireshark` [21] is a Graphical User Interface (GUI) application, aimed at both journeymen and packet analysis ninjas. It supports a large number of protocols, has a straightforward layout, excellent documentation, and can run on all major operating systems.

wireshark

applicability

Several studies have made use of this approach to analyze the network traffic patterns. The authors in [22], for instance use data mining methodologies to define clusters of behavior profiles by understanding the captured traffic of end hosts. These clusters are then fed into classifiers to automatically identify anomalous behavior patterns that are of interest to network operators. Similar profiling of end-hosts traffic in performed in [23], but at the transport layer. This effort focusses on making the approach tunable to strike out a balance between the amount of traffic classified and the accuracy achieved by analyzing the traffic at multiple levels of details.

pros and cons

This approach benefits from the astounding level of detail it can provide. It allows deep packet inspection of the traces, thereby exposing even the application content being exchanged across the network. This calls for privacy concerns and can even bring in legal repercussions to make this technique unattractive for traffic analyzers today. In addition, the actual usage of this method comes at a higher price of its storage overhead and its inability to scale to larger setups.

1.2 CAPTURING FLOWS

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

netflow

ipfix

NetFlow and IPFIX are the two popular standards of IP flow information export. NetFlow [1] is a proprietary network protocol designed by Cisco Systems. It allow routers to generate and export flow records to a designated collector. The latest version, NetFlow v9 provides flexibility of user-tailored export templates, Multiprotocol Label Switching (MPLS) and IPv6 support and a larger set of flow keys. IPFIX [2] on the other hand is an open standard by IETF deemed to be the logical successor of NetFlow v9 on which it is based. The novelty of the standard lies in its ability to describe record formats at runtime using templates based on an extensible and well-defined information model. The data transfer mechanism is also simplistic and extensible by being unidirectional and transport protocol agnostic.

applicability

The wide applicability of this approach is easily seen from the pervasive use of flow records for a vibrant set of network analysis applications. For instance, the authors in [24] use the flow characteristics in the traffic pattern to formalize a detection function that maps traffic patterns to different Denial of Service (DoS) attacks, whereas in [25], the authors use the flow-record data to exploit timing characteristics of webmail clients to classify features that could identify webmail traffic from any other traffic running over HTTPS.

pros and cons

This is has been possible largely due to the hardware-assisted aggregation of the packets that has helped solve the storage overhead and scalability limitation of packet capturing techniques. Overcoming these limitations have eventually allowed researchers to perform network analysis over a larger dataset passing across high-speed links. However, with the ever-increasing bandwidth demands, the speed of the network links in the internet backbone is only slated to increase further, therefore the time is not too far when this issue might again scares us of its homecoming.

1.3 REMOTE MONITORING

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

The Remote Network Monitoring (RMON) Framework [26] for Simple Network Management Protocol (SNMP) [27] defines a number of Management Information Base (MIB) objects to be used by these monitoring probes. The RMON-1 standard [28] for instance, defines a MIB module to collect statistics, capture and filter packet contents at the logical link layer. The architecture in this standard has been further extended with a feature upgrade by the RMON-2 standard [29] to support similar analysis up to the application layer.

rmon

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

pros and cons

1.4 REMOTE METERING

In this technique, meters are deployed at the network measurement points to capture flow data according to a predefined set of rules specified by the user. The model, as defined by the Realtime Traffic Flow Measurement (RTFM) working group [30] has been designed to be protocol agnostic and restrictive in the amount of flow data that can transmitted across the network and stored to reduce the processing time of network analysis applications.

The feature that sets this technique apart is the flexibility given to the user to specify their flow measurement requirements, thereby allowing them to filter out the traffic they do not care about. This calls for the users, to at the very outset analyze and freeze their requirements before they start off to capture the traffic. This is analogous to the flaws inherit in the waterfall model [31] of software design, whereby one need to design the design before one designs it.

pros and cons

2 FLOW EXPORT PROTOCOLS

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

2.1 NETFLOW

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

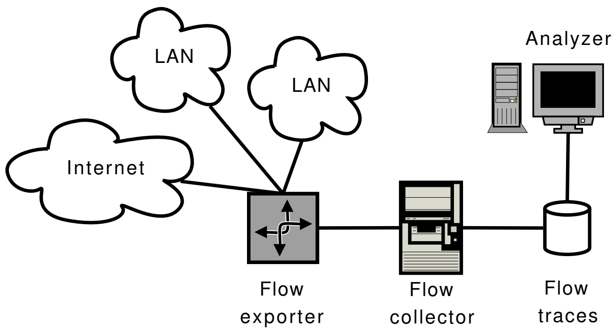

Figure 1: NetFlow: Overview [9]

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

protocol operation

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

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

Listing 2: A Flow Example

what is a flow?

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

version history

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

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

Table 1: NetFlow Version History

netflow v9

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

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

sampled netflow

Though sampled NetFlow does a good job in reducing the exported traffic volume, the sampling rate is still static which either reduces accuracy at low traffic volumes or increases memory use at high traffic volumes. An adaptive algorithm introduced in [32] helps overcome this difficulty. The introduced renormalization technique helps guarantee that the sampling rate can not only adjust to variable traffic mixes but also to network congestion. It also ensures that the flow records do not span over measurement bins to be able to guarantee statistical accuracy. The authors claim, that these updates are easily deployable to any NetFlow v9 router through a software update. In addition they say, a simple hardware add-on (flow counting extension), can also add capability to accurately count non-TCP flows, a feature long waiting to be seen in NetFlow v9.

adaptive netflow

Flexible NetFlow is the newest version of NetFlow v9 that incorporates Packet Sampling (PSAMP) [33] ideas to be able to select individual packets and export them in a packet record. The packet selection can be either deterministic or random depending on the chosen filters and sampling mechanism [34]. The exported packet records can even be authenticated and encrypted using either Transport Layer Security (TLS) [35] or Datagram Transport Layer Security (DTLS) [36] to prevent data manipulation across the route. Since PSAMP is based on IPFIX [37], only its limited feature set is currently supported by Flexible NetFlow. Additional features include ability to custom define flow-keys and flow-expiration rules to drastically reduce the amount of content exported by restricting it to only the needed flow-fields, and additional flows with immediate and permanent caches to suit the export timings to specific needs.

flexible netflow

The challenge to identify relevant records in gigantic collected datasets have fumed recent independent studies to discover flow dependencies. For instance, the authors in [38], describe a model that uses flow timing information by extending the PageRank [39] algorithm to rank and thereby extract the most relevant flows. Their model is weighted using parameters like the amount of bandwidth consumed and the likehood of security threat a flow might result in.

flowrank

Today, as the industry is moving towards data center virtualization, it has become inherently critical to obtain insights into the data center network behavior for optimizations and resource provisioning. Since, Flexible NetFlow's visibility is limited to the IP protocol it currently cannot be used to monitor data-center traffic. NetFlow-lite was thus introduced by Cisco Systems, to flows at the layer 2/3 level to increase data center visibility. NetFlow-lite uses similar packet sampling mechanisms as introduced in Sampled NetFlow along with the combined flexibility of Flexible NetFlow v9 at the switch level. NetFlow-lite captures the layer 2 traffic, encapsulates packet samples and pushes the NetFlow cache outside the switch into a element that can convert NetFlow-lite to Flexible NetFlow records. These flow-records are then later exported to legacy collectors from where they can be used for further processing. The authors in [40] provide the first implementation of NetFlow-Lite which works as an extension to nProbe [41] to seamlessly convert NetFlow-Lite records to NetFlow/IPFIX.

netflow-lite

2.2 IPFIX

IPFIX [2] by IETF is an interoperable protocol for IP flow export. It is deemed to be the logical successor of Flexible NetFlow v9. The working group defines IPFIX as, "a unidirectional, transport-independent protocol with flexible data representation and an information model covering most network management needs at layer 3 and 4". The PSAMP working group [37], that defines standards to individually sample packets in a flow export using statistical methods has adopted IPFIX as its underlying protocol for data transport.

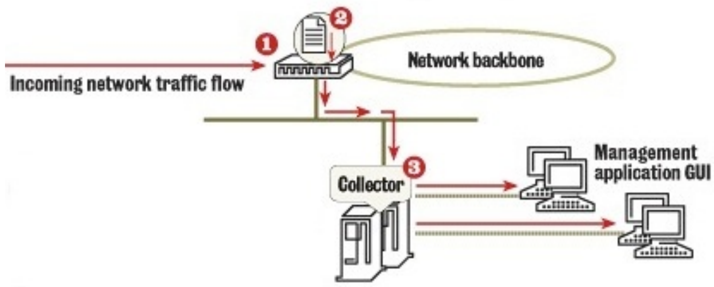

Figure 2: IPFIX: Overview [10]

The IPFIX architecture is described in [42] and is shown in figure 2. The architecture consists of three elements: a meter, which generates flows from IP packets, an exporter, which pushes these flows using IPFIX, and a collector, that collects and saves these flows for offline storage. All these elements have a one-to-many relationship among them. The group is also working to define an intermediary element, that might work to either aggregate or anonymize the flows.

architecture

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

messages and templates

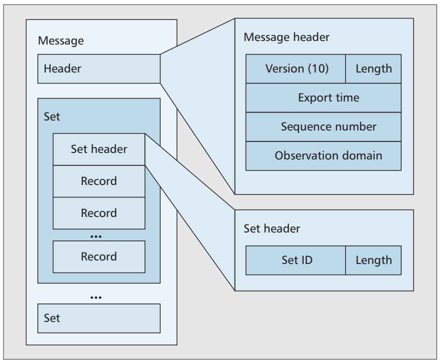

Figure 3: IPFIX: Messages [11]

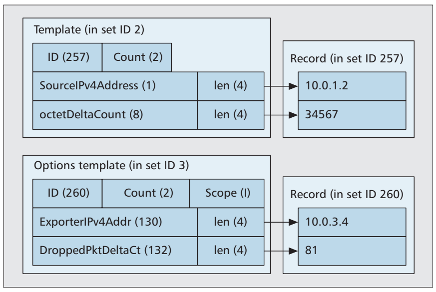

Figure 4: IPFIX: Templates [11]

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

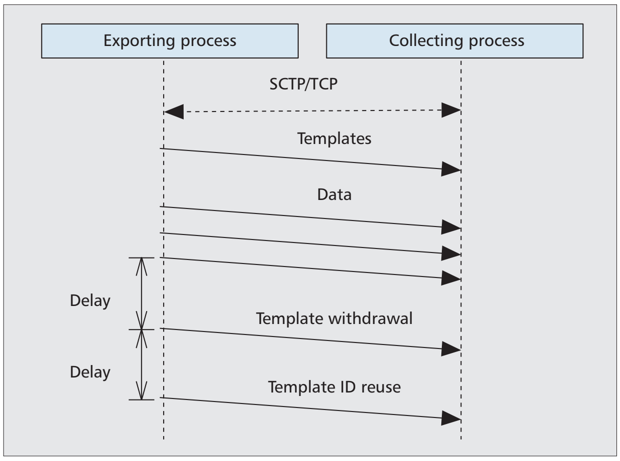

Figure 5: IPFIX: A Transport Session [11]

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

transport and security

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

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

management, extensions and future of ipfix

A MIB to monitor IPFIX devices using SNMP is defined in [43]. A similar configuration model to be used by NETCONF and YANG is being worked upon. In addition, several extensions have been defined to expand upon the protocol's functionality. For instance, [44] defines optional templates to allow bidirectional flows in a single IPFIX export whereas [33] supports aggregating common properties of multiple flows in a single record. IPFIX is even being looked upon as the future application-layer logging protocol as well as the underlying protocol for RESTful architectures. As a result, efforts to support structured data export over IPFIX are also under way.

2.3 SFLOW

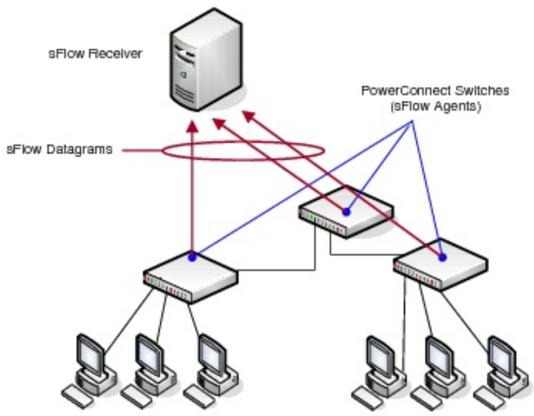

Figure 6: sFlow: Overview [12]

sFlow [45] by InMon Corporation is a competing technology used to capture traffic from switches and routers. It consists of an sFlow agent that captures traffic statistics and sends them across to a central data collector, called the sFlow analyzer as shown in figure 6. In order to be able to accurately monitor traffic at line speeds, the sFlow agent is built on a dedicated ASIC alongside the switching gear. In addition, the captured traffic is sampled before being encapsulated in sFlow datagrams and sent to the analyzer to provide scalability.

sampling mechanisms

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

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

sflow and snmp

sFlow provides a standard interface to configure and monitor the sFlow agents using SNMP. This subverts the need to telnet to every switch of the network infrastructure and use its Command Line Interface (CLI) to make subtle changes which can turn out to be overly complex and time-consuming. A MIB module to remotely control the sFlow agents is defined in [45]. The MIB module is Structure of Managed Information (SMI) v2 compliant and can be translated back to SMI v1 without incurring any semantic differences.

data format

sFlow uses a standard format to send sampled data from the sFlow agent to the sFlow analyzer. The data format is specified using External Data Representation (XDR) [46]. XDR allows compact representation and efficient encoding (or decoding) of the sampled data. The XDR specified sampled data is sent using UDP to a well-known host and port combination specified in the sFlow MIB. UDP is used as a transport mechanism owing to its less stringent buffer requirements and its robustness in delivering traffic information in a timely fashion.

limitations and future

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

Part II

STATE OF THE ART

The semantics and implementation of our in-house 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 3 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 4 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 5 and 6 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 7 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.

3 FLOWY

Flowy [47, 6] is the first prototype implementation of a stream-based flow record query language [3, 13, 48]. The query language allows to describe patterns in flow-records in a declarative and orthogonal fashion, making it easy to read and flexible enough to describe complex relationships among a given set of flows.

3.1 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.

3.1.1 *PyTables and PLY*

Flowy uses PyTables [49] 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.

3.1.2 *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 IPFIX or even newer versions of NetFlow [1] exports. The `FlowToolsReader` class instance (defined in `ftreader` module) provides an iterator over the records defined in `flow-tools` format. This can be plugged into the `RecordReader` class instance (defined in `record` module) to instantly get `Record` class instances.

3.1.3 *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).

3.2 PROCESSING PIPELINE

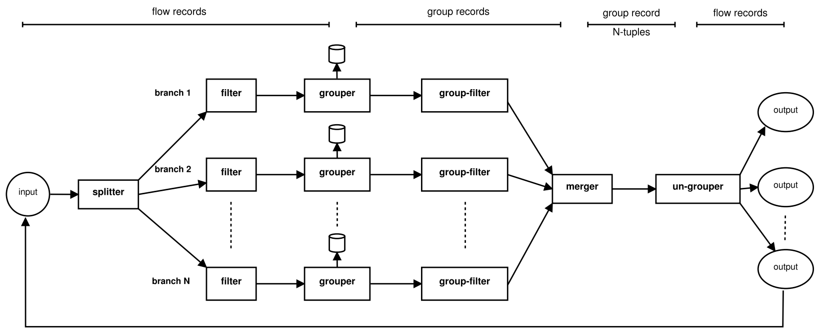

Figure 7: Flowy: Processing Pipeline [13]

The pipeline consists of a number of independent processing elements that are connected to one another using UNIX-based pipes. Each element receives the content from the previous pipe, performs an operation and pushes it to the next element in the pipeline. Figure 7 shows an overview of the processing pipeline. The flow record attributes used in this pipeline exactly correlate with the attributes defines in the IPFIX Information Model specified in RFC 5102 [50]. A complete description on the semantics of each element in the pipeline can be found in [3]

3.2.1 *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.

3.2.1.1 *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.

3.2.1.2 *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.

3.2.2 *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

3.2.2.1 *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.

3.2.2.2 *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.

3.2.3 *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.

3.2.3.1 *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.

3.2.3.2 *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.

3.2.4 *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

3.2.4.1 *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.

3.2.4.2 *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.

3.2.5 *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 [4]

3.2.5.1 *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.

3.2.5.2 *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.

3.2.6 *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.

3.2.6.1 *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.

3.2.6.2 *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.

3.3 FUTURE OUTLOOK

3.3.1 *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.

3.3.2 *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.

3.3.3 *Multithreaded Merger*

The merger stage in the processing pipeline is currently the most computation intensive operation and is unfortunately single-threaded. As suggested in [6] it should be possible to handle the outermost branch loop using multiple threads in a non-blocking fashion to improve performance.

4 FLOWY IMPROVEMENTS USING MAP/REDUCE

Flowy, although clearly setting itself apart with its additional functionality to query intricate patterns in the flows demonstrates relatively high execution times when compared to contemporary flow-processing tools. A recent study [14] 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 [14]

4.1 MAP/REDUCE FRAMEWORKS

Map/Reduce is a programming model for processing large data sets by automatically parallelizing the computation across large-scale clusters of machines [8]. 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).

4.1.1 *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 [51]. 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) [52] and then later streaming them using Hadoop Streaming utility would still be an issue as suggested in [14]

4.1.2 *The Disco Project*

Disco is a distributed computing platform using the Map/Reduce framework for large-scale data intensive applications [53]. 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.

4.2 PARALLELIZING FLOWY

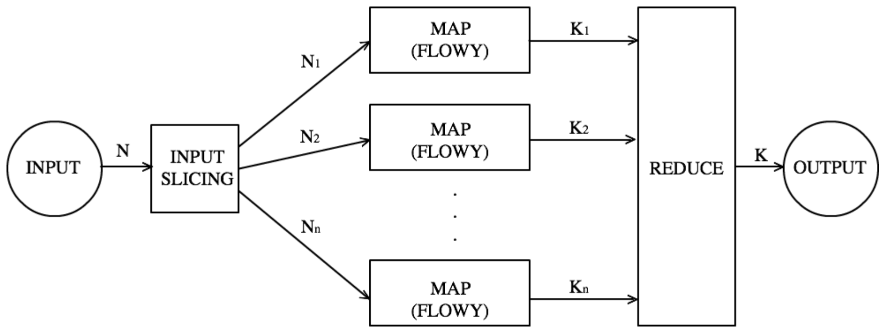

Figure 8: Parallelizing Flowy using Map/Reduce [14]

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

4.2.1 *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.

4.2.1.1 *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.

4.2.1.2 *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 9. It is advisable to slice the flow-records at low traffic spots to avoid the risk of cutting the records belonging to the same group. The idea of skipping pre-slices and sweeping across

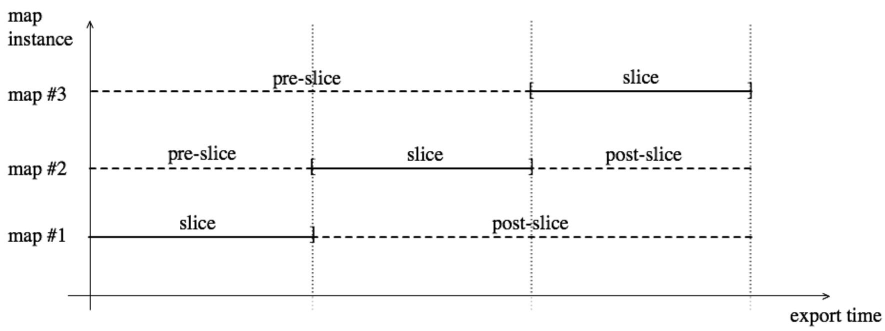

Figure 9: Slice Boundaries Aware Flowy [14]

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

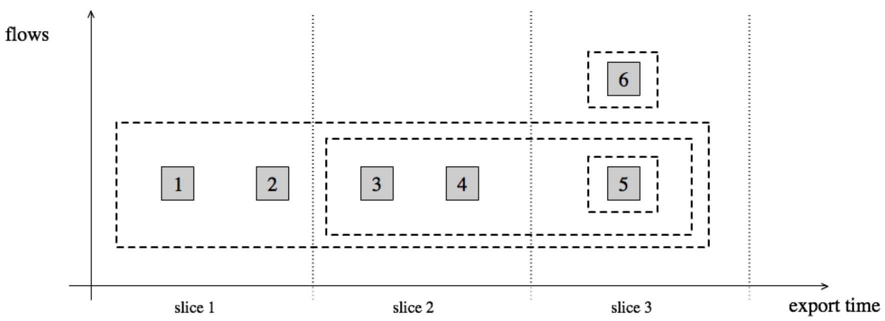

Figure 10: Flowy: Redundant Groups [14]

4.2.1.3 *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 [5]

4.2.2 *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.

5 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.

5.1 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 2: Runtime Breakup of Individual Stages [7]

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

deep nested loops

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

deep copy of flow records

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

pytables and hdf

5.2 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.

5.2.1 *Early Improvements*

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

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.

cython to connect c extensions to python

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 [54][55]. 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 [56].

5.2.2 *Data Format*

a custom c library to replace pytables

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.

5.2.3 *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.

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

Listing 3: Filter Rule Struct [7]

filter stage struct

A filter stage `struct` is shown in listing 3. The field to be filtered is indicated using a `field_offset` and `field_length` in the `char` array of a records. The value to be compared against with is also supplied which can be either a static value or another field of a record. `func` is a function pointer to the operation that is to be carried out on 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.

```c
struct merger_rule {
  size_t branch1;
  size_t field1;
  size_t branch2;
  size_t field2;
  uint64_t delta;
  bool (*func)(struct group *group1,
    size_t field1,
    struct group *group2,
    size_t field2,
    uint64_t delta);
};
```

Listing 4: Merger Rule Struct [7]

merger stage struct

Similarly, a merger stage `struct` is shown in listing 4. `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.

core limitations

The current core implementation also strictly adheres to the processing pipeline shown in figure 7. 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.

5.3 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 3: Flowy vs Flowy2 [7]

flowy 2.0 vs flowy

A flow query with the union aggregations stripped off was used as a sample to compare the runtime performance of Flowy [6] with Flowy 2.0 [7]. The benchmarks are shown in figure 3. It is conspicuous how well the replacement of the core algorithms from Python to C turned out to be.

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

Listing 5: Flowy2 vs `flow-tools` [7]

flowy 2.0 vs flow-tools

In another test, Flowy 2.0's functionality was reduced to absolute filtering to compare its performance with a state-of-the-art `flow-tools` analysis tool using 5. It turned out Flowy 2.0 performed just as comparable if not better on an average.

5.4 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.

5.4.1 *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.

5.4.2 *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.

5.4.3 *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 3, 4. 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.

5.4.4 *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.

5.4.5 *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.

6 FLOWY → 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 5.4. To mark this evolution of initial prototype to the current bleeding edge state, it was decided to rename the implementation to F [5] with an exhaustive performance evaluation against the state-of-the-art flow processing tools [57, 58] that operate on absolute filters.

6.1 RULE INTERFACES

flowquery struct

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 6.

```c
struct flowquery {
  size_t num_branches;
  struct branch_info *branches;
  struct merger_rule **mrules;
};
```

Listing 6: Flow Query Struct [5]

branchinfo struct

`branch_info` struct defines rules for each branch. It conglomerates filter, grouper and group-filter stages as shown in listing 7.

```c
struct branch_info {
  int branch_id;
  struct ft_data *data;
  struct filter_rule *filter_rules;
  size_t num_filter_rules;
  struct grouper_rule *group_modules;
  size_t num_group_modules;
  struct grouper_aggr *aggr;
  size_t num_aggr;
  struct gfilter_rule *gfilter_rules;
  size_t num_gfilter_rules;
  struct group **filtered_groups;
  size_t num_filtered_groups;
};
```

Listing 7: Branch Info Struct [5]

```c
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 8: Grouper Struct [18]

```c
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 9: Group Struct [18]

grouper and group struct

The group-filter struct is similar to the filter struct previously shown in listing 3. The grouper struct is shown in listing 8 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 9 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.

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

Listing 10: Grouper Aggregation Struct [5]

grouper aggregation struct

The grouper aggregation struct is shown in listing 10 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 4 and allows relative comparison between groups from different branches.

rules in cnf

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 5.4.5.

6.2 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 5.4.3 and 5.4.2.

6.2.1 *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.

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

Listing 11: Auto Generated Comparison Functions [5]

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 11. 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.

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

Listing 12: Auto Generated Switch Statement [5]

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 12.

1 `fun_gen.py`

6.2.2 *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.

using quick sort and binary search

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 search trees

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.

6.3 BENCHMARKS

f vs {flow-tools, nfdump}

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` [57] and `nfdump` [58]. 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 13 and evaluated on a set of 500K − 10M flow-records.

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

Listing 13: Queries to Benchmark F [5]

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 $O(n)$.

7

F: 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.

7.1 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 [59, 18]. 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 [47]. 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 14: Skype Application Signature [18]

A formalized Flowy query to identify Skype from the flow traces for an instance is described in listing 14. The filter, grouper and group-filter sections of each branch are shown separately in listings 16 and 15. Additional queries identifying variety of web browsers, mail clients, IM clients and media players can be found in [18].

```
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 15: Branch A [18]

```
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 16: Branch B [18]

skype application signature

The filter F_SSDP is used to identify the four identical UDP multicast messages the client sends out using Simple Service Discovery Protocol (SSDP) [60]. Similarly F_NAT_PMP filter is used to identify four Network Address Translation Port Mapping Protocol (NAT-PMP) [61] messages sent over UDP. The groupers G_SSDP and G_NAT_PMP group together flow records with the same source and destination IP address and the aggregate clauses describe the meta information with unique source IP addresses for each group records along with the total bytes carried within each group. The meta information is used to further filter the group-records in GF_SSDP and GF_NAT_PMP modules.

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

Table 4: Application Flow Signatures: Results [18]

success rate

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

7.2 CYBERMETRICS: USER IDENTIFICATION

The idea of identification of users based on biometric patterns such as keystroke dynamics [62], mouse interactions [63] or activity cycles in online games [64] 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 [9, 65]. Such a cybermetric user identification can be used for the purpose of providing secure access, system administration and network management. The feature extraction module of the analyzer as shown in figure 11 uses three distinct feature sets that could possibly be used to identify a user from a flow-record trace.

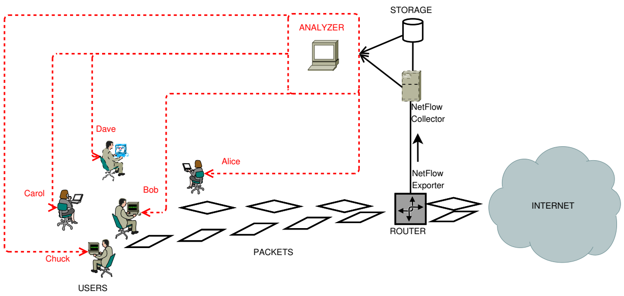

Figure 11: Cybermetrics: Overview [9]

Initial research efforts started with identifying application signatures in flow-records in [59, 18] and became relevant because different people have different preferences in the applications they use and as such a set of applications in flow-records is a characteristic feature of a user. Flowy queries were formalized for four different set of applications and tested against a known set of users. The evaluation results of the derived queries as shown in table 4 demonstrated a strong evidence of presence (or absence) of applications and thereby provided an eventual marker for user identification.

application signatures

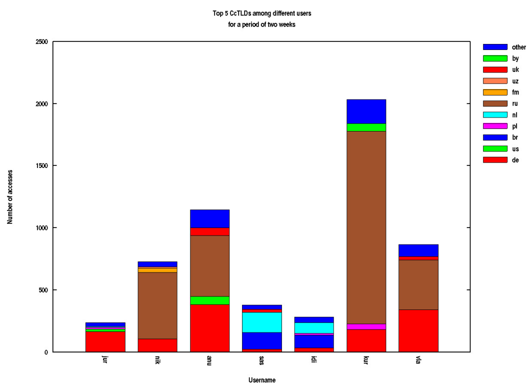

Figure 12: Geographical Preferences [9]

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

geographical preferences

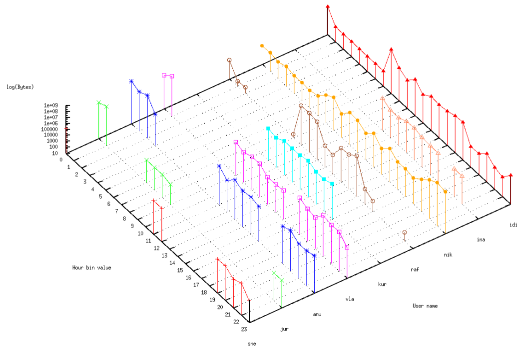

Figure 13: Daily Distributions for HTTP Traffic [9]

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

flow-record statistics

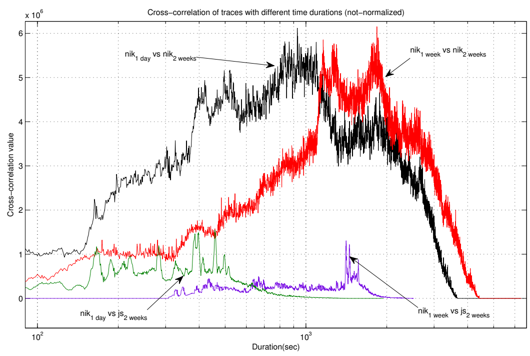

Figure 14: Cross Correlation of Traces with Varying Times [9]

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.

7.3 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 [15] we evaluated the compatibility of popular applications with such transitioning solutions: NAT64 [66] and Dual-Stack Lite [67]. The goal was to find potential failures by identifying application failure signatures left behind in the flow-record traces using Flowy. These failure signatures could later be used by service providers to automate the detection and eventually shorten the deployment verification cycle.

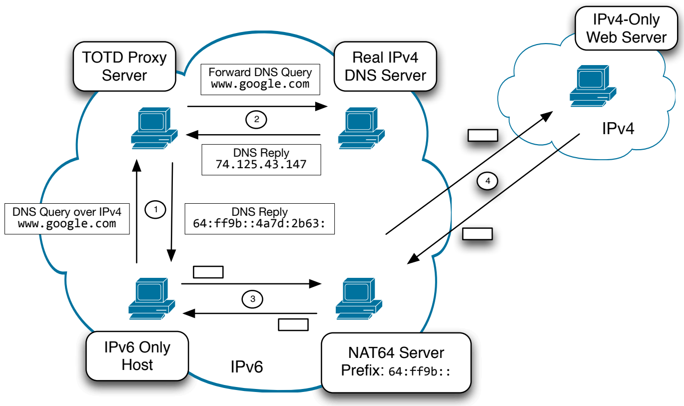

Figure 15: NAT64 Setup [15]

In the NAT64 deployment testbed as shown in figure 15, the authors witnessed failure in 3 applications: Skype, OpenVPN and Transmission. Flowy queries were defined to establish failure signatures for each application. A formalized Flowy query to identify Skype failure signature for an instance is described in listing 19. The filter sections of each branch are shown separately in listings 17 and 18.

application operation under NAT64

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

Listing 17: Branch A [15]

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

Listing 18: Branch B-C-D [15]

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 19: Skype Failure Signature [15]

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.

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).

failure when using IPv4 literals

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.

7.4 OPENFLOW

OpenFlow [68] is an open standards protocol that runs between an Ethernet switch and an OpenFlow controller (a software designed to run on a x86 server) to securely manage the forwarding plane of the switch over the network as shown in figure 16. This enables the controller to push out policies that dictate how to process flow-records crossing the networking infrastructure to eventually improve bandwidth, reduce latency and save power.

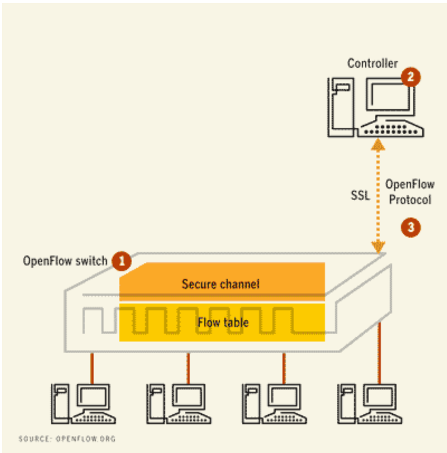

Figure 16: OpenFlow Architecture [16]

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 [69].

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.

software stack

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.

flowy and openflow

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.

7.5 FLOW LEVEL SPAM DETECTION

Feature	Description
Pkts	Packets
Rxmits	Retransmissions
RSTs	Packets with RST bit set
FINs	Packets with FIN bit set
Cwndo	Times 0-window advertised
CwndMin	Minimum window advertised
MaxIdle	Maximum idle time between packets
RTT	Initial round trip time estimate
JitterVar	Variance of inter-packet delay

Table 5: Features in Spam Flow [17]

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 [17, 70], the authors analyze the transport level characteristics of the email flows to differentiate between spam and legitimate email. These characteristics exploit the fundamental weakness of each spam: the requirements to send large amounts of the same email on resource constrained links owned by compromised botnets which is unlikely to change in the near future. They reason that a spammer's traffic is more likely to experience TCP timeouts, retransmissions, resets and variable Round Trip Time (RTT) estimates. Based on this hypothesis they extract 13 learning features as shown in table 5 to formalize a machine learning problem.

spamflow features

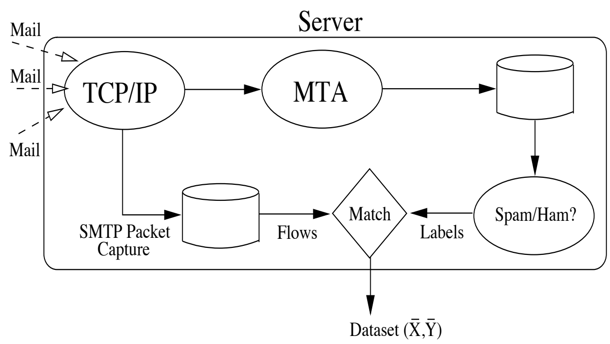

Figure 17: Spam Flow Classifier [17]

The data collection methodology is depicted in figure 17 where TCP packets corresponding to email messages are extracted and examined on a per-email flow basis. The packets in an email flow are coalesced together by using TCP port numbers in the email headers. Using machine learning feature selection, a spam classifier is built that matches each flow to a binary spam/ham ground-truth label. Support Vector Machines (SVMs) [71] are used for classification and Greedy Forward Fitting (FF) [72] is used for feature selection to find a set of features that provide the least training error. It turns out the classifier achieves 90% accuracy with 78% detectability of false-negatives from a particular content filter.

a spam classifier

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

flowy and spamflow

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

extending spamflow

Part III

WORK PLAN

The two implementations are branched off and currently live in their own parallel universe. The first task is to glue them to create a complete package that has the full-blown functionality and exploits the best of both worlds as discussed in section 5.4.1.

The custom C library that replaced PyTables can only process flow-records in `flow-tools` format. It needs to be extended to support `nfdump` format as discussed in section 5.2.2. `nfdump` will bring in the ability to read NetFlow v9 and thus IPv6 records. The possibility to further extend it to bleeding edge IPFIX support still needs to be investigated.

The grouper and the merger are the performance hit stages of the processing pipeline. Using quick sort and binary search, the complexity has been brought down to $O(n * k)$ for the grouper and $O(n^{m-1} * k)$ for the merger. However, having an actual search tree would bring it down to $O(n\log(n))$ as discussed in section 6.2.2 and 5.4.2.

The core C implementation has limited multithreading, which the merger currently cannot exploit. A `pthreads` wrapper is planned, that will create a thread pool and exploit the available cores as is discussed in section 5.4.4. The possibility to parallelize our prototype by making it mapreduce aware still needs to be investigated.

Several small prototype limitations and enhancements need to be looked into. For instance, the pipeline should be flexible to allow skipping of stages and the flow-query should be extended to allow more than one grouper and merger. The number of available operations also needs to be extended as discussed in section 5.4.5.

Part IV

APPENDIX

A

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

MPLS Multiprotocol Label Switching

RTT Round Trip Time

SVMs Support Vector Machines

FF Greedy Forward Fitting

SMTP Simple Mail Transfer Protocol

DDoS Distributed Denial of Service

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

RMON Remote Network Monitoring

MIB Management Information Base

SNMP Simple Network Management Protocol

RTFM Realtime Traffic Flow Measurement

GUI Graphical User Interface

IETF Internet Engineering Task Force

DoS Denial of Service

AS Autonomous Systems

CIDR Classless Inter-Domain Routing

SCTP Stream Control Transmission Protocol

PSAMP Packet Sampling

TLS Transport Layer Security

DTLS Datagram Transport Layer Security

IE Information Elements

IANA Internet Assigned Numbers Authority

PENs Private Enterprise Numbers

EP Exporter Process

CP Collector Process

SMI Structure of Managed Information

CLI Command Line Interface

XDR External Data Representation

BIBLIOGRAPHY

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

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

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

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

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

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

[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] 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.

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

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

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

[12] Dell, Texas, *Dell PowerConnect M6220, M6348, M8024, and M8024âk Switch Userâs Configuration Guide*.

[13] 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.

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

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

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

[17] 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.

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

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

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

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

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

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

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

[25] D. Schatzmann, W. Mühlbauer, T. Spyropoulos, and X. Dimitropoulos, "Digging into HTTPS: Flow-based Classification of

Webmail Traffic," in *Proceedings of the 10th annual conference on Internet measurement*, IMC '10, (New York, NY, USA), pp. 322–327, ACM, 2010.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[47] 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.

[48] 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.

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

[50] 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.

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

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

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

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

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

[61] 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.

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

[63] 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.

[64] 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.

[65] 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.

[66] 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.

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

[68] 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.

[69] 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.

[70] 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.

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

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