pmacct: steps forward interface counters

Paolo Lucente, paolo.lucente@ic.cnr.it

Abstract

Measuring and monitoring network traffic is of essential support to manage key operations in today's complex Internet backbones such as detecting hot spots, traffic engineering, usage-based pricing and gaining insight about the composition of the traffic mix. However, actual large-scale networks are able to produce, in very short times, huge amounts of data that quickly result difficult (either too slow or expensive) to be handled. Moreover, methods to log periodically randomly sampled packets have shortcomings (inaccuracy among the others) that hinder the analysis of traffic data; but sampling undoubtely allows to break the scalability barrier. Thus, a way to control carefully the produced dataset is required: the ability to select, aggregate and group IP flows plays a prime role in a today's network monitoring and measurement tool. This paper is about **pmacct**, a set of tools to select, account and aggregate IPv4 and IPv6 traffic; it endorses a simple aggregation method revolved around the basic concept of communication primitives (e.g. source and destination IP addresses, IP protocol, ToS/DSCP field): by arbitrarily reducing the set of primitives that unambiguously define a specific flow and by grouping flows logically, it is able to merge micro-flows into bigger macro-flows on the basis of the exploited similarities between them. Having summarized but still accurate reports of network activity, enables operators to gain a better understanding about the status of the observed network infrastructure. pmacct aims to be portable across Unix flavours and efficient in terms of resource management; to have minimal requirements to run and dependencies to satisfy; and finally to give meaningful insight by allowing to characterize the detail level of the produced dataset.

I. INTRODUCTION

IP networks are rapidly growing, covering almost every corner of the planet pushed also by their attractive wireless 'arm', getting faster and jointly stressed by the increasing demand for broadband access from research and educational institutions, offices, homes and co-location among the others in the global byte-hungry community. Moreover networks are plagued by a substantial background activity generated by Denial-of-Service attacks, viruses, and e-mail spam. The increasing number of users exposes the connected networks to an high replication factor of such activities and - thanks also to larger shares of bandwidth available per user (and a somewhat fervent creativity) - widens largely the tipology of applications in use. While the times of sudden hills in network graphs as unique fingerprint of past and unknown network behaviours are quite far in the memories, the actual ability to have deep knowledge of individual flows travelling back and forth the end-users is not of much help: the limit of a disruptive abundance of data to get through.

As a common result, for network operators is becoming even more important to obtain data which is ready to be analyzed or correlated by reporting and presentation applications in order to gain valuable insight about the network status, thus allowing to promptly extract the required informations. But because actual high-speed, large-scale networks produce, in short times, big amounts of raw data that cannot be quickly processed, it's becoming more important being able to control the size and the quality of the produced dataset. Traffic aggregation greatly alleviates the problem by reducing the set of primitives that define a specific flow and by

grouping flow primitives into larger network entities (for example individual IP addresses into either network prefixes or Autonomous Systems); this methodology allows to merge micro-flows into bigger macro-flows, also referred as aggregates, on the basis of the exploited similarities between them while continuing to preserve the required traffic detail. Intuitively, the new degree of similarity between the flows is established by the number of primitives into the reduced set. Summarized traffic reports are of valuable support for a range of critical operations like but not limited to determining the busiest segments of the network, thresholding sudden network events, monitoring the well-provisioning of the underlying infrastructure, SLA monitoring. Moreover, the ability to tag packets offers the opportunity for the deployment of innovative billing schemes (e.g., location-based) beyond the usage-based ones.

Some important requirements are at the root of a today's good passive monitoring software design: speed (the ability to deal smoothly high link rates), scalability (by allowing more sensors to work cooperatively in a distributed fashion) and an high degree of flexibility (by allowing to characterize the details of the produced dataset). pmacct has been developed with these goals in mind and more efforts have been pushed in developing and mantaining a clean architecture that allows for quick processing of incoming network data basing also over the observation that the computing speed will, in the very next future, no more compare favourably to network speed, further reducing the number of cycles available before the next packet arrives (the cycle budget) which is already almost minimal in today's networks[6].

II. BRIEF CONCEPTS

Let's give a definition of the primitives and their properties, applied to pmacct context:

- a primitive is a keyword that identifies a specific portion of the headers' stack of a packet,
- each keyword is unique into the alphabet: no two keys are identified by the same name,
- each portion of the stack is represented by just one keyword and viceversa,
- the portion identified by a keyword is strictly disjoint by any other portion identified by another keyword.

A *flow definition* is made of a group of primitives, plus few counters (e.g., bytes and packets counter). Let's now define the concept of *flow*: it is a group of packets that share the same instantiation of the flow definition. Basing on this, we will introduce the concept of *flow aggregation* as it is seen in pmacct. Let's give our first flow definition, μ*Flow*, the following way:

$$\mu Flow\{src_host, dst_host, ip_proto, src_port, dst_port, sum(bytes), sum(packets)\}$$

in real world, such grouping may represent a typical unidirectional group of packets transiting through the Internet and carrying a transport protocol like TCP or UDP. Let's assume, for the ease of our discussion, that values of both *src_host* and *dst_host* primitives are static; we will be able to obtain such behaviour by applying a filter (going practical again, such filter would enable us to count the packets transiting between two specific hosts). Then, generating some traffic for a time δ, we will populate *Set*α of *j* packets transiting between the two hosts; by applying our μ*Flow* definition to the produced $Set_j\alpha$, we will be able to classify its packets into an *x* number of flows:

$$Set_j\alpha(\mu Flow) = x$$

x is the cardinality of a new Setβ which is the set of all μFlows required to classificate the *j* packets into the Setα. Let's now give a new flow definition, λ*Flow*, which is effectively a superset definition of μFlow. Let's apply it to the produced $Set_x\beta$:

$$\lambda Flow\{src_host, dst_host, sum(bytes), sum(packets)\},$$

$$Set_x\beta(\lambda Flow) = 1$$

As a result, shown above, now we will be able to further classify μFlows in our Setβ into a single flow, λFlow, which represents the grand total of packets and bytes transferred between the two hosts. Therefore, we will term this unique flow an *aggregate* or macro-Flow and the incremental process of casting packets in a Setα into λFlows, an *aggregation process*. We may proceed with the above methodology to obtain arbitrary aggregates. However, it's important to notice that while the methodology itself allows to build arbitrary aggregates, in each software (thus, including pmacct) these are limited by the set of supported primitives (pmacct ones are listed in the Section X, Appendix A).

III. ARCHITECTURE OVERVIEW

pmacct daemons, *pmacctd* and *nfacctd*, sport a modular architecture which relies over a multi-process organization and a strict separation between the two main daemon operations: packet gathering and packet processing. The module which focuses on gathering packets from the netwrok is termed *core process* or simply core; it is also in charge of a few additional tasks including filtering, pre-tagging and sampling. One or more modules are employed in packet processing and are termed *plugin instances* or simply plugins. Each plugin communicates with the Core process, through a private shared memory segment which, along with its state variables and pointers, is termed *communication channel* or simply channel. It is structured as a circular queue of arbitrary size further divided in a number of chunks or *buffers*: as soon as a buffer gets filled up with fresh data, it's assigned to the plugin and the next buffer is used. The plugin is adviced that new data are available on the queue through an Out-Of-Band (OOB) signallation queue which is socket-based and is allocated a fraction of the size of the main queue. It's important to notice that not each buffer transiting from the Core process to the plugin is signalled through the OOB queue: the plugin, in fact, is able to get the Core process aware whether it's actually blocked waiting for new data to be available or it's still processing some backlogged buffer; if the Core process is able to fill the new buffer while the plugin is processing the previous one, it will not signal new buffer assignations: when the plugin will finish to process the old buffer, it will check for the arrival of new data before going to sleep again instead. While this mechanism have shown almost no benefits when few packets are processed, it has greatly helped in reducing the pressure over the kernel (because it effectively avoids to cross the userspace-kernel boundary) when exposed to heavy traffic rates.

The process of aggregating network data is distributed between the Core process and the plugin: the former applies the proper flow definition, the last is in charge of counters' accumulation and historical breakdown. In the next subsections we will see the working principles of the modules described above.

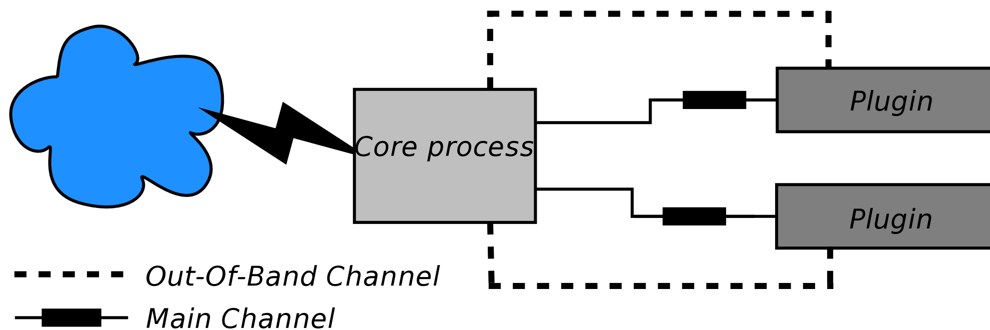

Fig. 1. daemon architecture overview

A. *Inside the Core process*

The Core process is broken in two parts: an upper one which collects data coming from the network and a lower one that handles plugin-related operations: it frames packets into aggregates, applies fine-grained filters, recollects resources if a plugin dies and has control on both (main and OOB) communication channels. While the last one is common for the two daemons - nfacctd and pmacctd - the former is distinct, thus implying the existence of two executables. This hard separation has been preferred following the consideration that each packet capturing framework (e.g., libpcap[1], [2]) and export protocol (e.g., NetFlow[4], sFlow) need peculiar operations in order to parse encapsulated data: for example, NetFlow v9[8] requires that packets have to be dissected basing over previously sent templates; libpcap instead captures a specified portion of each packet, prepending them an header which contains informations about the allocated buffer without offering any further support, so, IP packet fragmentation is entirely on the shoulder of the consumer application.

However, it's also important to notice few facts:

- the hard separation makes pmacct easily extendible to other export protocols, packet capturing frameworks and storage backends,
- because the lower part of the Core is common, it works effectively as an *abstraction layer*,
- writing a new Core will just require to write the new specific upper part,
- writing a new plugin, will just require to implement the common hooking interface to let it work in conjunction with all available Cores.

The Core process pipeline has been leaved as fast as possible, for example preferring pre-arranged routine pointers to conditionals, though few additional operations are handled on the critical path. The intuitive reasons beyond such choice are: a) the operation is of global scope thus doing it elsewhere would lead to resource wasting by repeating it multiple times unnecessarily, b) pushing an unuseful packet too deep in the pipeline stages would result in even worse effects.

1) The upper part, pmacctd: pmacctd acquires network traffic data using the well-known libpcap framework. pmacctd is also able to request filtering capabilities from the producer library: being usually placed directly into the kernel, this kind of filter is lightning fast; however its global scope makes this filtering tier of limited

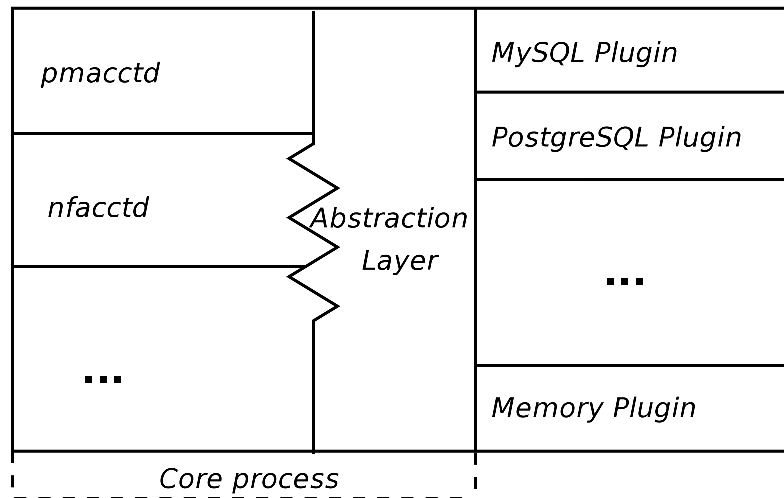

Fig. 2. Modularization overview

use. This is because the underlying abstraction layer supports a fine-grained second filtering tier.

The assembly line starts once a new packet is received; pmacctd validates it and sets base pointers to protocol headers until the transport layer (e.g., TCP, UDP): because almost all following operations will need to deal with specific header fields, such initial book-keeping activity will avoid later on searches through the full buffer body. The network layer (IPv4, IPv6) is handled properly, taking care of fragmentation; the newly arranged structure is then passed to the abstraction layer entry point, the *exec_plugins()* function.

2) The upper part, nfacctd: nfacctd acquires network traffic data by analyzing Cisco NetFlow packets sent to it from one or more exporting agents (e.g., NetFlow-enabled network equipments, NetFlow probes). The daemon first checks whether the agent is allowed to talk to it, then applies Pre-Tagging (which will be discussed more in deep later): it briefly allows to assign an *ID* (a small positive integer) comparing one or multiple fields of the NetFlow packet, such as the incoming or outcoming interface or the engine that have generated the packet, with an user supplied *Pre-Tag Map*. nfacctd is able to handle multiple versions of the NetFlow protocol at once so that not all exporters are forced to talk the same version. Once the NetFlow packet is validated and its version is recognized successfully, it is dissected by the proper routine. Each flow extracted successfully is then sent to the abstraction layer entry point, the *exec_plugins()* function.

3) The lower part, the abstraction layer: all operations in this layer are based over the set of pointers arranged by the upper part. Apart from the architectural consideration already done previously, the main tasks of the abstraction layer are to frame incoming data into aggregates, apply the second tier filters and feed the buffers to the attached plugins, turning in a round-robin fashion among the active circular queues. The flexible filtering capabilities available here allow to assign each plugin two different filters, one that matches Pre-Tags (if any) and the other that matches against a libpcap expression.

B. Inside the Plugins

Each plugin reads the packed buffers - containing the aggregates - sent by the Core process from its communication channel. The reception is quite straightforward: the plugin blocks until the arrival of new data is signalled through the OOB channel; once a new buffer is ready to be processed, the aggregates contained into it are processed sequentially. Then, before going to sit again, waiting for a new signal, the plugin will check whether a new buffer has already arrived in the meanwhile; if this is the case it will get processed immediately. The plugins are differentiated by the backend they use to store data. The following sections will present an overview of the In-Memory Table (IMT) plugin and the SQL one[1].

1) the In-Memory Table (IMT) plugin: stores aggregates into a memory structure, organized as an hash table. Such table is divided in a number of buckets and aggregates are direct-mapped to a bucket by the mean of a modulo function. Collisions in each bucket are solved building collision chains, organized as linked-lists. An auxiliar structure, a LSU cache (Last Recently Used), is provided to speed up searches and updates into the main table. The LSU saves last updated or searched element for each bucket: when a new operation on the bucket is required, the LSU cache is compared first; if it doesn't match, the collision chain gets traversed instead. Memory is requested from the underlying operating system in large chunks, called *memory pools*, to limit as possible the bad effects (e.g., trashing) that could derive from the dispersion through the memory pages of much more frequent tiny allocations. Memory pools are tracked via a linked list of descriptors to ease maintenance operations such as freeing unused memory.

Data stored into the memory structure can be accessed by a client tool communicating with the daemon through a Unix Domain socket. The queries may be atomic (they contain just a single request) or batch, allowing a single query to encapsulate up to 4096 requests. The available query types are 'bulk data retrieval', 'group data retrieval' (partial match), 'single entry retrieval' (exact match) and 'erase table'. Additionally both partial and full matches may supply a request for resetting the counters for the matched entries. The client query is evaluated by the plugin: requests that need just a short stroll through the memory structure are served by the plugin itself, the others (for example batch queries or bulk data retrieval) are fulfilled by a new process spawned by the plugin. Moreover, locks have been implemented to guarantee a successful cohesistence of long-lived queries and mutual-exclusive operations like full table erasure. Some batch queries may be fragmented by the operating system because exceeding the actual socket size. They will require a reassembly process for which is in charge the plugin as soon as it receives a `\x4` End of Message placeholder. Whether an incomplete message is received, it is discarded when the associated timeout expires.

2) the SQL plugin: stores aggregates into a direct-mapped cache, organized as an hash table. Such mapping is computed via a modulo function; if the bucket already contains valid data, the conflict is solved with the use of collision chains: the chain gets traversed and whether it does not contain any unused element, a new one is appended at the tail. New nodes are allocated exploring two chances: if any node has been marked stale (it happens when an allocated node is unused for some consecutive timeslots) it's reused by unlinking it from its

[1]As today, two distinct SQL plugins enable pmacct to operate with MySQL and PostgreSQL databases. However from the perspective of this document, this difference is not relevant. This is because we will refer to a generic *SQL plugin*.

old chain and then linking it to the current one; if no free nodes are available then a new one is created. Stale nodes are then retired (that is, unallocated) if they still remain unused for longer times (RETIRE TIME**2). To speed up lookups of candidate nodes for reallocation and retirement, an additional LRU list of allocated nodes is mantained.

The cache is also used for accumulation of counters and their historical breakdown. Aggregates are pushed into the DB at regular intervals; to speed up such operation, a queue of all pending aggregates is mantained as nodes are touched (either used or reused), avoiding long walks throught the whole memory structure. Actual data (aggregates) are framed into SQL queries which are sent to the DB. Because in this moment is not known whether an INSERT query would create a duplicate, an UPDATE query is launched first and if no rows are affected, the INSERT query is trapped. pmacct allows to revert this default behavior (it could be correct under certain circumstances), skipping directly to the INSERT query. When the purging event is finished, aggregates in the cache are not unallocated but simply marked as invalid: while data coherence is preserved, we greatly avoid to waste CPU cycles.

The SQL approach is undoubtely fashinating because it opens new chances for advanced operations like data correlation, trapping of alerts by thresholding specific network events, etc. However the added value of SQL flexibility has an high price in terms of performance constraints and resources consumed. This argument will be described in the following section *The SQL limits*.

Moreover, the health of the database server is ensured by checking for the successful result of each SQL query. Whether the DB becomes unresponsive, a recovery flag is raised. It remains valid, avoiding further checks, for the entire purging event. If transactions are being involved, an additional reprocess flag is raised: it signals that all SQL queries already sent successfully have to be reprocessed because the actual transaction will never get finalized. The actions available to recover data are two:

- aggregates are written into a structured logfile saved on the disk: a logfile is made of (a) an header which contains some configuration parameters, (b) a template header which contains the description of the record structure, followed by (c) records dumped by the plugin; the logfile may be handled a posteriori by the mean of the two player tools available: *pmmyplay* and *pmpgplay*. The template has been thought to solve backward compatibility issues: in fact, the record structure is very likely to change over the time, as new primitives may be introduced.
- aggregates are written to a backup database server.

IV. Tagging packets

The ability to tag aggregates is undoubtely one of the most powerful features offered by pmacct. Tagging is broken down into two parts, called *Pre-Tag* and *Post-Tag*. Pre-Tagging is done in the upper part of the Core proccess and it's actually available only in nfacctd while it's leaved as future work its implementation in pmacctd; Post-Tagging is done into the lower part - the abstraction layer - thus it works for both daemons. Pre-Tagging is done shortly after the NetFlow packet has been validated and unrolled: by looking up an user-supplied Pre-Tag map and comparing the map values with those in the NetFlow packet, nfacctd assigns a small positive integer, called *ID*, to the flow currently analyzed. Also valuable are the added Pre-Tag filtering

capabilities available in the abstraction layer. Post-Tag is not based on any lookup but fixed and assigned to a specific plugin so that each aggregate assigned to it, is marked properly. The Post-Tagging stage is reached after the aggregate has passed all filters, if any.

V. Filtering packets

Each active plugin is applied an aggregate definition. But it's important being able to control which traffic is assigned to which plugin; it could be desirable to apply the aggregate definition only to a specific fraction of the network traffic; this is a rather common case even in the most simple scenarios: for example, to split incoming from outcoming traffic it's required a pair of reverse filters. Core process abstraction layer sports two kind of filters aimed to satisfy such needs: one matches a libpcap expresson, the other matches the Pre-Tag assigned by the upper part. This last filter, because of the intrinsic logical value of a Pre-Tag, is greatly useful allowing to deal with even more advanced scenarios than the one presented above; in fact, by the mean of a simple integer comparison, it allows to spread the aggregates among the plugins, for example, to generate traffic matrices between interfaces, intercept the traffic directed to a specific link or to point out how multiple interfaces compare in terms of traffic generated.

VI. The SQL limits

SQL gives unparalled chances to combine, compare and analyze data but exposes some hard limits when applied to actual network traffic rates. Mantaining the database consistent and healty is a prime need and requires the use of mechanisms like transactions and indexes that have a considerable impact in terms of resource consumption and which impact may also scale negatively when the number of tuples stored into the database itself become huge. This is because both SQL plugins sport a memory cache; however often it does not suffice: in fact the cost of building and firing over the network a tiny packet (e.g., by a malicious user) is much cheaper compared to the cost of accounting such packet into the database. pmacct has a way to preprocess the purging event - when the memory cache is purged and SQL queries are created - effectively thresholding the impact that an enormous amount of SQL queries may have on the database. The queries may be just discarded or recovered temporarily to the disk, allowing to be analyzed, processed and finally sent to the DB a posteriori. This quantitative method has been preferred - leaving new approaches as future works - being not fully convinced by its direct alternative, a simple-systematic tail-dropping one (e.g., pick the first N *toptalkers*): thresholding hardly some specific aggregate value may alter consistently the quality of the dataset.

VII. Future works

While many efforts have already been spent to let the software to converge quickly to a stable and usable phase, much work remains still to do. Experimental results have demonstrated, as result of previous works[3], that a small number of heavy talkers account the largest share of traffic: this leave great chances to novel paradigms that concentrate only on macro-flows, defined in this context as those flows that exceed some threshold when compared to the total bandwidth available at a specific link. Such methods may endorse the concept of scalability while limiting the loss of accuracy[5], [7]. Moreover, recently there has been a wide

flourishing of interesting works in fields related to the network passive monitoring such as promising packet capturing frameworks and exporting protocols (e.g., IPFIX[10], [9]). The perspective would be to integrate pmacct with them.

Some interesting issues are still in the wild either requiring a comprehensive solution or further experimental results which may in turn drag to novel approaches; among them the SQL barrier, a refined sampling tier and a content-based Pre-Tagging scheme. A separate paper will cover broadly - in very next future - sampling topics and scenarios which are currently under evaluation.

VIII. Availability

pmacct is distributed free of charge and under GPLv2 licence. It can be downloaded from the pmacct homepage: `http://www.ba.cnr.it/~paolo/pmacct/` . Some Linux distributions and FreeBSD have already included pmacct in their userland software.

IX. Acknowledgments

The author would like to thank Martin Anderberg, Gianluca Guida, Wim Kerkhoff and the many users who have provided valuable comments, suggestions and support during the work.

X. Appendix A

The list of primitive codes supported by pmacct, along with their description is shown in the following table:

Primitive code	Description
src_mac	Source MAC (Physical) address
dst_mac	Destination MAC (Physical) address
vlan	VLAN id
src_host	Source IP address
dst_host	Destination IP address
tos	DSCP / IPv4 Type of Service / IPv6 Class of Service
proto	Transport layer protocol
src_port	Source UDP/TCP port
dst_port	Destination UDP/TCP port
src_net*	Source network prefix
dst_net*	Destination network prefix
src_as*	Source Autonomous system
dst_as*	Destination Autonomous system
sum_host!	Sum of incoming and outcoming traffic per IP address
sum_net*!	Sum of incoming and outcoming traffic per network prefix
sum_as*!	Sum of incoming and outcoming traffic per Autonomous system
sum_port!	Sum of incoming and outcoming traffic per UDP/TCP port
none	Enable no primitives: make global counters per interface

* The primitive requires requires the definition of a networks lookup map to work correctly.

! The primitive is mutually exclusive.

REFERENCES

[1] S. McCanne and V. Jacobson. The BSD Packet Filter: A new architecture for user-level packet capture, in *1993 Winter USENIX conference*, Jan. 1993.

[2] V. Jacobson, S. McCanne and C. Leres. libpcap, *The pcap(3) manual page*, LBL, Berkeley, CA, Oct. 1997.

[3] C. Estan and G. Varghese. New directions in traffic measurement and accounting, in *ACM SIGCOMM Internet Measurement Workshop*, Nov. 2001.

[4] Cisco Systems, Cisco NetFlow. `http://www.cisco.com/warp/public/732/Tech/netflow`, 2003.

[5] Sampled Cisco NetFlow, in *Cisco IOS release 12.0S*, Aug. 2003.

[6] H. Bos and G. Portokalidis. *FFPF: Fairly Fast Packet Filters* Technical report, Jan. 2004.

[7] C. Estan, K. Keys, D. Moore and G. Varghese. Building a better NetFlow, in *ACM SIGCOMM*, Aug. 2004.

[8] B. Claise, Cisco Systems NetFlow Services Export Version 9. *RFC 3954*, Oct. 2004.

[9] S. Leinen, Evaluation of candidate protocols for IP Flow Information eXport. *RFC 3955*, Oct. 2004.

[10] N. Brownlee and D. Plonka. *IP flow information export (ipfix)*. IETF working group.