This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



# Why Are My Flows Different? A Tutorial on Flow Exporters

Gernot Vormayr, Joachim Fabini, and Tanja Zseby

***Abstract*—Network flows build the basis of modern network data analysis by aggregating properties of network packets with common characteristics. A consistent and unambiguous definition of the network flow concept is an indispensable prerequisite and starting point for reproducible network research. However, in today's practice, the flow output of distinct flow exporters, which is software to generate flows from observed network packets, varies substantially on identical network packet stream input.**

**In this paper we present an in-depth comparison of different flow exporters and show how their outputs differ significantly. We argue that this substantially impairs reproducibility for traffic analysis research. We first present the detailed flow definition of the IP Flow Information eXport (IPFIX) standard including explanations and examples, analyze design and implementation of existing flow exporters, and explore the reasons why many projects and publications chose to implement their own flow exporters. Based on this analysis we highlight the main challenges in the flow exporting process and present a detailed tutorial on how to design and implement a flow exporter such that it yields consistent, reproducible output. Based on the tutorial's theoretical analysis and lessons learned we present design and main concepts of a versatile, flexible, and open source flow exporting solution called *go-flows* that generates deterministic, reproducible network flows. Finally, we present a flow-by-flow comparison of the analyzed flow exporters' output, explore the differences in terms of their generated flows, compare flow exporter performance, and conclude with guidelines on parameters that play a crucial role in improving the reproducibility of exported flows.**

***Index Terms*—Flow export, network monitoring, Internet measurements, IPFIX**

## I. Introduction

NETWORK data analysis is ubiquitous in modern computer networks. It is needed for network debugging, accounting, the provisioning and auditing of Quality of Service (QoS), and the detection of network intrusions. Current research mainly focuses on methods for anomaly detection, attack detection, and traffic classification.

Common to network data analysis is that some form of network data is required as input for the analysis. As seen in Fig. 1, this can either be in the form of network packets, device statistics, or network flows, irrespective of the network data analysis type. Network packets can be observed directly on the wire and, therefore, the contents of these packets (user data, also called payload, and control information) can be captured directly from network interfaces. Device statistics are

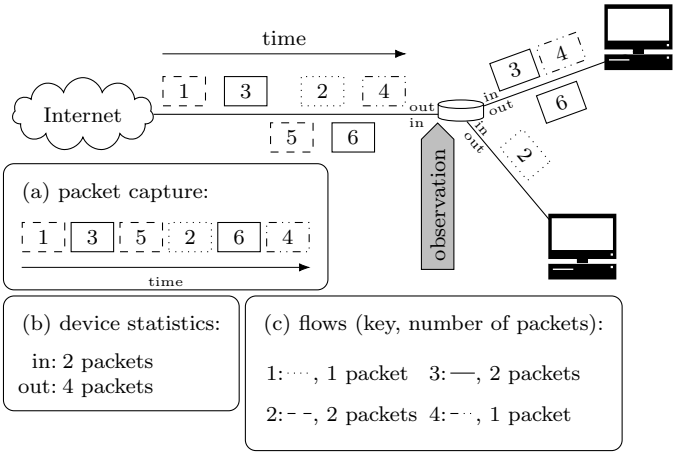


Figure 1. Network data representation forms, as observed in a small network connected to the Internet via a forwarding device. The upper host sends packets 4 and 3, and receives packet 6, while the lower host sends packet 2. Packet 1 is sent and packet 5 received by the forwarding device in the middle. The following observations can be made at the observation point in the network: (a) packet capture: contents (control information and, optionally, payload) of every network packet is recorded, (b) device statistics: only statistics of packets passing the observation point are recorded, and (c) flows: packets belonging to a flow (common property is the outline style of the packet) are aggregated (number of packets in this example).

counters (e.g., number of packets, number of bytes) aggregated from network packets that pass a single or multiple points of observation in the network (e.g., network interface, firewall). Full network packets give a complete picture of the whole communication, while device counters are the aggregation of network packet properties. Thus, device counters don't provide a detailed picture, like full network packets do, but on the other hand demands in terms of storage and processing power are vastly decreased.

Network flows provide a middle ground between full network packet analysis and device statistics. Analog to device statistics, flows are an aggregation of properties of network packets with common characteristics. Examples for aggregations are number of packets or number of transmitted bytes, while typical common characteristics are network addresses or transport ports.

Therefore, flows provide a better resolution since one flow consists only of a subset of packets passing a single point of observation in the network. Depending on which properties are used for combining packets to flows and which packet properties are aggregated, the resolution of the flow data can cover the full range from packet data (no common properties)

Manuscript received ...

G. Vormayr, J. Fabini, and T. Zseby are with Institute of Telecommunications, TU Wien, Vienna, Austria (email: {first.last}@tuwien.ac.at)

Digital Object Identifier ...

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



to device counters (network device as common property).

Network flows can therefore be tuned precisely to the required network data analysis requirement by selecting the used properties. For instance, time the flow occurred, flow duration, i.e., the time difference between first packet to last packet of a flow, and total number of bytes can be useful for network accounting and profiling.

Another example is detecting network based attacks, e.g., a Denial of Service (DOS) attack, where an attacker sends many unrelated packets with special characteristics that would overload a service, e.g., random packets, where each would potentially open a connection without sending any follow-up packets. This would result in many flows, each containing only one packet, while the legitimate ones would contain multiple packets. Analyzing just the network packets would make this task very hard, since it is nearly impossible to distinguishing between legitimate connection attempts and fake ones. Device counters can't give the full picture either due to a high number of packets or connection attempts being a possible result of a high number of users accessing the service.

Likewise, network flows enable easier passive QoS monitoring. For inferring parameters like two-way delay, packet retransmissions, and bandwidth, related packets must be matched, which is provided by network flows.

Applications or devices capable of aggregating packets to flows are called *flow exporters*. These flow exporters read packets from a source, which can either be live network traffic or network packets stored in a network trace. After packet input, the flow exporter has to decode the packet to acquire the packet properties. Next packets featuring common properties must be grouped and the desired properties aggregated to form flow records, i.e., a list of values per flow. These flow records are the output of the flow exporter, which can either be stored in a file or transmitted via a network. This data can then be used as basis for the required network data analysis task.

Several commercial and open-source variants with differing options and goals are available. Most flow exporters have their origin in network accounting or are a by-product of existing network functions (e.g., packet switching already groups packets by common properties – in this case packet source and destination). This severely limits their flexibility and makes them unsuitable for exploring novel aggregation properties or common characteristics required for research and will be explored in Section III.

This lead to the development of one-off or highly specialized flow exporter implementations employed in current research, which will be discussed in Section IV. In many scientific works researchers use own flow exporters, but the flow exporter is just a necessary tool and not the focus of the research. Therefore, details about the flow exporting process are often missing. This leads to non-reproducible and non-comparable results, as we will show later. Furthermore, although flow exporting is only a small task, the flows form the basis for the rest of the network analysis. Differences in aggregated value interpretations (e.g., which parts of a packet does the length encompass?) or mistakes in flow exporting (e.g., packet property errors, where does a flow end?) can therefore influence the results of the overall work. This makes it difficult to compare results produced with different flow exporters.

To improve future research, we provide this tutorial covering the entire range of flow exporting from flow definition and formats, the exploration and evaluation of existing flow exporters, to a complete tutorial for building a custom flow exporter including a reference implementation called *go-flows*. Additionally, we explore output and performance of the analyzed flow exporters and the reference implementation uncovering substantial differences that show severe implications on reproducibility and comparability of results.

### A. Contents and Structure of the Paper

The thorough review and detailed analysis of existing general-purpose and tailored flow exporters yield insights on why different implementations generate distinct flows for the same dataset despite using the same flow definitions. As a viable countermeasure we propose guidelines on how to export flows from packets in a complete in-depth tutorial that aims at supporting the reproducibility of future network data analysis. This tutorial is complemented by the introduction of a flexible and versatile open-source flow exporter named *go-flows* whose design and implementation bases on the tutorial's lessons learned. Finally, the paper explores differences in the flow output of existing flow exporting solutions and *go-flows* for the same publicly available input dataset in a detailed flow-by-flow and runtime comparison.

Since, depending on the targeted network analysis, not every aspect might be of interest to the reader, the paper is structured into the following four parts:

(a) Network flow description in Section II, including a historic overview of network flows, which describes the origins of the used definitions, as well as commonly used flow options, followed by the network flow definition based on the IP Flow Information eXport (IPFIX) standard including detailed explanations and examples. Section II contains an in-depth description of commonly used network-based and file-based network flow representations.

(b) A comprehensive analysis and comparison of existing flow exporters in Section III. Additionally, an investigation into reasons why custom-built flow exporters are deemed necessary is presented in Section IV, along with potential challenges and pitfalls that are inherent to such implementations, by examining existing scientific publications in the network analysis domain.

(c) A complete tutorial in Section V covering in detail the required steps, in particular: network packet acquisition, decoding, flow extraction, property aggregation, and record output. This is followed by the description of the reference implementation *go-flows*, which extends the tutorial with topics on performance considerations, flexibility, and reproducibility in Section VI.

(d) The final part in Section VII contains an in-depth flow-by-flow evaluation of the output of every flow exporter analyzed in Section III including *go-flows*. Included with the evaluation are steps required for using the different

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



flow exporters, common differences on the basis of examples, and root analysis of the observed issues and discrepancies.

Part (a) is recommended for all readers, since it contains the base definitions used throughout the remaining paper.

Developers interested in implementing a new flow exporter can use part (c) as a complete tutorial and starting point, since it covers each aspect of flow exporting. We also recommend part (d), which contains possible issues that can arise during the flow exporting process on the basis of existing implementations.

Researchers interested in finding a suitable flow exporter for their work should read the analyzed flow exporters in part (b) and the evaluation in part (d). If those do not fulfill the requirements, the flexible reference implementation described in Section VI should be considered. As a last resort, the tutorial in Section V can be used to implement a full-fledged flow exporting solution. However, we recommend extending *go-flows* with the required functionality via the provided extensive plugin architecture, which allows limiting the required development effort, increases interoperability, and supports reproducibility due to the open source nature of the project. Section VIII contains a summary of recommended guidelines to improve future scientific work.

Network operators can use part (d) as a reference for exploring flow exporting solutions and as a starting point for finding the cause of unexpected results or identifying issues in existing flow exporters.

### B. Related Work

Scientific publications analyzing the process of aggregating flows from packet data is scarce due to the abundant usage of existing flow exporters. Yet descriptions of those flow exporters are scarce. Even published works about those developed flow exporters highlight only parts of the flow exporting process. For example, Lampert et al. describe the flow exporter called *vermont* in [1], however, they present only a very high level architectural overview. Another publication covering a specific flow exporter by Inacio et al. [2] highlights only specific parts, like the flow table or using the packet timestamp as time base, of the flow aggregation process. A detailed analysis of existing flow exporters follows in Section III.

Most network data analysis publications that, however, do utilize a custom-built flow exporter don't describe the used flow exporting process due to the main topic of the publication being the analysis task. Additionally, crucial details regarding the flow-exporting process are often missing, therefore, limiting reproducibility. A selection of those works, including possible reasons why a custom flow exporter was created, will be discussed in Section IV.

Related publications covering only network flow formats, e.g., Brownlee et al. [3], who described IPFIX, present only the flow format itself. A comprehensive tutorial into flow monitoring was given by Hofstede et al. [4], which describes the whole process from packet observation, flow metering & export, data collection, to data analysis. However, due to the broad focus, only a high level overview of the flow exporting process is included, leaving out details crucial to implementing a custom exporter. Additionally, existing flow exporters were only analyzed based on documented basic features, but not their output.

Haddadi et al. [5] analyzed the effect of using different flow exporters in network analysis. However, the differing results were only analyzed based on the different flow properties supported by the flow exporters. Analysis with a common subset of supported flow properties to compare the differences in flow output was not attempted.

In this work we include a comprehensive analysis of existing flow exporters in Section III, including a more detailed and research focused overview of capabilities and possible extensibility in Table II, which enables selecting the suitable flow exporter for the given task, if possible. Additionally, a thorough analysis of flow output and performance of the analyzed flow exporters, and, additionally, the exporter *go-flows*, which was built in tandem with the tutorial, is presented in Section VII, showing major disparity in output and performance numbers.

## II. Network Flows

As described in Section I and Fig. 1, network flows provide a middle ground between network packet analysis and device counters by aggregating packets with common properties.

The section starts with a historic overview of network flows, describing origins of used terms and commonly used settings in Section II-A, followed by a detailed description of the flow definition given by the IPFIX standard in Section II-B. This is accompanied by a list of flow advantages in Section II-C, and, finally, network-based and file-based flow storage formats in Section II-C.

### A. Network Flow History

Early definitions of flows date back to 1991, where flows were proposed for accounting. In Request For Comments (RFC) 1272 [6] packets were assigned to flows by executing a set of rules. Depending on the desired granularity these rules could contain transport ports, or host or network addresses. Suggested aggregated values were number of packets, number of bytes, and start and stop time of the flow.

In 1997 this was refined by RFC 2063 [7], where a traffic flow was defined as an equivalent to a call or connection, meaning a stream of packets between two end points. In this proposed standard the term *flow key* was used for describing the flow a packet belongs to. It could consist of addresses used in the different network layers of a packet (e.g., network addresses, transport port numbers). Since data can flow in both directions, resulting in differing flow keys due to the reversing of source and destination address, the forward and reverse key must be matched if *bidirectional flows* are needed. These are flows where packets in the forward and backward direction are part of one flow. Additionally, packets could be filtered and ignored for the flow aggregation. Flow attributes were defined as aggregates of quantities reflecting events that occurred during the flow duration. The flow duration could be limited by a timeout.

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



Since publications and flow implementations treated flows and connections synonymously in the past, the use of the *5-tuple*:

- source network address,
- destination network address,
- transport protocol identifier,
- source transport port,
- and destination transport port,

became popular and is still in widespread use today. Since the 5-tuple is used to identify connections in hosts unambiguously, using it as the flow key, aggregates packets to a single flow that form a single connection. This even holds for techniques like Network Address Translation (NAT), which hides several hosts behind a single Internet Protocol (IP) address, therefore, sharing said address by rewriting outgoing packets to contain the shared source network address and related incoming packets to contain the correct address again. To ensure that the connections of the hosts don't interfere with each other, the host implementing NAT must ensure that connections from two different hosts don't share the same source port number. This could result in the same 5-tuple for connections from two different hidden hosts, if both connect to the same service at the same time. The host implementing NAT prevents this, by not only changing the source network address, but also the source transport port. Therefore, using the 5-tuple always results in flows representing connections. In flow exporters supporting both IPv4 and IPv6 network addresses, those would result in distinct flows due to the addresses being sized differently ($32\,\mathrm{bit}$ for v4 and $128\,\mathrm{bit}$ for v6).

In 2001 the IPFIX work group started publishing requirements for future network flow definitions and a protocol for transporting flow data, which was finalized in 2004 as RFC 3917 [8]. This resulted in the proposed standard RFC 5101 [9] (IPFIX), published in 2008. It is loosely based on the proprietary NetFlow v9 format (RFC 3954 [10]), extending it with a high grade of flexibility and very detailed definitions of flow related terminology, which is explained in Section II-B.

During the development of IPFIX even flow unrelated protocol standards tried to take flows into account. For instance RFC 3697 [11] proposes to use the *flow label* header in IP version 6 instead of the *transport ports*, to help with flow aggregation if decoding would be expensive or if the transport layer is encrypted, which would make decoding the transport layer impossible.

The accompanying proposed standard RFC 5103 [12], also published in 2008, extends the IPFIX definition with considerations for bidirectional flows.

The current version of IPFIX was promoted to Internet standard RFC 7011 [13] in 2013, which replaced the old RFC 5101 [9].

Future work includes the proposed standard for flow selection methods RFC 7014 [14], which describes limiting the data usage of flows even further with sampling. This results in using only a subset of the captured flows.

A more detailed history of the protocol was described by Hofstede et al. in [4].

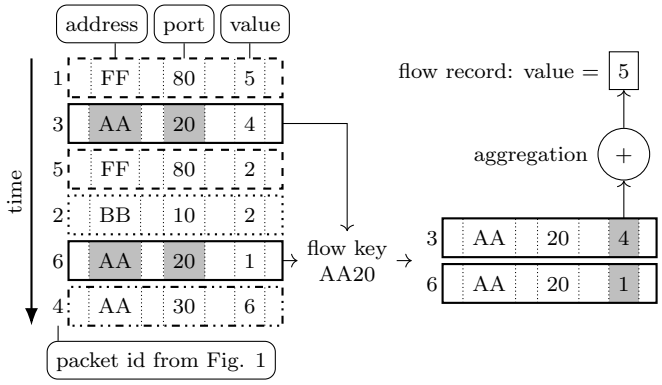


Figure 2. Flow aggregation example based on the packets from Fig. 1 with common properties (flow key) *address* and *port*, and exported property *value*. First, packets matching the flow key are coalesced to a flow. Afterwards, the packet property is aggregated with an aggregation function (*sum* in this example), resulting in the flow property.

### B. Network Flow Definition

The most frequently used definition for flows today is the flow definition by the IPFIX standard:

RFC 7011 [13] defines a *network flow* as the aggregation of network packets that share a set of common properties (see Fig. 2). These properties can be header fields from any layer (e.g., network source address, protocol type), characteristics of the packet itself (e.g., packet length), or fields derived from packet treatment (e.g., next hop address). This set of common properties is called a *flow key*.

Selected properties of the set of packets belonging to a flow can be aggregated into a so called *flow record*. See Fig. 2 for an example of such an aggregation.

According to RFC 5102 [15] a flow *expires* (further packets matching the same flow key start a new, unrelated flow) on detecting:

- an event derived from a specific packet within the flow, called *end of Flow detected*, (e.g., encountered Reset (RST) flag in a Transport Control Protocol (TCP) flow indicating the termination of the connection [16]),
- an *idle timeout*, which is the maximum time during which no packets with the same flow key are observed,
- an *active timeout*, which is the maximum time a flow is allowed to last,
- an event inside the flow exporter, called *lack of resources*, (e.g., out of memory), or
- an event external to the flow exporter, called *forced end*, (e.g., forced shutdown, end of input file).

Without flow expiry, flows would have an endless duration. Some existing protocols define flags or special sequences in packets indicating the end of a connection (e.g., RST and Finish (FIN) flags in TCP [16]). For such protocols, the flow exporter can use these indicators to expire flows, resulting in flows having a rough relation to connections (see Fig. 3 (a) for an example).

If indication for expiring a flow can't be obtained from packets (e.g., protocol without end marker like User Datagram Protocol (UDP) [17], due to the use of encryption, or due to the requirement of flows having no relation to connections),

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



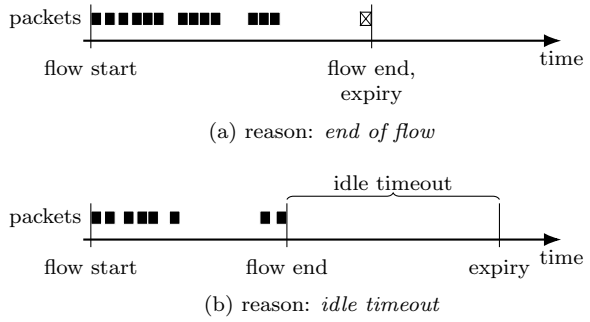


Figure 3. Flow expiry example. In (a) a packet containing an end marker (e.g., RST in a TCP connection [16]) is received, leading to an expiry of the flow at this point in time. The flow properties *flow start* mark the timestamp of the first packet and *flow end* the timestamp of the last packet. In (b) no packet indicates the end of the flow. After the last packet of the flow was encountered and the *idle timeout* passed, the flow is expired, leading to different *expiry* and *flow end* times.

Table I
COMMON FLOW PROPERTIES LISTED BY CATEGORY.

| | name | description |
|---|---|---|
| flow | flow start | timestamp of first packet |
| | flow end | timestamp of last packet |
| | flow end reason | reason why flow was expired |
| packet | source network address | source in network layer |
| | destination network address | destination in network layer |
| | transport protocol number | protocol type of transport layer |
| | source transport port | source in transport layer |
| | destination transport port | destination in transport layer |
| statistic | number of packets | sum of packets |
| | flow length | sum of transmitted bytes |
| | minimum length | length of smallest packet |
| | maximum length | length of biggest packet |
| | average length | average packet length |

an *idle timeout* can be used. An example for an idle timeout can be seen in Fig. 3 (b).

Further limits like *active timeout* and *lack of resources* might be necessary to limit the maximum number of concurrent flows. This might be required by limitations in the available computing environment, like maximum available memory.

Flows are expired as soon as the first expiry condition is encountered. This means, for example, that even flows comprised of packets that would allow for packet-based expiry can be expired due to a timeout event.

After expiry, the flow is finished, i.e., no further packets can be added to the flow. Eventual out of order packets arriving after expiry would start a new flow.

Due to performance optimizations or requirements in the computing environment, the actual time the flow record is exported, which means the flow record data is written to the result file, might occur at a later point in time.

RFC 5101 [9] defined a base set of flow properties, which was superseded by the Internet Assigned Numbers Authority (IANA) maintained list *IPFIX Entities* [18]. A small set of example flow properties is listed in Table I. The properties can be assigned to the following categories:

- **Flow-based:** properties of the flow itself, e.g., the reason the flow was expired, or the property of a significant packet in the flow (e.g., *flow start* in Fig. 3).
- **Packet-based:** common properties of all the packets in the flow, e.g., source network address, or transport protocol number (e.g., *address* in Fig. 2).
- **Statistics-based:** aggregation of packet properties, e.g., sum of transmitted bytes, minimum packet size in the flow (e.g., the aggregated *value* in Fig. 2).

Although the flow aggregation requires an additional step in the data preparation, it offers advantages over packet-based approaches.

### C. Flow Advantages

Flow aggregation *reduces the amount of data* needed for analysis [19]. Therefore, necessary computing power and, eventually, data storage needs are significantly lowered. For instance Hofstede et al. [4] report decreased storage requirements by a factor of 2000. Although this data reduction means a loss of information, it is still possible to keep the information dictated by the requirements of the analysis problem [20]. This can be caused by the increasing usage of encryption, which makes the packet payload unusable for analysis, anyhow.

This data reduction technique enabled simple network data analysis as early as 1990, where available computing power would have prohibited such an exercise at the packet level [21]. Frank et al. [22] enhanced this approach with machine learning as early as 1994.

Another advantage of network flows is that they innately contain less identifying information than packets (e.g., no payload). Therefore, it is easier to *anonymize flow data* [20].

Additionally, due to communication in current computer networks between applications not fitting into single packets and requiring complicated inter packet relationships, *flows are a better representation format* compared to packets. Where traditional port-based techniques fail, the added value of analyzing related packets allows the investigation of evolving computer networks and applications [23]–[25]. This even allows analyzing encrypted traffic where the payload is obfuscated [23]–[25].

### D. Flow Formats

Flow records can be exported either via a network-based protocol or stored in files. Some of the flow formats can be used for both network transmission and writing to files. Since several of these formats and protocols also dictate parts of the flow aggregation, e.g., possible flow keys, how flows are aggregated, or possible flow record values, not every format can be used in every situation.

*1) Network-based Flow Formats:* One of the earliest network-based protocols is the Cisco proprietary binary-based NetFlow, for which version 1 (v1) was introduced in 1996 [3]. Due to Cisco distributing the data format freely, it became widely used [3]. The first versions v1 to v7 support only a

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



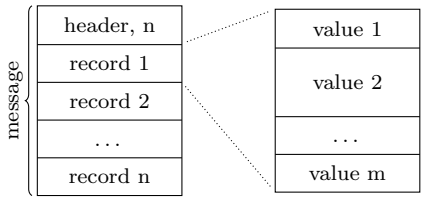


Figure 4. Simplified flow protocol with fixed record types and fixed templates as used in, e.g., NetFlow v1 - v8. *Records* are sent as a stream of *messages*, where each contains the *flow records*. A *flow record* consists of a fixed and ordered set of fixed size *values*. Different *values* can have a different length, however, the same *value* must always have the same length. Due to these restrictions, no record size is needed, since every record layout is identical. The *header* contains basic information such as current time and the number of records *n* in the message. If multiple different record types are supported, like in NetFlow v8, the *header* must contain the *record type*.

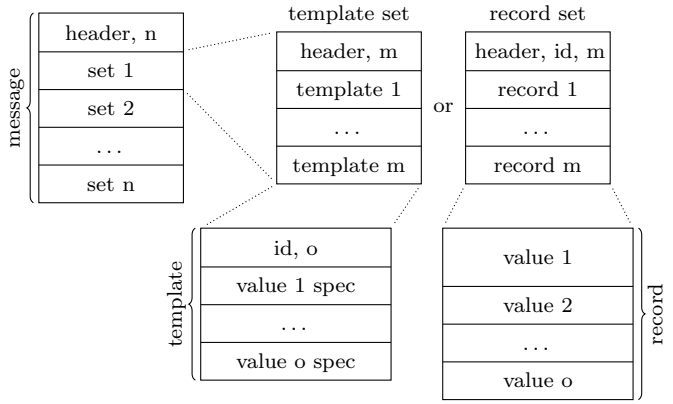


Figure 5. Simplified flow protocol with fixed record types and dynamic templates as used in, e.g., NetFlow v9 and IPFIX. Data is sent as a stream of *messages*, where each contains one or multiple *sets*. A *set* consists of a *header* specifying the set type and number of elements *m* and the set body, which depends on the set type. The body can contain *template specifications* or *flow records*. *Template specifications* contain a *template ID* and the number of values *o* followed by *value specifications*, which contain the length and type of single values. A *record set* starts with a header specifying which *template ID* is used and the number of records *m*, followed by *records* in the same way as Fig. 4.

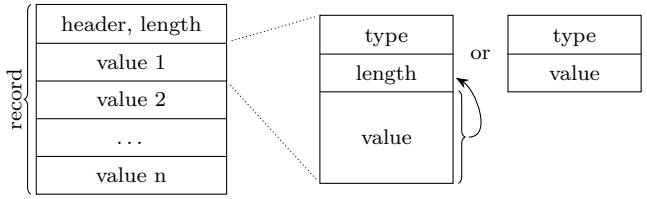


Figure 6. Simplified flow protocol with dynamic record types as used in, e.g., *argus*. Unlike NetFlow and IPFIX (Figs. 4 and 5), this format does not use and, therefore, does not need any template specification or inline specification mechanism. *Records* contain a *header* specifying type and *length* of the record, followed by the flow record *values*. *Values* are represented by type, length of the value, and value (TLVs). Small values fitting into the length field can also be encoded in the length field either with a special encoding discerning those from length values or without special encodings using a preshared list of short types.

fixed flow key and a fixed set of flow record values (e.g., number of bytes and packets of a flow, source and destination addresses and ports) [26], while v8 introduced a set of 14 different record types [26], and v9 extended the protocol to fully dynamic flow record specifications with a Cisco-specified list of possible values [10], [26].

Since NetFlow v1 to v8 use only a limited number of flow record types, flow records are sent as *messages* containing a *header* describing the message contents followed by one or multiple *records*, which is outlined in Fig. 4. Each record contains all flow properties one after another. Due to the protocol and header dictating value types and their corresponding lengths, no information about types and lengths is included. Different values can have different lengths, however, the length of the same value in different records is fixed. A single network packet contains one message, limiting the number of flow records per message with the maximum packet size.

Due to the flexibility of NetFlow v9 an additional level of abstraction, called *set*, is needed. A *set* allows the transmission of *templates* describing the *values* contained in *records*. Like older NetFlow versions, values inside a single record can have differing lengths. Since value order and types are fixed per template, only the used template has to be referenced. As can be seen in Fig. 5, *sets* consist of a *header* describing the set type (*record* or *template*) and the specified data.

Although NetFlow is only specified as a network protocol, it is possible to store NetFlow records in a file due to the included packet framing (i.e., every message starts with a header specifying the number of records) [27].

Other network device vendors implemented their own flow export protocol (e.g., Juniper Networks: Jflow, 3Com/HP: NetStream, Ericsson: Rflow), which also follow the basic structure in Fig. 4 and are to a certain extent compatible to NetFlow.

A newer, more flexible, and open standard is the Internet Engineering Task Force (IETF)-maintained IPFIX, which is defined in RFC 7011 [13]. As already mentioned in Section II-A, it is the successor to NetFlow v9. The IPFIX standard not only provides the definition for the network protocol, but also a very generic and extensive specification of flows, which is described in Section II-B.

The format of IPFIX extends the NetFlow v9 format, as described above and in Fig. 5, with option templates that allow adding metadata to the exported data, and a generalized and more flexible value type specification. RFC 7011 [13] contains examples for metadata like exporter properties (e.g., details about the used network interface), packet or flow statistics (e.g., number of packets the flow exporter received or dropped), or details specifying the flow aggregation process (e.g., properties that make up the flow key, timeouts).

IPFIX is specified for both network transport, capable of utilizing various transport protocols [13], as well as file storage [13], [27]. Due to IPFIX and NetFlow messages sharing some compatibility, it is even possible to mix different NetFlow versions and IPFIX in a single data stream [27]. Hofstede et al. [4] provide a more detailed overview of IPFIX, including a more detailed example record visualization.

A different network-based flow protocol is argus, which is used by the flow exporter *argus* (see Section III for an overview of *argus*). Unlike NetFlow and IPFIX this format is not based on templates. To keep the format future proof and

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



allow any possible flow record value type, this format uses Type-Length-Value (TLV) fields [28]. As explained in Fig. 6, this means, that instead of just sending the value, additionally a type and length must be transmitted. Flow records are therefore a list of TLVs.

Advantage of this format is, that the receiving software doesn't need to keep track of templates and no template exchange mechanism is needed, which keeps protocol implementations simpler. However, this comes at the cost of larger record sizes since every record must contain every type and length information. To improve this situation, TLV implementations typically include an optimization that encodes the length inside the type and uses the length field for the actual value (see Fig. 6). Nevertheless, this works only for small values fitting inside the length field, therefore limiting the scope of this optimization. Due to the high probability of flows needing similar record types, IPFIX is a better choice due to the vastly lower overhead. Additionally, there is no canonical specification of argus, only the reference implementation.

A flow-related network-based format is sFlow. This format was created to transport network interface statistics and parts of sampled packet data from flows, where a flow is defined as the set of packets passing from one network interface to another [29]. Due to the required sampling and the missing notion of flows or packet aggregations, sFlow is not suitable to transport flow data.

*2) File-based Flow Formats:* The simplest way to store flow records in files is to use network-based formats that include framing and, therefore, support file storage (e.g., IPFIX as mentioned above). However, this has the downside, that the software used for reading the flow records must support this format. Therefore, it might be necessary to use data formats that are more widely used for exchanging data, that support flow records.

The simplest format is the text-based Comma-Separated Values (CSV)-format, which organizes data in rows, which contain columns. Examples for row delimiters are combinations of line feed and carriage return, and examples for column delimiters are comma, semicolon, space, or tab. Since there are many different variants and implementations, RFC 4180 [30] provides a canonical definition using carriage return followed by line feed as row delimiter and comma as column delimiter.

As portrayed in Fig. 7, flow records can be represented by rows with the values contained in the columns. Optionally, the file can start with a header, describing the value types. Advantages of using CSV-files are the near universal support of this file format in most software, and the human readability. Disadvantages include the high overhead due to storing numbers as text instead of binary and the different format specifications. Additionally, if multiple record types are needed, a scheme for specifying record types must be implemented.

Flow records can also be stored in any database format capable of storing structured data, e.g., Sqlite. Sqlite is a free software library that supports reading and writing database files with a Structured Query Language (SQL) dialect [31]. SQL is a language for writing and querying structured data.

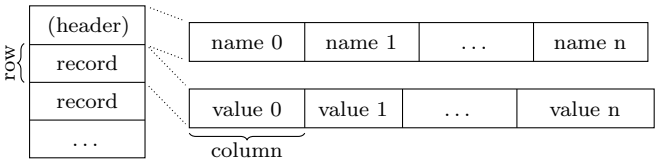


Figure 7. Simplified, structured format for flow records, e.g., CSV files. Files of this type consist of *rows*, which contain *columns*. Example separators for CSV files are *line feed* for rows, and *comma* for columns. If needed, a first row can be a *header*, naming the exported values. The rest of the file contains one *flow record* per *row*, with the record *values* contained in the *columns*.

This enables powerful search and filter capabilities in the resulting flow records.

## III. Flow Exporters

Flow exporters are applications capable of aggregating packets into flows. In this section we present a detailed analysis of *flow exporters* and precursors, called *flow exporter-like*, with a focus on research based usage.

Section III-A contains an introduction into these two types and description of the selection process. This is followed by a detailed description of flow exporter-like software in Section III-B and flow exporters in Section III-C. Additionally, Table II lists supported features of the analyzed software like supported flow directionality, protocols, expiry mechanisms, and properties and if the exporter can be extended to support possible required scenarios. Finally, Section III-D summarizes limitations.

### A. Introduction

Since firewalls and forwarding devices already need flow information for operating (e.g., tracking connections with the 5-tuple flow key inside a stateful firewall), many of those are capable of exporting flow data [4]. However, exporting flows is only an auxiliary function with low priority in those devices and many of them support flow export only via unreliable transport protocols [2].

Furthermore, this approach only supports aggregating flows from network traffic, which is forwarded by the device. Therefore, public datasets or previously captured network data can't be used directly, but instead must be simulated by an additional device. This approach would imply uncertainties caused by the simulation device besides the high overhead needed for the measurement setup. Additionally, such a setup would lack reproducibility due to the issues mentioned above and the impossible task of simulating the previously captured traffic precisely. Thus, using software for flow export is necessary.

There are commercial and open-source options available capable of exporting flows in various file formats or over networks. Depending on the type of software, a network interface or a file containing previously stored packets can be used as source. In addition to several flow exporters, there is also packet analysis software available that can export flows or precursors to flows.

Since most user manuals of software are lacking specific implementation details, which are needed for the following

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



analysis, closed source exporters like *nProbe* [32] are not included in this analysis. Additionally, flow exporters that are not capable of using files as input or have very limited functionality are also not included in the list. Those flow exporters are not usable in experimental settings and should only be considered for deployments of the finalized work. One example for this is *clerk* [33], which was developed at Google for monitoring their corporate network. It was built for high speed operation on commodity hardware, can therefore export a very limited and non-configurable amount of properties, and use packets from network interfaces only.

Another example are hardware-based flow exporters, which also lack the capability of using file input since those are specifically built to only be used in online configurations. Additionally, hardware-based flow exporters provide only a fixed set of features [4]. Examples for hardware-based flow exporters are special purpose hardware, flow-export-capable forwarding devices and firewalls. Hofstede et al. [4] compiled a list including supported basic features of such hardware devices.

The following analysis categorizes software into the two groups *flow exporter* and *flow exporter-like*.

*Flow exporters* are capable of aggregating flow records by reading packets from a source, i.e., file or network interface. This means those must have at least the following features:

(1) Support decoding the necessary parts of network packets.
(2) Use a flow key for grouping sets of packets into flows.
(3) Support some form of flow expiry.
(4) Extract packet properties.
(5) Aggregate packet properties to flow records.
(6) Output flow records in a suitable format.

*Flow exporter-like* software are applications that were not designed to be flow exporters, but provide some functionality required for building a flow exporter. This could be for instance being able to split a file containing packets into one file per flow containing packets belonging only to a specific flow, i.e., features (4) to (6) are missing. Another example is just being able to extract TCP connections, which would be the equivalent of a flow exporter only supporting the TCP protocol, 5-tuple as flow key, and expiry according to the TCP state and eventual timeouts. Those were developed before network flows were widely used. The following detailed analysis is comprehensively summarized in Table II.

### B. Flow Exporter-Like

One of the earliest network analyzers is *tcptrace*, which was developed around 1996. It can track TCP connections in network traces, which are files containing packet captures, and from live network capture. *tcptrace* can aggregate numerous TCP-based properties, e.g., window sizes, flags, Round Trip Time (RTT), which is the two-way delay for a packet from client to server and back to the client, Congestion Window (CWND) which is the maximum number of unacknowledged packets that can be in transit at one time on the network path (see RFC 793 [16] for further flags and definitions), as well as basic UDP properties, e.g., number of packets, number of bytes. These properties can either be plotted in graphs or output as text. It is possible to add further properties with the help of a C-based plugin Application Programming Interface (API). Only the 5-tuple can be used as flow key and flows are expired via TCP-state tracking or after a $4\,\mathrm{min}$ idle timeout [34]. Although development stopped in 2004, it is still in use today (e.g., Hagos et al. [45]).

*netdude* was developed in 2001 to provide an application for displaying and modifying network traces. Due to the fully structured architecture and plugin support, it can also fulfill further tasks like annotating traces with additional information (e.g., *conntrack* plugin), export flows (*demux* plugin), or combine flows (*mux* plugin). Since the focus of *netdude* is on trace files, the flows are output as network traces and therefore additional software for aggregating flow records is needed. Either source and destination address, destination port, destination address and port, or the 5-tuple can be used as flow key. Flows are expired according to the TCP specification, or after a $60\,\mathrm{s}$ timeout for non-TCP flows [35], [36]. The development of *netdude* seems to have stopped in 2010.

*netmate* is a highly customizable monitoring solution developed in 2004. It is extensible via a module-API and with Extensible Markup Language (XML)-based configuration files. XML-files are text-based files for storing structured data. It can be used to export flows in different text-formats, as well as IPFIX. The flow key is derived from a set of filters that can be specified via XML. Since those filters are specified as header offsets and bit-masks in an additional XML-file, it is possible to use any field from any protocol. Flows can only be expired after timeouts or by processing modules. No processing module is included for expiring TCP-based flows [37]. Development has stopped in 2009.

### C. Flow Exporter

In 1993 one of the first flow exporters called *argus* was released. It was built for network monitoring and supports a distributed architecture, where network traffic passing different points of observation in the network can be matched. It can also parse various protocols (e.g., TCP, Address Resolution Protocol (ARP), Internet Control Message Protocol (ICMP), Real-Time Transport Protocol (RTP), Generic Routing Encapsulation (GRE), Encapsulating Security Payload (ESP)), and uses its own proprietary flow format. *argus* does extensive state-tracking, to provide as much information as possible (e.g., request and response tracking – relating ICMP echo request and replies). The decoded and understood protocols are fixed and there is no plugin mechanism capable of extending the functionality. The flow key and the detail level can be modified but only in a limited way (e.g., include data link source and destination in the flow key, or more detailed state tracking) [28], [38]. No new version was released since 2016, however, a new version has been announced for Q1 2019 in January of 2019. *argus* was used by [46] to capture and aggregate the used data.

*pmacct* was first released in 2003 and is a modular network accounting system. The base system can aggregate basic properties (e.g., packets, bytes) and group those by a variable set of properties (e.g., source and/or destination network

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



Table II
LIST OF FLOW EXPORTERS AND FLOW EXPORTER-LIKE SOFTWARE.

| | name | first | latest[a] | flow key[b] | bidi | proto[c] | expiry[d] | filter | res[e] | output[f] | properties[g] | lang[h] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| flow exporter-like | tcptrace [34] | 1996 | 2004 | 5-tuple | ✓ | TCP, UDP, * | full-TCP; 4 min idle | ✓ | µs | plot, text, plugin | TCP, RTT, CWND, plugin | C |
| | netdude[1][35], [36] | 2001 | 2010 | 5-tuple; src, dst; dstPort; dst, dstPort | ✓ | –[2,3] | full-TCP[4]; 60 s | – | µs | PCAP[2] | –[2] | C |
| | netmate [37] | 2004 | 2009 | * | ✓/– | *[3] | timeout | ✓ | µs | text, PCAP, IPFIX, plugin | jitter, RTT, RTP, plugin | C, C++ |
| flow exporter | argus [28], [38] | 1993 | 2016 | 5-tuple + (VLAN) + (layer2) + (MPLS) | ✓ | TCP, UDP, ICMP, ...[5] | full-TCP, ICMP, DNS, timeout, ... | ✓ | µs | PCAP, argus | basic, state | C |
| | pmacct[6][4], [39], [40] | 2003 | current | 5-tuple | ✓[7] | TCP, UDP, * | (timeout)[8] | ✓ | ms | NetFlow, IPFIX | basic (config) | C |
| | Vermont [1], [4], [41] | 2005 | current | * | –[9] | *[3] | (active, idle)[8] | ✓ | µs | IPFIX, DB, text, plugin | payload (config) | C++ |
| | yaf [2], [4], [42] | 2006 | current | 5-tuple + (VLAN) + (MPLS) | ✓/– | TCP, SCTP, UDP, ICMP, * | TCP/SCTP, active, idle | ✓ | ms | IPFIX | payload, fingerprint, application, plugin | C |
| | Joy [43] | 2016 | current | 5-tuple | ✓/– | TCP, UDP, ICMP, * | active, idle[10] | ✓ | µs | JSON, IPFIX | TLS, fingerprint, application, API | C |
| | go-flows[11] | 2018 | current | *, plugin | ✓/– | TCP, UDP, ICMP, * | (TCP), active, idle, plugin | ✓ | ns | CSV, IPFIX, PCAP, plugin | aggregation (config), plugin | go |

[a] *current*, if the latest release was in the year this paper was written (2019).
[b] Supported flow key(s). *5-tuple*: source and destination IP address, protocol number, source and destination transport port. +: addition to the flow key. *()*: optional field. *: flow key can be freely configured. *src*: source address. *dst*: destination address. *dstPort* destination transport port.
[c] Supported protocols. * means that any protocol is supported but in a more limited way than the others specified.
[d] Flow expiry. Can be state-based (e.g., full-TCP for full TCP-state-machine as specified by RFC 793 [16], TCP for partial TCP-state-machine only tracking FIN/RST), or time-based (i.e., active or idle timeout). Plugin means that flows can be expired via API.
[e] Time resolution supported internally and in output.
[f] Supported output file formats or types. *plugin*: exporter provides an output-API that modules can use.
[g] Supported flow properties excluding basic properties like packet size, number of packets and protocol header fields. *plugin*: properties can be added via an API provided by the exporter.
[h] *lang* is the programming language used. This specifies the supported language for plugins.
[1] With the *demux* plugin.
[2] *netdude* supports only dumping the packets belonging to a flow to a PCAP file.
[3] Only IPv4 packets supported.
[4] Timeouts as specified by the TCP standard RFC 793 [16].
[5] *argus* supports a fixed and hard-coded set of protocols.
[6] *pmacct* supports flows via the *nfprobe* plugin.
[7] Bidirectional flows are exported as two unidirectional flows (flow start and end times of both flows match).
[8] Supports timeouts only during live capture.
[9] Documentation claims bidirectional flow support, but support has been broken since 2014 [44].
[10] Timeouts are fixed at 10 s idle and 30 s active.
[11] Described in Section VI.

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



address, data link address, Autonomous System (AS)). These aggregates can only be expired based on time and be exported via databases, text-files, or queried from the daemon. To support flows including TCP state-tracking, a plugin called *nfprobe* is provided that can only use the 5-tuple as flow key and has a configurable flow timeout. These flows can only be exported via IPFIX or NetFlow [39], [40].

*Vermont*, first released in 2005, is a network monitoring solution for packet and flow filtering, sampling, accounting, and aggregation. Flow key, aggregated properties, and active and idle timeout can be chosen via an XML-based configuration file. IPFIX, various Databases (DBs), text, and a plugin-API are provided for data output [1], [41]. Due to a bug introduced during the rewrite of the IPFIX export in 2014, *vermont* can only export unidirectional flows [44].

*yaf* was first released in 2016 and written with the goal of building an IPFIX compliant flow exporter. Since the IPFIX standard was not finalized in 2016 (the first draft was published in 2003, the proposed standard RFC 5101 [9] released 2008, and the standard RFC 7011 [13] released in 2013), the development of *yaf* was also used to gather experience with IPFIX and provide feedback to the standardization process. This flow exporter can only use the 5-tuple (optionally extended with Virtual Local Area Network (LAN) (VLAN)-tag and Multiprotocol Label Switching (MPLS)-labels) as flow key. Flows can be bidirectional or unidirectional. Basic properties, as well as Operating System (OS) fingerprints, payloads, and Deep Packet Inspection (DPI)-results can be exported. It is in active development and part of the CERT Network Situational Awareness (NetSA) security suite [2], [42]. *yaf* was used in [47] to export flows from Packet Captures (PCAPs) provided by the Measurement and Analysis on the Widely Integrated Distributed Environment (WIDE) Internet (MAWI) project. There is also another version available, that extends *yaf* with performance metrics for TCP flows called *Quality of Flow (QoF)*, but development seems to have stopped in 2013 [4], [48].

*joy* is an open source flow exporter and collector released by Cisco in 2016. It was created to support encrypted traffic analysis efforts and can therefore inspect Transport Layer Security (TLS)-records, Domain Name System (DNS)-records, as well as Hypertext Transfer Protocol (HTTP) elements. Only the 5-tuple can be used as flow key. Bidirectional as well as unidirectional flows can be aggregated. In addition to basic flow properties, properties relevant to TLS, DNS, HTTP, as well as fingerprinting can be exported. Only time-based active and idle flow expiration is supported. Flow records are exported as Javascript Object Notation (JSON). The output can be filtered and further aggregated with an included tool and optionally converted to IPFIX [43]. *joy* was used in [49], [50] for property export.

### D. Summary

As shown in Sections III-B and III-C and Table II, there is a wide range of flow exporters and flow exporter-like applications with different capabilities available. The analyzed exporters were built for a specific goal like accounting (e.g., *pmacct*, *Vermont*), implementing a standard (e.g., *yaf*), or for a specific analysis (e.g., *joy* for TLS-analysis). Therefore, most flow exporters lack flexibility and support only fixed functions or a limited set of functions. For example, support for using statistics on packet properties (e.g., standard deviation, median) is usually very limited or non-existent.

This leads to differing implementations of flow exporters with varying flow definitions. Additionally, important details like flow expiry are often not mentioned in the resulting publications, leading to non-reproducible and non-comparable scientific research.

However, implementing a flow exporter needs substantial implementation effort, which is why many researchers revert to using existing non-flexible solutions accepting the downside of limited choices for properties and flow keys. This is one reason why features like the hard coded 5-tuple for flow keys are commonly used. Despite those barriers there are existing works employing custom flow exporters trying to overcome those limitations.

## IV. Publications using Custom Flow Exporters

Despite the extensive list of available flow exporters listed in Section III, there are many scientific publications implementing their own custom flow exporter. Possible reasons like inflexibility regarding flow key, properties, or available statistical functions have already been mentioned previously. In this section these topics are explored further, including a look at how the custom flow exporters are built, described, or specified.

The following list of network data analysis papers used either flow exporter-like software or custom code for aggregating packets to flows. Application domains of the analysis include traffic classification, attack detection, anomaly detection, or providing a dataset. These scientific publications are a subset from the Network Traffic Analysis Research Curation (NTARC) database [58], that use custom flow exporting to assess reasons why custom flow exporters are implemented in the scientific community. Additionally, we include one of the earliest papers that uses network flows, to demonstrate that despite the long history of network flows, the challenges are still the same today. A comprehensive summary and comparison of these papers is also provided in Table III.

Although the authors didn't mention the term flows, a machine-learning-based traffic classification system was evaluated as early as 1994 by Frank et al. [22]. The authors demonstrated with their experiment that such a system could be used to improve intrusion detection systems by reducing the data needed for analysis, or augment rules generated by an expert with learned patterns. As source for flows the authors used the *Network Security Monitor (NSM)* proposed by Heberlein et al. [21] which exports flow-like records from a live network capture. This system uses source and destination address, service type, and connection identifier as flow key. Frank et al. used a subset of the properties (e.g., time, number of packets, number of bytes) provided by *NSM* for their work.

Moore et al. [51] aggregated network packet traces recorded in 2003 into flows in 2005. The authors used *netdude* to split

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



Table III
LIST OF PAPERS IMPLEMENTING CUSTOM FLOW EXPORTERS.

| pub. | year | goal | flow key | bidi[1] | flow expiry | filter | custom[2] | statistics[3] | flow exporter |
|---|---|---|---|---|---|---|---|---|---|
| [22] | 1990 | traffic classification | src, dst, service, connID | ✓ | □ | TCP | – | – | NSM [21] |
| [51] | 2005 | provide dataset | 5-tuple | ✓ | complete TCP/□ | TCP | ✓ | ✓ | netdude, tcptrace, custom |
| [52] | 2006 | traffic classification | 5-tuple | ✓ | TCP/600 s idle | TCP, UDP | ✓ | ✓ | netmate |
| [53] | 2007 | traffic classification | src, dst, srcPort, dstPort | ✓ | TCP/□ | TCP | ✓ | ✓ | tcptrace, custom |
| [54] | 2009 | traffic classification | 5-tuple | ✓ | TCP/600 s idle | TCP, UDP | ✓ | ✓ | netmate |
| [25] | 2012 | traffic classification | 5-tuple | ✓ | TCP/600 s idle | TCP, UDP | ✓ | ✓ | netmate, custom |
| [55] | 2013 | traffic classification | 5-tuple | ✓ | □ | – | ✓ | ✓ | custom |
| [56] | 2015 | attack detection | □ | □ | □ | □ | ✓ | ✓ | custom |
| [23] | 2015 | traffic classification | 5-tuple | – | TCP/900 s idle | TCP | ✓ | ✓ | custom |
| [24] | 2016 | anomaly detection | tstamp (minute), src, dst | – | – | – | ✓ | – | tshark, custom python/perl |
| [57] | 2017 | traffic classification | 5-tuple | ✓ | □ | TCP, UDP | ✓ | ✓ | custom java code |

✓ Used. – Not used. □ Not mentioned in paper.
[1] Bidirectional flows were used.
[2] Custom properties were used.
[3] Advanced statistical functions, e.g., *standard deviation* or *median*, were used on properties.

the traces into flows, and *tcptrace* and custom programs for aggregating the properties. As flow key the 5-tuple was used and only complete TCP connections were analyzed due to the missing implementation for flows without an end (e.g., UDP flows). Since the aim of the work was to provide as much data as possible for research uses, a total of 249 properties were included into the dataset which was provided as Weikato Environment for Knowledge Analysis (WEKA) input files, which is a framework for machine learning [59].

Williams et al. [52] compared the performance of five different machine-learning algorithms on flow data. This data was aggregated with *netmate* utilizing the 5-tuple flow key and a 600 s idle timeout on TCP and UDP connections. Since all flow properties and the full flow aggregation is supported by *netmate*, no custom processing was necessary.

Auld et al. [53] proposed a flow-based traffic classification that can identify traffic despite not using payload, IP-addresses, or port numbers. Their approach used *tcptrace* for aggregating packets to flows and custom code for aggregating properties. Although Auld et al. briefly mention challenges with flow expiry regarding TCP streams (e.g., incomplete streams) and non-TCP streams (e.g., no clear flow boundaries), parameters like, e.g., idle timeout are not provided. It can only be assumed that the default values of *tcptrace* were used.

Alshammari et al. [54] evaluated five different machine-learning algorithms for identifying Secure Shell (SSH) as well as Skype network traffic. Like in [52], *netmate* with a 5-tuple flow key, 600 s idle timeout, and TCP and UDP connections was used to aggregate flows. No additional properties were added to the *netmate* implementation.

Nguyen et al. [25] developed a traffic classification scheme for QoS based on flows. Since a decision is needed for QoS before a traffic flow is finished, the authors explored the idea of using sub-flows, which are flows aggregated from a subset of the packets belonging to a full flow (e.g., first 10 packets). This was first evaluated with flows aggregated by *netmate* (5-tuple flow key, 600 s idle timeout, TCP and UDP traffic) and custom code to split the flow into the proposed sub-flows. After verifying the performance, a custom classifier called Distributed Firewall and Flow-shaper Using Statistical Evidence (DIFFUSE) was developed that includes flow aggregation and classification for life traffic.

Zhang et al. [55] proposed a flow-based traffic classification network. Due to the need for custom properties and the use of statistical methods, a preexisting flow exporter couldn't be used. No further details about the used flow exporter were included in the paper.

Qin et al. [56] implemented a Distributed Denial of Service (DDoS) attack detection framework with machine learning and clustering. Due to the shortcomings of *NetFlow* (e.g., limited properties, no statistical functions), a custom flow exporter was created. The paper only mentions the exported properties and that the exporter was modeled after *NetFlow*. Therefore, other flow exporter properties like flow key and expiry can only be guessed, which leads to very limited reproducibility.

Zhang et al. [23] implemented a flow-based traffic classification framework. Only TCP flows were analyzed and a 900 s idle-timeout was adopted to limit the maximum number of concurrent active flows which determines the peak memory usage. Despite mentioning software used for verification, the flow exporter is not described in more detail than flow key, exported properties, and above mentioned flow expiry.

Iglesias et al. [24] analyzed the temporal behavior of communication and found a small set of seven properties which allow to fully categorize the observed traffic. Due to increasing use of encryption, privacy concerns, and availability, packets without payload and identifying properties were aggregated to flows. Since this work uses a custom flow key (timestamp in minutes, source address, destination address) and custom properties, a custom flow exporter was developed using *python* and *perl* utilizing *tshark* [60] for packet decoding.

Vlăduţu et al. [57] proposed an Internet traffic classification platform based on flows. A custom flow exporter in java was built using *jpcap* [61] for packet parsing. Due to the focus of the paper on flow properties and machine learning, key aspects of flow export like flow expiration (e.g., timeout values, if a

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



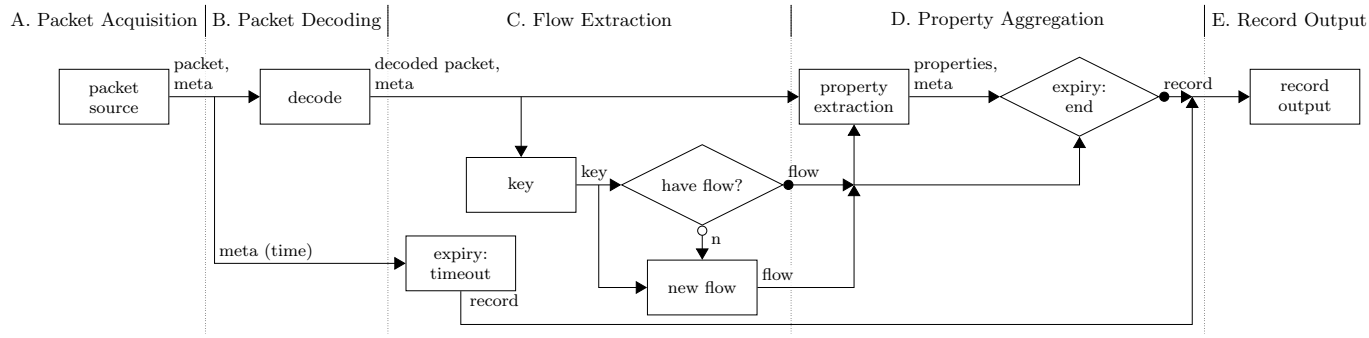


Figure 8. Flow exporter processing pipeline including control and data flow, and references to the sections describing the sub-parts. Processing starts with acquiring a *packet* and *metadata* (e.g., timestamp of arrival) from a *packet source*. With this *timestamp*, existing flows are checked for *timeout*-based *expiry* conditions. Next, this *packet* is *decoded* and a *flow key* calculated. If a *flow*, representing the partially aggregated data, with this *flow key* does not exist yet, a *new* one is created. This is followed by *property extraction*, which adds data from the *decoded packet* and *metadata* to the *flow*. After *property extraction*, *end of Flow* conditions are checked, and if fulfilled, the record is *expired*. The last step handles *record output* to the desired destination.

TCP state-machine was used) were omitted despite mentioning challenges in flow export. A preexisting flow exporter probably wasn't used due to the need for custom properties and statistic-based aggregation functions.

Despite the wide range of flow exporters with differing capabilities, with a subset presented in Section III and Table II, many publications and research projects, including the list above, implement custom flow exporters. Reasons for this are missing functionalities like using non-standard flow keys or employing more advanced aggregation functions. However, all these flow exporter implementations are not published, and seldom reused in future projects. This leads to every research project reinventing flow exporting.

## V. Flow Exporter Building Blocks

Flow exporters aggregate flow records from network packets. A high level overview of a specific implementation of the flow exporter *yaf* was given be Inacio et al. [2], but due to the broad focus and the description of one specific implementation only example parts like timestamp handling are explained. Further guidelines are presented by Hofstede et al. [4], however, the focus is on the complete flow monitoring process and existing applications, leaving out implementation details. The following tutorial covers the necessary steps in more detail, including considerations for reproducible flow export.

In this section we present the required steps and specifications to build a flow exporter. We point difficulties and pitfalls out that need to be taken care of to support reproducible results. This tutorial covers the steps *Packet Observation* and *Flow Metering & Export* presented as a generic overview by Hofstede et al. [4] in more detail, allowing the generation of reproducible flow-based scientific data. On top of that, we provide definitions for specifications which must be provided in conjunction with experiments to ensure reproducibility of the generated flows.

The required steps are presented as a pipeline from *packet* to *flow record* in Fig. 8. These steps are explained in the following sections:

**V-A Packet Acquisition:** Network packet input from files or life capture.

**V-B Packet Decoding:** Network packet decoding.

**V-C Flow Extraction:** Keeping track of flows, assigning packets to flows, and timeout based flow expiry.

**V-D Property Aggregation:** Aggregating packet properties to flow records and packet-event-based flow expiry.

**V-E Record Output:** Formatting and writing the resulting flow records to an output stream.

### A. Packet Acquisition

The first step required for flow exporting is to acquire network packets. Network packets can be read from a file or directly life from a network device. Common to both approaches is that not only the packet contents, but also metadata is required.

*1) Network Packet Contents:* Network packet contents are structured following a layered approach in line with the International Organization for Standardization (ISO) Open Systems Interconnection (OSI) reference model. This model defines the seven layers (1) Physical, (2) Data Link, (3) Network, (4) Transport, (5) Session, (6) Presentation, and (7) Application.

Data transmitted via the network hardware is formatted according to the protocol used in the physical layer and include the three parts: header and footer enclosing the payload (see Fig. 9). The header is needed for synchronizing and determining the beginning of a packet, while the footer might be needed to provide a pause between packets. Since header and footer look the same for every packet and are only processed in hardware, these parts can not be captured and will not be part of a capture file. For these reasons, as shown in Fig. 9, the *frame size* considers only the payload of the physical layer.

Contained in the payload of the physical layer is the data link layer, which, like layer 1, typically uses the header, payload, and footer structure (see Fig. 9). The header of the data link layer already contains useful information (e.g., hardware addresses) and can be observed in the network capture. The footer commonly includes a Frame Check Sequence (FCS) needed to verify the correct transmission of the packet. Since the FCS is verified by the networking hardware and network

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



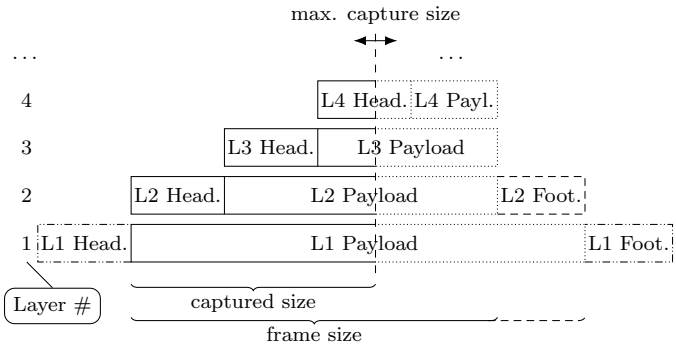


Figure 9. Captured packet including stacked layers. Physical layer (*1*) and data link layer (*2*) enclose the *payload* between *header* and *footer*. Layer *1* header and footer are normally not captured, resulting in the *frame size* being the length of layer *2*. Layer *2* footer might be missing in a capture, which can be caused by OS, hardware, or driver capabilities or settings. *Maximum capture size* might be limited due to performance, data protection, or privacy reasons, which results in the *captured size* being smaller than the *frame size*.

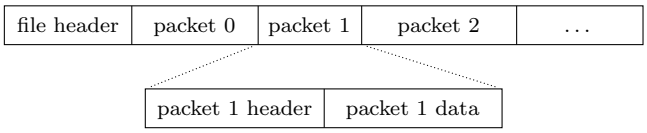


Figure 10. Simplified packet capture file, e.g., PCAP-format. These files contain a *file header*, followed by the captured *packets*. The *file header* typically contains the protocol type of the captured packets, the maximum capture length, and the timestamp accuracy. Captured packets consist of a *packet header* followed by the actual *packet data*. Packet headers typically contain the *timestamp* of the packet, the original frame size, and the captured frame size.

packets failing the check are dropped, network hardware does not forward the FCS to software in most cases.

It depends on several factors whether faulty FCS or malformed packets are reported by the hardware, in particular on:

- OS,
- driver software,
- network hardware (e.g., specialized network capture hardware like Endace Data Acquisition and Generation (DAG) cards support capturing malformed packets and the FCS),
- settings (e.g., some Wireless LAN (WLAN) hardware can report FCS depending on a configuration setting),
- or even on the packet type (e.g., there are WLAN hardware and OS combinations where the FCS is reported only for certain types of packets).

Layers above the data link layer include only a header, followed by the payload. Furthermore, since the full payload of the data link layer is provided to the OS, no further fields are missing from the capture. One exception to this rule is, that the maximum capture size can be limited for improved performance. As demonstrated in Fig. 9, this can even lead to truncated headers, i.e., only a part of the header is captured. Additionally, due to privacy concerns, the payload of the transport layer or even the network layer might be deleted, resulting in missing upper level layers.

Therefore, in addition to the captured packet data, the *frame size* (i.e., the original size of the lowest captured layer) and the *captured frame size* (i.e., the captured size) is needed. To allow decoding the packet, the *protocol type* of the lowest captured layer must also be included in the capture. This even allows only capturing a higher layer and it's contents, e.g., only recording the network layer, to reduce the amount of storage needed.

In addition to the size information, the point in time the packet was captured must be included in the form of a *timestamp*. Even online capture sources might be required to provide a *timestamp* together with the captured packet data, which could be recorded by the network hardware or the OS for improved accuracy.

If more than one packet source is combined, care must be taken regarding clock synchronization of the different packet capture points. Deviations in clock synchronization can lead to packets being processed in the wrong order after they are merged for flow exporting, which can lead to packets being assigned to the wrong flow, wrong number of flows, wrong flow expiry, or even wrong flow properties.

*2) File-based Packet Source:* There are no standardized file-based packet capture formats. The most widely used format is the PCAP-format, for which the only specification is the reference implementation of the *libpcap* library [62].

As illustrated in Fig. 10, PCAP-files start with a file header containing information common to every packet like protocol type of the capture packets, maximum capture length, and timestamp accuracy. Packets are stored as a header containing packet metadata like original frame size, captured frame size, and timestamp, followed by the raw packet data.

This format is widely used due to its simplicity, the ubiquity of the libpcap library, and the availability of wrappers for libpcap, as well as read and write capabilities available in many programming languages. However, PCAP has the following shortcomings:

- There is no specification, only a reference implementation.
- All packets must have the same type of lowest layer, since this information can only be specified in the file header.
- Although packets from multiple network interfaces can be written into one PCAP-file, all network interfaces must be of the same type and have the same properties. Additionally, it is not possible to store an interface Identifier (ID) in the packet meta data.
- Despite including a field for timestamp resolution in the file header, the original version supported only millisecond resolution. In combination with the missing specification, this resulted in several PCAP-formats that differ in how the resolution and the timestamps are stored.
- It is not possible to store additional metadata, like, e.g., capture filter, how many packets were filtered, or provide additional data with labels (e.g., packet classifications).

This lead to the development of the PCAP Next Generation (PCAPNG) format.

Unlike PCAP, there is a specification of PCAPNG maintained by Tuexen et al. [63]. However, as of today, the specification is still incomplete and under development, leaving parts of the format unspecified and resulting in diverging

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



implementations. Despite these issues, it is widely supported, since libpcap is capable of using this file format.

PCAPNG has the following features:

- It is possible to use high precision timestamps.
- Packet captures from different interfaces with varying properties and even differing base protocol types can be stored.
- This file format does not mandate the data stored within to be network packets. For instance, it is possible to also store log messages. Furthermore, different message types can be stored in one single file.
- Arbitrary metadata can be added to files and even packets.

According to [63] PCAPNG-formatted files are structured similarly to PCAP (see Fig. 10), with the difference that the *file header* (called *section header* in PCAPNG) is allowed to occur multiple times in a file. This enables combining multiple files by simply appending the data, which is not possible with PCAP. Additionally, there are multiple variants of the *packet header*, enabling the storage of the different data types mentioned above (e.g., *interface header*, *interface statistics header*). Furthermore, not every field in the header is fixed, and additional fields can be added with TLVs (see Section II-D for an explanation of TLVs).

Besides the generic packet capture formats PCAP and PCAPNG, there are also proprietary formats that were created to match specific capture hardware. One such format is the Extensible Record Format (ERF), used by Endace DAG capture cards. As described in [64], this format uses the same format as depicted in Fig. 10, but without a *file header*. *Packet headers* include timestamp, interface number, captured size, and frame size. As part of the *packet header*, further headers specific to the packet protocol type, and *extension headers* supporting optional values can be present.

*3) Life Capture Packet Source:* Like file-based packet sources, there are no standardized life capture packet sources. Mechanisms depend on OS and Hardware. There are two main categories for capture mechanisms: Either via a *read-like function*, or via a *shared buffer ring*.

Most OSs provide a mechanism for capturing packets via a *read-like function*. This function blocks, i.e., does not return, until a network packet arrives. Besides the raw packet data, depending on OS and settings, an additional header containing timestamp, captured size, and frame size might be provided by the *read-like function*.

Examples for these mechanisms include *AF_PACKET* and *AF_INET* raw sockets in Linux [65], *bpf* special devices in Berkeley Software Distributions (BSDs) [66], and *Winsock raw sockets* in Windows (raw sockets in windows require an additional driver installed) [67].

Functions calling in to the OS incur a high overhead for executing and, therefore, require high execution times. Since the *read-like function* must be executed for every packet the packet-throughput of this mechanism is limited. Therefore, a newer, but more complicated approach of *sharing a buffer ring* between the application and the OS was created.

For the *shared buffer ring* approach, applications must first set up a huge buffer, capable of storing multiple packets, and share the buffer with the OS. The OS writes the received packets directly to the buffer, and the client application can read the packets directly from the buffer. Therefore, the call overhead mentioned above is eliminated. But, since both OS and application access the buffer at the same time, a synchronization mechanism is required, which adds some overhead again. *Buffer sharing* and synchronization mechanisms available depend on OS and OS version.

Examples for mechanisms capable of *buffer ring sharing* are *AF_PACKET* sockets in Linux [65], *bpf* special devices in BSDs [66], and the driver for proprietary endace DAG capture cards. There are further mechanisms, that use *buffer ring sharing* and bypass the OS drivers to achieve higher performance, e.g., the proprietary *PF_RING* driver for linux [68], and the *Data Plane Development Kit (DPDK)* [69].

Moreno et al. [70] provide an in-depth analysis, detailed explanations about how the different mechanisms work, can be used, and performance comparisons.

Due to the specialized mechanisms and exact usage depending on OS version, hardware capabilities, and used programming language, it is recommended to use a dedicated packet source library.

*4) Packet Source Libraries:* As explained earlier, there are no standardized mechanisms for file-based network packet storage or life capture. Additionally, all packet capture mechanisms are highly specialized with respect to hardware, OS, OS version, and even computer architecture. Therefore, it is recommended to use packet source libraries which provide a unified API and handle the various OS interactions. This approach is even recommended by OS API documentations [65].

The most widely used and supported packet source library is libpcap [62]. It was created in 1994 with the goal of providing an OS independent library for capturing network packets [71]. Although it is written in C, various libraries implement bindings to libpcap for many programming languages [71] (like, e.g., Java: pcap4j [75], Python: scapy [76], or Go: gopacket [78]).

Despite originally being developed for Linux and BSD systems, there are also two ports for the Microsoft Windows OS called winpcap [82] and npcap [83]. Winpcap is an older implementation and supports only an outdated version of libpcap, which even lacks PCAPNG file format support. Development has stopped and npcap is recommended as an alternative [82]. Although npcap is an up to date alternative, its license is more restrictive and might prevent one from using it [83].

Another possibility for reading packet files is to use a program that supports reading packets from capture files or life packets and writing out a list of packets in a different format. One example for such a program is tshark [60], which is capable of writing packet data in JSON or CSV format.

Which packet source library can be used depends on the desired programming language, the supported OSs, supported network hardware, or supported file formats. Additionally, many of these libraries support packet decoding, which will be explained in the next section. An overview of a selection of packet source libraries can be seen in Table IV. Libraries that are outdated and no longer supported, e.g., the libraries

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



Table IV
PACKET SOURCE AND DECODE LIBRARIES.

| name | language | libpcap[a] | native capture | native file formats | decode |
|---|---|---|---|---|---|
| libpcap [62], [71] | C | – | Linux (AF_PACKET ring, DAG), BSD (bpf ring), Windows[1] | PCAP, PCAPNG | – |
| libtrace [70], [72] | C++ | ✓ | Linux (DPDK, DAG, AF_PACKET ring, ...), BSD (bpf) | PCAP, PCAPNG, ERF, ...[2] | ✓ |
| libtins [73] | C++ | ✓[3] | – | – | ✓ |
| Pcap++ [74] | C++ | ✓[3] | Linux (PF_RING, DPDK), Windows | PCAP, PCAPNG | ✓ |
| pcap4j [75] | Java | ✓[3] | – | – | ✓ |
| scapy [76] | Python | ✓[3,4] | Linux, BSD | PCAP, PCAPNG | ✓ |
| dpkt [77] | Python | – | – | PCAP, PCAPNG | ✓ |
| gopacket [78] | Go | ✓[3,4] | Linux (PF_RING, AF_PACKET ring) | PCAP, PCAPNG | ✓ |
| tshark [79][4] | - | ✓[3] | – | PCAP, PCAPNG, ERF, ...[6] | ✓ |

[a] Libpcap wrapper. Libraries wrapping libpcap support every native function of libpcap (except for winpcap implementation on Windows, which only supports the PCAP file format).
[1] Microsoft Windows support is provided by winpcap and npcap.
[2] A full list is provided by [80].
[3] Supports Microsoft Windows via winpcap (only PCAP file format supported).
[4] Supports Microsoft Windows via npcap.
[5] tshark is part of the Wireshark packet analyzer and can write decoded network packets formatted as CSV or JSON.
[6] A full list is provided by [81].

Table V
COMMON PROTOCOLS NEEDED FOR THE 5-TUPLE ASSIGNED TO LAYERS.

| Layer | Protocols |
|---|---|
| Data Link (2) | IEEE 802.3 (Ethernet), IEEE 802.1Q (VLAN) |
| Network (3) | IPv4, IPv6, ICMP, ICMPv6 |
| Transport (4) | TCP, UDP |

jpcap [61] and libnetdude [36] mentioned in Section III, were left out in this selection.

### B. Packet Decoding

After the raw network packet data is acquired via the packet source, it must be decoded. Depending on the layer type, decoding is a resource intensive process. Therefore, only the required parts should be decoded. Which parts are required depends on the used key function and properties to extract.

For example, a key function using the network source address, requires decoding of the network layer, since it contains this field. As can be seen in Fig. 9, the network layer (3) is the payload of the data link layer (2). If the captured network contains the data link layer (2) (e.g., the Ethernet protocol) as the lowest layer, this layer must also be decoded, to find the position of the network layer (3).

A selection of common protocols needed for the 5-tuple, sorted by layer, is listed in Table V. Depending on where the flow exporter is deployed, further protocols, e.g., MPLS, Stream Control Transmission Protocol (SCTP), might be of interest. The data link layer is actually not part of the 5-tuple, which only contains addresses, ports, and transport layer type from layers three and four. Nevertheless, most packet captures contain the data link layer as the base layer, requiring decoding of this layer, as explained above.

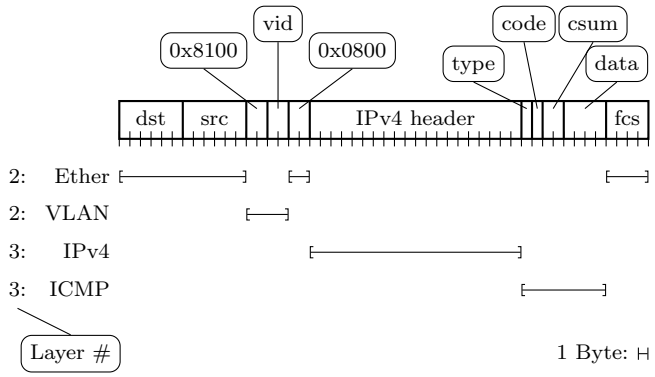


Figure 11. Detailed view of an ICMP/IPv4 packet inside a VLAN-tagged Ethernet packet with byte resolution. *0x8100*: Ethernet type code for VLAN; vid: VLAN ID; *0x0800*: Ethernet type code for IPv4; *type*: ICMP type; *code*: ICMP code; *csum*: ICMP checksum; *data*: ICMP data. Fields of the IP header are not described. Since the layers are assigned by function in the ISO OSI model, there are multiple protocols in the same layer and the model presented in Fig. 9 is not strictly followed.

Since the layer model assigns protocols to layers based on function and not based on which layer is contained in which layer, in many cases network packets don't follow the layout in Fig. 9 stringently. For example, as depicted in Fig. 11, an ICMP packet employing VLAN contains the two layer two protocols Ethernet and VLAN, and the two layer three protocols IP and ICMP. Ethernet follows the layer structure closely, since it contains a header and a footer enclosing the next layer protocol, IPv4. However, VLAN does not fit into the layer model, because it actually replaces the type field in the Ethernet header with an extended version. This leads to a peculiarity that is visible in Fig. 11: the Ethernet header line changes to VLAN and then back to Ethernet before it progresses to the IPv4 header line.

Notwithstanding, decoding libraries (Table IV) treat the first

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



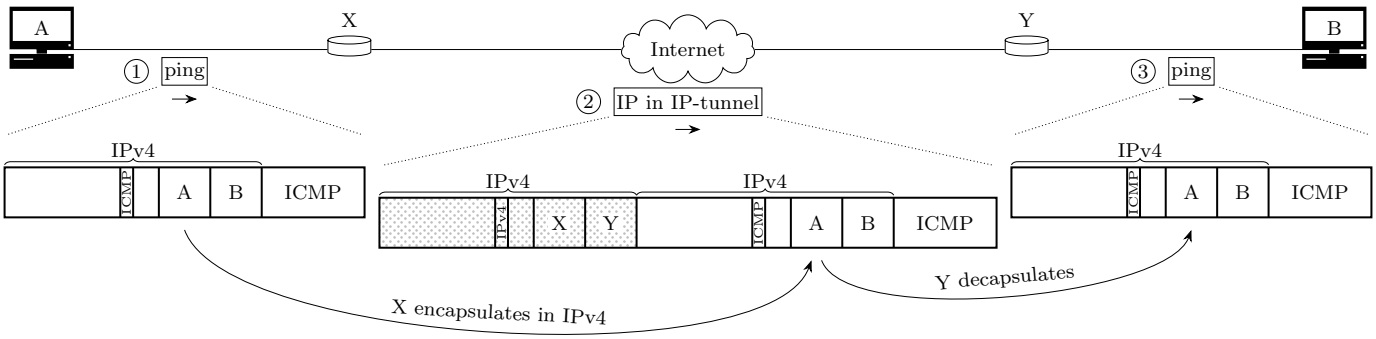


Figure 12. Packet in packet encapsulation example: IPv4 packet tunneled within IPv4. Hosts *A* and *B* are connected via routers *X* and *Y*. (1) Host *A* sends a *ping*-packet (ICMP inside IPv4) with protocol type *ICMP*, source address *A*, and destination address *B*. (2) Router *X* encapsulates this packet inside a new IPv4 layer with protocol type *IPv4*, source *X*, and destination *Y*. (3) Router *Y* removes the outer IPv4 packet and transmits the inner IPv4 packet to host *B*.

Ethernet protocol type (0x8100) as part of Ethernet and the second Ethernet protocol type (0x0800) as part of VLAN. This results in an output, that follows the layered model in Fig. 11 more closely. Additionally, this view allows step by step decoding of packets only depending on the type information extracted from the previous layer.

IPv4 fits into the strictly layered model, while ICMP is a payload of IPv4 although it is a layer 3 protocol. This is caused by the fact, that ICMP doesn't contain ports, which is required by layer 4.

Many flow exporters treat ICMP as being layer 4 by assigning zero as source port and a combination of ICMP-*type* and ICMP-*code* as the destination port (e.g., *yaf*, *go-flows*) or source port (e.g., *argus*).

A simple way of decoding protocol headers is to treat those as C structures, which are combined data types in the C language that aggregate multiple fields. The Ethernet header can be represented by a structure containing the fields dst and src, each being a list of 6 bytes, followed by a two byte number, the Ethernet type code. With this structure, the fields can be directly accessed by their names in the code. This approach is used by most flow exporters written in the C language, e.g., *pmacct*, *yaf*, *argus*, *Vermont*, and *Joy*.

Depending on the processor, the byte order might require reversing before the fields can be used. Network packets traditionally use big-endian order, which means the most significant byte is sent first (e.g., the hexadecimal number 0x8100 is sent as 0x81, followed by 0x00), while, e.g., Intel x86 processors use the opposite order (e.g., 0x8100 is sent as 0x00, followed by 0x81). This is only required for numeric fields, since non-numeric fields like IP-addresses are always in network byte order.

Although this approach is simple and incurs very little overhead in terms of processing power and memory usage, there is the downside that all further protocol intricacies and corner cases must be manually handled. For instance, the IP length field of TCP packets captured on Microsoft Windows can be zero, although the minimum is 20, which is the length of the IP header [84]. This is caused by the usage of TCP Segmentation Offload (TSO), where splitting data, that is larger than the maximum packet size, into multiple packets, is handled by the network hardware. In this case, the captured packet contains headers that are used as template for creating the individual packets by the network hardware, and the payload is the unsplit data. Since the length field has no meaning in this case, it is set to 0 on Microsoft Windows. However, on the other OSs, this is handled by the network driver and not the OS, leading to a correct length field.

Another challenge arises if packets are truncated in the middle of a header, as depicted in Fig. 9. Depending on the protocol type, there can still be enough information left. In ICMPv6 type, code, and checksum use 4 bytes total [85] (type, code, and checksum have the same structure as in ICMP [86], as depicted in Fig. 11), which is sufficient to successfully decode the header. However, *Joy* and *yaf* treat ICMPv6 only as valid if it contains at least 8 bytes.

A similar problem arises with protocols containing optional fields. For instance, IPv4 and TCP might contain multiple options at the end of the header. While all decoding libraries in Table IV except *tshark* treat packets that are truncated just before or in the middle of the options part as invalid, *yaf* and *pmacct* treat those as if they were complete.

These and similar issues can also happen due to faulty values in packet fields, especially fields specifying lengths. In addition to failures in hardware or software, this could also be caused intentionally by malicious software. If the flow exporter is used for detecting network intrusions, such packets must be considered by the flow exporter. Olovsson et al. [87] compiled a list of observed header anomalies.

It is also possible to put the contents of a packet inside another packet. An example for this is a tunnel, where two networks are connected via a different network with a different addressing scheme. This is demonstrated in Fig. 12, where two hosts are connected via two routers that encapsulate packets in between. In this example an IPv4 layer is stored inside another IPv4 layer, which is called an IP in IP-tunnel [88]. Since the encapsulated packet layers don't need to match, it is also possible to encapsulate IPv4 in IPv6 or the other way around. There are other protocols that have been explicitly designed for supporting encapsulation – for instance GRE, where an example structure is IP inside GRE inside IP.

Depending on the use case of the flow exporter, these inner packets must also be decoded. Care must be taken with the inner addresses, since there might be multiple unrelated tunnels

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



using the same inner addressing scheme. In this case, different computers have the same network address, e.g., addresses from the private IP ranges as defined in RFC 1918 [89], which is not a problem, since they are in distinct networks. If network traffic of tunnels used by these computers is captured, using just the inner addresses for the flow key might assign unrelated packets to the same flow, which is done by *yaf*.

Packets that could not be successfully decoded must be reported by the flow exporter, and also mentioned in scientific research using this data. This is important since different packet decoding mechanisms tolerate different errors, which could lead to non-reproducible results due to the usage of a different flow exporter. Most exporters support this via some form of statistics including number of packets that failed decoding, which is output after program termination or at regular intervals. The IPFIX protocol supports such statistics as part of the data stream via option templates [13], which is supported by *yaf*.

### C. Flow Extraction

After the packets have been decoded, the next step is flow extraction. This step is split into the two parts timeout-based expiry and assigning the decoded packet to a flow.

Timeout-based expiry must happen before assigning packets, since a timeout could have occurred between two packets which have the same flow key. For these two steps, flows must contain at least the following properties:

- *Timestamp* of the next timeout, needed for timeout-based expiry.
- *Flow key*, needed for finding the flow for a given key.
- *Flow record*, containing values that will be exported upon flow expiry.

All active flows must be stored in a table, called flow cache [4], supporting lookup of flows by key.

*1) Timeout-based Expiry:* For timeout-based expiry, the table of flows must be scanned for flows that exceed one of the possible timeout limits (e.g., idle timeout, or active timeout).

Instead of using the real time (also called wall clock time), the timestamp of the current packet, as acquired by Section V-A, should be used. This makes the flow extraction process independent of the packet source used, e.g., stored packet captures can be used as input, and packets can be buffered, if they arrive faster than the flow extractor can process them. Additionally, timestamps attached to packets have higher resolution than the acquired real time.

This is still the case even if no hardware-based packet timestamping is available, since the OS can acquire more accurate timestamps compared to a user space application. Additionally, the point in time, the timestamp was acquired, will be much closer to when the packet was captured.

The drawback of this system is that time doesn't progress continuously, but in discrete steps forward on every packet arrival. Therefore, multiple flows could experience a timeout between two subsequent time instances when the flow exporter is triggered. Furthermore, a flow with the same flow key as the current packet could still be in the flow cache, even though it should have already been expired. Hence, as mentioned above, timeouts must be checked before assigning packets to flows. This approach is used by, e.g., *yaf* [2].

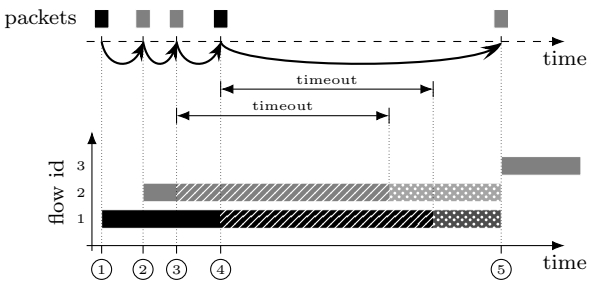


Figure 13. Packet timestamp as time source leading to discontinuous time and the need for handling timeouts before assigning packets to flows. This example shows the steps that have to be carried out upon the capture of the packets with flow keys black and gray: (1) Black: Create flow with key black, ID 1. (2) Gray: Create flow with key gray, ID 2. (3) Gray: Add packet to flow ID 2. (4) Black: Add packet to flow ID 1. (5) Gray: Handle timeouts between (4) and (5) (dotted flows should have been exported at this point in time): Export flow ID 2, followed by ID 1. Create new flow with key gray, ID 3.

If flows possessing a timeout value greater than the current time are encountered, the following steps must be carried out for every flow fulfilling this condition:

- The type of timeout must be written to the flow record assigned to the flow whenever the *flow end reason* is part of the record.
- The flow must be removed from the flow cache.
- The flow must be forwarded to the record output step (Section V-E).

If multiple flows are expired at the same time, the flows to expire must be sorted by increasing timeout. Otherwise, the flow output order would depend on the traversal order of the flow cache, resulting in a possibly non-deterministic flow output sequence.

An example, demonstrating this by means of multiple flows expiring between two instances of time can be seen in Fig. 13. Between timestamp (4) and (5), both flow ID 2, followed by flow ID 1 should expire, but due to the step-wise progress of time this is noticed at the later point, (5), at which both flows must be exported in the correct order (first 2, which expired earlier). If this time-based expiry would not be carried out first, then the last packet in Fig. 13 would be wrongfully assigned to flow ID 2.

*2) Assigning Packets to Flows:* After the timeout-based expiry step, the current packet must be assigned to a flow. This is done based on the flow key. As has been explained in Section II, the flow key defines the common packet properties used for matching packets to flows. The most commonly used flow key is the 5-tuple consisting of source and destination network address and transport ports, as well as the protocol identifier. This particular flow key results in flows resembling connections. However, it is not strictly necessary to use the 5-tuple as flow key. For example leaving out transport ports and protocol identifier would result in flows that represent all the data exchanged between two network addresses. Care must be taken in the results, since network addresses do not necessarily relate with hosts, since one host can have multiple addresses and hosts can hide behind other hosts, i.e., one address can be

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



shared by multiple hosts. An example for using a non-standard flow key is the work by Iglesias et al. [24].

For achieving this task, the flow exporter must maintain a flow cache that allows finding an active flow for a given flow key. Since this lookup has to be done for every packet, the lookup must be fast. This makes hash tables an ideal candidate for flow caches.

Hash tables allow storing values associated with a key. Values are found by mapping the key, which can be arbitrarily sized, via a hash function onto data of fixed size. This transformed key is then used as an index into a table, where the value is contained.

Theoretically, this makes the lookup fast and efficient since only the hash function has to be calculated instead of comparing every key in the table. However, the table can only have a limited size due to memory constraints, leading to the necessity of multiple keys mapping to the same index. Additionally, all occurring flow keys would need to be known beforehand to construct a hash function that doesn't result in the same value for multiple keys, called hash collisions. One way to solve this, is to store a list of values instead of a value inside the hash table entry. The correct value is then found by first using the hash as index into the table, followed by searching this list. This method is called chaining [90]. Since the search space is significantly reduced due to splitting the whole data into smaller parts based on the hash, the performance is still improved considerably compared to a naive search through all the keys.

An example of the whole process of finding the flow belonging to the decoded packet can be seen in Fig. 14. First, the flow key (five-tuple in this example) of the packet must be calculated. The simplest way to achieve this, is to concatenate the necessary fields, which is done in step (1). Next, the hashed flow key is calculated in step (2), followed by using it as an index into the flow cache in step (3). Since there is no value at this index, no flow with this key is found.

However, if bidirectional flows, i.e., flows that contain packets from both directions, are needed, the flow exporter has to repeat the lookup with a reversed key. This happens for all packets, where the direction is the opposite from the first encountered packet in the flow (i.e., source and destination properties are swapped).

In Fig. 14, the reverse key process is shown on the right side, starting with step (4). First, the reverse key is calculated by swapping source and destination of the network addresses A and B, as well as the transport ports 1 and 2. Next, the hashed flow key is calculated in step (5) and used as an index in the flow cache in step (6). Since there are multiple flows with the same keys, the flow key has to be compared to every flow in step (7), resulting in a flow with the same flow key as the decoded packet.

Many programming languages contain data structure implementations for handling the hash table functionality as part of the language. For instance *Python* provides the builtin **`dict`** datatype [91], *Go* the **`map`** datatype [92], and *Java* `Map<k, v>` and related classes [93]. All the implementations handle the complete hash table functionality, including hashing, table lookup, and remedies for possible hash collisions.

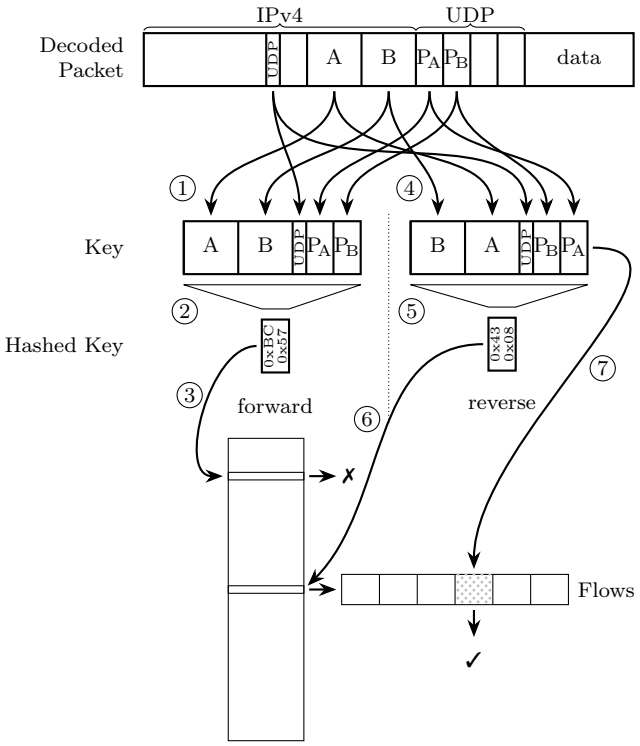


Figure 14. Flow lookup for IPv4 packet containing UDP and data. (1) First, the *forward* flow key is built (5-tuple – source and destination addresses and ports, and transport layer type). (2) From this key a hash value is calculated, which is used as lookup (3) into the *flow cache*. Since there is no entry, the *reverse* hashed flow key is calculated by exchanging source and destination (4) - (5). The lookup (6) with this hash value succeeds, leading to a list of flows. The flow belonging to this packet is found by comparing the flow key with all the flows in the list (7).

Since these implementations handle the complete lookup algorithm, it is not necessary to store the flow key inside the flow.

If a flow has been found, the flow together with the decoded packet and the metadata can be used for the next step, V-D Property Aggregation. However, if no flow is found, a new flow containing the required parameters must be initialized and inserted into the flow cache. This new flow must then be forwarded to the following processing stage.

### D. Property Aggregation

The previous steps V-B Packet Decoding and V-C Flow Extraction yield a *decoded packet*, *packet metadata*, and the *related flow*. With this information, the desired flow properties can be aggregated.

Flow properties can fit the categories flow-based, packet-based, or statistics-based (see Section II-B). Property aggregation has to be carried out for every property individually and depends on this category.

The category a property belongs to can also depend on which properties are part of the flow key. For instance, if the flow key does only include the network source address instead of both source and destination, a single flow possibly contains packets with multiple different destination addresses. Instead of packet-based, the network destination address must now

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



be a statistics-based property, since some form of selection has to be used to arrive at a property value. Possibilities could include creating a list of all encountered destination addresses, selecting an arbitrary address, or using an empty or illegal value if more than one address is encountered. An example for this is the illegal value solution used by Iglesias et al. [24].

Since the decoded packet might contain an indicator for flow termination, flow expiry might have to be processed after the property aggregation.

*1) Flow-based Property Aggregation:* As explained in more detail in Section II-B, flow-based properties are properties of the flow itself or of a significant packet in the flow. Therefore, these properties have to be set only *once* for the entire flow, regardless of the number of packets belonging to the flow. However, implementation details might break this rule, as will be explained below.

Examples for these properties are flow properties related to start and end of a flow:

- **Flow start:** the timestamp of the first packet in the flow.
- **Flow end:** the timestamp of the last packet in the flow. Beware that this is not necessarily the time, the flow expired, which is the case with timeout-based expiry.
- **Flow end reason:** the reason why this flow was stopped.

One way to implement *flow start* is to set the property to an undefined value during flow initialization. If this property has the undefined value during aggregation, the value has to be set to the timestamp contained in the packet metadata.

Since knowing in advance which packet will be the last packet of a flow is impossible, *flow end* could be implemented by always setting it to the timestamp contained in the packet metadata. With this implementation, *flow end* is always equal to the timestamp of the last packet in the flow.

Another flow-based property that might be set more than once is the *tcpSequenceNumber* property as defined by RFC 5102 [15]. The TCP sequence number is part of every TCP packet. During the connection setup, sender and receiver pick a random sequence number as start and send this number in the initial packets. These initially sent packets are marked with the Synchronize (SYN) flag [16]. After the connection setup phase, the sequence number increases with the number of bytes sent. Therefore, the initial sequence number, i.e., the sequence number in the first sent packet by sender or receiver, is meant with this property, although this is not specified in RFC 5102 [15].

However, multiple TCP connection attempts could be part of the flow, leading to the situation, that the first packet does not contain the initial sequence number of the connection. Furthermore, the connection setup could be missing from the packet observation (e.g., caused by the packet capture starting while a TCP connection is active). The real initial sequence number could be possibly contained in a different packet than the first one. The flow exporter *yaf* solves this issue by replacing the *tcpSequenceNumber* property with the current value, if the current value is lower than the recorded one. After all, the sequence number should increase for the duration of a connection.

However, caution is necessary since according to RFC 793 [16] the size of the TCP sequence number field is limited to $32\,\mathrm{bit}$, resulting in the highest possible sequence number of $4\,294\,967\,295 = 2^{32}-1$. To overcome this limitation, RFC 793 [16] states that all operations on this field must be performed modulo $2^{32}$, which leads to sequence numbers that would overflow the field, to start from $0$ again [15]. Additionally, the initial sequence number should be random, leading to a high possibility of potential overflows. Therefore, sequence numbers can't be simply compared and this might lead to differing results for distinct implementations.

*2) Packet-based Property Aggregation:* Packet-based properties are properties, that are common to every packet belonging to a flow. Like flow-based properties, packet-based properties have to be set only *once* for the whole flow.

These properties are typically packet header values that are also part of the flow key, e.g., network addresses, transport ports, or protocol numbers. Since these properties are the same for every packet, they can be set on the first packet.

Like the *flow start* flow-based property, this can be achieved by setting the property to an undefined value during flow initialization, followed by copying it from the packet if it is undefined during aggregation.

*3) Statistics-based Property Aggregation:* Common to all statistics-based properties is that a packet property of every packet of the flow is used as input for a statistical function. Packet properties can be fields from packet headers or part of the metadata (e.g., timestamp of the packet, or size of the packet).

The simplest approach would be to accumulate the desired packet property of all packets belonging to the flow in a list, followed by applying the function to this list at flow expiry. However, this has the drawback of requiring huge amounts of memory, since all required packet properties of every packet in every currently active flow must be stored.

If appplicable or available it is preferred to use online algorithms, which are algorithms that can process the input piecewise instead of requiring the whole data at once. This might require tracking more than just the required packet property. However, the algorithm will still use substantially less memory compared to recording every property since only the state of the algorithm needs to be stored.

A simple example is the *mean* value of a property, for which the exporter just needs to sum up all the packet property values, count the number of packets, and divide the sum by the number of packets at expiry time. For the online version summing and counting needs to be implemented with a partial sum and a partial count that are stored in the flow record after every packet. At flow expiry time those two fields contain the full sum and the full count, and the calculation can be finished with the required division operation. This algorithm is very close to the naive implementation, except for storing the interim results, therefore, replacing the possibly huge list of packet properties with the two partial values.

The same method can be applied for *minimum*, *maximum*, *sum*, *counting* number of values, and *mode*. However, using this approach might lead to numerical problems due to the limited accuracy of floating-point numbers in computers. For

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



Table VI
COMMON FILE-BASED FLOW FORMATS.

| format | overhead | complexity | records | support | human readable | data types |
|---|---|---|---|---|---|---|
| CSV [30] | high | low | fixed[1] | universal[2] | yes | text [3] |
| JSON [94], [95] | very high | high | flexible | universal[2] | yes | numeric, text, binary, structured [4] |
| IPFIX [4], [13], [27] | low | very high | flexible | limited | no | numeric, timestamp, IP, text, binary, structured |

[1] Every record must have the same set of fields.
[2] Libraries available for every environment.
[3] Everything must be converted to human readable text.
[4] Timestamps, IP-addresses not supported – those must be converted to text.

instance a simple online algorithm for the *variance* would be to subtract the squared mean of values from the mean of the squared values ($\sigma = \overline{x^2} - \overline{x}^2$). Both parts could be calculated with the online version presented above, but since both means can be very similar numbers, the precision of the subtraction could be very limited, leading to unexpected results. Welford [96] presented an alternative algorithm that can be used to achieve accurate results.

One statistical function that can not be computed by an online function is the *median*. For calculating the sample median, one needs to find the $n/2$ smallest value in the list of property values from every packet belonging to the flow. There are only estimators available that don't need every value, resulting in seldom use of the median [97], which is still true for flow exporters today.

The simplest approach to calculating the median would be to sort the packet properties and select the middle element, or the arithmetic mean of the two middle elements in case of even number of values. Although the memory requirements can't be reduced, at least this process can be sped up by using a selection algorithm instead of sorting. Selection algorithms find the $k$ smallest element by partially sorting the elements. Those algorithms work similar to sorting, but instead of sorting the whole set of numbers, only the part of interest is sorted. This means in the case of the median, that only the middle element is required to have the correct position, i.e., all elements to the left of the median must be lower than the median and all elements to the right of the median must be higher than the median. However, the list to the left and the list to the right can be in any order. One efficient example for such an algorithm is the Floyd-Rivest algorithm [98].

*4) Packet-based Expiry:* After property aggregation, packet properties need to be checked for an eventual *end of flow detected* expiry condition (see Section II-B for an overview of all expiry conditions). Unlike time-based expiry, this has to be done after property aggregation, since even if the current packet would cause flow expiry, it is still part of the flow.

The packet-based expiry implemented by most flow exporters is TCP-based expiry. TCP-packets contain special flags, that signal the end of a connection. One of these flags is the RST flag, that signals the immediate termination of the connection [16]. If this flag is encountered, the flow can be immediately expired.

In addition to reset, TCP contains a mechanism for graceful connection tear down. If one of the communication partners wants to end the connection they can signal stopping sending data by setting the FIN flag in a packet. This packet has to be acknowledged by the other communication partner. If both communication partner have sent a FIN packet and both have acknowledged this, the connection is terminated [16], and the flow can be expired. Unlike RST, this requires a state machine, to keep track of this tear down process.

Although RFC 793 [16] includes additional states in this connection termination process, most flow exporters use the simple model explained above. The additional states in RFC 793 handle the case for out of order packets, that could arrive after the connection has been already terminated. But if one wants to use this model, the full state-machine (including proper connection initialization with SYN) has to be implemented, to prevent spurious flows caused by packet loss or reordering. This can't be done by strictly following the instructions in the RFC, since packets might be missing from the packet observation, e.g, due to connections starting before the observation started, or packets being lost by the capture process. Therefore, only specialized flow exporters use the full TCP state-machine (see also Table II).

Packet-based expiry utilizing higher layer protocols is also possible, but expensive in terms of computational power since this requires decoding the whole packet. Additionally, this might not be possible due to the higher layers being encrypted or omitted in a packet capture. Therefore, most flow exporters only support TCP-based expiry. From the analyzed exporters, only *argus* implements expiry based on additional protocols (e.g., DNS, ICMP, HTTP).

### E. Record Output

The expiry steps explained in Sections V-C and V-D yield the final flow records for exporting. As already explained in Section II-D, flow record data can be written to a file or exported via network. Both approaches require converting the data to the data representation required by the target format.

In the following we present three alternative solutions for output formats: CSV-formatted, JSON, and IPFIX. An overview including format features can be seen in Table VI.

Due to it's simplicity CSV is a widely used format for writing flow data out to a file. Although standards for the exact format specification exist (e.g., RFC 4180 [30]), there are many different variants in use.

As explained in Section II-D, CSV is a text-based format where values must be formatted in a human readable way and delimited by a column delimiter (e.g., comma, semicolon,

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



or tab), and records delimited by a row delimiter (e.g., new line). Hence, values must be converted to human readable representations, e.g., numbers converted to text, IPv4 addresses to dot-decimal notation, and timestamps to text (e.g., RFC 3339 [99] compatible time and date representations). Typically, this capability is already provided by the language environment, for instance:

- functions `printf`, `inet_ntoa`, and `strftime` in C [100],
- function **`str`** and packages `ipaddress` and `datetime` in Python [91],
- packages `fmt`, `net` and `time` in Go [101].

Caution is needed whenever processing values that might contain one of the delimiters. If a field contains one of the delimiters, it must be bracketed by double quotes. Additionally, if a field enclosed with double quotes contains itself a double quote, the double quote must be escaped by two double quotes.

Delimiter types and exact field formats depend on the implementation of the software intended for consuming the file. If double quotes are not necessary, the CSV-format can be written directly by writing the fields converted to text with the delimiters in between. However, that would limit the flexibility of the output system to supporting only one specific variant and limited types of record values. Therefore, it is better to use libraries for this task (e.g., builtin package `csv` in Python [91], and builtin package `encoding/csv` in Go [101]).

Additionally, most software implementations are built on the premise that the number of fields is the same for every row in a CSV file. Therefore, workarounds are required for record values that are not applicable for every flow (e.g., TCP-based values like TCP-flags in non-TCP-flows). This could be as simple as leaving the value empty in the output, although, this requires that the target software can handle these fields.

An alternative approach to CSV is to write one JSON document per record to a file, delimited by new lines. JSON, like CSV, is a text-based human readable format, but unlike CSV, it supports structured data. According to [94], [95], JSON documents support the basic data types text, number, boolean, arrays (i.e., lists of values), objects (i.e., unordered set of key-value pairs), and an empty value. Text values must be enclosed with quotation marks, while number, boolean, and empty values are written directly in their human representation form (e.g., *true*, *false*, or *null*). Arrays must be enclosed with square brackets and can contain an arbitrary number of fields of any type, which must be delimited by a comma. Objects must be enclosed with curly brackets and contain an arbitrary number of combination of key, which can only be a string, followed by a colon and an arbitrary value, separated by a comma.

A simple approach of utilizing JSON for flow export would be to use an object, with the key identifying the record value type (e.g., "start time") and the corresponding value the record value. Additionally, this format allows to structure the data in a hierarchical way, e.g., use nested objects for representing the layers. Like with the CSV format, record value types not supported directly by JSON, e.g., IP addresses, and timestamps, must be converted to text and the required exact formatting depends on the software used to consume the data. A list of libraries, providing encoding functionality for JSON documents, is provided by [102].

Due to their text-based nature, CSV and JSON files are human readable, but this comes at the cost of required processing time for the binary to text conversion and larger file sizes due to text requiring more space than binary data (e.g., an IPv4 address requires $4\,\mathrm{B}$ in binary compared to a maximum of $15\,\mathrm{B}$ in text format). If JSON is used with the above mentioned method of utilizing objects with property names as keys, even more space is required, since every property name is contained in every exported record. Additionally, CSV and JSON provide no network transmission capabilities, but they could be transmitted via a TCP connection. However, handling of setup, unreliable or slow connections, and failures would have to be implemented with additional mechanisms.

This could be solved by using the IPFIX protocol/file format, which was specifically designed for exporting flow data. As explained in Section II-D, IPFIX aggregates multiple sets into messages, which can be either written to disc, or sent over the network via UDP, TCP, or SCTP. A set can either contain one or multiple templates, or one or multiple records. Templates specify the formatting, i.e., size and order of values and an ID identifying a value. Therefore, a template must be sent or written, before it is used as part of a record. IPFIX IDs are assigned by IANA in [18], but there is also the possibility to define custom enterprise IDs. Since these IDs specify which binary format to use for a given value, these additional specifications are also necessary for encoding. Hofstede et al. [4], as well as RFC 7011 [13] provide detailed examples and definitions.

Additional considerations for storing IPFIX in files is given by RFC 5655 [27]. This standard enhances IPFIX with additional information elements, which improve reliabilty for data at rest (e.g., checksums), information elements augmenting files with additional metadata (e.g., earliest timestamp, latest timestamp, information about the exporter), and compression options.

A software component, called *fixbuf*, capable of encoding and decoding IPFIX (C-library, and Python-package) is provided by the Carnegie Mellon University [103]. This library is used by *yaf*.

## VI. Reference Implementation: go-flows

As explored in Section IV, there is demand for features like non-standard flow keys, advanced statistical methods, calculations on flow properties, and non-standard flow properties that aren't covered by existing flow exporting solutions (see Section III). This lead to hand crafted solutions being implemented as part of scientific publications. These implementations add a huge overhead to the original work due to being a necessity and not the topic of the actual publication. This often resulted in non-reproducibility of the final work because code is not released as part of the publication or parts of the flow exporting process specification missing. Furthermore, those re-implementations lead to lots of duplicated effort, since core parts of flow exporting could be shared by every implementation.

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



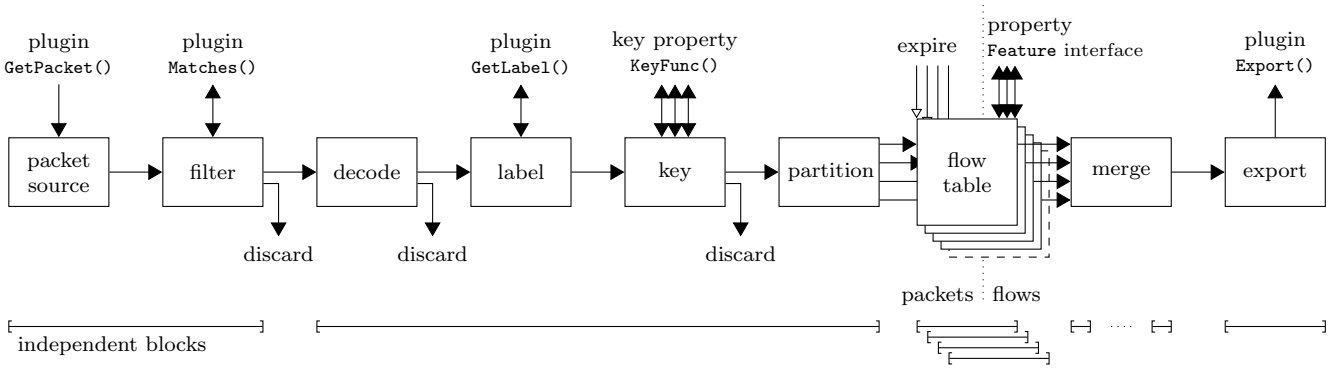


Figure 15. *go-flows* pipeline overview. Like in the tutorial (Fig. 8), packets pass through the steps *packet source*, *decode*, *key* function, flow extraction and property aggregation in *flow tables*, and finally *export*. In *go-flows* the additional steps *filter* (optional packet filtering) and *label* (optional label attachment to packets) are included. The plugin and property arrows in the figure identify well-defined interfaces that *go-flows* offers to users for seamlessly integrating their own custom-tailored method implementations for, e.g., `GetPacket()` for input, `Matches()` for filtering, or `Export()` for exporting. For increased performance, flow extraction and property aggregation is executed concurrently in multiple tables, requiring an additional *partition* step based on the key, and a *merge* step. Time-based expiry is not done continuously, but carried out in regular intervals, based on the timestamp derived from packets. All steps including an alternative *discard* step can discard packets (e.g., *key* if the packet is missing a property required by the flow key). Steps with a connection to *plugin* can be provided via a plugin and steps with a connection to *property* can be provided via property definitions, called features. A common set of plugins is provided with *go-flows*. The *independent blocks* are executed concurrently.

To address these issues, we developed a fully flexible open source flow exporter, called *go-flows*, based on the tutorial presented in Section V with the following features:

- Fully flexible flow exporter: Flow key, flow properties, and flow life cycle can be changed via configuration and plugins.
- Flow properties can be specified as combination of packet properties (e.g., packet size, header values), operations (e.g., statistical functions, arithmetic, logic), and selections (e.g., apply packet properties only to a subset of packets in a flow).
- Fully customizable flow exporter: Packet input, filter, optional labels, key properties, packet properties, operations, selections, and record export are extensible via a plugin system.
- Cross platform: Windows, Linux, and BSDs are supported.
- Reproducibility: Configurations must be supplied using the flow specification part of the NTARC file format, a machine-readable format for storing data about research publications in the network traffic analysis field [104], [105].
- Although the flexibility incurs a performance penalty, *go-flows* was tuned for performance, and is therefore on par with the analyzed flow exporters in Section III (see Section VII for performance and flow evaluations).

In the following Section VI-A the architecture is explained in more detail. This includes a description of how plugins must be specified and details about internal processes in the flow table. This is followed by an in depth description of the property extraction model in Section VI-B, explaining how properties are implemented, followed by a description how property combinations must be specified in Section VI-C. The description of the reference implementation *go-flows* is closed by the summary in Section VI-D which includes a reference for obtaining the full code of the implementation.

### A. Architecture

The overall architecture of *go-flows*, which follows the one described in the tutorial in Section V and Fig. 8 closely, can be seen in Fig. 15. Additional steps have been inserted into the architecture, as described in the tutorial, to support packet filtering and labeling. Furthermore, *go-flows* can partition, i.e., distribute, packets based on the flow key over multiple tables, which assigns all the packets belonging to the same flow to the same table. Due to flows being completely independent, the tables can handle flow extraction concurrently, therefore increasing processing speed. Since this causes non-deterministic output order induced by process scheduling, an additional merge step is inserted before record output.

The steps *flow extraction* and *property aggregation* from the tutorial are handled inside the flow table blocks. This block will be explained in detail in the next section.

Overall speed is further increased by splitting the execution pipeline into the following blocks, which are also executed concurrently (see independent blocks in Fig. 15):

- packet source and filter,
- decode, label, key, and partition,
- single merge stages,
- and output.

Moving data between concurrently running blocks requires synchronization, which lowers the performance. This can even lead to a performance that is worse than the one of a non-concurrent pipeline. Hence, packets are moved between those blocks in batches of 1000 packets to lower the ratio between synchronization time and packet processing time.

Memory required for the packet data and the decoded fields is provided by a dynamically resizing ring buffer. This means that instead of allocating memory upon receiving and decoding a packet and releasing it again after processing is finished, packet-memory is reused. This technique increases the performance further due to memory allocations being an expensive operation. Additionally, most decoded header fields

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



Listing 1
EXAMPLE *go-flows* INVOCATION SPECIFYING PLUGINS.

```
go-flows run features vladutu.json export csv out.csv source libpcap input.pcap
```

don't contain copies of the values, but only point into the position in the original raw packet data.

Therefore, a single block must not store references to specific packets or header fields for later use (e.g., for packet reordering), since these might be overwritten by other packets. Hence, header fields must always be copied. Additionally, blocks requiring packet storage must use a copy function specially designed for this case, which marks the packet as in use, preventing it from being overwritten. Packet copies acquired via this mechanism must be freed after they are no longer required, otherwise those packets can't be reused anymore, which might result in exhausting the available memory.

*go-flows* has been designed to be fully modular. Every step, except for decode, partition, and merge can be customized. This is implemented via a plugin architecture providing simple APIs for the specific tasks, while the *go-flows* internal blocks handle the above mentioned details.

This plugin-based architecture allows supplementing *go-flows* with potentially required functionality and customizations. Every aspect of the flow exporting process is implemented this way, therefore, enabling the usage of the flow exporter in every possible flow-based research and even some non-flow-based cases. The open source nature of the project allows contributing these improvements back to the project, which increases reproducibility and enriches possible future research.

Due to differing requirements, this plugin architecture is split into the different categories plugins, key properties, and features. While plugins handle a complete task (e.g., read a packet from a source, export a flow), key properties and features are responsible for only part of a task. For instance, the key property *sourceIPAddress* extracts only the network source address. The five tuple requires therefore five key properties. One feature can potentially compute just one part of a property. *Mean* would just calculate the mean of another feature.

*1) Plugins:* Common to all plugins is that they must implement `ID`, `help`, `new`, and `Init` functions as part of their interface. For debugging, visualization purposes, and user interaction, the `ID` function should return a human readable identifier and the `help` function must provide a usage description. Plugin initialization must be handled by the `new` function, which will be provided with the user specified input arguments. This function will be called during argument parsing and should therefore not create files or start execution. These initializations should instead be carried out in the `Init` function, which will be called after the initialization is finished and before the actual flow extraction process starts.

Additional functions must be provided depending on the plugin type. The following plugin types are available:

**Source:** Source plugins read raw packets and metadata from a packet source. The plugin-provided function `ReadPacket` must return the type of the lowest layer, packet contents, packet metadata, how many packets were skipped, and how many packets were filtered or an error value. Skipped packets must be the number of packets that could not be captured, e.g., caused by buffer underrun, while filtered must be the number of ignored packets due to filtering since the last call to `ReadPacket`. If all the packets have been read, an end of file error must be returned. If multiple source plugins are used, only one will be used at any given moment. Execution switches to the next one upon encountering an end of file error.

**Filter:** Filters must provide the `Matches` function which will be called for every packet acquired from a packet source. This function is provided with layer type, raw packet data, metadata, and packet number of every packet and must return true, if the given packet should be used. In case of multiple filter plugins, they will be called in order until one filter does not match. If all filters match, the packet will be used.

**Label:** Label plugins can provide additional data that will be attached to the packets used for flow exporting. The provided function `GetLabel` is provided with the decoded packet and metadata and can return arbitrary data that will be attached to the packet. Additionally, an error value must be returned. If there are no more labels left, an end of file error must be returned. Multiple label plugin handling is the same as with multiple source plugins.

**Export:** Export plugins must provide the three functions `Fields`, `Export`, and `Finish`. `Fields` is called before the first packet is processed and is provided with a list of textual representations of every property that has to be exported. This can be used to write a file header if necessary (e.g., for CSV-based export). For every flow that got exported, the function `Export` will be called with a template, the property values, and the export timestamp. The template contains the field descriptions of the given data (e.g., for IPFIX-based export). After every packet has been fully processed and every flow exported, the `Finish` function is called, which can be used to flush unwritten data. If multiple export plugins are used, the exported flows are passed to every export plugin.

Several plugins for packet source, filter, label, and export are already included with the source code.

Plugins have to be specified via command line arguments upon flow exporter invocation. The general usage is `type name arguments`, where `type` is the plugin type, `name` the plugin name and `arguments` the arguments needed by the plugin. At least one source and one export plugin must be provided, the others are optional. An example invocation with *CSV*-file export to `out.csv` and *libpcap*

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



packet source reading from `input.pcap` can be seen in Listing 1. *Features* specifies a file containing a description of all flow properties, which will be described in Section VI-C.

Instead of requiring one plugin per key property combination, each key property or source and destination property pair is specified on its own. This enables more flexibility compared to the existing flow exporters, but requires a different mechanism than the generalized plugin mechanism described above.

*2) Flow Key Properties:* Like plugins, key properties can also be provided externally, but use a different mechanism compared to plugins due to different usage patterns. Several `Register*` functions are available for registering different key property types with the flow extractor.

Key properties must provide one or multiple names or a regular expression, a description, directionality, layer type, and a function capable of returning a key property function. Directionality can be either source, destination, or unidirectional. Together with layer type, the information is required for possible bidirectional flow extraction.

As has been explained in the tutorial Section V-C, one way to implement bidirectional flows is to calculate the flow key for forward and reverse direction. Since this adds the performance overhead of possibly calculating the flow key twice and doing the lookup twice, *go-flows* uses the following improved mechanism: Instead of one monolithic key, the key is split into the three parts source key, destination key, and unidirectional key. Source key and destination key contain source properties (e.g., source network address) respectively destination properties (e.g., destination network address) ordered by OSI layer, if both source and destination are part of the key properties. The unidirectional key contains all unidirectional properties, i.e., properties that are the same for both directions (e.g., protocolIdentifier), arbitrarily ordered. The final key is a concatenation of the three keys source, destination, and unidirectional.

In the bidirectional case, the flow key is computed similar to the unidirectional case, by concatenating the three keys source, destination, and unidirectional. However, in the bidirectional case the ordering of key parts is important. Before concatenation, source key and destination key are compared bytewise. If the bytewise comparison yields that the destination key is smaller than the source key, then the position of source and destination is swapped. This key composition method safeguards identical keys for forward and for reverse direction.

In case multiple different protocols are encountered on the network layer, the resulting flow keys are distinct, if the address sizes are different. For example IPv4 and IPv6 packets would result in IPv4 and IPv6 flows due to IPv4 using $32\,\mathrm{bit}$ addresses and IPv6 $128\,\mathrm{bit}$ addresses. In this case it is not possible that IPv4 packets end up in IPv6 flows, even when IPv6 mapped IPv4 addresses are encountered.

Following the key calculation, the decoded packet is forwarded to a flow table based on the flow key. This ensures that packets belonging to a single flow are always handled by the same table. Since all flows are independent of each other, the multiple flow tables (see Fig. 15 and Section VI-A) can handle flow extraction concurrently.

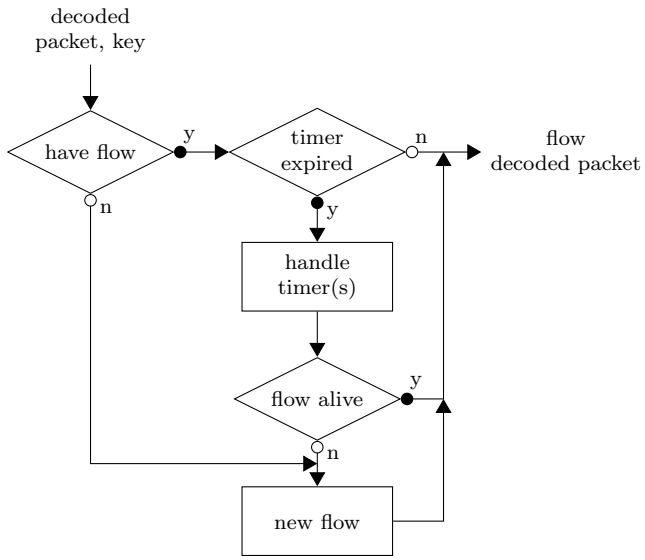


Figure 16. *go-flows* flow extraction pipeline inside the flow table block from Fig. 15. First, the flow key is used for finding the flow in a hash table. If a flow is found, its *timers* are checked for expiry conditions and executed if necessary. After timer handling, the flow is checked, if it is still *alive*. If the flow was expired, or no flow has been found, a *new flow* is created. In the last step, the *decoded packet* is forwarded to property extraction inside the *flow*.

*3) Flow Extraction/Flow Table:* The flow extraction step from Section V-C is distributed over multiple tables in *go-flows*. Since, as explained above, packets are partitioned over tables based on the flow key, all packets belonging to a single flow will always be assigned to the same table. This might result in packets not being balanced perfectly over all tables, which further causes non-optimal speed-ups due to tables receiving less work than others. However, this technique ensures that every table can work fully independent, resulting in a simpler and faster implementation.

Flow extraction performance is further improved by handling timeout based expiry at a configurable time interval instead at the arrival of every packet. The default interval is $100\,\mathrm{s}$. Larger intervals increase memory usage, since flows that would have been expired need to be kept longer in memory, while lower intervals increase processor usage due to more frequent expiry checks. Therefore, the interval length represents a trade-off between performance and memory usage.

However, this method requires an additional step compared to Fig. 8 after a flow has been found for a given key. Since timeout based expiry can be delayed significantly, the flow stored in the flow table belonging to the current packet could already be in a state where it should have been expired and a new one should have been created.

If a flow belonging to the key is found, first eventually expired timers have to be checked. Due to the delayed expiry handling, the flow found could be in a state where it should have already been expired. If there have been no missed timer events, or the flow is still alive after timer expiry, the existing flow is used. Otherwise, a new flow is created and inserted into the flow table (see Fig. 13 for an example). Finally, the decoded packet and the flow is forwarded to the property

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



extraction, which will be explained in Section VI-B.

Flows are stored in a hash table, using the flow key as key.

The flow extraction steps used in *go-flows* are outlined in Fig. 16. First, the flow is searched in the flow table using its flow key. If no flow was found, a new one is created and inserted into the flow table. Since, as explained earlier in Section VI-A2, the flow key in *go-flows* is the same for both directions in bidirectional flows, no additional checks are required.

*4) Flow merging:* Flow records from expired flows, need to be merged into a single stream, since *go-flows* utilizes multiple tables for flow extraction concurrently (see also Fig. 15 and Section VI-A). Due to OS scheduling being non-deterministic, the record output order would also be non-deterministic, if records would be appended in appearance order to the output queue. Furthermore, flow expiry might not have happened in the expected order due to the delayed processing of expired flows, as explained above.

To remedy this issue *go-flows* supports sorting record output by flow start time, flow stop time, and flow expiry time. This is implemented in two parts: inside each flow table and in the *merge* step.

Inside each flow table, empty flow records are prepended to a list upon flow start. For flow-stop-based sorting, the flow record is moved to the front of the list on every packet. With this technique, the list of flow records is already sorted in reverse order. Therefore, after processed events and after expiry handling, the flow table removes finished records from the back of the list and forwards those in reverse order to the merge queue for start and stop sorting.

Expiry sorting works like stop sorting, except that flows are also moved to the front of the queue upon expiry and after full table expiry processing all finished records are forwarded to merging. This list will only be partly sorted and requires therefore an additional sorting step. This sorting step is implemented with *natural merge sort*, which has the best performance with partially sorted data, due to reusing already sorted sub-sequences.

The merge step is implemented in a tree, with single stages merging two streams of flow records into one. These steps are implemented like the merge step from *merge sort*, which means that only records from the input queue with the smaller time value are selected for output.

Thus, flow sorting is a cheap process since, except for expiry sorting, an algorithm was implemented where no actual sorting is needed. However, stop sorting can incur a high memory overhead, since a high number of records must be kept in memory due to the stop time becoming only known after expiry timeouts for most flows.

Stop sorting is the default, with the possibility to turn off sorting, if output order is irrelevant to further flow processing. Turning off sorting improves memory usage and limits required processing power.

### B. Property Extraction Model

In all the existing flow extractors and the tutorial above, property extraction is carried out in fixed functions (e.g., minimum packet count, total number of packets, average packet size), combining statistical function and property extraction. This leads to inflexible flow exporter solutions and was a contributing factor to the development of custom flow extractors for scientific works, as explained in Section IV.

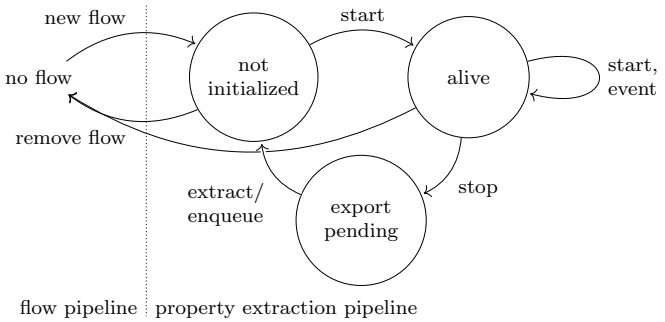


Figure 17. Flow states in *go-flows*. Flows can be in the three states *not initialized*, *alive*, and *export pending*. Flows are created by the flow table in the *not initialized* state, where features are not yet initialized. Feature initialization is done with start, which marks the flow *alive*. In the *alive* state events modify the features state, but also start can be called, which reinitializes all features. For exporting the flow, stop is carried out for every feature, finalizing their values, bringing the flow into the *export pending* state. Finally, extracting and exporting the values changes the flow back into the *not initialized* state. If the flow is still *not initialized* at the end of event processing, the flow is removed from the flow table.

The examples given above are all a combination of a statistical function (min, sum, average) and a packet property. In *go-flows*, flow properties are therefore a combination of operations, packet properties, and flow properties. Those are called *features* in *go-flows*. With this model, only a small number of features implemented in the flow extractor lead to a huge possible number of combinations. Additionally, more advanced functionality like, e.g., extracting properties from a subset of packets of a flow is possible. This allows *go-flows* to be used as exporter for the QoS work by Nguyen et al. [25]. Furthermore, operations can modify the flow life cycle by filtering packets (*filter features*), expiring flows, removing flows, and resetting flows (*control features*).

Like key properties, features can be implemented by external, user-provided libraries and modules. Those can be registered via `Register*` functions, which must be provided with name, description, type, data type, and implementation.

Several functions are called on every requested feature during the flow life cycle. Only the required functions must be provided, leading to most of the features just implementing one function (`Event`). These functions are best described by means of the flow life cycle as described by Fig. 17 and the property extraction pipeline in Fig. 18.

Flows start out in the *not initialized* state, which means that all features are uninitialized and might contain stale or incorrect values. Starting the flow leads to *start* being called for every feature, changing flow state to *alive*. Even if the flow is already *alive*, start can be called, to reinitialize every feature, which is equivalent to resetting the flow. In the *alive* state, the flow can also be removed, without exporting, by a control feature. In the *alive* state every packet is processed by the features. For exporting the flow, first the features are *stopped*, where the values can be finalized, changing flow state

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



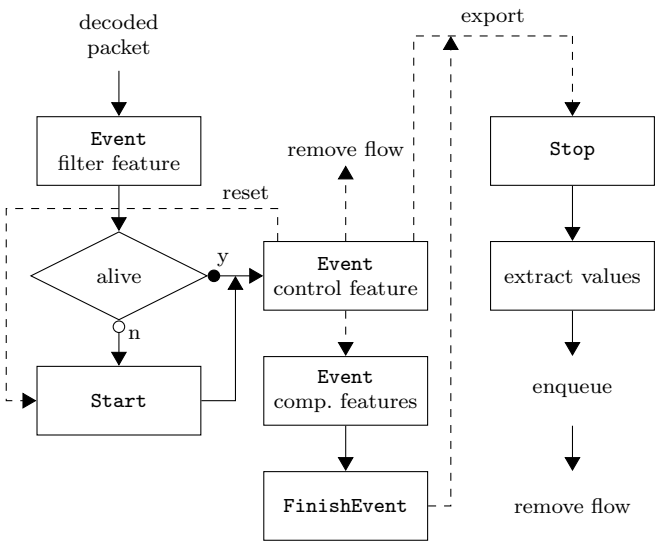


Figure 18. *go-flows* property extraction pipeline. First, *filter*-features are tested for packet rejection. Next, the flow is checked, if it is *alive*. If it is not *alive*, all features are *started*, initializing all values. This is followed by processing all *control* features, which can reset, stop, or export the flow. After filter and control, all *compute features* requiring a decoded packet as input are executed in the *event* step. Those features call dependent features, e.g., features that aggregate the just extracted packet property (see Fig. 19 for an example), with their newly generated values as input. Finally, internal state tracking, responsible for tracking which feature was executed, is reset in *finish event*. Upon flow export, *stop* is called on every feature, allowing final calculations, followed by *extracting the values* of features that get exported. After this, the record is *enqueued* for export and the *flow removed* from the table.

to *export pending*. After this, the feature values, i.e., the flow properties, are *extracted*, the final flow record built, and the flow record *enqueued* for export, which changes flow state to *not initialized*. If the flow is in the *not initialized* state at the end of processing, the flow is removed from the table.

The mapping from these state transitions to plugin functions can be seen in Fig. 18. First, for every packet, every filter-feature is queried, if the packet should be processed for this flow. Only if all required filter-features allow the current packet, processing continues. This further means, that if the flow was *not initialized* and a filter-feature disallowed the current packet, the flow is removed.

If the current packet is approved, and the flow is *not initialized*, `Start` is called on every feature. Features must initialize their internal data, if necessary, in this function (e.g., *sum* function must reset their accumulated value to 0). After this, the flow state is changed to *alive*.

Next, the `Event` functions of control features, which are features that are capable of changing the flow life cycle, are called. This could lead to resetting the flow, meaning `Start` is called, resetting the flow, followed by control feature processing. Therefore, control features resetting the flow must not reset the flow more than once for the same packet, which would create an endless loop. Other possibilities are removing the flow or exporting the flow now or after all features have been processed for the current packet.

If processing continues after the control features, `Event` is called for all features operating on the decoded packets (e.g., features that extract packet properties). Those might set a value, which in turn calls the `Event` function on dependent features, leading to values trickling down a tree of features. For instance, calculating the average packet size would first call the `Event` function on the packet size feature, extracting the packet size and set it as the current value. This in turn calls the `Event` function on the average feature providing the packet size (see Section VI-C for more examples).

Since features might not generate a new value on every packet and features can use multiple other features as input `FinishEvent` is called after the current packet has been processed. This is part of the internal tracking if all arguments of multi-input features produced a new value. Multi-input features can use this mechanism to only act on provided values, if every feature, used as input, provided a value for the current packet.

On flow export, first *stop* is executed. As explained earlier in the tutorial in Section V-D3, many pipeline-based statistical property implementations need to carry out work before a flow is exported. Therefore, the first step during export is to call `Stop` on every feature. This can be used to, e.g., divide the accumulated sum by the accumulated event count for calculating the average.

After post-processing of every feature is finished, the values are extracted from the features marked for export. Those values are the final flow properties, which are combined into a flow record. This is followed by enqueuing the flow record for export together with a template describing the properties, and removal of the flow from the flow table.

### C. Property Specification

Due to the generalized way properties, called features, can be combined to build meta features (e.g., combination of multiple operations and packet or flow properties), an unambiguous machine-readable format for specifying properties and key properties is needed. The NTARC file format, a machine-readable format for storing data about research publications in the network traffic analysis field [104], [105], fulfills this property. Additionally, it enables specifying parameters like active timeout, idle timeout, and directionality. This can be used as a simple format for providing accompanying metadata to scientific research that improves usability and repeatability. Furthermore, the existing database of curated papers at [58] can be used as input for *go-flows*.

Since the NTARC format contains additional non-flow related data, a simpler format containing only the flow specification can also be used. However, if a complete NTARC file is used, it might be necessary to specify which flow specification should be used, since more than one flow specification can be contained in a single file.

One feature of the NTARC format that is not allowed by *go-flows* is that parts of the specification can be missing. In the file format this is allowed due to the original intention of using it for curating existing scientific publications in network traffic analysis. This is required due to not every publication containing all information necessary for reproducing the experiment, which was also explored in Section IV. *go-flows* intentionally disallows this to increase reproducibility of the experiments it is used for.

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



Listing 2
PARTIAL INPUT EXAMPLE FOR [57]

```
{
    "active_timeout": 1800,
    "idle_timeout": 300,
    "bidirectional": false,
    "features": [
        "packetTotalCount",
        "flowDurationMicroseconds",
        "octetTotalCount",
        {"apply": [
            "octetTotalCount",
            {"select_slice": [0, 10]}
        ]},
        {"stdev": ["ipTotalLength"]},
        {"minimum": ["ipTotalLength"]}
    ],
    "key_features": [
        "sourceIPv4Address",
        "destinationIPv4Address",
        "sourceTransportPort",
        "destinationTransportPort",
        "protocolIdentifier"
    ]
}
```

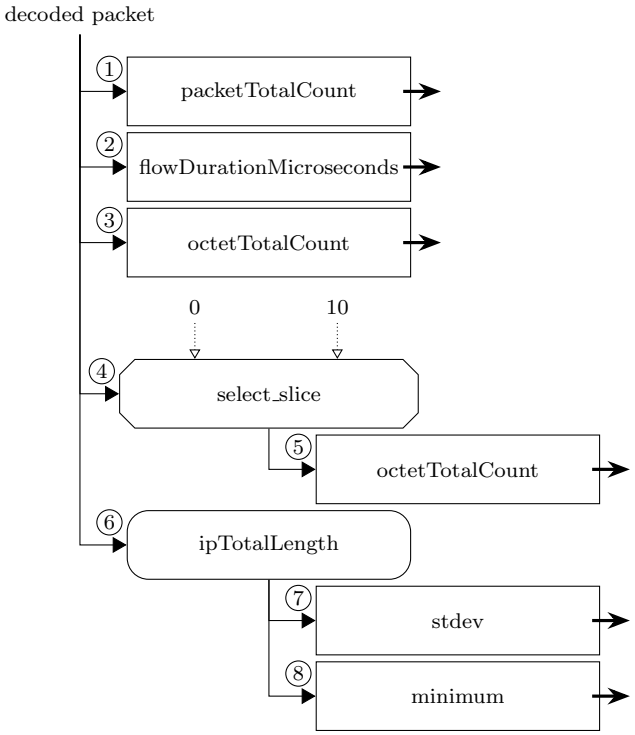


Figure 19. *go-flows* call graph computed from Listing 2. Features using the decoded packet as input will be called one after another. The order of execution is depicted with circled numbers. Constant values are provided to features at initialization (e.g., the numbers *0* and *10* to *select_slice*). Features that compute values based on the result of other features will be called in a tree, e.g., *ipTotalLength* calls *stdev* and *minimum* and provides both with the calculated length. Computed values are shared if possible to conserve memory usage and required processing time.

The NTARC format is based on JSON. A small partial example, using the simple format, implementing parts of the flow specification of [57], which was mentioned in Section IV, can be seen in Listing 2. This example was taken directly from the NTARC database at [58], shortened for clarity, and made usable by replacing the missing timeout specification with commonly used values.

The simple format consists of a JSON object with the key value combinations describing the different flow specifications. For instance Listing 2 specifies an *active timeout* of 1800 s, an *idle timeout* of 300 s, *unidirectional flows*, and the five-tuple as *flow key*, called `key_features`.

Standard key features (i.e., flow key) and standard features (i.e., flow properties) must use the IANA naming convention at [18]. However, it is also possible to use non-standard features, which must start with a leading underscore (e.g., `_interPacketTimeNanoseconds` representing the time difference between two packets).

The *features* part can contain operations, allowing for combinations and variations of existing features. For example Lines 13 and 14 in Listing 2 contain the combination of the packet feature `ipTotalLength` with the operations *standard deviation* and *minimum*. *Features* must contain a list, where each entry represents a single value in the final flow record (e.g., Line 14 will result in the `minimum` of the `ipTotalLength` and not `minimum` and `ipTotalLength`).

Another example demonstrating the use of only a subset of the packets for calculating a flow property can be seen in Lines 9 to 11 in Listing 2. Here the flow feature `octetTotalCount` is *applied* to the first 10 packets (packets 0 to 9 – NTARC uses Python-like indexing: two indices specify a range, where the second index is one higher than the maximum index; indices start at 0) in the flow.

As explained in Section VI-B in detail, the specified features are executed in a tree, where features receiving the decoded packet are called, and those in turn call the features depending on their output values. The resulting call tree for Listing 2 can be seen in Fig. 19, which contains features that generate a value per packet (rounded edges) and features that generate a value per flow (rectangle). Another feature, providing a subset of decoded packages, is marked with cut edges.

The first three features are called one after another, since there are no additional dependencies.

`apply` does not appear in the call graph, because it just changes the way packets are provided to the flow property in the first argument. In Listing 2 this means that instead of calling the second `octetTotalCount` (Line 10 in Listing 2) on every packet, the packets are filtered first with the `select_slice` feature, which in turn provides the filtered packets to `octetTotalcount`. This can be seen in Fig. 19, where the feature in step 5 receives packets from step 4 instead of *decoded packet*.

After this, the `ipTotalLength` packet feature will be called in step 6 with a decoded packet. Due to both `stdev` and `minimum` using the same value as source, `ipTotalLength`

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



will be computed only once and the value shared by the two operations.

During flow expiry, the values of the features marked with an arrow will be copied to the flow record.

### D. Summary

The flow exporter *go-flows* was built with the help of the tutorial in Section V with a focus on flexibility, extensibility, reproducibility, and an eye towards performance allowing for short evaluation cycles. It supports arbitrary flow key and property combinations, including operations on packet properties and statistical functions.

Specifications of key properties and flow properties as well as all necessary parameters required for the flow exporting process (e.g., expiry timeouts) must be provided using the NTARC format. This spurs reproducibility by providing a simple mechanism for exchanging flow specifications and forcing the selection of every necessary parameter.

Due to the flexibility of *go-flows*, and the detailed description of the flow exporting process, it was possible to re-implement the custom built flow exporter used by Iglesias et al. [24] with *go-flows*. This publication used the non-standard flow key consisting of timestamp in minutes, source network address, and destination network address for flow extraction and custom properties for flow aggregation. Since this publication exported custom flow properties, some of the features had to be implemented with the extension mechanism described above. Even in this case the implementation effort was reduced substantially by using *go-flows'* property specification mechanism that allows features to share common functionality. The original implementation required a combination of multiple programs (*editcap*, *tshark*) and scripts written in different languages (python, perl) implementing a complete flow exporter from scratch.

All the publications listed in Section IV could use *go-flows* instead of the used flow-exporter-like software or custom implementations. Due to the property extraction pipeline allowing control features to modify flow behavior (see Fig. 18) it would also be possible to implement the first step of the QoS work by Nguyen et al. [25], which requires the flow to be expired after the first couple of packets of a flow were captured.

The flexibility even enables the usage of *go-flows* for flow related tasks, like creating one PCAP file per flow, which contains the packets belonging to the flow. Combined with the possibility for a wide range of flow key combinations, *go-flows* can therefore be used for PCAP-splitting and filtering tasks.

The complete source code of *go-flows* including plugins and features is provided at [106] under the GNU Lesser General Public License (LPGL). This repository additionally contains examples, including the above mentioned features for the scientific work by Iglesias et al. [24]. The open source nature of this project will help to spur innovation in the field of flow exporting. It provides an easily extendable base set of flow and packet properties, functions that can be arbitrarily combined, and addresses future challenges by seamlessly integrating user-defined plugin extensions for most of the processing stages as detailed by the tutorial in Sections V and VI.

Table VII
PACKET STATISTICS OF MAWI SAMPLEPOINT-F 201702011400.

| protocol | packets/# | packets/% |
|---|---|---|
| TCP | 57 161 109 | 65.57 |
| UDP | 7 706 500 | 8.84 |
| ICMPv4 | 21 094 217 | 24.20 |
| ICMPv6 | 97 073 | 0.11 |
| other | 1 114 653 | 1.28 |
| total | 87 173 552 | 100.00 |

## VII. EVALUATION

Flow generation and flow exporting are commonly used preprocessing steps for exploring network data and determine the input for subsequent analysis steps. Therefore, they play a central role in network traffic analysis and can highly influence results and findings. Due to the importance of these processing steps we compare the output of different flow exporters and show that results differ vastly, which also inhibits the comparability of scientific results that were produced based on different flow exporters.

The flow exporters described in Section III and Table II (*go-flows*, *argus*, *pmacct*, *Vermont*, *yaf*, and *Joy*) were evaluated with regard to flow output and runtime performance. For this comparison, we presented every flow exporter with the same PCAP file and compared the results to *go-flows*. Since not every exporter supports every possible combination of extracted properties, the common subset consisting of i) five-tuple, ii) octetTotalCount, iii) packetTotalCount, iv) tcpSequenceNumber, v) and reverseTcpSequenceNumber was specified for flow extraction.

All tests were conducted on a dual Intel Xeon E5-2260 v2 (total of 20 physical/40 virtual cores) with $64\,\mathrm{GiB}$ system memory and a $512\,\mathrm{GiB}$ Samsung Solid State Drive (SSD) 850.

As input file, MAWI sample data [107] published at [108] was chosen, since it contains a large amount of real world traffic that represents multiple challenges to flow exporters:

- High number of packets and flows, e.g., the MAWI samplepoint-F trace 201702011400 lasting $15\,\mathrm{min}$ contains $\approx 87 \times 10^6$ packets resulting in $\approx 25 \times 10^6$ flows. A more detailed packet statistic can be seen in Table VII.
- High number of packets that can only be expired via timeout (e.g., UDP, ICMP).
- Truncated packets. Due to privacy concerns no packet contains a payload. Additionally, the capture size was limited to $96\,\mathrm{B}$ resulting in partial packet headers in many cases.
- Missing and duplicate packets.
- Malformed packets.

All the following analysis was based on MAWI samplepoint-F 201702011400, which is freely accessible at [108] and described by [107].

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



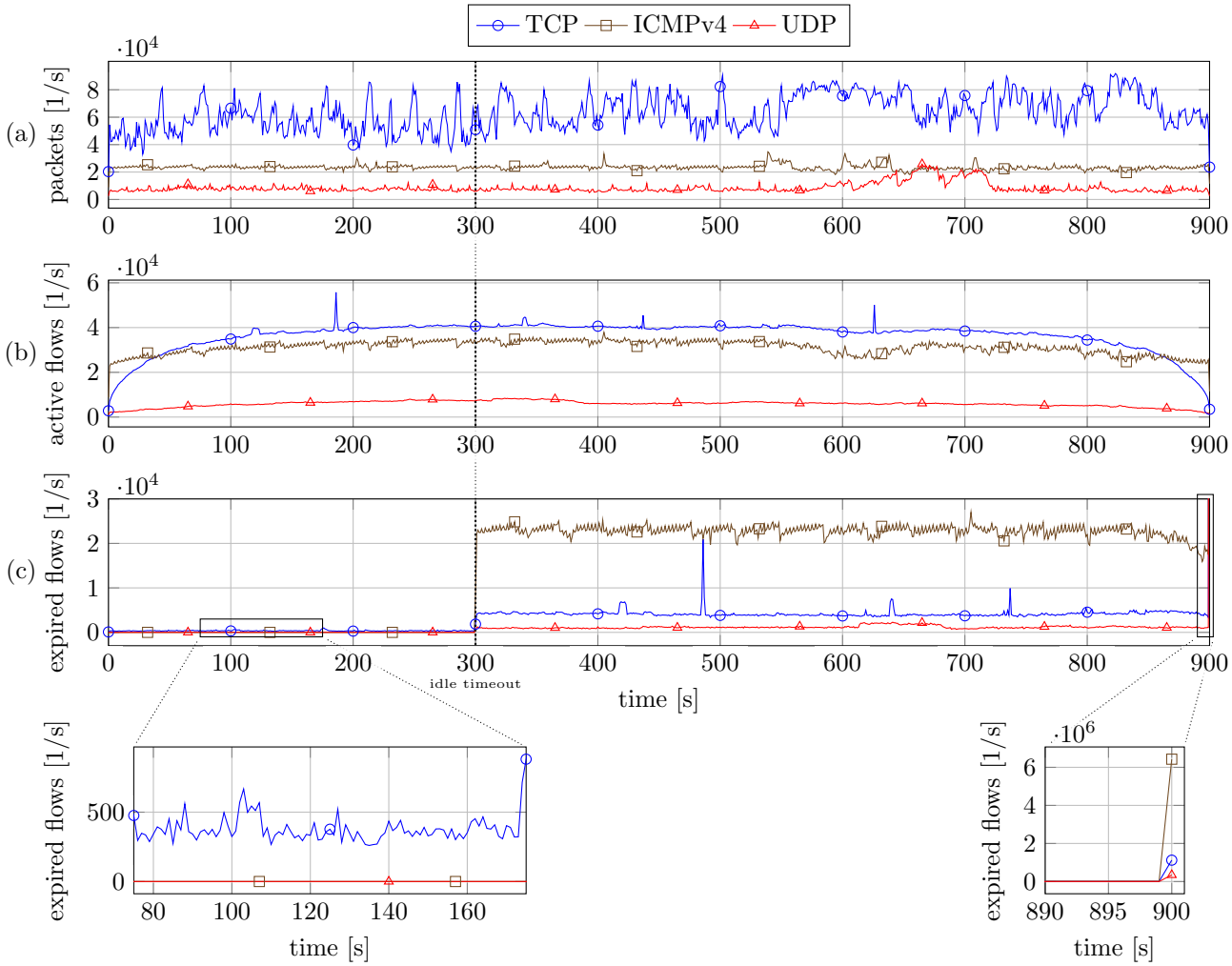


Figure 20. Non-stacked packets (a), active flows (b), and expired flows (c) per second over time of MAWI samplepoint-F trace 201702011400 for TCP, ICMPv4, and UDP traffic. The active flows per second (b) depict flows where the start time is before the point in time in the plot and the end time is after it. Since this resembles a moving average of the packets per second over an expiry sized window, these curves are flattened at the edges. Expired flows per second (c) is the more important metric for flow exporters, since flows must be stored until they expire, which might be long after their stop time. This can be seen in the third plot (c), where ICMPv4 and UDP flows start expiring after the 300 s mark, which was the idle timeout used. Part of the TCP flows expire before this mark due to TCP-based FIN and RST expiry. At the end of the file, all remaining flows (1 123 211 TCP-flows, 6 426 145 ICMPv4-flows, 345 416 UDP-flows) must be expired, which can be seen in the zoomed-out plot on the lower right.

A packet and flow over time plot of the MAWI samplepoint-F 201702011400 can be seen in Fig. 20. The first plot contains the non-stacked number of packets per second over time for the protocols TCP, ICMPv4, and UDP. In the second plot, the resulting number of concurrently active flows per second can be seen, which is flattened at the edges due to active flows per second resembling a moving average of the packets per second. Although, the number of flows is in the order of $10 \times 10^4$, the number of flows that must be kept in the flow table is much higher: As is evident by the third plot, which shows expired flows per second, most of the flows can only be expired after the idle timeout (300 s in this example), which is long (300 s!) after the flow stop time. This can be seen at the beginning of the plot, where only a fraction of the TCP flows are expired (zoomed representation shown in the lower left corner of Fig. 20). At the end of the file $\approx 8 \times 10^6$ flows are still in the flow table (zoomed part shown in the lower right corner of Fig. 20).

The following evaluations start with a description of the test setup in Section VII-A including how each flow exporter was used and which steps were necessary to get viable output. This is followed by a detailed flow record comparison including a flow by flow comparison and a cause analysis for expected and unexpected output deviations in Section VII-B. These findings are summarized in Section VII-C. After the full output comparison, a runtime analysis comparing memory and runtime requirements of the analyzed flow exporters is presented in Section VII-D. The evaluation is completed with a runtime comparison between *go-flows* and the custom exporter built for analyzing time-activity footprints by Iglesias et al. [24] in Section VII-E.

### A. Test Setup

For the flow comparison, each flow exporter was used for obtaining flows from the sample MAWI samplepoint-F 201702011400. Since there is no common flow format that

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



Table VIII
COMPARED VERSIONS OF FLOW EXPORTERS.

| nr | name | version number / commit hash |
|---|---|---|
| 1 | argus [28] | 3.0.8.2 |
| 2 | pmacct [40] | v1.7.3-rc2 / 52ad016 |
| 3 | Vermont [41] | 32e684a |
| 4 | yaf [42] | 2.10.0 |
| 5 | Joy [43] | a544976 |
| 6 | go-flows [106] | 50fe3ba |

is supported by the analyzed flow exporters, the output of the exporters was converted to a common format. Due to the high number of flows and none of the supported flow formats supporting fast flow property comparison or search capabilities, an indexed Sqlite database [31] was chosen for comparison.

Table VIII lists all analyzed exporters, the used version, and the order of appearance in the comparison. In case no version numbers are used by the project or not yet released code was necessary, commit hash IDs are given.

The following list describes steps necessary for acquiring the flows used in this evaluation:

*1) argus:* For a successful compilation, the line `#include <rpc/types.h>` had to be removed from `argus_util.c`. This is necessary with newer versions of glibc, which removed support for sunrpc. This modification didn't affect argus in any way.

Besides specifying the PCAP input file and the output file, no special command line arguments or configuration was used.

*Argus* uses its own proprietary binary format, therefore, requiring the usage of the *ra* tool for converting its output to CSV. Care must be taken with the output of the *ra* tool. It claims to support XML output, but only a subset of fields can be exported via XML (e.g., TCP sequence numbers are not supported). Beware: No error is generated if one selects fields for export, which are not supported in XML, instead those are just missing from the output.

*2) pmacct: pmacct* can export bidirectional flows using the *nfprobe* plugin. One bidirectional flow is exported as two unidirectional flows by *pmacct* – one for each direction. Both flows have the same start time, stop time, and protocol identifier, reversed source and destination addresses and ports, and flow properties representing the respective direction. Due to start and stop time being the same, these are not real unidirectional flows (e.g., the flow start time does not match the time of the first packet in the given direction if the first packet was in the opposite direction). For this analysis the flows were merged back into a single bidirectional flow.

With the default configuration values, *pmacct* uses an internal queue for passing packets between the plugins, which was built for live packet capture. This means that packets get lost on queue overflow. However, reading from the used capture file is faster than the *nfprobe* plugin can handle. Even adjusting the buffer size as suggested by the warning message upon *nfprobe* encountering lost sequence numbers still resulted in packets missing from flows.

Switching to the zeromq buffer mechanism remedied this issue. However, this resulted in a crash due to a failed assertion in version v1.7.2, leading to the usage of v1.7.3-rc2, which is the second release candidate for version v1.7.3, for this test, which includes a fix for the problem.

*Nfprobe* is capable of exporting flows using the NetFlow or IPFIX protocol, but only via UDP transport. Due to UDP being an unreliable transport and the sample trace containing a high number of packets, it was not possible to capture all resulting flows. With unreliable connections, such as UDP, this can happen even if both sender and receiver are on the same host. This can be caused by scheduling issues, buffers being too small, the receiving application being not fast enough, or a combination of these issues.

Since it was not possible to overcome those issues, even with a high performance receiver writing only to memory, the `send` function was overridden via `LD_PRELOAD` with one that writes into a file instead of to the network for this test. Testing for flow completeness was achieved by looking for gaps in IPFIX sequence numbers.

One additional setting that needed to be changed was the number of maximum concurrent flows, since the default limit is 8000.

Used configuration parameters were:

- plugins: nfprobe
- aggregate: src_host, dst_host, src_port, dst_port, proto, timestamp_start, timestamp_end
- plugin_pipe_zmq: true
- plugin_pipe_zmq_profile: xlarge
- nfprobe_version: 10
- nfprobe_maxflows: 26214400

Like *argus*, *pmacct* supports IP defragmentation, however individual fragments are not counted as individual packets. *pmacct* assigns protocol number 0 to IPv6 packets that contain only a fragment header and no payload, which maps to IPv6 hop-by-hop options. To keep the result comparable, fragmented packets were filtered out of the input file.

*3) Vermont:* Although the documentation lists bidirectional flow support, it has been broken since 2014. *Vermont* tries to find reverse information elements, which stopped working during the last IPFIX support rewrite [44]. Since *Vermont* supports neither IPv6 nor IP fragments, both were removed from the analysis.

The following modules and configuration were used:

**observer:** The sample file was given as filename and offlineAutoExit was set to 1. Next module: packetQueue.

**packetQueue:** Maximum size of 100 packets. Next module: packetAggregator.

**packetAggregator:** sourceIPv4Address, destinationIPv4Address, protocolIdentifier, sourceTransportPort, and destinationTransportPort as flowKey; flowStartNanoseconds, flowEndNanoseconds, octetTotalCount, packetTotalCount, and icmpTypeCodeIPv4 as nonFlowKey; 300 s inactiveTimeout, 1800 s activeTimeout, and 10 s pollInterval. Next module: ipfixQueue.

**ipfixQueue:** Maximum size of 1000 flows. Next module: ipfixFileWriter.

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



**ipfixFileWrite:** Maximum file size 10 000 000 000 000.

The high maximum file size was needed to get a manageable number of files. Although the documentation states that this number should be the maximum output file size in KiB with a default of 20 MiB, using no value here results in numerous files with one or even no flow per file. The setting given above resulted in files with a maximum file size of ≈50 MiB.

*4) yaf:* Although *yaf* supports reassembling IP fragments, this was turned off, to match the other flow exporter tests. Besides the fragmentation option and specifying input and output file, *yaf* was used with the default configuration.

*5) Joy: Joy* was configured with bidirectional flows enabled (bidir=1), limit packet array to zero (num_pkts=0), include packets with zero data length (zeros=1), and include retransmits (retrans=1).

*Joy* uses a fixed 10 s idle and 30 s active timeout, which can't be changed with configuration. Furthermore, packet-property-based expiry is not supported. Although *joy* includes a tool called *sleuth* for flow post processing, which supports stitching flows together, it does not support flow expiry, resulting in endless flows.

Since fragmented packets are not supported, those were filtered out of the used packet trace. Beware, that various older revisions don't support truncated packets, since those ignore packets where the IP total length exceeds the captured length.

*6) go-flows:* Unlike the other exporters, *go-flows* has no default values for settings that modify flow behavior. An idle timeout of 300 s and an active timeout of 1800 s, and the five-tuple as flow key was chosen to match most of the other exporters.

Exported flow properties were the five-tuple, packetTotalCount, octetTotalCount, flowStartNanoseconds, and flowEndNanoseconds, since those properties are common to all the analyzed flow extractors. Additionally, tcpSequenceNumber, reverseTcpSequenceNumber, and flowEndReason was exported, since some of the analyzed exporters support those.

Since the flow exporter *Vermont* supports only unidirectional flows, a set of unidirectional flows and a set of bidirectional flows was generated with *go-flows*.

For settings that don't modify flow behavior, the default options were used.

### B. Output Comparison

For the flow-by-flow output comparison, every flow exporter described in Section III (*go-flows*, *argus*, *pmacct*, *Vermont*, *yaf*, and *Joy*; see Table VIII) was used to generate flows from MAWI packet traces described in [107] published at [108]. As representative sample, the sample number 201702011400 from samplepoint-F was chosen. In the following, every flow exporter difference to the reference *go-flows*, is discussed in detail, including used version, configuration, and modifications, if required.

The tables listing the flow differences use the following terminology:

**exact match:** Every exported flow property of the compared flow records matched exactly. For flow exporters that don't support ICMP-based properties, the comparison is split into two parts: non-ICMP flows, where every flow property matches exactly, and non-ICMP flows, where every flow property except ICMP type and code match exactly.

*** mismatch:** These flows could be reliably matched to ones from the other exporter, but the given flow property differed (e.g., TCP sequence number).

**multi y:** Due to different flow expiry techniqes, one flow in flow exporter *x* results in multiple flows in flow exporter *y*. This means, that flow key, the sum of packets, and packet sizes of multiple flows from *y* matches one flow from *x*. Furthermore, start and stop times of the flows from *y* fall within the range start to stop time from the flow from *x*.

**multi x/multi y:** Same as above, but multiple flows in *x* match multiple flows from *y*.

*** reversed:** Source and destination properties are exchanged, i.e., the flow would be the same, but in the other direction.

**non-matching:** These flows could not be matched to ones from the other exporter.

Except for Vermont, which only supports unidirectional flows, all the following comparisons use bidirectional flows.

*1) argus:* Before comparing the output to *go-flows*, the following possible issues were noticed:

- The protocol field, which should contain the IP protocol number according to the documentation, can also contain numbers outside the IP protocol number range (e.g., for non-IP packets).
- Like other flow exporters, ICMP type and code are encoded in one of the transport ports (destination transport port in *argus*). However, care must be taken during parsing *ra* output, since port numbers are formatted as decimal numbers, but ICMP type and code information is in hexadecimal.
- Host byte order is used for encoding ICMP type and code information (e.g., on x86 hardware, the type and code are reversed compared to the network packet), leading to non portable text output.
- There is an issue with IPv6 reassembly, since IPv6 packets containing only a fragment header (protocol number 44) and no payloads are reported as containing ICMPv4 (protocol number 1).
- One field that can't be compared with the other flow exporters is the total bytes field. *Argus* uses the sum of the full packet sizes, while other flow exporters and the IANA definition at [18] use the size of the IP header plus the size of the IP payload. Since the full packet length might include padding (resulting in no packet smaller than 60 B) or an FCS, those two can not be compared. Another possibility would be to use the appbytes field in *argus*, which contains the size of the application layer. Nevertheless, this is not possible with anonymized traffic traces, since the application layer and, therefore, the size of the application layer is missing.
- In most flow exporters and in the IANA definition at [18], flow end time marks the time of the last packet, while in *argus* this field contains the time, the flow expired. This

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



Table IX
ARGUS BIDIRECTIONAL FLOW COMPARISON.

| | go/# flows | argus/# flows |
|---|---|---|
| exact match | 16 139 384 | 16 139 384 |
| exact match reversed | 49 953 | 49 953 |
| TCP sequence mismatch | 38 915 | 38 915 |
| TCP sequence mismatch reversed | 291 | 291 |
| end time mismatch | 318 161 | 318 161 |
| ICMP match[a] | 387 780 | 387 780 |
| ICMP multi go-flows[a] | 7 326 741 | 3 663 531 |
| multi go-flows | 92 633 | 19 593 |
| multi argus | 403 996 | 1 702 344 |
| multi go-flows/multi argus | 66 830 | 139 529 |
| ICMP code mismatch | 3151 | 18 029 |
| non-matching TCP[b] | 29 093 | 69 006 |
| non-matching UDP | 551 | 4815 |
| non-matching ICMPv4 | 8989 | 58 227 |
| non-matching ICMPv6 | 6484 | 16 200 |
| Total | 24 872 952 | 22 625 758 |

[a] Includes total of 354 678 ICMPv4 and 8 ICMPv6 flows with wrong type and code information.
[b] Includes 552 duplicate flows (i.e., flows that have the same five tuple and flow start time).

means, that if the flow was expired due to a timeout, this field contains the timestamp of the last packet plus the timeout.

- Unlike other flow exporters, *argus* contains a full TCP state machine, and also state machines for other protocols, e.g., ICMP and DNS. This results in flows that represent connections better, e.g., duplicate TCP-RST packets are merged into a single flow, which would result in multiple flows in other flow exporters. Another example are ICMP echo requests and replies, which are merged into one flow in *argus*. Due to this finer grained flow expiry, part of the flows will have a different flow direction compared to other flow extractors, e.g., ICMP echo replies, since those are in the same flow as the ICMP echo requests.

Although *argus* uses extensive state-machines for flow expiry, a high number of flows have an exact match, as can be seen in the full flow-by-flow comparison in Table IX. This resulted from the high number of flows in the MAWI dataset, where the packets containing control flags (e.g., TCP SYN) of a connection didn't experience packet loss or reordering. The remaining flows that do contain those real world packet issues represent a good stress test for these state-machines. Most of reversed, split flows, TCP sequence mismatches, and non-matching flows were caused by the differing state-machines between *argus* and *go-flows*. Additionally, ICMP echo request and matching ICMP echo replies (i.e., ping), are matched to one flow in *argus*, but multiple flows in *go-flows* due to ICMP type and code being part of the flow key. Those are accounted for in ICMP matches and ICMP flow splits.

However, the complete state-machines in argus are more complex and thus have issues with edge cases. One example are flows where two or more TCP RST packets with the same five-tuple are encountered that don't belong to any connection. Those packets will result in one flow where start time and end time is the timestamp of the first packet, thus ignoring the rest of the RST-packets. Another example are ICMP echo replies without matching ICMP echo requests. These result in flows with random numbers as ICMP type and code, which is probably caused by uninitialized memory.

Table X
PMACCT BIDIRECTIONAL FLOW COMPARISON.

| | | go/# flows | pmacct/# flows |
|---|---|---|---|
| non-ICMP | exact match | 4 462 739 | 4 462 739 |
| | exact match reversed | 122 434 | 122 434 |
| | multi go-flows | 196 284 | 63 665 |
| | multi go-flows #packet mismatch | 10 303 | 4088 |
| | non-matching TCP | 0 | 54 |
| ICMP | exact match | 12 553 746 | 12 553 746 |
| | exact match reversed | 5 | 5 |
| | multi go-flows | 7 527 188 | 3 759 675 |
| | multi go-flows #packet mismatch | 14 | 14 |
| | non-matching | 239 | 241 |
| | Total | 24 872 952 | 20 966 661 |

A total number of 354 678 ICMPv4 and 8 ICMPv6 flows with wrong type and code information was discovered in the output. Since this issue was worked around during flow comparison by matching on parts of the five-tuple, timing information, and packet counts, these flows are part of *ICMP match* and *ICMP multi go-flows* in Table IX.

Further non-matching flows are caused by better packet decoding in *argus*, e.g., *argus* can decode TCP packets, where the TCP header is truncated in the middle. These packets are discarded by other flow exporters.

*2) pmacct:* During test setup it was noticed that *pmacct* assigns protocol number 0 to IPv6 packets containing only a fragment header, but no payload, which maps to IPv6 hop-by-hop options. Since fragmented packets were removed for the final comparison, these packets do not show up in the results. Furthermore, the *nfprobe pmacct*-plugin does not support expiry of any kind for packet dumps due to time-based expiry being implemented with real time instead of packet-based-timestamps. Additionally, unlike other flow exporters, *nfprobe* does not support including ICMP type and code in the flow key or as a property in the output.

Due to the missing ICMP information in the flow output, the resulting flow comparison in Table X is split into ICMP and non-ICMP flows. ICMP flows were matched based on source and destination address, protocol type, timing information, and packet counts.

Despite the missing flow expiry and ICMP decoding, a high number of flows matched exactly. This is a result of the high number of distinct flows in the MAWI dataset, which is a result of base network traffic characteristics. For instance, TCP streams use random source ports, which will result in distinct flow keys for most TCP flows due to the source port being part of the flow key in these tests. However, there is still a

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



Table XI
VERMONT UNIDIRECTIONAL FLOW COMPARISON.

| | go/# flows | Vermont/# flows |
|---|---|---|
| exact match | 24 483 056 | 24 483 056 |
| multi Vermont | 356 228 | 1 204 212 |
| multi go-flows/multi Vermont | 9974 | 26 579 |
| non-matching TCP | 2 | 0 |
| non-matching ICMP | 0 | 300 |
| Total | 24 849 260 | 25 714 147 |

Table XII
YAF BIDIRECTIONAL FLOW COMPARISON.

| | | go/# flows | yaf/# flows |
|---|---|---|---|
| distinct | exact match | 24 824 602 | 24 824 602 |
| | TCP sequence mismatch | 388 | 388 |
| | flowEndReason mismatching | 2 | 2 |
| | multi go-flows | 12 | 6 |
| | non-matching TCP | 0 | 2 |
| | non-matching ICMPv6 | 2439 | 0 |
| dup | exact match | 45 498 | 45 498 |
| | TCP sequence mismatch | 11 | 11 |
| | Total | 24 872 952 | 24 870 509 |

Table XIII
JOY BIDIRECTIONAL FLOW COMPARISON.

| | | go/# flows | joy/# flows |
|---|---|---|---|
| non-ICMP | exact match | 4 344 891 | 4 344 891 |
| | end time mismatch | 50 582 | 50 582 |
| | multi joy | 227 714 | 823 067 |
| | multi go | 86 193 | 19 081 |
| | multi go/multi joy | 55 464 | 50 626 |
| | multi go/multi joy #packet mismatch | 15 915 | 31 930 |
| | #packet and time mismatch | 10 986 | 8464 |
| ICMP | exact match | 12 640 855 | 12 640 855 |
| | end time mismatch | 462 | 462 |
| | multi joy | 28 427 | 115 235 |
| | multi go | 7 394 059 | 3 696 856 |
| | multi go/multi joy | 17 223 | 42 601 |
| | multi go/multi joy #packet mismatch | 97 | 89 |
| | #packet and time mismatch | 67 | 72 |
| | non-matching | 17 | 1533 |
| | Total | 24 872 952 | 21 826 344 |

significant number of flows, for which this difference in flow expiration results in differences. Those are part of the reversed flows and merged flows.

Finally, like other exporters, *pmacct* employs its own packet decoding. This results in the flow mismatches and missing packets in Table X due to differences in decoding incomplete or wrong TCP and ICMP headers.

*3) Vermont:* Even though documentation and configuration suggest that *Vermont* can export bidirectional flows, bidirectional flow export does not work. As has been explained earlier, this is the result of the IPFIX support rewrite, which occurred 2014 [44]. Therefore, the full flow comparison in Table XI was done with unidirectional flows.

Most of the flows output by *Vermont* are an exact match to *go-flows*. *Vermont* features it's own packet decoding code, which results in the non-matching TCP and ICMP flows, caused by differences in rejecting partial headers.

The discovered split flows are a result of an error in the time-based flow expiry. Instead of the packet time, *Vermont* uses the real time to check if flows have expired. This means, that during this test, after every pollInterval, which was $10\,\mathrm{s}$, all flows were expired. Despite this implementation issue, a high number of flows matched, since most of the flows are short-lived in the used MAWI sample.

*4) yaf:* The full flow-by-flow comparison with *go-flows* can be seen in Table XII, which had to be split into distinct and duplicate flows. This is caused by *yaf* supporting only $\mathrm{ms}$ resolution, which results in some duplicate flows, i.e., multiple flows having the same five tuple and flow start time. Additionally, this low time resolution resulted in two flows with a timeout as expiry reason in *go-flows*, while having end of file in *yaf* due to not reaching the timeout with the lower resolution. The same reason caused the split flows.

Due to *go-flows* and *yaf* using an incomplete TCP state machine, both can't reliably detect the initial TCP sequence number. Both use the sequence number of the first packet, which is replaced by the sequence number from a later packet, if it has a lower sequence number. This can compensate for packets that are out of sequence. TCP sequence numbers are stored as unsigned $32\,\mathrm{bit}$ integer numbers, which wrap back to zero upon reaching the maximum value. This must be accounted for in the TCP sequence number comparison, which is not done in *yaf*, leading to the different results (see Section V-D).

*Yaf* and *go-flows* use different packet decoding implementations – *yaf* employing its own and *go-flows* using gopacket. This leads to differing capabilities in decoding incomplete headers, which are caused by the packet anonymization in the MAWI trace.

*5) Joy:* Before the flow comparison was started, the following issues were noticed:

- *Joy* supports neither ICMP type and code as part of the flow key nor as flow property, therefore, the analysis was split between ICMP flows and non-ICMP flows.
- The field bytes_count represents the sum of transport layer payload sizes, or IP payload size if the transport layer is unknown, which can't be compared with the other flow exporters.
- The protocol field in the output contains the whole range of possible numbers although the trace file contains only a very small subset.
- The output also contains flows with an empty packet count, which is caused by an overflow of the packet count. An unsigned $8\,\mathrm{bit}$ integer is used as packet count, allowing a maximum of $255$ packets per flow.

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



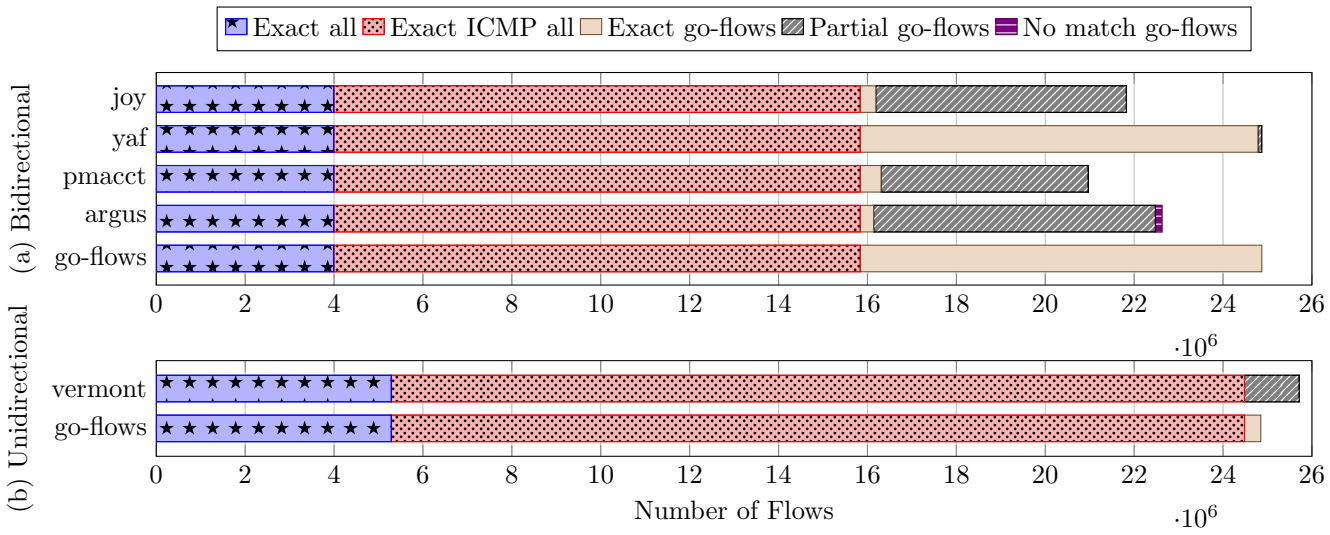


Figure 21. Overview of flow-by-flow comparison. Exact matches are flows that match exactly between all the flow exporters (exact all) or the given exporter and *go-flows* (exact go-flow). Exact ICMP are flows that match exactly except for ICMP type and code since those are not supported by every exporter. Partial matches are flows that match except for one or more properties. The varying numbers of total flows results from the differing flow expiry mechanisms, where multiple flows in one exporter are one flow in the other flow exporter.

- *Joy* uses a fixed 10 s idle and 30 s active timeout, which can't be changed with configuration. *Joy* includes a tool called *sleuth* for flow post processing, which supports stitching flows together. However, *sleuth* does not support any flow expiry, resulting in endless flows.
- Packet-property-based expiry is not supported.

As described above, *joy* doesn't support ICMP features, therefore, the flow-by-flow comparison, as can be seen in Table XIII, was split into non-ICMP and ICMP. Despite the differences in flow expiry described above, a high number of flows output by *go-flows* and *joy* matches exactly. However, unlike the already discussed flow exporters, the fixed flow expiry with low limits caused a significant number of split and multi flows.

The end time mismatches were caused by an error in handling bidirectional flows consisting of only two packets, with one packet going into one direction and the other packet in the opposite direction. In this case, the flow end time is the same as the flow start time, therefore ignoring the second packet.

As stated above, the packet counter in *joy* is capable of only representing up to a maximum of 255 packets per flow. The analyzed MAWI dataset contains a non negligible number of flows exceeding this limit, resulting in the flows marked #packet mismatch.

Just like with the other flow exporters, there are differences in packet decoding, resulting in differences accepting packets with truncated headers as valid. This resulted in the mismatched flows.

### C. Summary of Output Comparison

Although the flow extraction process described in Section V consists only of a few simple steps, using different flow exporters leads to different results. An overview of the flow comparison of all analyzed flow exporters can be seen in Fig. 21. Due to different expiry mechanisms and differing timeouts, the total number of flows varies between the different flow exporters.

The most differences are caused by varying degrees of supported features. The main difference leading to completely different flows is the support for bidirectional flows, which is not universally available. Although one could find similar flows, i.e., flows resulting from packets that were observed in only one direction, flow exporters supporting only unidirectional flows were separately analyzed in Fig. 21.

Further differences result from fragmentation support, packet decoding, differing time resolution, supported properties, different interpretations of properties, and implementation errors.

*1) Fragmentation:* Flow exporter differences can result from differing support of fragmentation. Not every flow extractor supports defragmenting IP packets. Additionally, flow extractors supporting fragments, count the number of packets differently. While some count each fragment as individual packet, others count just the reassembled packet as one. Therefore, fragmented packets were removed from the input dataset before the flow comparison.

*Argus* can reassemble IPv4 and IPv6 packets and counts every fragment as an individual packet. Like *argus*, *pmacct* and *yaf* support IP defragmentation, however individual fragments are not counted as individual packets. *Vermont* and *joy* supports neither IPv6 nor IP fragments.

*2) Packet Decoding:* Another difference in flow output results from decoding challenges of incomplete and malformed packet headers. For privacy reasons, the payload is missing for all packets in the used packet trace MAWI samplepoint-F 201702011400. Besides the missing payload, a high number of packets are truncated in the middle of a header due to privacy concerns (e.g., some ICMP packet types contain part of a related packet, several TCP options), and due to the maximum

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



Table XIV
TIMESTAMP DIFFERENCES FOR SAME FLOW.

| exporter | file format | timestamp |
|---|---|---|
| trace data | PCAP | 1 485 925 200.600 172 |
| Vermont | IPFIX | 1 485 925 200.600 172 203 267 |
| go-flows | IPFIX | 1 485 925 200.600 172 000 006 |
| go-flows | CSV | 1 485 925 200.600 172 000 |
| argus | argus | 1 485 925 200.600 172 |
| joy | JSON | 1 485 925 200.600 172 |
| yaf | IPFIX | 1 485 925 200.600 |
| pmacct | IPFIX | 1 485 925 200.600 |

capture size of 96 B. Which parts are removed, is described in the frequently asked questions at [108].

The different flow extractors can handle those packets to a varying degree. For example some implementations treat ICMP packets (v4 and v6) only as complete when those contain at least 8 B (e.g., *yaf*), while others consider 4 B as sufficient (e.g., *pmacct*).

*3) Time Resolution:* One base difference between the different flow exporters is the time resolution. The time resolution is a product of the following components:

- The time resolution provided by the source: PCAP supports µs and ns resolutions, while PCAPNG is capable of storing arbitrary resolutions. MAWI traces are stored with µs-resolution.
- The flow-exporter-internal timing representation: Most flow exporters use Unix timestamps for storing time, which uses the number of seconds since 1st of January 1970 for representing a point in time. For instance 2nd of February 2017 5:00:00 UTC is the Unix timestamp 1 485 925 200. Using floating-point numbers for timestamps will cause a loss in precision due to the high number of significant digits required, e.g., single-precision floating-point numbers are only capable of storing this Unix timestamp with a resolution of about ±128 s, while double-precision would support a resolution of ±119 ns.
- The flow-format time representation: IPFIX supports s (with second-resolution), ms (with ms resolution), µs (with ±232 ps resolution), and ns resolution (with ±232 ps resolution). The differences in resolution are due to s and ms being represented as number of seconds and milliseconds, while µs and ns use the Network Time Protocol (NTP)-timestamp-format, which uses fixed point base two numbers.
- The influence of conversion steps: Precision might be lost during time representation conversion between source, internal, and output representation.

An example for the single ICMP packet from 77.189.16.129 to 203.141.119.12 with type and code 0 in MAWI samplepoint-F 201702011400 at 5:00:00.600172 UTC can be seen in Table XIV. This packet was chosen since it results in a single flow in all flow extractors, resulting in a comparable timestamp. As mentioned above, the original timestamp in the PCAP trace has µs resolution. The same is true for *argus* and *joy*, which write µs resolution timestamps. Care has to be taken to not loose precision while reading *joy* timestamps, since those are formatted as JSON numbers, which are floating-point numbers.

*Go-flows* and *Vermont* support writing timestamps with ns resolution in the IPFIX-format. However, *Vermont* uses µs resolution internally. Furthermore, *Vermont* stores timestamps internally as the two 32 bit numbers seconds and micro seconds and uses the following factorization to convert the µs-part to the NTP fractional part[1]:

$$t_{\text{nsNTP}} = t_{\mu\text{s}} \cdot \frac{2^{32}}{10^6} \approx t_{\mu\text{s}} \cdot \left(4096 + 256 - \frac{1825}{32}\right)$$
$$\approx t_{\mu\text{s}} \ll 12 + t_{\mu\text{s}} \ll 8 - (t_{\mu\text{s}} \cdot 1825) \gg 5$$

With this factorization, the whole calculation can be carried out with 32 bit numbers, but the accuracy is limited to ±300 ns, which fits the result from Table XIV. *go-flows* on the other hand supports ns resolution internally, using one 64 bit number to store the Unix timestamp in ns since 1st of January 1970. For converting the fractional part to NTP *go-flows* uses[1]:

$$t_{\text{nsNTP}} = t_{\text{ns}} \cdot \frac{2^{32}}{10^9} \approx (t_{\text{ns}} \ll 32)\,/10^9 + 1$$

This conversion requires 64 bit integer arithmetic, but results in accurate ns timestamps. The +1 is needed to prevent rounding errors during conversion back from the NTP timestamp, e.g., without this, the example in Table XIV would be 1 485 925 200.600 171 999 773, which would be off by 1 ns after converting back from NTP due to the lack of rounding in conversion routines. Additionally, both *go-flows* and *Vermont* timestamps have trailing digits due to the usage of base 2 fractions in NTP timestamps.

*4) Supported Properties:* Further differences arise from the support of exported properties and how the flow key is handled. Not all flow exporters support exporting ICMP type and code properties, e.g., *joy*, *pmacct*. Additionally, those that do support this, employ different strategies for computing the flow key for ICMP flows. While *yaf* and *go-flows* treat ICMP type and code as if those were the source transport port, *argus* merges related ICMP packets into one stream (e.g., echo request and echo reply). Due to these differences, ICMP flows could only be compared according to source and destination network address, start and stop time, and number of packets in Fig. 21.

*5) Interpretation of Properties:* Different interpretations of flow property definitions lead to further flow mismatches between different flow exporters. One example is the flow size in bytes. *Yaf*, *go-flows*, *Vermont*, and *pmacct* use the sum of the IP header sizes and the IP payload sizes as flow size, while *argus* uses the sum of the complete packet sizes, and *joy* only the sum of the transport layer payload sizes. This leads to different results since transport layer payloads can be empty (e.g., TCP acknowledgment packets), while complete packet sizes can not be smaller than 60 B for Ethernet due to padding. Another example for different interpretations is the flow end time, where most exporters use the timestamp of

[1] ≪ and ≫ represent bit-wise left and right shift.

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



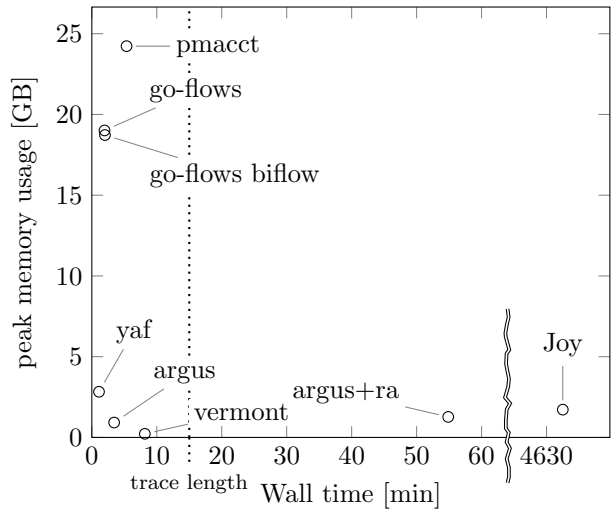


Figure 22. Performance overview of the analyzed flow exporters comparing execution time and used peak memory usage. The trace length marks the covered time span of the tested network trace.

the last packets, except *argus*, which also adds the timeout if applicable.

*6) Implementation Errors:* Wrong flow properties caused by errors in the implementation were also discovered.

*Argus* for example can provide incorrect numbers in ICMP type and code properties if echo replies without echo requests are observed, which is probably caused by uninitialized values in the state machine.

*Joy* will report incorrect packet counts if the packet count exceeds 255 for a single flow due to the packet count value being stored in a field that is too small. Flow start or end time can be wrong in *joy* due to packet times being handled incorrectly.

### D. Runtime Comparison

The analyzed flow exporters differ not only in their flow output, but also in terms of memory requirements and runtime performance. Memory requirements are not a primary concern, but excessive memory usage might prohibit the usage of a specific flow exporter due to available hardware. However, the packet traces from the MAWI projects contain exceptionally large numbers of concurrent flows, which directly impacts peak memory usage. Therefore, depending on the used packet trace, memory usage might not be an issue.

Like memory usage, runtime performance might also not be a primary concern. Nonetheless, if a single run takes too long, this might prohibit evaluating differing feature combinations, flow keys, or parameters due to time constraints. Furthermore, depending on the required scenario, executions times that take longer than real time, excludes possible usage in online scenarios.

All runtime tests were conducted multiple times on a dual Intel Xeon E5-2260 v2 (total of 20 physical/40 virtual cores) with 64 GiB system memory and a 512 GiB Samsung SSD 850. To limit the influence of disk access times in the analysis, the sample trace was stored on the SSD and flow output written back to the SSD.

An overview of the results can be seen in Fig. 22. This plot contains peak memory usage and the execution time needed for every flow exporter. The vertical line at 15 min marks the time span covered by the sample trace, showing that most exporters can process the file faster than real time.

*Argus* is included two times, since the *ra* tool is needed to convert the argus trace into a usable format. *Argus* alone can process the sample trace faster than real time, but the *ra* tool adds a significant performance overhead. The mean combined execution time required to acquire CSV formatted flow data was ≈55 min.

*Joy* is a serious outlier, needing three to four days for processing the sample trace. This is caused by the flow expiry code, which checks every flow for expiry on every read packet. All other flow exporters either use a sorted list of flows or check for flow expiry at fixed intervals instead of at every packet.

Although, as explained in Section VI, *go-flows* is fully flexible, including several choices of output flow format, fully customizable flow key, and fully customizable feature selection and aggregation, it is still faster than the analyzed exporters except for *yaf*. Execution speed might not be important in a research setting, especially if only packet traces are used as input. However, implementations that are too slow can prohibit exploring new features, flow keys, or parameters due to high turnaround times.

Peak memory usage is dominated by a combination of the peak number of flows and the number of extracted flow properties in most flow exporters. This leads to lower memory usage in *argus*, *vermont*, and *joy*, due to the differing flow expiry, leading to smaller peak flow table usage. However, in *pmacct* the high memory usage is caused by the used architecture. The input module reads and decodes packets as fast as it can and puts those into a buffer, while the flow aggregation is too slow, causing the buffer to grow considerably. In *go-flows*, the high memory usage is caused by the number of concurrent flows, the flexible implementation, and due to sacrificing memory usage for increased performance.

### E. Comparison to Custom-built

Only one of the related works listed in Section IV contains a runtime analysis of the custom developed flow exporter. Iglesias et al. [24] used a combination of *mergecap*, *tshark*, and custom scripts written in python and perl. The authors implemented their own flow exporter due to the usage of a custom flow key consisting of the timestamp in minutes, source transport address, and destination network address as well as custom flow properties.

Iglesias et al. [24] used traces from the MAWI samplepoint-F [108] and observed required run times of the flow exporting part of about 16 h for a single 15 min network trace. The main part of this long runtime was caused by *tshark* which does extensive packet analysis including trying to relate packets to connections and large scale protocol analysis (e.g., find duplicate TCP packets, reassemble TCP streams, match ICMP echo requests and replies).

As described in Section VI-D, the same exporter was implemented in *go-flows* by implementing missing custom

This article is the accepted version of "Why Are My Flows Different? A Tutorial on Flow Exporters" in the journal "Communications Surveys & Tutorials".
Citation information: DOI 10.1109/COMST.2020.2989695 Abstract: http://ieeexplore.ieee.org/document/9076310/



features and writing a flow specification. Exporting this type of flows would not have been possible without considerable effort with any other of the flow exporters described in Section III. The *go-flows* version takes about $1.5\,\mathrm{min}$ and requires about $\approx 8.4\,\mathrm{GiB}$ memory for a single $15\,\mathrm{min}$ MAWI samplepoint-F network trace [108] on a dual Intel Xeon E5-2260 v2 (total of 20 physical/40 virtual cores) with $64\,\mathrm{GiB}$ system memory and a $512\,\mathrm{GiB}$ Samsung SSD 850. This example also shows, that as stated in Section VII-D, memory consumption is highly dependent on the employed flow expiry, which resulted in smaller flow tables in this scenario, compared to Section VII-D, and, therefore, less memory consumption.

Using *go-flows* instead of the custom solution results therefore in a speed up of several orders of magnitude ($1.5\,\mathrm{min}$ compared to $16\,\mathrm{h}$).

## VIII. Recommended Guidelines

The following rules provide guidelines for improving reproducibility in flow-based research like network traffic analysis:

- Whenever possible, use an existing flow exporter and do not develop an own one.
- If existing flow exporters do not support the features you need, extend an existing flow exporter and make all changes explicitly clear and openly available to the scientific community.
- Whenever extending or modifying a flow exporter or flow exporter modules, make i) the abstractions, ii) goals, and iii) non-goals of the model explicit to support a scientifically sound reuse of the contributed flow exporter extensions.
- Document in detail which exporter (version, configuration, extensions, etc.) you are using. Additionally, make all assumptions, prerequisites, and configurations with respect to flow exporter functionality and features explicit.
- Be aware that flow exporter default configurations may change with time or flow exporter versions.
- Explicitly state all the implications that the use of this flow exporter has to your research.
- Verify the correctness of the flow exporter output and check if those values match the expected outcome.
- Take into account the differences in flow exporters and their potential effects on results when comparing results with others who used other flow exporters or might not have documented all settings.

## IX. Summary & Conclusion

This tutorial provides a definition for network flows based on the IPFIX standard in Section II-B, which is followed by possibilities to store or transmit flow records in Section II-D. In Section III existing flow exporters and in Section IV possible reasons why researchers implemented their own flow exporters have been explored, with the main causes being unsupported flow properties, key properties, or statistical methods. Additionally, most flow exporters were written for a specific purpose and are thus inflexible and can not be accommodated to the given problem.

The main part presents an in-depth tutorial covering all required steps to implement a flow exporter from packet to flow in Section V. The tutorial is complemented by a description of the flow exporter reference implementation *go-flows* covered in Section VI, which discusses reusability, versatility, performance considerations, and reproducibility issues.

A comprehensive evaluation of output and performance of the analyzed flow exporters and the introduced *go-flows* is presented in Section VII.

Although the flow definition presented in Section II-B suggests that any flow exporter supporting the required features can be used, this tutorial and, in particular, the evaluation in Section VII uncover that flow records from different flow exporters can diverge considerably. This is caused by one or more of the following uncertainty factors:

- Differing settings, e.g., used timeouts for flow expiry. Even though this can cause vastly different flows, not all the required settings are reported by the papers analyzed in Section IV.
- Not using a standardized definition for flow properties. One example for this is the packet size, which could mean size of the entire packet (e.g., *argus*), or just a specific layer (e.g., size of network layer in most exporters, payload size of transport layer in *joy*). Another example is the flow stop time, which is the expiry time instead of the time of the last packet in some flow exporters (e.g., *argus*).
- Different interpretation of under-specified standardized flow properties. For example the tcpSequenceNumber in RFC 5102 [15] does not specify if this should be the sequence number of the first packet, the first packet starting the connection, or an arbitrary one.
- Differing capabilities, which can be support for ICMP as part of the flow key, or state tracking of protocols.
- Implementation errors and limitations. These can even depend on the flow exporter version used.

Although just a single uncertainty factor can already lead to different results of the entire network analysis research the flows are used for, those factors are often neglected leading to poor comparability and reproducibility of results. This is especially true for custom flow exporter implementations, which are often not published, since those are not the main part of the scientific work.

Though the usage of an existing flow exporting solution can help to avoid common pitfalls, we uncovered huge differences in widely used flow exporters in Section VII. Therefore, the flow exporters should always be evaluated before considering their usage. Furthermore, with some small exceptions, existing flow exporters support only fixed features, flow keys, output formats, and some even only fixed parameters (e.g., hard coded active timeout, hard coded idle timeout), making those unsuitable for future research. As shown in Section IV this leads to custom implementations, which are insufficiently described causing non-reproducible work, and some even cause missed research opportunities due to very slow performance prohibiting parameter and feature exploration.

To improve this situation, we provide an open source reference implementation called *go-flows* at [106] as part

of this tutorial. This flow exporter was built to specifically support reproducible scientific research, and, therefore, uses the NTARC data format for specifying flows, which provides a way to unambiguously specify flow exporter settings used in scientific publications. Furthermore, *go-flows* is highly flexible, enabling customizing every aspect of the flow exporting process. Additionally, flow and packet properties, as well as operations like statistical functions, are not hard coded and can thus be user-configured and combined arbitrarily to fulfill every requirement, with only minimal development effort needed for non-standard flow properties.

Despite this flexibility, *go-flows* is still faster than all other flow exporters explored in Section VII, except for *yaf*. This enables exploring new features, flow keys, and parameters due to low turnaround times. The only possible downside of *go-flows* is the higher memory usage than most other exporters. However, this is only of concern for packet traces containing a huge number of concurrent flows and only if limited resources are available. For example, for the packet trace used in Section VII containing $\approx 8.7 \times 10^6$ concurrent flows, *go-flows* required $\approx 18.7\,\mathrm{GiB}$ peak memory. However, the same dataset analyzed with the settings, as taken from Iglesias et al. [24] required only $\approx 8.4\,\mathrm{GiB}$ peak memory. Therefore, the memory usage won't be an issue in most settings. Despite this limitation *go-flows* is the only choice, if the desired flow key, features, or aggregations, are not covered by another flow exporter. As was shown in Table II, this is mainly the case for flow keys differing from the standard 5-tuple.

## Acknowledgment

Part of this work was funded by the Austrian Research Promotion Agency (FFG) IKT der Zukunft research project synERGY (security for cyber-physical value networks exploiting smart grid systems, grant number 855457).

## List of Acronyms

**API** Application Programming Interface
**ARP** Address Resolution Protocol
**AS** Autonomous System
**BSD** Berkeley Software Distribution
**CSV** Comma-Separated Values
**CWND** Congestion Window
**DAG** Data Acquisition and Generation
**DB** Database
**DDoS** Distributed Denial of Service
**DIFFUSE** Distributed Firewall and Flow-shaper Using Statistical Evidence
**DNS** Domain Name System
**DOS** Denial of Service
**DPDK** Data Plane Development Kit
**DPI** Deep Packet Inspection
**ERF** Extensible Record Format
**ESP** Encapsulating Security Payload
**FCS** Frame Check Sequence
**FIN** Finish
**GRE** Generic Routing Encapsulation
**HTTP** Hypertext Transfer Protocol
**IANA** Internet Assigned Numbers Authority
**ICMP** Internet Control Message Protocol
**ID** Identifier
**IETF** Internet Engineering Task Force
**IP** Internet Protocol
**IPFIX** IP Flow Information eXport
**ISO** International Organization for Standardization
**JSON** Javascript Object Notation
**LAN** Local Area Network
**LPGL** Lesser General Public License
**MAWI** Measurement and Analysis on the WIDE Internet
**MPLS** Multiprotocol Label Switching
**NAT** Network Address Translation
**NetSA** Network Situational Awareness
**NSM** Network Security Monitor
**NTARC** Network Traffic Analysis Research Curation
**NTP** Network Time Protocol
**OS** Operating System
**OSI** Open Systems Interconnection
**PCAP** Packet Capture
**PCAPNG** PCAP Next Generation
**QoF** Quality of Flow
**QoS** Quality of Service
**RFC** Request For Comments
**RST** Reset
**RTP** Real-Time Transport Protocol
**RTT** Round Trip Time
**SCTP** Stream Control Transmission Protocol
**SQL** Structured Query Language
**SSD** Solid State Drive
**SSH** Secure Shell
**SYN** Synchronize
**TCP** Transport Control Protocol
**TLS** Transport Layer Security
**TLV** Type-Length-Value
**TSO** TCP Segmentation Offload
**UDP** User Datagram Protocol
**VLAN** Virtual LAN
**WEKA** Weikato Environment for Knowledge Analysis
**WIDE** Widely Integrated Distributed Environment
**WLAN** Wireless LAN
**XML** Extensible Markup Language

## References

[1] R. T. Lampert, C. Sommer, G. Münz, and F. Dressler, "Vermont-A Versatile Monitoring Toolkit for IPFIX and PSAMP," in *Proc. of IEEE/IST Workshop on Monitoring, Attack Detection and Mitigation (MonAM 2006)*, 2006.
[2] C. M. Inacio and B. Trammell, "Yaf: Yet another flowmeter," in *Proc. 24th Int. Conf. LISA*, 2010, pp. 1–16.
[3] N. Brownlee, "Flow-based measurement: IPFIX development and deployment," *IEICE Trans. Commun.*, vol. 94, no. 8, pp. 2190–2198, 2011.
[4] R. Hofstede, P. Čeleda, B. Trammell, I. Drago, R. Sadre, A. Sperotto, and A. Pras, "Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX," *IEEE Commun. Surv. Tutor.*, vol. 16, no. 4, pp. 2037–2064, Fourthquarter 2014.
[5] F. Haddadi and A. N. Zincir-Heywood, "Benchmarking the Effect of Flow Exporters and Protocol Filters on Botnet Traffic Classification," *IEEE Syst. J.*, vol. 10, no. 4, pp. 1390–1401, Dec. 2016.
[6] C. Mills, D. Hirsh, and G. Ruth, "INTERNET ACCOUNTING: BACKGROUND," RFC 1272, Sep. 1991.

[7] N. Brownlee, C. Mills, and G. Ruth, "Traffic Flow Measurement: Architecture," RFC 2063, Jan. 1997.
[8] J. Quittek, T. Zseby, B. Claise, and S. Zander, "Requirements for IP Flow Information Export (IPFIX)," RFC 3917, Oct. 2004.
[9] B. Claise, "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information," RFC 5101, Jan. 2008.
[10] ——, "Cisco Systems NetFlow Services Export Version 9," RFC 3954, Oct. 2004.
[11] J. Rajahalme, A. Conta, B. Carpenter, and S. Deering, "IPv6 Flow Label Specification," RFC 3697, Mar. 2004.
[12] B. Trammell and E. Boschi, "Bidirectional Flow Export Using IP Flow Information Export (IPFIX)," RFC 5103, Jan. 2008.
[13] B. Claise, B. Trammell, and P. Aitken, "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information," RFC 7011, Sep. 2013.
[14] Salvatore D'Antonio, Tanja Zseby, Christian Henke, and Lorenzo Peluso, "Flow Selection Techniques," RFC 7014, Sep. 2013.
[15] J. Quittek, S. Bryant, B. Claise, P. Aitken, and J. Meyer, "Information Model for IP Flow Information Export," RFC 5102, Jan. 2008.
[16] "TRANSMISSION CONTROL PROTOCOL," RFC 793, Sep. 1981.
[17] J. Postel, "User Datagram Protocol," RFC 768, Aug. 1980.
[18] "Internet Assigned Numbers Authority (IANA), IP Flow Information Export (IPFIX) Entities." [Online]. Available: https://www.iana.org/assignments/ipfix/ipfix.xhtml
[19] A. Sperotto, G. Schaffrath, R. Sadre, C. Morariu, A. Pras, and B. Stiller, "An Overview of IP Flow-Based Intrusion Detection," *IEEE Commun. Surv. Tutor.*, vol. 12, no. 3, pp. 343–356, Third 2010.
[20] W. John, S. Tafvelin, and T. Olovsson, "Passive internet measurement: Overview and guidelines based on experiences," *Computer Communications*, vol. 33, no. 5, pp. 533–550, Mar. 2010.
[21] L. T. Heberlein, G. V. Dias, K. N. Levitt, B. Mukherjee, J. Wood, and D. Wolber, "A network security monitor," in *Proceedings. 1990 IEEE Computer Society Symposium on Research in Security and Privacy*, May 1990, pp. 296–304.
[22] J. Frank, "Artificial intelligence and intrusion detection: Current and future directions," in *Proceedings of the 17th National Computer Security Conference*, vol. 10. Baltimore, MD, 1994, pp. 1–12.
[23] J. Zhang, X. Chen, Y. Xiang, W. Zhou, and J. Wu, "Robust Network Traffic Classification," *IEEEACM Trans Netw*, vol. 23, no. 4, pp. 1257–1270, Aug. 2015.
[24] F. Iglesias and T. Zseby, "Time-activity footprints in IP traffic," *Computer Networks*, vol. 107, pp. 64–75, Oct. 2016.
[25] T. T. T. Nguyen, G. Armitage, P. Branch, and S. Zander, "Timely and Continuous Machine-Learning-Based Classification for Interactive IP Traffic," *IEEEACM Trans. Netw.*, vol. 20, no. 6, pp. 1880–1894, Dec. 2012.
[26] (2007) NetFlow Services Solutions Guide. Cisco Systems, Inc., San Jose, CA.
[27] B. Trammel, E. Boschi, L. Mark, T. Zseby, and A. Wagner, "Specification of the IP Flow Information Export (IPFIX) File Format," RFC 5655, Oct. 2009.
[28] C. Bullard. Audit Record Generation and Utilization System (ARGUS). [Online]. Available: https://www.qosient.com/argus/index.shtml
[29] P. Phaal and M. Lavine, "sFlow Version 5," Jul. 2004. [Online]. Available: https://sflow.org/sflow_version_5.txt
[30] Y. Shafranovich, "Common Format and MIME Type for Comma-Separated Values (CSV) Files," RFC 4180, Oct. 2005.
[31] SQLite. [Online]. Available: https://www.sqlite.org/index.html
[32] nProbe. [Online]. Available: https://www.ntop.org/products/netflow/nprobe/
[33] G. Connell. Clerk. [Online]. Available: https://github.com/google/clerk
[34] S. Ostermann. Tcptrace. [Online]. Available: http://www.tcptrace.org/
[35] C. Kreibich, "Design and implementation of netdude, a framework for packet trace manipulation," in *Proc. USENIX/FREENIX*, 2004.
[36] Netdude. [Online]. Available: http://netdude.sourceforge.net/
[37] S. Zander and C. Schmoll. Network Measurement and Accounting Meter (NETMATE). [Online]. Available: https://sourceforge.net/projects/netmate-meter/
[38] C. Bullard, "ARGUS-2.0.1 IPFX BOF Presentation," Proceedings of the Fifty-First Internet Engineering Task Force, London, England, Aug. 2001. [Online]. Available: https://www.ietf.org/proceedings/51/slides/ipfx-2/sld001.htm
[39] P. Lucente, "Pmacct: Steps forward interface counters," 2008.
[40] ——. Pmacct. [Online]. Available: http://www.pmacct.net/
[41] Vermont - VERsatile MONitoring Tool. [Online]. Available: https://github.com/tumi8/vermont
[42] B. Trammel, C. M. Inacio, and M. Duggan. Yet Another Flowmeter (yaf). [Online]. Available: https://tools.netsa.cert.org/yaf/
[43] D. McGrew, B. Anderson, P. Perricone, and B. Hudson. Joy - A package for capturing and analyzing network flow data and intraflow data, for network research, forensics, and security monitoring. [Online]. Available: https://github.com/cisco/joy
[44] Vermont - VERsatile MONitoring Tool — Issue #53 — exceptions with default example file. [Online]. Available: https://github.com/tumi8/vermont/issues/53
[45] D. H. Hagos, P. E. Engelstad, A. Yazidi, and O. Kure, "Towards a Robust and Scalable TCP Flavors Prediction Model from Passive Traffic," in *2018 27th International Conference on Computer Communication and Networks (ICCCN)*, Jul. 2018, pp. 1–11.
[46] G. Nychis, V. Sekar, D. G. Andersen, H. Kim, and H. Zhang, "An Empirical Evaluation of Entropy-based Traffic Anomaly Detection," in *Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '08. New York, NY, USA: ACM, 2008, pp. 151–156.
[47] C. G. Cordero, S. Hauke, M. Mühlhäuser, and M. Fischer, "Analyzing flow-based anomaly intrusion detection using Replicator Neural Networks," in *2016 14th Annual Conference on Privacy, Security and Trust (PST)*, Dec. 2016, pp. 317–324.
[48] B. Trammel. Quality of Flow (QoF). [Online]. Available: https://github.com/britram/qof
[49] B. Anderson and D. McGrew, "Machine Learning for Encrypted Malware Traffic Classification: Accounting for Noisy Labels and Non-Stationarity," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '17. New York, NY, USA: ACM, 2017, pp. 1723–1732.
[50] ——, "Identifying Encrypted Malware Traffic with Contextual Flow Data," in *Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security*, ser. AISec '16. New York, NY, USA: ACM, 2016, pp. 35–46.
[51] A. Moore, D. Zuev, and M. Crogan, "Discriminators for use in flow-based classification," Queen Mary University of London, Department of Computer Science, Tech. Rep. RR-05-13, Aug. 2005.
[52] N. Williams, S. Zander, and G. Armitage, "A Preliminary Performance Comparison of Five Machine Learning Algorithms for Practical IP Traffic Flow Classification," *SIGCOMM Comput Commun Rev*, vol. 36, no. 5, pp. 5–16, Oct. 2006.
[53] T. Auld, A. W. Moore, and S. F. Gull, "Bayesian Neural Networks for Internet Traffic Classification," *IEEE Trans. Neural Netw.*, vol. 18, no. 1, pp. 223–239, Jan. 2007.
[54] R. Alshammari and A. N. Zincir-Heywood, "Machine learning based encrypted traffic classification: Identifying SSH and Skype," in *2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications*, Jul. 2009, pp. 1–8.
[55] J. Zhang, C. Chen, Y. Xiang, W. Zhou, and A. V. Vasilakos, "An Effective Network Traffic Classification Method with Unknown Flow Detection," *IEEE Trans. Netw. Serv. Manag.*, vol. 10, no. 2, pp. 133–147, Jun. 2013.
[56] X. Qin, T. Xu, and C. Wang, "DDoS Attack Detection Using Flow Entropy and Clustering Technique," in *2015 11th International Conference on Computational Intelligence and Security (CIS)*, Dec. 2015, pp. 412–415.
[57] A. Vlăduţu, D. Comăneci, and C. Dobre, "Internet traffic classification based on flows' statistical properties with machine learning," *Int. J. Netw. Manag.*, vol. 27, no. 3, p. e1929, May 2017.
[58] D. C. Ferreira, M. Bachl, G. Vormayr, F. I. Vázquez, and T. Zseby, "Curated Research on Network Traffic Analysis," [Data set]. Zenodo., Nov. 2018. [Online]. Available: http://doi.org/10.5281/zenodo.1483964
[59] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA Data Mining Software: An Update," *SIGKDD Explor Newsl*, vol. 11, no. 1, pp. 10–18, Nov. 2009.
[60] Tshark - The Wireshark Network Analyzer. [Online]. Available: https://www.wireshark.org/
[61] Jpcap - a network packet capture library. [Online]. Available: http://jpcap.sourceforge.net/
[62] TCPDUMP/LIBPCAP public repository. [Online]. Available: https://www.tcpdump.org/
[63] M. Tuexen, F. Risso, J. Bongertz, G. Combs, and G. Harris. PCAP next generation file format specification. [Online]. Available: https://github.com/pcapng/pcapng
[64] "ERF Types Reference Guide," Endace, Tech. Rep. EDM11-01 - Version 21. [Online]. Available: https://www.endace.com/erf-extensible-record-format-types.pdf

[65] A. Kleen, "Packet - packet interface on device level," packet(7) manual page, Linux 4.16, Sep. 2017. [Online]. Available: http://man7.org/linux/man-pages/man7/packet.7.html
[66] "Bpf - Berkeley Packet Filter," bpf(4) manual page, FreeBSD 12.0, Oct. 2016. [Online]. Available: https://www.freebsd.org/cgi/man.cgi?query=bpf&sektion=4&manpath=FreeBSD+12.0-RELEASE+and+Ports
[67] Raw Sockets - Windows applications — Microsoft Docs. [Online]. Available: https://docs.microsoft.com/en-us/windows/desktop/winsock/service-provided-raw-sockets-2
[68] PF_RING - ntop. [Online]. Available: https://www.ntop.org/products/packet-capture/pf_ring/
[69] DPDK - Data Plane Development Kit. [Online]. Available: https://www.dpdk.org/
[70] V. Moreno, J. Ramos, P. M. S. del Río, J. L. García-Dorado, F. J. Gomez-Arribas, and J. Aracil, "Commodity Packet Capture Engines: Tutorial, Cookbook and Applicability," *IEEE Commun. Surv. Tutor.*, vol. 17, no. 3, pp. 1364–1390, thirdquarter 2015.
[71] L. M. Garcia, "Programming with libpcap-sniffing the network from our own application," *Hakin9-Computer Security Magazine*, vol. 3, no. 2, 2008.
[72] WAND Network Research Group: Libtrace. [Online]. Available: https://research.wand.net.nz/software/libtrace.php
[73] Libtins - packet crafting and sniffing library. [Online]. Available: http://libtins.github.io/
[74] PcapPlusPlus. [Online]. Available: http://seladb.github.io/PcapPlusPlus-Doc/
[75] Pcap4J - A Java library for capturing, crafting, and sending packets. [Online]. Available: https://www.pcap4j.org/
[76] Scapy Project. [Online]. Available: https://scapy.net/
[77] Dpkt. [Online]. Available: https://dpkt.readthedocs.io/en/latest/
[78] G. Connell. GoPacket. [Online]. Available: https://github.com/google/gopacket
[79] Wireshark - Go Deep. [Online]. Available: https://www.wireshark.org/
[80] S. Alcock, P. Lorier, and R. Nelson, "Libtrace: A Packet Capture and Analysis Library," *SIGCOMM Comput Commun Rev*, vol. 42, no. 2, pp. 42–48, Mar. 2012.
[81] Capture File Format Reference. [Online]. Available: https://wiki.wireshark.org/FileFormatReference
[82] WinPcap. [Online]. Available: https://www.winpcap.org/
[83] Npcap: WIndows Paket Capture Library & Driver. [Online]. Available: https://nmap.org/npcap/
[84] "INTERNET PROTOCOL," RFC 791, Sep. 1981.
[85] A. Conta, S. Deering, and M. Gupta, "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification," RFC 4443, Mar. 2006.
[86] J. Postel, "INTERNET CONTROL MESSAGE PROTOCOL," RFC 792, Sep. 1981.
[87] T. Olovsson and W. John, "Detection of malicious traffic on back-bone links via packet header analysis," *Campus-Wide Info Systems*, vol. 25, no. 5, pp. 342–358, Nov. 2008.
[88] C. Perkins, "IP Encapsulation within IP," RFC 2003, Oct. 1996.
[89] Yakov Rekhter, Robert G Moskowitz, Daniel Karrenberg, Geert Jan de Groot, and Eliot Lear, "Address Allocation for Private Internets," RFC 1918, Feb. 1996.
[90] S. S. Skiena, *The Algorithm Design Manual*, 2nd ed. London: Springer-Verlag, 2008.
[91] The Python Standard Library - Python 3.7.3 documentation. [Online]. Available: https://docs.python.org/3/library/
[92] Effective Go - The Go Programming language. Build version go1.12.5. [Online]. Available: https://golang.org/doc/effective_go.html
[93] Map (Java Platform SE 8). [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/Map.html
[94] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," RFC 8259, Dec. 2017.
[95] "The JSON Data Interchange Syntax," ECMA-404, Dec. 2017.
[96] B. P. Welford, "Note on a Method for Calculating Corrected Sums of Squares and Products," *Technometrics*, vol. 4, no. 3, pp. 419–420, Aug. 1962.
[97] P. J. Rousseeuw and G. W. Bassett Jr, "The remedian: A robust averaging method for large data sets," *J. Am. Stat. Assoc.*, vol. 85, no. 409, pp. 97–104, 1990.
[98] R. W. Floyd and R. L. Rivest, "Algorithm 489: The Algorithm SELECT—for Finding the Ith Smallest of N Elements [M1]," *Commun ACM*, vol. 18, no. 3, pp. 173–, Mar. 1975.
[99] C. Newman and G. Klyne, "Date and Time on the Internet: Timestamps," RFC 3339, Jul. 2002.
[100] Function Index (The GNU C Library). [Online]. Available: https://www.gnu.org/software/libc/manual/html_node/Function-Index.html
[101] Packages - The Go Programming language. Build version go1.12.5. [Online]. Available: https://golang.org/pkg/
[102] "JSON." [Online]. Available: https://www.json.org
[103] Libfixbuf. [Online]. Available: https://tools.netsa.cert.org/fixbuf/index.html
[104] D. C. Ferreira, M. Bachl, G. Vormayr, F. I. Vázquez, and T. Zseby, "NTARC specification," NTARC specification (Version v3.0.0). Zenodo., Nov. 2018. [Online]. Available: http://doi.org/10.5281/zenodo.1484190
[105] D. C. Ferreira, F. I. Vázquez, G. Vormayr, M. Bachl, and T. Zseby, "A Meta-Analysis Approach for Feature Selection in Network Traffic Research," in *Proceedings of the Reproducibility Workshop*, ser. Reproducibility '17. New York, NY, USA: ACM, 2017, pp. 17–20.
[106] G. Vormayr. Go-flows. [Online]. Available: https://github.com/CN-TU/go-flows
[107] Kenjiro Cho, Koushirou Mitsuya, and Akira Kato, "Traffic data repository at the WIDE project," in *Proceedings of USENIX 2000 Annual Technical Conference: FREENIX Track*, San Diego, CA, Jun. 2000, pp. 263–270.
[108] MAWI packet traces. [Online]. Available: http://mawi.wide.ad.jp/mawi/

**Gernot Vormayr** received his B.Sc. degree in electrical engineering and his Dipl.-Ing. degree (M.Sc. equivalent) in computer technology from the TU Wien, Vienna, Austria. Since 2016, he has been employed as a University Assistant at the Institute of Telecommunications, TU Wien, where he is pursuing his doctoral degree. His current research interest is in malware communication with a focus on botnets.

**Joachim Fabini** received his Dipl.-Ing. degree in computer sciences and his Dr.Techn. degree in electrical engineering from the TU Wien. After five years of research and development with Ericsson Austria, he joined the Institute of Telecommunications, TU Wien, in 2003. He is a Senior Scientist with the Communication Networks Group with research focus on active measurement methodologies.

**Tanja Zseby** received her diploma degree (Dipl.-Ing.) in electrical engineering and her doctoral degree (Dr.-Ing.) from TU Berlin, Germany. She worked as Scientist and Head of the Network Research Group at the Fraunhofer Institute for Open Communication Systems, Berlin, Germany and as a visiting scientist at the University of California, San Diego (UCSD). She now is a full professor of communication networks at TU Wien.