Design of DIFFUSE v0.1 – DIstributed Firewall and Flow-shaper Using Statistical Evidence

Sebastian Zander, Grenville Armitage
Centre for Advanced Internet Architectures, Technical Report 101223A
Swinburne University of Technology
Melbourne, Australia
szander@swin.edu.au, garmitage@swin.edu.au

***Abstract*—In recent years a growing number of researchers investigated the performance of machine learning based traffic classification using statistical properties – classification techniques that do not require packet payload inspection. Such techniques assist Internet Service Providers to work within any legal or technical limitations on direct payload inspection. Potential new applications include automated 'market research', automated traffic prioritisation, and Lawful Interception. For many of these new applications a de-coupling between the flow classification and subsequent flow treatment, such as blocking or shaping, is highly desirable. In the DIFFUSE project we are developing extensions for an existing packet filter that provide ML-based traffic classification based on statistical properties and de-couple flow classification from flow treatment. This report describes the selection of the existing packet filter extended, the design of the overall architecture and key components, as well as the machine learning techniques supported.**

***Index Terms*—Statistical Flow Classification, Machine Learning, Quality of Service, Traffic Prioritisation**

I. Introduction

During recent years a body of research emerged around the identification and classification of traffic flows based on statistical properties (features) and in particular the application of Machine Learning (ML) techniques to generate such classifiers [1]. Statistical properties, such as distributions of packet sizes or inter-packet arrival times, can be calculated without accessing packet payloads (payload inspection). Such techniques assist Internet Service Providers (ISPs) to work within any legal or technical limitations on direct payload inspection. Potential new applications include automated 'market research', characterising traffic for Lawful Interception [2], or automated prioritisation of real-time traffic [3].

For many of these new applications a de-coupling between flow classification and subsequent flow treatment (the actions performed on flows), such as blocking or shaping, is highly desirable. For example, a single high performance classifier near the core of an ISP network may control multiple low-power nodes near the network edge (perhaps embedded within Asynchronous Digital Subscriber Line or Cable modem gateways) so that centralised traffic classification can automatically modify the Quality of Service (QoS) treatment experienced by packets at the network edge. This de-coupling also enables potentially computationally intensive per-flow statistics calculations to be offloaded from the packet forwarding path.

Common open-source packet filters that combine firewall and traffic shaping (such as IPFW [4], PF [5], Netfilter [6] and others) currently do not use traffic statistics, instead relying on direct inspection of packets passing through the filtering node's local interfaces. Furthermore, these filters couple the flow classification and treatment tightly, i.e. the actions are executed locally immediately after the flow classification.

In the DIFFUSE project [7] we are designing and developing extensions for an existing packet filter that provide ML-based traffic classification based on statistical properties and de-couple flow classification and treatment. In our architecture there are *classifier nodes* that classify traffic flows and then instruct *action nodes* via a *control protocol* to carry out actions for the classified flows. In this report we describe the design of the system and its key components. To avoid 'reinventing the wheel' our system will be based on an existing packet filter. As main development platform we selected the FreeBSD operating system since it is often used for building firewalls and/or traffic shapers, and a number of existing packet filters run on FreeBSD.

The report is organised as follows. First we define fundamental terms and concepts in Section II. In Section III we compare existing FreeBSD packet filters and choose the most suitable as basis for our new system.

In Section IV we describe the overall architecture of the system and its key components, and show example scenarios illustrating how the system could be used.

In the remaining sections we describe the design of the system components in more detail. In Section VI we describe the extended command language that enables the configuration of classifier and action nodes. In Section VII we describe the design of the control protocol used to exchange data between classifier and action nodes.

In Section VIII we describe the initial choice of ML techniques supported. Since our system is designed to be flexible other ML techniques can be added in the future. In Section V we outline the software design of DIFFUSE v0.1. Section IX concludes the report.

II. Definitions

First we define some fundamental terms and concepts used throughout the report.

A. Flows

A *flow* is a number of consecutive packets that have the same values for a defined set of packet header fields within a certain time frame. The set of packet header fields is usually the commonly used 5-tuple (source and destination IP addresses, source and destination ports, protocol), but it could be a different set of fields (such as only the source and destination IP addresses).

For connection-oriented protocols (like TCP) the flow start and end is usually marked by the establishment and teardown of a connection. For non connection-oriented protocols (like UDP) the first packet seen marks the start and no packets arriving for a certain duration (flow timeout) marks the end of the flow.

A *unidirectional* flow is a flow where packets flow only in one direction (e.g. only packets matching a 5-tuple), whereas a *bidirectional* flow is packets flowing in both directions (e.g. all packet matching a 5-tuple and the same 5-tuple with source/destination addresses and ports reversed).

A *subflow* is part of a flow. For our purposes a subflow is a sliding window of n consecutive packets within a unidirectional flow (as in [8]).

Bidirectional flows have two directions which we refer to as *forward* and *backward*. For connection-oriented protocols (and if the initial handshake can be observed) packets from the originator of the connection are going in the forward direction, and packets from the other end are going in the backward direction. For connection-less protocols or when the handshake could not be observed the first packet defines the forward direction.

If a rule defines a source or a destination (by specifying a match pattern, e.g. matching against the source IP address), packets matching the pattern are considered to flow forward, whereas packets that are going in the reverse direction are considered to flow backward.

B. Features

Previous work usually used the two level hierarchy of *features* and *feature sets*, where a feature is a characteristic of a flow or subflow (such as the mean packet length) and a feature set is a number of different features. For the DIFFUSE v0.1 architecture we extended this hierarchy. *Feature statistics* are statistics of a series of feature values (such as the minimum, mean or maximum), features are characteristics of flows, subflows or packets (such as packet length or inter-arrival times) and feature sets are a number of features (as before).

The main reason for this three-level hierarchy is that DIFFUSE v0.1 supports different independent feature modules, but for performance reasons different statistics of the same feature are part of the same module.

III. Choice of Firewall

Here we discuss our choice of the existing packet filter we extended. Since DIFFUSE v0.1 is based on FreeBSD we have a choice between three packet filters: IP Firewall (IPFW) [4], IPFilter (IPF) [9], [10] and Packet Filter (PF) [5]. We compare these based on a number of criteria.

A. Functionality

All three packet filters support the basic functions of filtering based on network and transport layer information, network address translation, logging etc. IPFW and PF can tag packets for implementing policy-based rules and have interfaces to traffic shapers that can queue and prioritise packets. IPFW mostly uses Dummynet [11] but also has an interface to ALTQ [12], whereas PF uses ALTQ. IPFW/Dummynet can also be used to emulate certain network link conditions by limiting link capacity, emulating delay and packet loss etc.

PF has more advanced functionality than IPFW and IPF, most notably the ability to implement redundant firewalls (state transfer and failover protocol [5]), load-balancing, logging to tcpdump files, and filtering on operating system fingerprints. However, IPFW now also supports tables and in-kernel NAT and has reduced the functionality gap to PF.

DIFFUSE needs packet queuing and prioritising support, which rules out IPF. While the advanced functions of PF are nice they are not really required for DIFFUSE.

B. Portability

IPFW has been developed and used in FreeBSD over many years, is the network firewall in MacOSX, and has recently been ported to Linux and Windows [13]. IPF runs on the BSD family (FreeBSD, OpenBSD, NetBSD) as well as on Solaris, HP-UX, IRIX and Linux. PF is the main firewall of OpenBSD, and it has been ported to FreeBSD and NetBSD.

As far as portability is concerned IPFW and IPF are the best options for DIFFUSE. PF falls behind as the number of operating systems it runs on is very limited.

C. Support

IPFW is the FreeBSD sponsored firewall; it is authored and maintained by FreeBSD volunteer staff members. IPF was the main packet filter of OpenBSD, before it was replaced by PF in 2001. Given that we have chosen FreeBSD as development platform in terms of future support IPFW is most promising. The PF firewall comes second being OpenBSD's official firewall and given that it is part of the FreeBSD source tree. IPF is also part of the FreeBSD source tree.

However, the PF sources part of FreeBSD are always lagging behind the latest OpenBSD version. For example, the PF version in FreeBSD-9.0-current checked out in July 2010 are the PF sources from March 2007. Similarly, the IPF sources in FreeBSD lack one major version behind the latest release. For example, the IPF version in FreeBSD-9.0-current checked out in July 2010 are the IPF sources from October 2007 (version 4.1) and much older than the latest release from May 2010 (version 5.1). For IPF this is less of a problem because the IPF sources are released independently of FreeBSD and should compile on all supported operating systems.

All three packet filters are actively maintained and used. For DIFFUSE IPFW and PF have a slight edge as they are the main packet filters of FreeBSD/OpenBSD.

D. Performance

Measuring the performance of packet filters is not straightforward, as the performance depends heavily on the scenario, i.e. the actual firewall rules and network traffic. Nevertheless, one can compare different packet filters in particular scenarios. Previous work compared IPF, PF and Linux Netfilter [14], [15]. According to these studies PF performs similar to IPF and whether they performs better or worse than Netfilter depends on the scenario. We were unable to find a study of IPFW or a recent performance comparisons of all three firewalls.

The performance of the packet filter should be reasonably good, but this criteria is not of very high importance for DIFFUSE. Since IPFW, IPF, and PF are are all deployed we believe they all provide sufficient performance in practice.

E. Usability

The rule set language of PF is better designed than the languages of IPF and IPFW, which both look somewhat organically grown. The structure of PF and IPF rulesets differs from IPFW (and Linux Netfilter). By default for IPF and PF the last rule that matches determines the action, whereas for IPFW (and Netfilter) the first rule that matches determines the action. This makes it more difficult to convert IPF and PF rulesets to IPFW rulesets and vice versa.

While the rule language of PF is better designed, the languages of IPF and IPFW are still logical, easy to read and use. As far as usability is concerned all three firewalls are adequate for DIFFUSE.

F. Extensibility

IPFW, IPF and PF have nicely written user documentation, but for all three there is not much developer documentation. None of them has a fully modular framework that can be extended easily. However, IPFW's Dummynet now has a modular framework for adding schedulers. All three packet filters have nicely written code, but IPFW stands out because it also has a lot of useful comments inside the code, whereas comments in IPF and PF code are rather sparse. Furthermore, here at CAIA we have some in-house expertise for extending IPFW. Hence, IPFW wins this category.

G. Decision

IPF does not have functions for packet queuing and prioritisation and hence cannot be used. We chose IPFW over PF because IPFW supports the three arguably most popular operating systems (FreeBSD, Linux, Windows) and it appeared to be easier to extend than the others due to well documented code, relatively modular structure and existing in-house expertise.

IV. System Description

A. Architecture

In order to provide ML-based traffic classification and de-couple the classification from the subsequent action our system has several key components:

- A *Classifier Node* (CN) computes statistical features from flows identified by their 5-tuple and classifies them based on local machine-learning rules.
- An *Action Node* (AN) performs configured actions (block, redirect, rate shape, etc.) on packets belonging to flows that have been classified by a local or remote CN.
- An IP-layer *Control Protocol* (CP) between CNs and ANs to enable real-time coordination, such as alerting ANs when to start and stop acting on identified flows.
- An extended set of *Packet Filter Rules* (PFRs) to express ML-based traffic matching based on statistical attributes at CNs and specify the actions to be taken by nominated ANs.

A CN records flow identification information (5-tuple) and computes flow characteristics, such as packet length and inter-arrival time statistics. A CN continuously compares the statistics of observed flows to a configured set of rules and uses this information to generate traditional header-only inspection rules for ANs. When a flow (flow X) matches a statistical rule, the CN passes the flow's 5-tuple and class to AN(s) to actually instantiate the flow class' associated action. The action is then applied to all subsequent packets belonging to flow X. The rule is removed from the AN(s) once flow X has stopped.

CNs and ANs automatically establish IP based control links via a CP to share information as matching flows come and go. CNs and ANs are different logical entities, but they can be co-located on the same physical network device. For example, a traditional packet filter combines them in a single device. If CN and AN are instantiated on the same host, equivalent to a traditional packet filter, this control link is inside the host.

Small networks with a few CNs and ANs can be configured manually by creating and distributing rulesets. In large networks comprising many CNs and ANs it is desirable to have a management system that automatically translates network policies into rulesets and distributes these rulesets to CNs and ANs. Such a management system is out of scope of this project, but it can be designed and build on top of our developed system in future work.

B. Classifier Node

A CN consists of an extended packet Filter/Classifier in kernel space and an userspace daemon process (called *Exporter*) that exports the flow specification (5-tuple), class and (optionally) an action to the AN(s) via the CP (see Figure 1). We call this information *flow rules*.

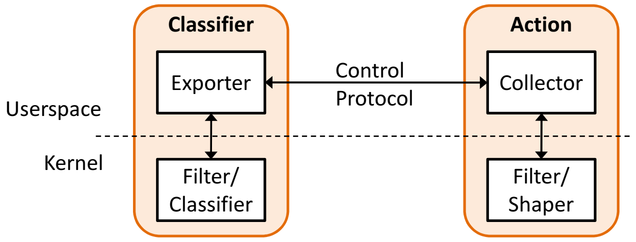

Figure 1. Classifier node and action node components, and control protocol

The extended packet filter computes statistical features for packet flows. These features can be used directly as patterns for matching packets or as input for an ML-based classifier that assigns classes to packets based on these features.

The feature computation and classification is done on a per-packet basis inside the kernel to maximise performance. The sending of flow rules to remote ANs is done by a userspace daemon, because this task is less performance critical[1] and a userspace daemon is easier to develop, test, and port to other operating systems (OSs). Also it has access to a much larger set of functionality via libraries.

The Exporter needs access to flow information generated by the classifier. Since the IPFW control interface (based on socket options) does not allow unsolicited messages from kernel to userspace and frequent polling of the in-kernel classifier is impractical, a separate interface (UDP socket) is used to convey the flow information from classifier to Exporter (see Section V).

C. Action Node

The AN consists of an userspace daemon (called *Collector*) that listens for flow information from CNs and configures the packet filter and traffic shaper accordingly using the existing configuration interface(s). The Collector consists of a frontend and a backend. The frontend handles the control protocol communication and manages the addition/removal of flow rules stored in an internal database. The backend is responsible for generating packet filter or traffic shaper specific rules based on the flow rules in the database.

The advantage of a userspace daemon is that no kernel code needs to be modified, again easing development, testing, porting to other OS and access to userspace libraries. Furthermore, different firewalls or traffic shapers

[1] We assume the number of simultaneous active flows that trigger remote actions is typically only a few thousand and there is a limit on how often actions are updated.

can be supported by modifying or extending the backend of the Collector. Besides packet filter or traffic shaper specific actions, other actions could be implemented, such as logging of classified traffic in a database.

D. Ruleset operations

On the CN the rule language of the packet filter needs to be extended to allow the specification of features to be computed, use of feature values in match patterns, use of ML classifiers, and the configuration of remote ANs (see Section VI). On the AN only the existing packet filter and traffic shaping functionality is used and no rule set language modifications are required.

However, in addition to the extended rule set language we need new commands to configure the Exporter and Collector (see Section VI).

E. Control protocol operations

CNs send "*add rule*" messages to ANs (see Figure 3), which contain (partial) rules that have a match part (flow specification), one or more class tags, and optionally an action part. Our notion of "add rule" messages includes the updating of existing rules. If a CN sends an "add rule" message to an AN with the same flow specification but different action as a rule sent previously, the AN must replace the previous rule with the new rule.

If a CN is configured such that it does not send actions in "add rule" messages, the receiving AN(s) must be configured so that they have a list of flow classes and associated actions. In this case the AN(s) will determine the actions based on the classes identified by the CN. If the AN(s) are configured with such action lists, the configured actions always overrule any actions specified in "add rule" messages.

Rules can have a *timeout* which will cause the AN to remove rules after a specified duration has elapsed since the rule became active (*rule timeout*), or when no packets have matched the rule for the specified duration (*flow timeout*). This is shown in Figure 2. If rule timeouts are used CNs need to periodically refresh rules (so that long running flows are properly handled).

CNs can also send explicit "*remove rule*" messages to ANs (see Figure 3). An AN will remove all rules that match the rule specification in the message. If the rule specification is broad, e.g. only the protocol is specified, this may trigger the removal of many rules. If one wants to remove exactly one rule, the flow specification in the "remove rule" message must be specified exactly as in the "add rule" message.

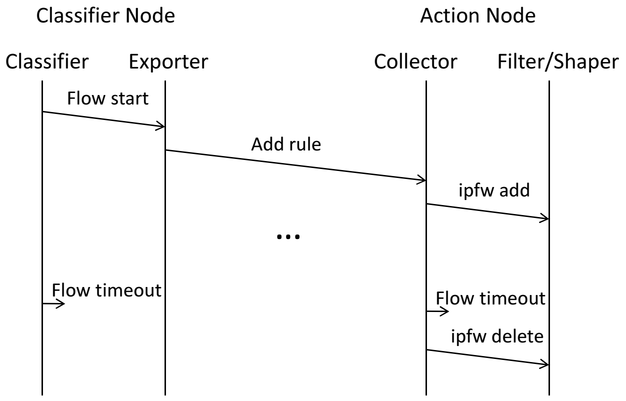

Figure 2. Rule creation and timeout based rule removal

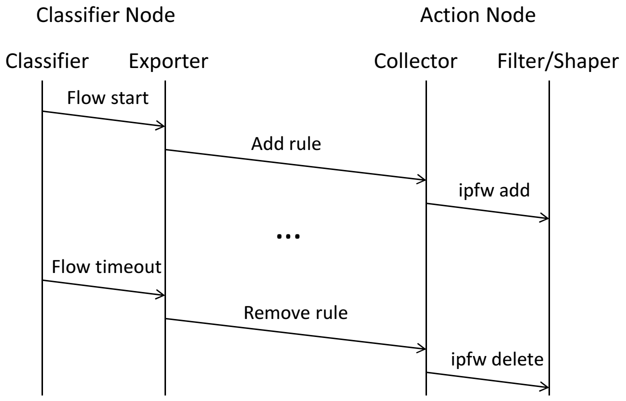

Figure 3. Rule creation and explicit rule removal messages

The use of flow timeouts has advantages over explicit "remove rule" messages. Firstly, they save network capacity as the number of messages can be reduced. Secondly, they can also prevent control loops that may occur otherwise because actions like blocking or shaping affect packet flows and their characteristics and hence the decisions of CNs.

Imagine the following scenario. A CN identifies a flow to block, triggers an AN, and the AN blocks the flow. The flow times out at the CN triggering a rule removal from the AN, which unblocks the flow. The CN identifies the flow again and so on. If the AN does the flow timeout locally (using a timeout provided by the CN), the rule will be active as long as packets are matching and then it will timeout. This approach presupposes that the flow times out at the AN before it does time out at the CN, but the CN can ensure this by controlling the AN's timeout value. This approach also covers the case when the rule is (prematurely) removed due to resource constraints at the AN.

However, flow timeouts may not be available everywhere (they may not be efficient to implement on all devices) and then “remove rule” messages must be used. But the delivery of “remove rule” messages cannot be guaranteed in all cases, e.g. a CN may crash. To avoid rules living forever in the AN, the AN purges old rules based on last recently used information (flow timeout). Should the AN run out of memory it is up to the AN to decide what to do, i.e. what rules to purge, unless it was advised by the CN what to do. For example, if the CN attached priorities to rules, then the AN must purge rules according to their priority.

In our prototype implementation by default the CN uses explicit “remove rule” messages, and the AN uses flow timeouts to time out rules. Explicit “remove rule” messages can be turned off, then both the CN and AN use flow timeouts.

The CN classifies flows all the time (in the extreme case for each packet arriving), but the CN will only notify AN(s) for new flows or changed flows (same flow specification but different class or action). Note, that a new flow may have the same flow specification as a previous flow if the previous flow has ended before the new flow started. Furthermore, there are options to limit the sending of rules even further, see Section VI-H. The protocol developed for communication between classifier nodes and action nodes is described in Section VII.

F. Example Scenarios

We illustrate the DIFFUSE v0.1 architecture in an example scenario, where the ISP differentiates a customer’s traffic into real-time and non-real-time traffic and subsequently uses this information to prioritise the real-time traffic. Figure 4 shows the customer and the ISP network. A CN with a rule database is located on or connected to an edge router inside the ISP’s network. Two ANs are located on the ISP’s edge router and customer’s router.

During operation the system does the following. The CN continuously classifies traffic flowing between the customer and ISP networks based on statistical characteristics and stored rules. For each new real-time flow detected, the CN sends the flow’s 5-tuple, class and action to the ANs. The ANs then create a new rule for the real-time flow that will prioritise its traffic over non-real-time flows. After the real-time flow has stopped, the rule is removed from the ANs.

V. Software Design

Now we briefly describe the software design of DIFFUSE v0.1. Figure 5 shows the main building blocks of the CN. Inside the kernel there is a new DIFFUSE module. At load time the DIFFUSE module registers a new raw socket option which is used to configure feature, classifier and export instances, as well as for showing and deleting them using the raw socket interface. This is the same interface used by IPFW and Dummynet.

The DIFFUSE module also registers itself with the IPFW module (which must be loaded before DIFFUSE can be loaded). After DIFFUSE has registered the IPFW module will call DIFFUSE hooks every time an IPFW rule is added or removed with an action or option unknown to IPFW. This allows the DIFFUSE module to handle the instantiation and removal of new rule actions and options, such as the `mlclass` action or the `match-if-class` option.

The IPFW module also calls a DIFFUSE hook for every packet that is checked when there are rule actions and options unknown to IPFW. This allows the DIFFUSE module to process the new actions or options, and decide whether a packet matches or not.

The raw socket interface is a pull interface only. Unsolicited messages from kernel to userspace are not possible. However, for exporting flow information, a push interface is needed. The DIFFUSE module exports flow information via the control protocol over a UDP socket, if there are any rules with export actions. The Exporter receives the flow information and forwards them to ANs, possibly using different transport protocols, such as SCTP or TCP.

Users use DIFFUSE specific config, show and delete commands as well as new rule actions and options via an extended ipfw userspace tool. A modified version of WEKA is used to generate classifier models based on previously collected and labelled traffic data. The extended ipfw userspace application parses the model files and sends the data to the DIFFUSE kernel module as part of the classifier instance configuration.

Figure 6 shows the internals of the DIFFUSE module (dashed lines indicate relations between objects and solid lines indicate message flows). DIFFUSE feature and classifier algorithms are separate modules. The config commands create configured instances of these algorithms, which are kept in linked lists. Configured export instances are also stored in a linked list. DIFFUSE actions and options in IPFW rules point to the instances.

Flow information, such as 5-tuples, flow state (e.g. TCP state, timeouts), computed features and flow classes are stored in a flow table, which is realised as hash table with last recently used sorting of the bucket lists. Export rules create flow rules that are stored in a first in first out (FIFO) list and later exported via the control protocol.

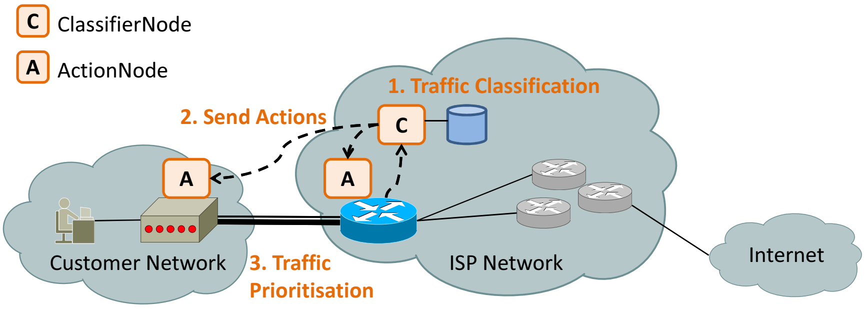

Figure 4. Using the DIFFUSE architecture for distributed traffic prioritisation

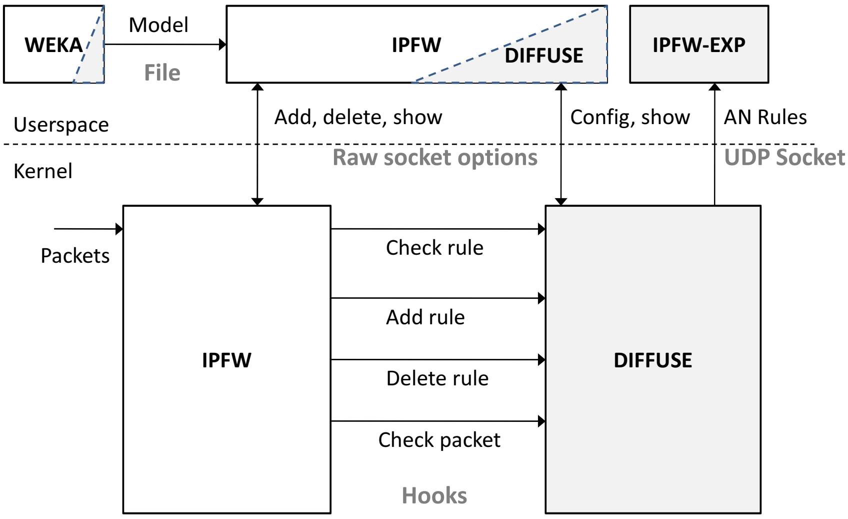

Figure 5. Classifier node design, main building blocks

Figure 7 shows the main building blocks of the AN. The Collector receives classified flows from CNs and stores them in an internal database (hash table). For new flows the collector creates according IPFW/Dummynet rules via command line (ipfw add command). Flow information is deleted based on rule removal messages or timeouts. When rules are removed from the database the Collector also removes the corresponding IPFW/Dummynet rules using the ipfw delete command. On the AN an unmodified IPFW/Dummynet can be used (without DIFFUSE extensions).

The kernel part of DIFFUSE must be implemented using the C programming language. To be able to share code between userspace and kernel and achieve high performance we decided to also implement the Exporter and Collector using C/C++. Since WEKA is implemented in Java, our WEKA extension is also implemented in Java.

VI. Design of Command Set Extensions

Here we describe the extended IPFW command set used to configure CN and AN.

A. Notation

We use the following notation based on ABNF [16]. **`Bold typewriter font`** identifies parameter names (terminal symbols) and *`italic typewriter font`* identifies parameter values chosen by the user (non-terminal symbols that do not contain spaces). Symbols in double quotes “” are also terminal symbols.

`Normal typewriter font` identifies non-terminal symbols that are broken down into parameter names and values at a later point. Parameters and values in square brackets [] are optional, a slash / defines alternatives, and round brackets () are used for grouping.

A preceding n*m means a symbol or group is repeated a minimum n and a maximum m times. If n is zero it is omitted, and if m is infinity it is omitted as well. This allows shortforms such as * for 0–infinity or 1* for 1–infinity.

B. Existing IPFW rules

First we define existing IPFW rules and a number of symbols for parts of existing IPFW rules that we later use

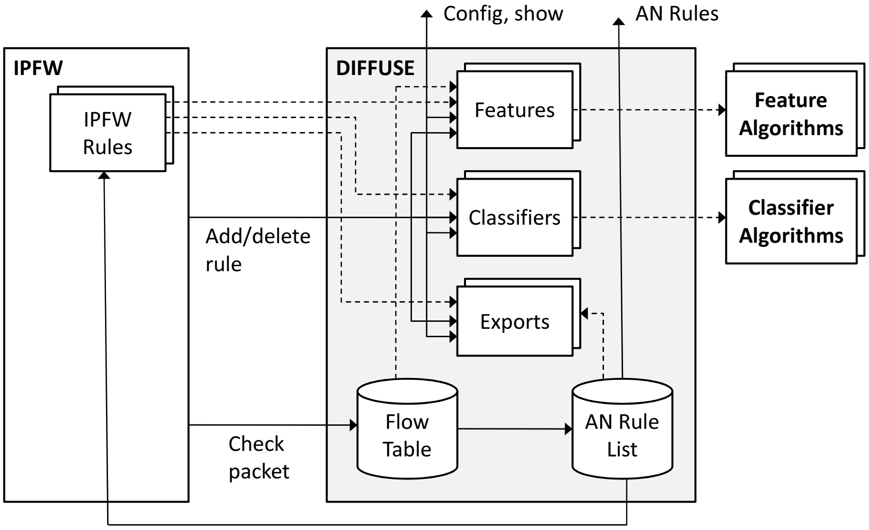

Figure 6. Classifier node design, DIFFUSE module

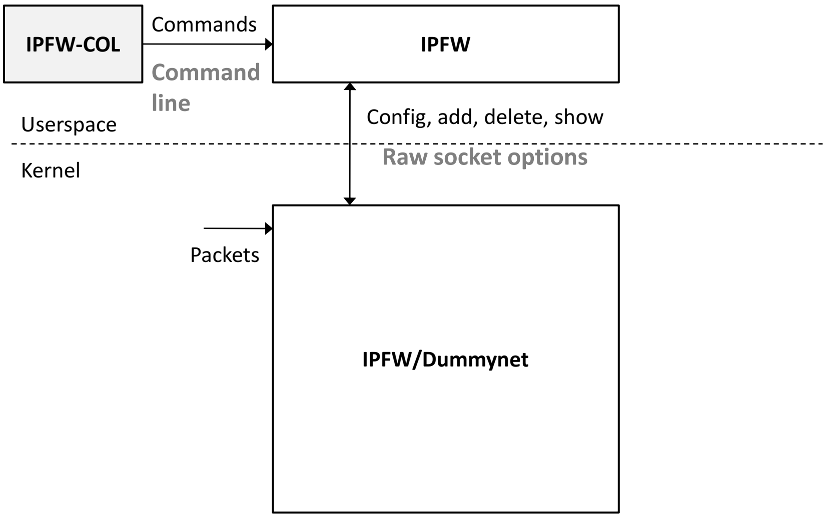

Figure 7. Action node design

in the definitions of extended rules (see Figure 8). The symbol `ipfw-rule-id` is the optional rule number, the rule set number and it also includes IPFW's match probability. The symbol `ipfw-log-altq-tag` comprises the log, ALTQ and tag options. `ipfw-action` is one of the actions executed when the pattern part of the rule matches (note that ... is a placeholder for the other actions not shown here [4]). The symbol `ipfw-patterns` describes the patterns that are used to match a packet and `options` describes all possible options, such as `keep-state`, `tagged` [4].

C. Configure, delete and show features

Figure 9 shows the command to configure a feature. The argument `feature-name` is an user-defined name for the feature. The argument `module-name` is the name of the feature as defined in the feature's implementation (basically an implemented feature module is the class and a configured feature is an instance of the class). The feature's kernel module must be available, i.e. loaded into the kernel.

The symbol `options` stands for further options provided by the feature module. Each option is either a flag (enabling or disabling a property of a feature) or a parameter name followed by an argument. Our prototype implementation creates default instances for all existing features with `feature-name` equal to `module-name`. When a feature has no options or the

```
ipfw-rule = ipfw-rule-id ipfw-action ipfw-log-altq-tag ipfw-patterns

ipfw-rule-id = [rule-number] [set set-number] [prob probability]
ipfw-log-altq-tag = [log [logamount number]] [altq queue] [(tag / untag) number]
ipfw-action = (allow / deny / count / ...)
ipfw-patterns = proto from source to destination [ipfw-options]
ipfw-options = (keep-state / tagged tag-list / ...)
```

Figure 8. Existing IPFW rules

```
ipfw feature feature-name config module module-name [options]
options = *((option-flag / (option-name option-value)) )


ipfw feature ( delete / show ) feature-name
```

Figure 9. Configure, delete and show features

default options are deemed acceptable the default feature instances can be used straight-away.

A feature can be viewed or deleted with the commands shown in Figure 9. The reserved feature name `all` can be used to show or delete all defined features.

D. Current feature options

There are two types of features: unidirectional and bidirectional. Unidirectional features compute statistics for unidirectional flows. For bidirectional flows these are computed separately for each direction. Bidirectional features compute statistics over both directions of bidirectional flows (or only one direction of unidirectional flows).

The initial protype has the features shown in Table I.

All of our initially implemented features provide the `window` and `partial-window` options. The option `window` defines the window in packets over which a feature is computed. By default features are only computed for full windows, unless `partial-window` is set. Any match for a feature statistic that has not been computed yet due to a partial window will fail. Note that a feature does not necessarily need to be computed over windows of packets, i.e. there could be features computed for single packets only.

By default the packet length computed by "plen" is the packet length of an IP packet including the IP header. If the option `ipdata-len` is used, the packet length computed is the length of an IP packet excluding the IP header. If the option `payload-len` is used the length computed is the payload length (UDP or TCP data).

The "iat" (inter-packet arrival time) feature has an option `accurate-time`. If set the computed timestamps are more accurate, at the cost of more processing time.

Figure 10 shows examples how features are defined.

E. Compute and use features for matching

Features are computed and used for matching as shown in Figure 11. Such rules compute the specified features for all subflows that match `ipfw_patterns`. Once the feature statistics are available they can be used for matching and when all IPFW patterns and feature patterns match the rule's action will be executed.

A `feature-list` is a comma-separated list of one or more feature names, specifying the features to be computed. Note that if one or more `feature-test` are present all features used in the tests will be implicitly added to `feature-list` if not already present. This means that normally one does not need to specify `feature-list`.

By default (without `unidirectional`) unidirectional features are computed separately for both directions of bidirectional subflows and feature tests can be used with an optional direction attribute. If `unidirectional` is specified, features are computed for unidirectional subflows and hence a rule matches unidirectional subflows ("fwd" or "bck" are not used in feature matches). Bidirectional features are computed over both directions of bidirectional flows and hence "fwd" or "bck" are also not used in feature matches.

If feature matches are used but `features` is not specified an implicit `features` is generated, listing all features used in the tests and producing bidirectional flows. Note that if `unidirectional` is not set rules match against features computed in both directions and if a rules matches the action will be executed for packets in *both* directions as well. With `unidirectional` a rule's action is only applied to unidirectional flows.

Table I
FEATURES AND THEIR OPTIONS

Feature	Type	Options
plen	uni	**window** size, **partial-window**, **payload-len**, **ipdata-len**
iat	uni	**window** size, **partial-window**, **accurate-time**
pcnt	bidir	**window** size, **partial-window**

```
ipfw feature myplen config plen window 10
ipfw feature myiat config iat window 20 accurate-time
```

Figure 10. Feature configuration examples

```
ipfw add ipfw-rule-id ipfw-action ipfw-log-altq-tag ipfw-patterns [features feature-list]
[1*feature-test] [unidirectional] [every / once / sample packets]
feature-list = feature-name[*(”,”feature_name)]
feature-test = [(fwd / bck)”.”]feature-statistic”.”feature-name(<=/</=/>/>=)value
```

Figure 11. Matching with features

Features are often computed for subflows and not for separate packets. Hence the matching is usually more on a per-flow basis rather than on the basis of separate packets. Mutually-exclusive options exist for controlling how often rules are re-evaluated. By default and if `every` is used a rule is evaluated for every packet. If `once` is specified a rule is evaluated only once (when all feature statistics have been computed for the first time). With `sample` a rule is evaluated every `packets` packets.

If a rule uses these options and based on them rules are not evaluated for a packet, the checking for the packet terminates immediately and the packet will be treated the same as the last packet for which the complete checking was performed. Note that these parameters only limit how often rules are evaluated, features are still computed for each and every packet.

The ipfw commands `delete`, `flush`, `list`, `show`, `zero` and `resetlog` work with feature-enabled rules as before. Figure 12 shows example commands for using feature matches.

F. Use machine learning classifier

An ML classifier takes a set of features of a subflow F ($n = |F|$) as input and returns a classification c, where c is exactly one class out of the set of classes C ($m = |C|$). A classifier is configured as shown in Figure 13.

The argument `classifier-name` is the name of the classifier. The argument `algorithm-name` defines the classification algorithm to use (e.g. C4.5, see Section VIII) and must be a name of a classifier module present (loaded into the kernel). The argument of `model` defines the classifier model to use for the classification. The model file is a text file created by a modified version of WEKA [17] (see Section VIII-C). The model file is parsed and the resulting classifier is setup in the kernel.

The model file contains the list of statistics needed, but the names may not be valid DIFFUSE names. Hence the parameter `use-feature-stats` allows to specify the list of features statistics the classifier uses for classifying flows. This parameter is also handy if one wants to use the same classifier model with differently generated feature statistics, for example the same statistics but generated over different window sizes. If used this parameter always overrules the statistics listed in the model file.

The argument of `use-feature-stats` must list the same number of feature statistics in exactly the same order they were used when building the classifier. Feature statistics are listed in the same format as for feature matches explained above. As for feature matches, for unidirectional features and bidirectional flows the direction is assumed to be forward (`fwd`) if not specified.

The parameter `class-names` can be used to overwrite the class names specified in the model file. The class names are used to match packets based on their class (see below).

Classification results can be used to match packets as shown in Figure 14. One can either use rules with the new `mlclass` action and use IPFW tags, or use rules with the new `match-if-class` option, or a combination of both.

For rules with `mlclass` action or `match-if-class` option(s) feature lists are generated implicitly

```
ipfw feature myplen config plen window 25
ipfw add deny udp from any to any features myplen min.myplen<100 bidirectional
ipfw add deny udp from any to any fwd.min.myplen<100 # same as above rule
```

Figure 12. Feature matching examples

```
ipfw mlclass classifier-name config algorithm algorithm-name model file-name
[use-feature-stats feature-stat(n-1)*(“,”feature-stat)] [class-names name(m-1)*(“,”name)]
feature-stat = [(fwd / bck)”.”]feature-statistic”.”feature-name


ipfw mlclass ( delete / show ) feature-name
```

Figure 13. Configure, delete show classifier

```
ipfw add ipfw-rule-id ( mlclass classifier-name / ... ) ipfw-log-altq-tag ipfw-patterns
[class-tags tag(m-1)*(“,”tag)] [match-if-class class-name:class[*(m-1)(“,”class)]] [every
/ once / sample packets]
class = ( name / “#”number )
```

Figure 14. Classes-based matching

in the same way as for feature matches (see previous sub section). For bidirectional flows actions of matching rules will be applied to both directions of flows.

The parameter `classifier_name` references a classifier configured previously. The parameter `class-tags` defines a list of IPFW tags, where each tag is associated to one class in C. As the result of the classification each packet is tagged with the tag configured for the class it matches. The tags can be used in subsequent rules for matching (with the `tagged` option of IPFW).

The new `match-if-class` parameter specifies a classifier name and a list of class names or indices that make a rule match. A preceding hash symbol “#” differentiates between class names and class numbers (see Figure 14). If the current class of a packet or subflow matches any of the classes listed the `match_if_class` option matches.

The optional `every`/`once`/`sample` was discussed in Section VI-E. A classifier can be viewed or deleted with the commands shown in Figure 13. The reserved classifier name `all` can be used to show or delete all defined classifiers. Note that classifier models are only shown for single classifiers, but not when `all` is used.

Without IPFW tags one can write rules that use an ML-based classifier as shown in Figure 15. If IPFW tags are used one can write rules as shown in Figure 16.

Flows can be classified by multiple classifiers. Hence multiple `match-if-class` with different `class-name` can be used in one rule. The standard IPFW options can be used to select what type of flows are classified. For example, if one wants to classify only UDP flows the “ip” in the example rules can be replaced by a “udp”.

Note that classification is only performed once all feature statistics needed are available. For example, if one or more of the statistics have not been computed yet because windows are only filled partially, the flow will not be classified. (In future work some of the classifiers may be extended to handle missing features.)

Besides recording a flow’s class for each classifier that classified the flow, our prototype implementation also keeps track of the number of consistent consecutive classifications. This allows adding hysteresis to the export of rules and prevents flapping. For example, “rule add” messages are only send after a flow was classified as class X for the n-th time.

G. Flow table

State for all flows for which features are computed is stored in a flow table. The flow table can be displayed with the show command (see Figure 17). This command shows information for all flows, such as the rule that generated the flow, packet and byte counters, the 5-tuple, a list of all computed features and their current values and a list of all class labels. By default only active flows are shown, but if the `expired` parameter is specified expired flows are also shown. (Expired flows are no longer active but their state is still in the table. State is not immediately deleted when flows expire, but only when more space is needed for adding new flows.)

```
ipfw mlclass myclass config algorithm c4.5 model /etc/ipfw/realtime.model use-feature-stats
fwd.min.myplen,fwd.mean.myplen,fwd.max.myplen,bck.min.myplen,bck.mean.myplen,bck.max.myplen
class-names rt,nonrt
ipfw add pipe 1 ip from any to any match-if-class myclass:rt
ipfw add pipe 2 ip from any to any match-if-class myclass:nonrt
```

Figure 15. Matching using the new match-if-class option

```
ipfw mlclass myclass config algorithm c4.5 model /etc/ipfw/realtime.model use-feature-stats
fwd.min.myplen,fwd.mean.myplen,fwd.max.myplen,bck.min.myplen,bck.mean.myplen,bck.max.myplen
class-names rt,nonrt
ipfw mlclass myclass ip from any to any class-tags 1,2
ipfw add pipe 1 ip from any to any tagged 1
ipfw add pipe 2 ip from any to any tagged 2
```

Figure 16. Matching using IPFW tags

```
ipfw flowtable show [expired]
ipfw flowtable ( zero / flush )
```

Figure 17. Flow table commands

All entries in the flow table can be removed (flushed) or the packet and byte counters can be zeroed by using the flush or zero commands (see Figure 17).

H. Export flow rules to ANs

On the CN we need to configure rules that decide what information the classifier sends to the Exporter or to remote AN(s).

Firstly, an export target needs to be configured as shown in Figure 18. The argument `export-name` is the name of the new export target instance. The argument of `target` is the destination of the flow rules. The protocol must be UDP, `host` is the fully qualified host name (or IP address), and `port` is the port number the target is listening on. The arguments `action-name` and `action-params-val` are the action name and parameters that are send for matching flows. Note that action name and parameters must specify valid IPFW actions[2]. Note that the receiving AN(s) may overrule these with locally specified actions.

The argument of `min-batch` is the minimum number of flow rules exported in one batch. Similarly, the argument of `max-batch` is the maximum number of flow rules exported in one batch (must be equal or larger than `min-batch`). Note that increasing `min-batch` also increases the delay for delivering flow rules.

The argument of `max-delay` specifies a maximum delay between the generation of flow rules and their export. Note that if `max-delay` is set (value larger than zero) the minimum batch size is still enforced, but the maximum batch size can now be exceeded (if at the time of exporting more rules are over the maximum delay than the size of the maximum batch).

The argument of `confirm` specifies how many times a flow has to be consecutively classified as the same class before flow information is exported. For example, if `confirm` is set to 2, information is only exported if the class was confirmed twice (three consecutive classifications resulting in the same class).

By default ANs treat flows as bidirectional, i.e. apply actions to both directions of a flow, or not. Setting `unidirectional` instructs ANs to treat flows as unidirectional, but only if they were unidirectional at the CN as well. However, based on local configuration the receiving AN(s) may still decide to treat flows differently.

Secondly, we need to define the rules that, if they match, will send flow rules to the configured AN(s) as shown in Figure 18. A rule with the new `export` target will export flow rules according to the configured exporter `export-name` for all flows that match the rule. Figure 19 shows an example where all flows classified as real-time are exported to localhost.

The classifier in the kernel can only export information via the UDP transport protocol. If UDP is sufficient the classifier can send rules to ANs directly. However, in many cases UDP will not be appropriate, for example if reliable transport is required (see Section VII). In

[2] In the initial prototype implementation these are opaque values that are not checked.

```
ipfw export export-name config target udp://host:port [action action-name] [action-params
action-params-val] [min-batch number] [max-batch number] [max-delay delay] [confirm number]
[unidirectional]
ipfw add ipfw-rule-id ( export export-name / ... ) ipfw-log-altq-tag ipfw-patterns
diffuse-patterns
```

Figure 18. Configure export target and trigger export of rules

```
ipfw export myexp config target udp://localhost min-batch 1 max-batch 5
ipfw add export myexp ip from any to any match-if-class myclass:rt
```

Figure 19. Export flow rules example

this case the classifier needs to send the information to the userspace Exporter via UDP, which then forwards the information to the Collector via SCTP or TCP (see Section VII).

The Exporter is configured as shown in Figure 20. By default the Exporter listens for flow rules from any (kernel) classifier on the default port 3191. The -c switch can be used to specify a particular classifier host and change the default port number[3]. The flow information is forwarded to a number of ANs specified as list of URLs with the -a switch. The -q switch turns off any output to stdout.

Note that not only the 5-tuple describing the flow, the class tags and the action is exported to ANs. A number of other data is sent as well, such as a bidirectional flag that specifies if actions should be executed for both directions of bidirectional flows, rule timeouts etc. (see Section VII).

I. Listen to remote CN

On the AN we need to configure the Collector to listen to remote Exporter(s) as shown in Figure 21. The parameters -s, -t and -u specify on which SCTP, TCP, UDP ports the Collector is listening (at least one of these must be specified). The -n switch turns off the IPFW rule generation (useful for testing as non-root). The -q switch turns off any output to stdout. The -r switch specifies the IPFW rule number space used by rules generated by the Collector (default 1000–2000). The Collector will create as many IPFW rules as fit into this space. Note that it is the users responsibility to ensure that the range specified is available.

The -c switch defines a file that defines a mapping between classes and actions (*actions file*). If flow rules are received with one of the classes specified in the actions file, the specified actions will always overrule any actions given by the CN. The syntax of the actions file is shown in Figure 22. Figure 23 shows an example actions file.

The Collector is independent of the firewall or traffic shaper used to treat flows. However, the initial version of the Collector can only be used with IPFW. Note that in the first version of the prototype action names and parameters are opaque values, which are not checked by the Collector.

The Collector has its own dynamic flow table. For each rule received from a CN via an “add rule” command the Collector checks if the rule is already present in the table. If a flow rule is present with same flow specification and action, the collector only updates the timeouts (if any). If a flow rule is present with the same flow specification but different action and there is no actions file, the Collector replaces the old rule with the new rule and updates timeouts (if any). If no rule is present with the same flow specification the collector inserts the new rule into the table.

If a new rule was inserted or an existing rule was updated the Collector will create a new IPFW rule or update an existing IPFW rule by using the IPFW command line interface. Removal of rules occurs upon timeout or explicit request (“remove rule” messages). In both cases the Collector removes the rule from its internal database and then removes the IPFW rule via IPFW’s command line interface.

Figure 24 shows an example for configuring an Exporter and Collector.

VII. Design of Remote Actions Protocol

We first discuss requirements on the transport protocol and message encoding. Then we describe the design of the protocol used to transmit flow rules from CNs to ANs.

[3] Typically the Exporter runs on the same host as the kernel classifier, but it could run on a different host.

```
ipfw_exp [-c host:port] [-a list-of-urls] [-q]
list-of-urls = url*(“,”url)
url = (udp / tcp / sctp)”://”host”:”port
```

Figure 20. Userspace exporter configuration

```
ipfw_col [-c actions-file] [-r min-rule-no[“-”max-rule-no]] [-s sctp-port] [-t tcp-port] [-u
udp-port][-nq]
```

Figure 21. Collector configuration

```
actions-file = 1*line
line = ( # comment / default / classifier-name:class_number) action [action_parameters]
```

Figure 22. Actions file syntax

```
# class 0 is rt and class 1 is non-rt
default queue 2
myclass:0 queue 1
myclass:1 queue 2
```

Figure 23. Actions file example

A. Transport Protocol

UDP and TCP are the main transport protocols currently used in IP networks. UDP is a connectionless protocol providing unreliable data transport without flow and congestion control. Due to its simplicity the overhead (both in terms of network capacity and CPU utilisation) is lower than for TCP. Furthermore, it allows more precise timing of messages by the sender.

TCP on the other hand is a connection-oriented protocol and provides flow and congestion control as well as reliable transport of data at the cost of higher overhead and less control for the sender. TCP overhead can be somewhat reduced by measures such as persistent connections (e.g. used between web browsers and web servers).

SCTP is a newer transport protocol that provides a number of advanced features that are very useful for DIFFUSE v0.1. SCTP is more reliable than TCP, as it has a stronger checksum, and supports transparent fail-over a) over different network interfaces of one host and b) over different hosts due to its multi-homing capability. SCTP allows out of order delivery of data which prevents the head of the line blocking problem inherent in TCP. Furthermore, SCTP allows to bundle different independent data transfers (called streams) into one connection (called association), so only a single socket is needed. PR-SCTP is an extension of SCTP that also provides timely unreliable data transport (avoiding unnecessary retransmissions) with congestion control.

By default SCTP will use multiple network interfaces for one association, so it provides fail-over in case an interface on either side of an association becomes (temporarily) unusable. Furthermore, SCTP allows one-to-many socket associations, which can be used by a CN to transmit the same message to multiple ANs simultaneously. One-to-many socket associations can also be used to provide fail-over, e.g. if there are redundant ANs.

Our main criteria for the transport protocol are reliability, timeliness of message reception, congestion control and overhead (network and computational at the sender/receiver). We now discuss our requirements on the transport protocol taking into account different scenarios.

For traffic prioritisation one might not need maximum reliability, but it depends on the business case. If customers pay money for an improved QoS then it should be a very reliable service. For security-based applications often very high reliability is required. For market research high reliability is probably not needed. Very importantly a timely message delivery is needed for traffic prioritisation or security-based applications.

In a closed network which is dimensioned properly congestion control may not be necessary. But the Internet Engineering Taskforce (IETF) mandates the use of congestion control in the Internet (as demonstrated

```
ipfw_exp -c localhost -a sctp://action1.node:5000 -q
ipfw_col -c class_actions.txt -r 10000-20000 -s 5000 -t 5000 -q
```

Figure 24. Exporter and Collector configuration example

during the standardisation of the IPFIX protocol [18]). The overhead of the transport protocol is a less important criteria, since we usually would not have extremely short messages. With SCTP reliability is tunable and inverse proportional to overhead, but even when completely unreliable SCTP still has more overhead than UDP.

Another issue is current deployment. UDP and TCP are generally supported by every end host and network device. SCTP is generally available on all end hosts using common modern operating systems and supported by many network devices [19]. Table II classifies UDP, TCP, and SCTP according to the criteria identified above on the scale: $--$ (worst), $-$, +, ++ (best).

In conclusion we select SCTP as default transport protocol for DIFFUSE v0.1 because it is very reliable, provides timely message delivery, with SCTP-PR reliability and overhead can be tuned, and it provides congestion control even in unreliable mode. In situations where reliability is not an issue or there is no packet loss and congestion control is not an issue (closed well dimensioned network), UDP may be used to provide a timely message delivery with minimum overhead. TCP may be used if reliability or congestion control are required and SCTP is not available (backwards compatibility). Which transport protocol is used can be controlled by the configuration of CNs and ANs.

B. Message encoding

Text-based protocols are easier to read and debug, easier to extend with new commands or fields and easier to handle with high-level script languages, which are often used to develop prototypes. While they facilitate quick prototyping they are likely less efficient in terms of processing time. Binary encoding on the other hand has less overhead and is more efficient to parse with low-level programming languages (like C) that generate more efficient code than script languages.

Since we decided to use C/C++ for implementing DIFFUSE (see Section V), we decided to use binary encoding because with C/C++ binary protocols can be handled more easily and efficiently than text-based protocols. Also since many data fields are stored as binary numbers inside the kernel classifier, a text-based protocol would require a large number of binary to text conversions and vice versa.

Furthermore, binary encoding has significantly less network overhead compared to text-based encoding. To minimise overhead but still have a flexible and extensible protocol we decided to use a template-based encoding, where templates define the fields present in datasets and datasets only contain the actual data.

C. Protocol format

The protocol is designed to have minimum overhead while still being flexible enough to allow further extension in the future. Flexibility is crucial, because although we outlined some scenarios for which the DIFFUSE architecture could be used, we think there are many other possible scenarios.

To avoid gratuitously reinventing a wheel, our protocol is conceptually based on the IP Flow Information Export (IPFIX) protocol [18], which was developed for very similar requirements [20]. Our protocol uses the same template-based approach and similar binary encoded messages. However, the format of protocol headers and fields is not identical to IPFIX.

1) Fixed header: Every message of the protocol has a fixed header comprised of (see Figure 25):

- Version (16 bit)
- Message length (16 bit)
- Sequence number (32 bit)
- Timestamp (32 bit)

Version specifies the protocol version. Message length is the total length of the message in octets including the fixed header. The sequence number numbers all messages. It is required to determine the order of messages (in case UDP or unordered SCTP is used and packets are reordered), can be used for retransmission of information over UDP and also provides weak security against insertion attacks with UDP as packets with sequence numbers out of the acceptable window will be silently ignored.

The Timestamp contains the time the message was generated (in seconds since Unix epoch). It allows the Collector to determine when a message was sent by the Exporter, i.e. how old the information is (assuming clocks are synchronised). The collector can use this information to adjust timeouts or ignore outdated information.

Table II
CRITERIA OF TRANSPORT PROTOCOLS

Protocol	Reliability	Timeliness	Cong. Control	Overhead	Deployment
UDP	−	+	−	++	++
TCP	+	−	+	−	++
PR-SCTP	− to ++	+	+	+ to −−	+

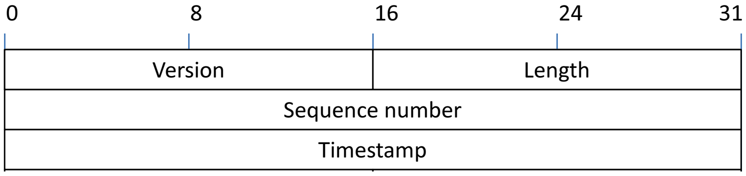

Figure 25. Fixed header of the DIFFUSE v0.1 control protocol

2) Templates and data sets: After the fixed header each message contains a number of sets. Currently there are three types of sets:

- Options template
- Flow rule template
- Option and flow rule data

Flow rule data sets are used to transmit flow rules. Option data sets are used to transmit optional data. Optional data can be transmitted with different frequencies, e.g. on a per connection/association basis or on a per message basis. Options and flow templates specify the types of information elements (IEs) contained in options/flow datasets.

Each dataset has a fixed header which contains the following fields:

- Set ID (16 bit)
- Length of set (16 bit)

Set ID specifies whether the data is an options template (set ID = 0), a flow rule template (set ID = 1) or data (set ID $\geq$ 256). Length is the length of a template or data set in octets including the set header.

An options or flow rule template set contains the following fields:

- Template ID (16 bit)
- Flags/reserved (16 bit)
- A number of field definitions each consisting of an IE ID (16 bit) and optionally the length of the data in octets (16 bits)

The template ID specifies an ID for the particular template that is then referenced in a dataset (values 256–65535). The following 16 bits are reserved for future use. Each IE ID specifies an information element, e.g. the source IP address. The length defines the length of the data in octets, e.g. length is 4 bytes for an IPV4 source address.

There are three types of IEs: fixed-length, variable-length and dynamic-length. For fixed-length IEs the IE ID also specifies the length (e.g. source IP address) and the next field is another IE ID. For variable-length IEs (e.g. a string) the length of the IE must be specified in the template in the length field following the IE ID. The length of dynamic-length IEs varies with each entry in a dataset. The first octet of a dynamic-length field in a dataset specifies the length of the field in octets (including the length field).

Whenever possible fixed-length and variable-length IEs should be used. Dynamic-length IEs should only be used if the IE length is unknown in advance and can vary significantly between entries in a data set. The highest two bits of the IE ID specify the type. If set to 00 or 01 the IE is fixed-length, if set to 10 the IE is variable length, and if set to 11 the IE is dynamic-length. This means IDs 0–32767 are for fixed-length IEs, IDs 32768–49151 are for variable-length IEs, and IDs 49152–65535 are for dynamic length IEs.

A dataset contains the data for all the IEs specified in the template in exactly the same order as specified in the template. Note that sets must aligned on 32-bit boundaries. Padding octets must be added at the end of sets as needed to ensure this.

One could reduce the overhead of the protocol by using 8-bit integers instead of 16-bit integers for IDs. However, it has been shown many times that original protocol designers severely underestimated the future need for numbering space thus necessitating protocol redesign or the use of bad hacks later on. Instead of trying very hard to optimise the overhead by reducing the size of fields one could use lossless compression to reduce the size of the data on the wire, e.g. adaptive Huffman coding is successfully used by First-person Shooter Games to reduce message sizes. However, compression increases the complexity and requires extra computational resources at the sender/receiver.

3) Template management: Depending on the transport protocol there are two ways of handling templates:

- Transmit templates in every packet (UDP)
- Transmit templates only at the start of connections (SCTP, TCP)

If UDP is used, templates are transmitted in each packet. Packets are self-contained and loss of packets with templates will never cause loss of data beyond the lost packet (data that cannot be used because the template is not known). No additional reliability is needed for templates and there are no issues if an AN restarts. Also, an AN does not need to store templates. However, network protocol overhead is higher.

If TCP is used the Exporter only needs to transmit templates once at the start of a connection, because TCP provides ordered reliable transport. The Collector must store the templates for the duration of the TCP connection. However, if a connection is closed and re-established, the Exporter must send all templates at the start of the new connection again, because it cannot know if only the connection was closed or the Collector had to restart and may have lost templates. Transmitting the templates only at the start reduces the overhead, but requires ANs to store the templates.

If SCTP is used templates and data are transmitted reliably as for TCP. If SCTP-PR is used templates only need to be transmitted once at the start, but they must be transmitted reliably using ordered reliable SCTP. The receiver must store templates for the duration of an association. Data may be transmitted unreliably if reliable data transport is not needed. Templates and data must be send over two different SCTP streams, so there will be two streams per SCTP association (templates are send over stream 0 and data is send over stream 1).

4) Information elements (IEs): The following IEs are defined (the numbers in parenthesis are the ID and the size in octets or "V" for variable-length or "D" for dynamic-length IEs):

- IPv4 source address (1, 4)
- IPv4 destination address (2, 4)
- Source port (3, 2)
- Destination port (4, 2)
- Protocol (5, 1)
- IPv6 source address (6, 16)
- IPv6 destination address (7, 16)
- IPv4 Type of Service (ToS) (8, 1)
- IPv6 flow label (9, 4)
- Class label (10, 2)
- Match direction (11, 1)
- Message type (12, 1)
- Timeout type (13, 1)
- Timeout (14, 4)
- Action flags (15, 1)
- Action (32768, V)
- Action parameters (32769, V)
- Classifier name (32770, V)
- Export name(32771, V)
- Class tags (49152, D)

A number of IEs are fields taken straight from the IPv4 or IPv6 headers. Other IEs are explained briefly in the following paragraphs. Class label is the flow's class assigned by the classifier. Classifier name specifies the classifier that classified the flow. Match direction indicates whether matched flows are unidirectional or bidirectional. Message type specifies whether a message is an "add" or "remove" message. Timeout type specifies whether the timeout is a flow timeout or a rule timeout. Timeout is the timeout value in seconds.

Action flags are used to indicate whether the AN should apply actions to unidirectional or bidirectional flows. Action is the action name and action parameters specify the parameters of the action. Export name is the name of the export that generated the flow rule. Class tags defines a list of classifier name and class tuples (if flows were classified by multiple classifiers).

The prototype implementation uses a standard template with the following fields: export name, message type, IPv4 source and destination address, source and destination port, protocol, list of classes, timeout type and value, action name, flags and parameters. With a single class tag the size of one entry is 54 bytes. (Note that IPv6 is not fully suppported yet.)

5) Keep-alive: To minimise the overhead of establishing and shutting down connections repeatedly, a connection between CN and AN is kept open even if there is nothing to send for a while. If PR-SCTP or TCP is used this keep-alive mechanism is based on the SCTP association or TCP connection keep-alive mechanism. UDP is connection-less and there is no overhead, hence no keep-alive mechanism is used in the case of UDP.

D. Example

Figure 26 shows an example message consisting of the fixed header, a template set defining the IEs for template 256, and a flow rule data set for template 256 with multiple rules.

E. Security Considerations

We assume that often CNs and ANs are part of the same trusted network, which includes being connected via a secure Virtual Private Network (VPN). This prevents alteration or eavesdropping attacks on messages in flight. However, if CNs and ANs are connected via an untrusted network the integrity of messages must be protected against attackers by using digital signatures or encryption. If messages contain sensitive information encryption must be used to preserve message confidentiality.

An AN needs to authenticate messages; it must verify that messages were created by a trusted CN. Otherwise, attackers could send fake messages to ANs for various purposes including but not limited to obtaining services that they have not paid for (e.g. prioritisation of traffic) or mounting Denial of Service (DoS) attacks by blocking a victim's legitimate flows.

In the first version of our prototype message authentication is based on IP addresses. An AN only accepts messages from specified CN IP addresses on the specified ports using the specified protocols. If IP spoofing can be prevented the system will be reasonably secure. If IP spoofing is possible the sequence numbers provide some protection against blind insertion attacks. However, if strong protection against such attacks is required cryptographic authentication of messages must be used.

Like IPFIX strong security for our protocol can be provided by the Transport Layer Security (TLS) [21] or Datagram Transport Layer Security (DTLS) [22] protocols. For UDP and PR-SCTP DTLS must be used. For TCP TLS must be used.

VIII. Selected ML Technique

We leverage the Waikato Environment for Knowledge Analysis (WEKA) [17] to perform the initial data analysis and to build classifier models used for classification. WEKA provides an easy to use GUI as well as a command line interface to inspect the data, experiment with different classification techniques and build models from training data. After a classifier model has been trained in WEKA it can be saved and used with DIFFUSE v0.1.

WEKA provides access to many different classification techniques, but our first prototype only supports two algorithms because of the effort required for implementing them. Not only must we port WEKA's classification functions, but we must do this working around the restrictions imposed on kernel modules, such as the absence of floating point arithmetic and mathematical functions in the kernel. For each classifier we must also implement functions to parse and output a model.

There are many different ML algorithms [1]. Previous research showed that for classification of network traffic the better techniques provide similar accuracy, but differ greatly regarding training time and classification speed [23]. Our initial implementation supports the C4.5 and Naïve Bayes techniques.

We use the C4.5 decision tree classifier [24] because it provided good accuracy for network traffic classification previously [23], the classification function is fast (tree search) and relatively easy to implement (unlike other algorithms it does not require mathematical functions not implemented in the FreeBSD kernel). Using a decision tree algorithm has the advantage that a human can interpret the resulting classifier (classification tree), although with increasing size this becomes difficult.

The Naïve Bayes technique also was previously used to classify network traffic [23]. While the achieved accuracy of Naïve Bayes was lower than for C4.5, the classification function is fast and very easy to implement. Due its simplicity Naïve Bayes is significantly quicker than C4.5 in building a classifier (training).

It may be hard to implement more complex classifiers in kernel because of the lack of mathematical functions. Future work could investigate ways of diverting packets (or a list of features) to a userspace application that performs the classification.

A. C4.5

C4.5 creates a classifier based on a tree structure of nodes, branches and leaves [24]. Nodes in the tree represent features, and branches represent value tests. A series of nodes and branches is terminated by a leaf, which represents the class. Determining the class of an instance is simply a matter of tracing the path of nodes and branches to a terminating leaf node.

C4.5, as other decision tree learners, uses the 'divide and conquer' method to construct a tree from a set of training instances S. If all cases in S belong to the same class, the decision tree is a leaf labelled with that class. Otherwise the algorithm will use tests to divide S into several non-trivial partitions.

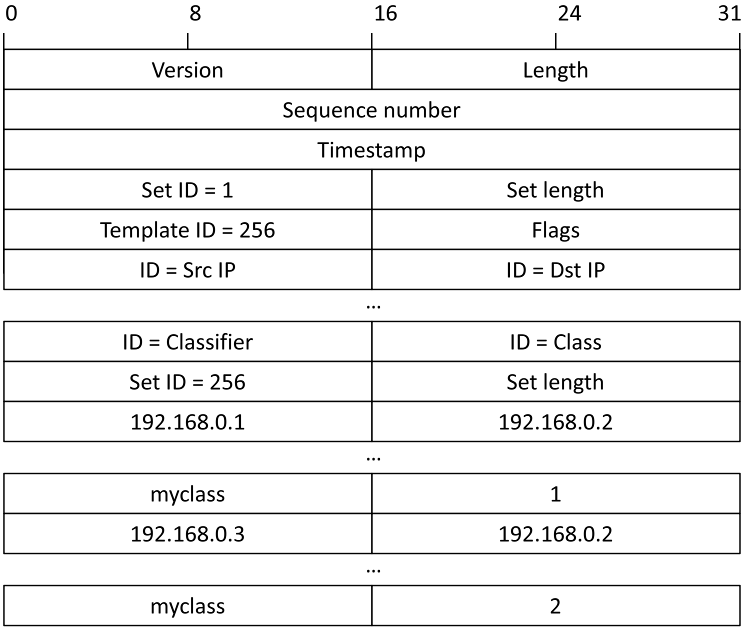

Figure 26. Example of DIFFUSE v0.1 control protocol message including a flow rule template and dataset

Each of the partitions becomes a child node of the current node and the tests to separate S are assigned to the branches. C4.5 uses two types of tests each involving only a single attribute A. In case of discrete attributes the test is $A = ?$ with one outcome for each value of A. For real attributes the test is $A \leq \theta$ where θ is a constant threshold. To find the optimal partitions C4.5 relies on greedy search and selects the test set that maximizes an entropy-based gain ratio [24].

The divide and conquer approach partitions the data until every leaf contains instances from only one class or a further partition is not possible because two instances have the same features but different class. If there are no conflicting cases the tree will correctly classify all training instances. This over-fitting leads to a decrease of the prediction accuracy. C4.5 attempts to avoid over-fitting by removing some structure from the tree after it has been built (tree pruning) [24].

Because C4.5 selects feature tests in order of maximising the entropy-based gain ratio it is not adversely affected by unimportant or irrelevant features like other techniques. The most useful features are always used at the top of the tree and irrelevant features are ignored. Feature pre-selection is not necessary, although sometimes it still improves accuracy slightly [23].

B. *Naïve Bayes*

Naïve-Bayes is based on the Bayesian theorem [23]. It analyses the relationship between each feature and the class for each instance to derive a conditional probability for the relationships between feature values and class. During training, the probability of each class is computed by counting how many times it occurs in the training dataset (called the prior probability). The prior probability is the probability that an instance belongs to a class without taking any features into account.

In addition the algorithm also computes the probability for an instance given its features and class. This probability cannot be directly computed but under the assumption that the features are independent it becomes the product of the probabilities of each single feature. The probability that an instance belongs to a certain class can then be computed by combining the prior probability and the probability from the density function for each class using the Bayes formula [23].

The Bayes formula is only applicable if all features are qualitative (nominal). A qualitative feature takes a small number of values. Then the probabilities can be estimated from the frequencies of the instances in the training data set. Quantitative features can have a large number of values (possible infinite) and the probability cannot be estimated from the frequency distribution. Instead these features must be modelled by some continuous probability distribution (often the Gaussian distribution is used). An alternative approach is to use discretisation, which transforms quantitative features into qualitative features.

Since the true density is usually unknown for real-world data, unsafe assumptions often occur when using continuous probability density functions. Discretisation circumvents this problem. On the other hand kernel density estimation can be used instead of simple density functions to model complex distributions.

In theory, a Naïve-Bayes prediction will only be correct if all the features are statistically independent of each other and the quantitative features behave according to the probability density models. However, in practice the algorithm often produces good results even when these assumptions are violated [23].

C. Classifier Model File Format

WEKA saves classification models produced during the training as Java serialised objects. This format is relatively complicated and no C/C++ parsers exist. To use a model generated with WEKA in DIFFUSE v0.1 we have extended WEKA. A command line switch (-y) was added that saves WEKA models in an ASCII format, that is easily readable for C/C++ applications.

A new interface class Diffusable was added to WEKA that needs to be used by all classifiers supported by DIFFUSE. For each classifier using the Diffusable interface we have implemented functions to export a classifier model in ASCII format. We now describe the export format using the same syntax as in Section VI.

As shown in Figure 27 each model file first lists the class names and feature/attribute names. The lines following these lists are classifier algorithm specific. (Spaces are explicitly indicated here by SP.)

For C4.5 each line in the model file represents a tree node and associate tests. The parameter `node` specifies the name of the node (always n_X) and the parameter `feature` specifies the name of the feature/attribute (always a_X), where X corresponds to the numeric index of a feature in the feature list or the number of the node in the tree (starting with zero). The next parameter specifies if the feature is nominal or real. The parameter `missing-class` specifies the resulting class (always c_X) if the feature is undefined (missing), where X corresponds to the numeric index of a class in the class list (starting with zero). Then the feature test is specified. There are three different cases:

- Nominal features with non-binary splits: a list of pairs of values and class/node names. Each value specifies a feature value and is followed by either a class name or node name.
- Nominal features with binary splits: a value followed by the class name or node name for equal feature values and the class name or node name for non-equal feature values.
- Real features: a real split value followed by the class or node name for lesser equal feature values and the class or node name for greater feature values.

For Naïve-Bayes the first line defines the prior probabilities of each class. The following lines define the conditional probabilities for feature value intervals (discretised features) or the parameters of the Normal distribution (non-discretised features). Naïve-Bayes with kernel-density estimation is currently not supported.

For nominal features or discretised features there is one line for each feature value (or feature value range). For each real feature there are four lines specifying the mean and standard deviation of the Normal distribution, and the weight sum and precision of the feature for each class.

As for C4.5 `feature` is the feature/attribute name (always a_X), where X corresponds to the numeric index of a feature in the feature list (starting with zero). The parameters `class-prior-prob` and `class-cond-prob` are the prior probabilities of classes and the conditional probabilities of classes depending on the feature values. The `class-cond-value` are the class values for the different attributes of real features.

Figure 28 depicts an example of a C4.5 classifier model generated for WEKA's iris dataset [17]. Figure 29 shows the first part of a classifier model generated by Naïve-Bayes for the same dataset [17].

IX. Conclusions and Future Work

This report presented the DIFFUSE v0.1 system, an extension for the IPFW packet filter and shaper [4] that provides ML-based traffic classification based on statistical properties and de-couples flow classification and treatment (distributed firewalling). We described the basic architecture and outlined the design of the software. We also defined and explained the main interfaces of the system: the extended ruleset language, the control protocol, and the format of classifier model files.

This report is not a manual. Man pages and HOWTOs are provided as part of the DIFFUSE v0.1 open source software release, which can be obtained from http://caia.swin.edu.au/urp/diffuse.

In future work we will analyse the system's classification accuracy, performance and scalability. We also will explore whether automatic (re)training of classifiers may be practically achieved using live IP traffic going past particular points inside an ISP network, and the degree

```
classes 1*(class-name SP)
attributes 1*(attribute-name SP)
( c45-model / nbayes-model )
c45-model = 1*( node feature (n / r) missing-class 1*(value (class / node) SP) /
1*(split-value (le_class / le_node) (gt_class / gt_node) SP) / 1*(value (match_class /
match_node) (no_match_class / no_match_node) SP))
nbayes-model =
prior 1*(class-prior-prob SP)
1*(feature (feature-value 1*(class-cond-prob SP)) / ((mean / stddev / weightsum / precision)
1*(class-cond-value SP)))
```

Figure 27. Model file format

```
classes Iris-setosa Iris-versicolor Iris-virginica
attributes sepallength sepalwidth petallength petalwidth
n_0 a_3 r c_0 0.6 c_0 n_1
n_1 a_3 r c_1 1.7 n_2 c_2
n_2 a_2 r c_1 4.9 c_1 n_3
n_3 a_3 r c_2 1.5 c_2 c_1
```

Figure 28. Example C4.5 model

```
classes Iris-setosa Iris-versicolor Iris-virginica
attributes sepallength sepalwidth petallength petalwidth
prior 0.33333 0.33333 0.33333
a_0 -inf-5.55 0.90566 0.22642 0.03774
a_0 5.55-6.15 0.07547 0.45283 0.20755
a_0 6.15-inf 0.01887 0.32075 0.75472
a_1 -inf-2.95 0.0566 0.66038 0.41509
a_1 2.95-3.35 0.35849 0.30189 0.4717
a_1 3.35-inf 0.58491 0.03774 0.11321
...
```

Figure 29. Example Naïve Bayes model

to which noise (packet loss and jitter) in the live traffic negatively impacts on the system's ability to recognise the same class of traffic in the future.

Acknowledgements

This project has been made possible in part by a grant from the Cisco University Research Program Fund at Community Foundation Silicon Valley for a project titled "Exploring the efficacy of distributed statistical traffic classification using modified open source packet filters".

References

[1] T. Nguyen, G. Armitage, "A Survey of Techniques for Internet Traffic Classification using Machine Learning," *IEEE Communications Surveys & Tutorials*, vol. 10, no. 4, pp. 56–76, 2008.

[2] P. Branch, "Lawful Interception of the Internet," *The International Journal of Emerging Technologies and Society*, Spring 2003.

[3] J. But, N. Williams, S. Zander, L. Stewart, G. Armitage, "Automated Network Games Enhancment Layer - A Proposed Architecture," in *Proceedings of 5th Workshop on Network & System Support for Games (NetGames) 2006*, October 2006.

[4] The FreeBSD Documentation Project, "FreeBSD Handbook, Section 30.6 IPFW." http://www.freebsd.org/doc/en/books/handbook/firewalls-ipfw.html.

[5] The OpenBSD Project, "PF: The OpenBSD Packet Filter." http://www.openbsd.org/faq/pf/.

[6] The netfilter.org Project, "Netfilter – Firewalling, NAT and Packet Mangling for Linux." http://www.netfilter.org/.

[7] S. Zander, G. Armitage, "DIstributed Firewall and Flow-shaper Using Statistical Evidence (DIFFUSE)." http://caia.swin.edu.au/urp/diffuse/.

[8] T.T.T. Nguyen, G. Armitage, "Training on Multiple Sub-flows to Optimise the Use of Machine Learning Classifiers in Real-world IP Networks," in *Proceedings of 31st IEEE Conference on Local Computer Networks*, November 2008.

[9] D. Reed, "IP Filter." http://coombs.anu.edu.au/ipfilter/.

[10] P. Dibowitz, "IPF FAQ." http://www.phildev.net/ipf/index.html.

[11] L. Rizzo, "Dummynet." http://www.iet.unipi.it/~luigi/ip_dummynet/.

[12] The FreeBSD Documentation Project, "FreeBSD Handbook, Section 30.4.6 Enabling ALTQ." http://www.freebsd.org/doc/en/books/handbook/firewalls-pf.html.

[13] Wikipedia, "IPFW – ipfirewall." http://en.wikipedia.org/w/index.php?title=Ipfirewall.

[14] D. Hartmeier, "Design and Performance of the OpenBSD Stateful Packet Filter (pf)," 2002. http://www.benzedrine.cx/pf-paper.html.

[15] M. Adamo, M. Tablò, "Linux vs OpenBSD – A Firewall Performance Test," *;LOGIN:*, vol. 30, pp. 35–42, December 2005.

[16] D. Crocker, Ed., and P. Overell, "Augmented BNF for Syntax Specifications: ABNF," RFC 5234, IETF, Janauary 2008. http://www.ietf.org/rfc/rfc5234.txt.

[17] I. H. Witten, Eibe Frank, *"Data Mining: Practical Machine Learning Tools and Techniques – 2nd Edition*. Morgan Kaufmann, San Francisco, 2005.

[18] B. Claise, Ed., "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information," RFC 5101, IETF, Janauary 2008. http://www.ietf.org/rfc/rfc5101.txt.

[19] Wikipedia, "Stream Control Transmission Protocol." http://en.wikipedia.org/wiki/Stream_Control_Transmission_Protocol.

[20] J. Quittek, T. Zseby, B. Claise, and S. Zander, "Requirements for IP Flow Information Export (IPFIX)," RFC 3917, IETF, Oct. 2004. http://www.ietf.org/rfc/rfc3917.txt.

[21] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246, IETF, August 2008. http://www.ietf.org/rfc/rfc5246.txt.

[22] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security," RFC 4347, IETF, August 2006. http://www.ietf.org/rfc/rfc4347.txt.

[23] N. Williams, S. Zander, G. Armitage, "A Preliminary Performance Comparison of Five Machine Learning Algorithms for Practical IP Traffic Flow Classification," *SIGCOMM Computer Communication Review*, vol. 36, October 2006.

[24] R. Kohavi, J. R. Quinlan, *Decision-tree Discovery*, ch. 16.1.3, pp. 267–276. Oxford University Press, 2002.