TIE: a community-oriented traffic classification platform

Alberto Dainotti, Walter de Donato, Antonio Pescapé, and Giorgio Ventre
University of Napoli "Federico II", Italy
{alberto,walter.dedonato,pescape,giorgio}@unina.it

***Abstract*— During the last years the research on network traffic classification has become very active. The research community, moved by increasing difficulties in the automated identification of network traffic and by concerns related to user privacy, started to investigate and propose classification approaches alternative to port-based and payload-based techniques. Despite the large quantity of works published in the past few years on this topic, very few implementations targeting alternative approaches were made available to the community. Moreover, most approaches proposed in literature suffer of problems related to the ability of evaluating and comparing them.**

In this paper we present a novel community-oriented software for traffic classification called TIE, which aims at becoming a common tool for the fair evaluation and comparison of different techniques and at fostering the sharing of common implementations and data. Moreover, TIE supports the combination of more classification plugins in order to build multi-classifier systems, and its architecture is designed to allow online traffic classification. In this paper, we also present the implementation of two basic techniques as classification plugins, which are already distributed with TIE. Finally we report on the development of several classification plugins, implementing novel classification techniques, carried out through collaborations with other research groups.

I. Introduction

The problem of traffic classification (i.e. associating traffic flows to the applications that generated them) has attracted increasing research efforts in recent years. This happened because, lately, the traditional approach of relying on transport-level protocol ports has become largely unreliable [1], pushing the search for alternative techniques. At first, research and industry focused on approaches based on payload inspection. However, such techniques present several drawbacks preventing their deployment under realistic scenarios, e.g.: (i) their large computational cost makes difficult to use them on high-bandwidth links; (ii) requiring full access to packet payload poses concerns related to user privacy; (iii) they are typically unable to cope with traffic encryption and protocol obfuscation techniques. For these reasons, the research community started investigating and proposing classification approaches that consider other properties of traffic, typically adopting statistical and machine-learning approaches [2] [3] [4]. Despite the large quantity of works published in the past few years on traffic classification, aside from port-based classifiers ([5]) and those based on payload inspection ([6] [7] [8]), there are few implementations made available to the community that target alternative approaches. NetAI [9] is a tool able to extract a set of features both from live traffic and traffic traces. However it does not directly perform traffic classification, but relies on external tools to use the extracted features for such purpose. To the best of our knowledge the only available traffic classifier implementing a machine-learning technique presented in literature is Tstat 2.0 [10] (released at the end of October 2008). Besides supporting classification through payload inspection, Tstat 2.0 is able to identify Skype traffic by using the techniques described in [11]. However such techniques have been specifically designed for a single application and can not be extended to classify overall link traffic. The lack of available implementations of novel approaches is in contrast with two facts: (i) scientific papers seem to confirm that it is possible to classify traffic by using properties different from payload content; (ii) there are strong motivations for traffic classification in general, and important reasons to perform it without relying on packet content. It has been observed that the novel approaches proposed in literature suffer of problems related to the ability of evaluating and comparing them [12]. A first reason for this difficulty is indeed the lack of implementations allowing third parties to test the techniques proposed with different traffic traces and under different situations. However, there are also difficulties related to, e.g., differences in the objects to be classified (flows, TCP connections, etc.), or in the considered classes (specific applications, application categories, etc.), as well as regarding the metrics used to evaluate classification performance.

To overcome these limitations, in this work we introduce a novel software tool for traffic classification called *Traffic Identification Engine* (TIE). TIE has been designed as a community-oriented tool, inspired by the above observations, to provide researchers and practitioners a platform to easily develop (and make available) implementations of traffic classification techniques and to allow fair comparisons among them. In the following sections, when presenting TIE's components and functionalities, we detail some of the design choices focused on: multi-classification, comparison of approaches, and online traffic classification.

This paper is organized as follows. Section II illustrates the main architecture of TIE. In Section III and IV we give some definitions, and explain its main functionalities. Section V describes two basic classification techniques we implemented as TIE plugins: port-based classification and payload inspection through pattern matching. Section VI describes the involvement of TIE in several international collaborative projects. Section VII ends the paper presenting future activities.

II. Architecture Overview

TIE is written in C language and runs on Unix operating systems, currently supporting Linux and FreeBSD platforms. The software is made of a single executable and a series of plugins that are dynamically loaded at run time. A collection of utilities and scripts are distributed with the sources and are part of the TIE framework.

TIE is made of several components, each of them responsible for a specific task.

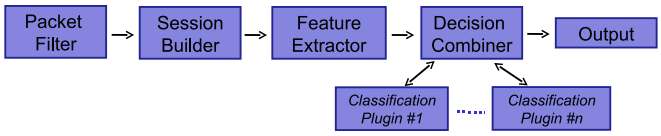

Fig. 1: TIE: main components involved in classification

Figure 1 shows the main blocks composing TIE. The *Packet Filter* is able to both capture live traffic or read from a traffic trace, and it can filter packets depending on several criteria. Packets are then aggregated into separate sessions (as explained in the following, these can be flows, biflows, etc.) by the *Session Builder*, which keeps updated the status of each session. A set of feature extraction routines (e.g. updating statistics on inter-packet times) are performed by the *Feature Extractor*. The classification is performed by the *Decision Combiner*, which coordinates the activities of several classification plugins (each one executing a different classification technique). The *Output* generates final output files with modalities and in data formats that depend on the operating mode (explained in the following). In the next sections we describe in detail each component and related tasks.

III. Operating modes

TIE supports operation on various kinds of data and different operating modes. In this section we briefly introduce the three available operating modes. Their operation will be further defined in the next section.

- **Offline Mode**: information regarding the classification of a session is generated only when the session ends or at the end of TIE execution. This operating mode is typically used by researchers evaluating classification techniques, when there are no timing constraints regarding classification output and the user is interested in obtaining information regarding the entire session lifetime.
- **Realtime Mode**: information regarding the classification of a session is generated as soon as it is available. This operating mode implements *online* classification. The typical application is policy enforcement of classified traffic (QoS, Admission Control, Billing, Firewalling, etc.). Strict timing and memory constraints are assumed.
- **Cyclic Mode**: information regarding the classification is generated at regular intervals (e.g. each 5 minutes) and stored into separate output files. Each output file contains only data from the sessions that generated traffic during the corresponding interval. An example usage is to build live traffic reporting graphs and web pages.

All working modes can be applied to both live traffic and traffic traces. Obviously, *realtime* mode is the one imposing most constraints to the design of TIE's components. We highlight that TIE was designed since the beginning targeting online classification, and this affected several aspects, described through the next section, of its architecture.

IV. TIE Functionalities

A. Packet Collection and Filtering

As regards packet capture, TIE is based on the Libpcap library [13], which is an open source C library offering an interface for capturing link-layer frames over a wide range of system architectures. It defines a common standard format for files in which captured frames are stored, also known as $tcpdump$ format, a *de facto* standard [13].

Modern kernel-level frameworks for traffic capture on Unix operating systems are mostly based on the BSD (or Berkeley) Packet Filter (BPF) [14], which allows to discard unwanted packets specifying filtering rules. Libpcap, by supporting the BPF syntax, allows programmers to write applications that transparently support a rich set of constructs to build detailed filtering expressions for most network protocols. Moreover, Libpcap allows to read packets from files in $tcpdump$ format rather than from network interfaces without modifications to the application's code except for a different function call at initialization time. This allows to easily write a single application which can work both in realtime and offline conditions.

As regards packet filtering, besides supporting the powerful BPF filters, which are called inside the capture driver, we implemented in TIE additional filtering functionalities working in user-space. Examples are: skipping the firs m packets, stopping the analysis after n packets, selecting traffic within a specified time range, and checking for headers integrity (TCP checksum, valid fields etc.).

B. Sessions

TIE decomposes network traffic into sessions, which are the objects to be classified. In literature approaches that classify different kinds of traffic objects have been presented: classifying flows, TCP connections, hosts, etc. To make TIE support multiple approaches and techniques, we have defined the general concept of session, and specified different definitions of it (selected using command line switches):

- **flow**: defined by the tuple $\{source_{IP}$, $source_{port}$, $destination_{IP}$, $destination_{port}$, *transport-level protocol*$\}$ and an inactivity timeout, with a default value of 60 seconds.
- **biflow**: defined by the tuple $\{source_{IP}$, $source_{port}$, $destination_{IP}$, $destination_{port}$, *transport-level protocol*$\}$, where source and destination can be swapped, and the inactivity timeout is referred to packets in any direction (default value is 60 seconds).
- **host**: a host session contains all packets it generates or receives. A timeout can be optionally set.

When the transport protocol is TCP, biflows typically approximate TCP connections. However no checks on connection handshake or termination are made, nor packet retransmissions are considered. This very simple heuristic has been adopted on purpose, because it is computationally light and therefore appropriate for online classification. This definition simply requires a lookup on a hash table for each packet. However, some approaches may require stricter rules to recognize TCP connections, able to identify the start and end of the connections with more accuracy. This may be the case, for example, of a classifier relying on features extracted from the first few packets (as TCP options, or packet sizes) [15] [16]. Moreover, explicitly detecting the expiration of a TCP connection avoids its segmentation in several biflows when there are long periods of silence. This behavior is typical for interactive applications like Telnet and SSH.

For these reasons, we implemented also additional heuristics, which can be optionally activated, to follow the state of TCP connections by looking at TCP flags:

- if the first packet of a TCP biflow does not contain a SYN flag then it is skipped. This is especially useful to filter out connections initiated before traffic capture was started.
- The creation of a new biflow is forced if a TCP packet containing only a SYN flag is received (i.e. if a TCP biflow with the same tuple was active then it is forced to expire and a new biflow is started).
- A biflow is forced to expire if a FIN flag has been detected in both directions.
- The inactivity timeout is disabled on TCP biflows (they expire only if FIN flags are detected).

These heuristics have been chosen in order to trade-off between computational complexity and accuracy, keeping in mind TIE's ability to work in *online* mode. Some applications, however, may require a more faithful reconstruction of TCP connections. For example payload inspection techniques used for security purposes, may require the correct reassembly of TCP streams in order to not be vulnerable to evasion techniques [17]. For these tasks, a user-space TCP reassembly state machine may be adopted and integrated into TIE, however this would significantly increase computational complexity.

Both *biflow* and *host* session types contain traffic flowing in two opposite directions, which we call *upstream* and *downstream*. For both biflow and host session types, upstream and downstream are defined by looking at the direction of the first packet (upstream direction). Information regarding the two directions must be kept separate, for example to allow extraction of features (e.g. IPT, packet count, etc.) related to a single direction. Therefore, within each session with bidirectional traffic, counters and state information are also kept for each direction.

In order to keep track of sessions status according to the above definitions we use a chained hash table data structure, in which information regarding each session can be dynamically stored.

Each session type is identified by a key of a fixed number of bits. For example, both keys of the *flow* and *biflow* session types contain two IP addresses, two port numbers, and the protocol type.

Figure 2 shows the simple hash function used for biflows. The function has been written so that source and destination hosts' IP addresses/ports can be swapped and still generate the same key.

```
/* source ip */
for (i = 12, j = 0; i != 16; i++) {
        j = (j * 13) + packet[i];
}
/* source port */
for (i = 20; i != 22; i++) {
        j = (j * 13) + packet[i];
}

/* dest ip */
for (i = 16, k = 0; i != 20; i++) {
        k = (k * 13) + packet[i];
}
/* dest port */
for (i = 22; i != 24; i++) {
        k = (k * 13) + packet[i];
}

return ((j + k + L4_PROTO(packet)) % BIFLOW_TABLE_SIZE);
```

Fig. 2: TIE: hash function used to identify and store biflow sessions.

For each session it is necessary to keep track of some information and to update them whenever a new packet belonging to the same session is processed (e.g. status, counters, features). Also, it is necessary to archive an expired session and to allocate a new structure for a new session. We therefore associate to each item stored in the hash table a linked list of sessions structures. That is, each element of the hash table, which represents a session key, contains a pointer to a linked list of session structures, with the head associated to the currently active session.

In order to properly work with high volumes of traffic, TIE is also equipped with a Garbage Collector component that is responsible of keeping clean the session table. At regular intervals it scans the table looking for expired sessions. If necessary it dumps expired sessions data (including classification results) to the output files and it then frees the memory associated to those sessions. Working in offline mode the Garbage Collector is responsible of appending classification results to the output file. In cyclic mode its work is synchronized with the dumping process made at regular intervals. Under realtime mode instead, it is only responsible to free memory of expired sessions.

C. Feature Extraction

In order to classify sessions, TIE has to collect the features needed by the specific classification plugins activated. For instance, a technique may need to access the payload of the first packet of a session in order to perform pattern matching. The Feature Extractor is the component in charge of collecting classification features and it is triggered by the Session Builder for every incoming packet. To avoid unnecessary computations and memory occupation, most features can be

collected on-demand by specifying command line options. This is particularly relevant when we want to perform online classification. The calculation of features is indeed a critical element affecting the computational load of a classifier. In [15] the computational complexity and memory overhead of some features in the context of online classification are indeed evaluated.

We started implementing basic features used by most classifiers, considering techniques of different categories: port-based, flow-based, payload inspection. We plan to enlarge the list of supported features by considering both new kinds of features and sets published in literature [18].

Classification features extracted from each session are kept in the same session structure stored in the hash table previously described. In general, each session structure in the table contains: (i) basic information (e.g. the session key, a session identifier, partial or final classification results, status flags, etc.); (ii) timing information (e.g. timestamps of the last seen packet for each direction); (iii) counters (e.g. number of bytes and packets for each direction, number of packets without payload, etc.); (iv) optional classification features (e.g. payload size and inter-packet time vectors, a payload stream from the first few packets, etc.). This structure can be easily extended to collect additional features. Moreover the collection of each on-demand feature is implemented as an inline function which can be also enabled/disabled at compile time.

D. Classification

TIE provides a multi-decisional engine made of a Decision Combiner and one or more Classification Plugins (or shortly classifiers) implementing different classification techniques. Each classifier is a standalone dynamically loadable software module. At runtime, a *Plugin Manager* is responsible of searching and loading classification plugins according to a configuration file called $enabled\_plugins$.

```
typedef struct classifier {
    int (*disable) ();
    int (*enable) ();
    int (*load_signatures) (char *);
    int (*train) (char *);
    class_output *(*classify_session) (void *session);
    int (*dump_statistics) (FILE *);
    bool (*is_session_classifiable) (void *session);
    int (*session_sign) (void *session, void *packet);

    char *name;
    char *version;
    u_int32_t *flags;
} classifier;
```

Fig. 3: TIE: interface of classification plugins.

Classification plugins have a standard interface, shown in Figure 3. To help plugin developers, a $dummy$ plugin with detailed internal documentation is distributed with TIE. Moreover the other classification plugins distributed with TIE (e.g. the Port-based classifier) can serve as sample reference code.

After loading a plugin, the Plugin Manager calls the corresponding $enable()$ function, which is in charge of verifying if all the features needed are available (some features are enabled by command line options). If some features are missing, then the plugin is disabled by calling the $disable()$ function. After enabling a plugin, the $load\_signatures()$ function is called in order to load classification fingerprints. If the loading process encounters an error then the plugin disables itself.

The *Decision Combiner* (DC in the following) is responsible for the classification of sessions and it implements the strategy used for the combination of multiple classifiers. Whenever a new packet associated to an unclassified session arrives, after updating session status information and extracting features, TIE calls the Decision Combiner. For each session, the Decision Combiner must make four choices: if a classification attempt is to be made, when (and if) each classifier must be invoked (possibly multiple times), when the final classification decision is taken, how to combine the classification outputs from the classification plugins into the final decision. To take these decisions and to coordinate the activity of multiple classifiers, the Decision Combiner operates on a set of session flags and invokes, for each classification plugin, two functions in the *classifier* structure: *is_session_classifiable()* and *classify_session()*. The $is\_session\_classifiable()$ function asks a classifier if enough information is available for it to attempt a classification of the current session. The $classify\_session()$ function performs the actual classification attempt, returning the result in a *class_output* structure, shown in Figure 4.

```
typedef struct class_output {
    u_int16_t id;        /* Application id */
    u_int8_t subid;      /* Application sub id */
    u_int8_t confidence; /* Confidence value */
    u_int32_t flags;
} class_output;
```

Fig. 4: TIE: the class_output structure stores the output of a classification attempt.

To highlight the central role of the DC, and how, thanks to few functions and structures, it allows a flexible design of its operating strategy, in the following we illustrate some sample situations regarding the four main decision mentioned above.

- **When to attempt classification**. The DC could decide to not evaluate the current session depending both on information from the classification plugins or on a priori basis. The latter may happen, for example, when the target of classification is a restricted set of traffic categories. In the first case, instead, the DC typically asks each of the active classification plugins if it is able to attempt classification on the current session. Depending on the replies from the classifiers the DC can decide to make a classification attempt. For instance, the DC may wait for all classifiers to be ready before making an attempt.
- **When each classifier must be invoked**. Depending on the classifiers that are available, the DC could decide to invoke only some of them, and only at some time, for a certain session. For example, there could be classification techniques that are applicable only to TCP biflows or some classifiers may be invoked only when

certain information is present. This is the case of payload-based classifiers. In general, we can design combination strategies with more complicate algorithms, in which the invocation of a specific classifier depends on several conditions and on the output of other classifiers. For example, a payload inspection technique is launched on a session only after that another classification plugin has suggested it is Peer-to-Peer traffic. Or, if a session is recognized as carrying encrypted traffic by a classification plugin, then the DC may start a specific classifier designed for encrypted traffic. The algorithm chosing the sequence of the classifiers to be invoked can be very simple or much more complex depending on the nature of the classification problem and on the classification techniques available.

- **When the final classification decision is taken**. This choice is usually connected to the previous one. The DC must decide when TIE has to assign a class to a session. This can happen at the arrival of any packet from the considered session. Simple strategies are, e.g., when at least one classifier has returned a result, or when all of them have returned a classification result, etc. In more complicate approaches, this choice can vary depending on the features of the session (e.g. TCP, UDP, number of packets, etc.) and the output of the classifiers. Moreover, if working in *online* mode, a limit on the time elapsed or the number of packets seen since the start of the session is typically given. If such limit has been passed, a final classification result (even if labeled as *Unknown*) is assigned.
- **How to combine the classification outputs from the classification plugins into the final decision**. The DC receives a *class_output* structure (Figure 4) from each of the classification plugins invoked. These must then be *fused* into a single final decision. The *class_output* structure contains also a confidence value returned by each of the classifiers, which can be helpful when combining conflicting results from different classifiers, and it determines the final confidence value returned by the DC. The criteria used by each classification plugin to assign a value to the confidence value is defined by the designer of the classification plugin and must be clearly reported in the plugin documenation, unless it is always set to the maximum (default). Effectively combining conflicting results from different classifiers is a crucial task. The problem of combining classifiers actually represents a research area in the machine-learning field *per se*. Simple static approaches are based on majority and/or priority criteria, whereas more complex strategies can be adopted to take into account the nature of the classifiers and their per-class metrics like accuracy [19].

We distribute TIE with a basic combination strategy as a first sample implementation. For each session, the decision is taken only if all the classifiers that are enabled are ready to classify it. To take its decision the combiner assigns priorities to classifiers according to the order of their appearance in the $enabled\_plugins$ file. If all the plugins agree on the result, or some of them classify the session as *Unknown*, the combination is straightforward and the final confidence value is computed as the sum of each confidence value divided by the number of enabled plugins. Instead, if one or more plugins disagree, the class is decided by the plugin with highest priority. To take into account the conflicting results of the classifiers, the confidence value is evaluated as before, and then divided by 2.

All the code implementing the decision combiner is in separate source files that can be easily modified and extended to write a new combination strategy. After future addition of further classification plugins, we plan to add combination strategies that are more sophisticated.

Finally, it is possible to run TIE with the purpose to train one or more classification plugins implementing machine-learning techniques with data extracted from a traffic trace. To do this, we first need pre-classified data (ground truth). These can be obtained by running TIE on the same traffic trace using a ground-truth classification plugin (e.g. the l7-filter classification plugin illustrated in Section V-B). The same output file generated by TIE is then used as pre-classified data and given as input to TIE configured to perform a training phase. For each activated classification plugin, two kinds of training functions can be invoked: the first one can be called each time the status of a session changes, the second is called after the entire traffic trace has been analyzed and all available features have been collected.

E. Data definitions and Output format

One of the design goals of TIE, was to allow comparison of multiple approaches. For this purpose a unified representation of classification output is needed. More precisely we defined IDs for application classes (we simply call them *applications*) and propose such IDs as a reference. Moreover, several approaches presented in literature classify sessions into classes that are categories grouping applications that offer similar services. We therefore added definitions of *group* classes and assigned each application to a group. This allows to compare a classification technique that classifies traffic into application classes with another that classifies traffic into group classes. Moreover, it allows to perform a higher-level comparison between two classifiers that both use application classes, by looking at differences only in terms of groups.

To build a valid application database inside TIE, we started by analyzing those used by the CoralReef suite [5], and by the L7-filter project [7], because they represent the most complete sets that are publicly available and because such tools represent the state of the art in the field of traffic analysis and classification tools. By comparing such to application databases, we then decided to create a more complete one by including information from both sources and trying to preserve most of the definitions in there.

To each application class, TIE associates the following information:

- An application identifier that univocally identifies the application.
- A human readable label to be used for readable output.
- A group identifier that associates the application to a category.

Moreover, to introduce a further level of granularity, inside each application class we allow the definition of sub-application identifiers in order to discriminate among sessions of the same application generating traffic with different properties (e.g. signaling vs. data, or Skype voice vs. Skype chat, etc.). To each sub-application the following information is associated:

- A sub-application ID.
- A human readable label to be used for readable output.
- A long description.

Each application class has at least the default generic sub-application ID "0". To obtain an easily manageable and portable application database we adopted an ASCII file format. Figure 5 shows portions of the *tie_apps.txt* file. Each line defines one application identified by the pair $(AppID, SubID)$. To properly define the application groups we started from the

```
#AppID SubID GroupID Label      SubLabel         Description
0,     0,    0,      "UNKNOWN", "UNKNOWN",       "Unknown application"
#
1,     0,    1,      "HTTP",    "HTTP",          "World Wide Web"
1,     1,    1,      "HTTP",    "DAP",           "Download Accelerator Plus"
1,     2,    1,      "HTTP",    "FRESHDOWNLOAD", "Fresh Download"
1,     7,    1,      "HTTP",    "QUICKTIME",     "Quicktime HTTP"
[...]
10,    0,    3,      "FTP",     "FTP",           "File Transfer Protocol"
10,    1,    3,      "FTP",     "FTP_DATA",      "FTP data stream"
10,    2,    3,      "FTP",     "FTP_CONTROL",   "FTP control"
[...]
4,     0,    1,      "HTTPS",   "HTTPS",         "Secure Web"
5,     0,    9,      "DNS",     "DNS",           "Domain Name Service"
```

Fig. 5: TIE: definitions of application classes from the file *tie_apps.txt*.

categories proposed by [20] and then we extended them by looking at those proposed by CoralReef [5] and L7-filter [7]. The resulting database, as shown in Figure 6, uses the same format adopted for the applications database file and contains a label and a description for each group.

```
#GID Label                 Description
0,   "UNKNOWN",            "Unknown group"
1,   "WEB",                "World wide web"
2,   "MAIL",               "Mail"
3,   "BULK",               "File transfer"
4,   "MALICIOUS",          "Malicious applications"
5,   "CONFERENCING",       "Conferencing and chat"
6,   "DATABASE",           "Database"
7,   "MULTIMEDIA",         "Multimedia (streaming)"
8,   "VOIP",               "Voice over IP"
9,   "SERVICES",           "Generic services"
10,  "INTERACTIVE",        "Interactive (login)"
11,  "GAMES",              "Games"
12,  "P2P",                "Peer-to-peer"
13,  "GRID",               "Grid"
14,  "NETWORK_MANAGEMENT", "Network management"
15,  "NEWS",               "News"
16,  "FILE_SYSTEM",        "File system"
17,  "ENCRYPTION",         "Encryption"
18,  "TUNNELING",          "Tunneling"
```

Fig. 6: TIE: file format for definitions of group classes.

The main output file generated by TIE contains information about the sessions processed and their classification. The output file is composed by a header and a body. The header contains details about the whole traffic results, the plugins activated, and the options chosen. The body is a column-separated table whose fields contain the following session related information: an unique identifier, the 5-tuple, the start/end timestamps, the packets/bytes count for both upstream and downstream directions, a $(AppID, SubID)$ pair and a confidence value as resulting from classification process. The output format is unique but counters and timestamps semantics depend on (i) the operating mode in which TIE was run and (ii) the session type.

In *offline* mode those fields refer to the entire session. In *realtime* mode they refer only to the period between the start of the session and the time the classification of the session has been made. This is done in order to reduce computations to the minimum after a session has been classified. Finally, in *cyclic* mode an output file with a different name is generated for each time interval, and the above-mentioned fields refer only to the current interval.

V. Classification plugins

We distribute the first beta version of TIE along with two basic classification plugins, implementing a port-based classifier and a deep payload inspection classifier. We implemented them for first because they represent the state of art of more traditional approaches, therefore such classification plugins can be used for comparison and evaluation purposes. As briefly illustrated in Section VI, we are currently developing more classification plugins, also through collaborations with other research groups, implementing techniques based on machine-learning and statistical approaches.

A. Port-based classification plugin

The Port-based classification plugin relies on source and destination port numbers as features. Several tools performing port based classification were available. In our search the CoralReef suite [5], developed by CAIDA, was the one with the largest and up-to-date port-based application database. To "not reinvent the wheel", the classification plugin we implemented from scratch relies on the CoralReef signature file.

To import signatures from this file we implemented a simple parser that retrieves only needed information and stores it into a hash table, in which the generic element has the structure shown in figure 7. Being that TIE determines the direction of a session differently compared to CoralReef (i.e. we consider the source port the one from the host generating the first packet), our parser swaps source and destination ports. Moreover, because TIE manages applications using an integer identifier, the parser does the mapping of each application by looking at its label.

When a signature contains port ranges or more source-destination combinations, the parser creates an entry in the hash table for each of them. This approach speeds up the classification process at the expense of few additional bytes of memory.

The algorithm implemented by the classifier on each session is very simple. It performs three lookups into the hash table by specifying the following information combinations:

- transport protocol and both source and destination ports
- transport protocol and destination port only
- transport protocol and source port only

The lookup, if successful, will return the corresponding entry containing the application identifier. The confidence value is always set to 100 when a hit occurs or set to 0 otherwise.

B. L7-filter classification plugin

Another classification plugin distributed with TIE is based on a deep payload inspection technique. We chose to implement the same technique used by L7-filter [7] inside a TIE classification plugin for the following reasons: (i) we wanted to support at least one payload-inspection technique to compare it against completely different approaches; (ii) among the publicly available tools, L7-filter is one of the most popular; (iii) we needed to implement at least one ground-truth technique in TIE, and L7-filter routines are often used in literature to build ground truth [21] [22]; (iv) the current version of L7-filter is not easy to use on traffic traces. Indeed, because of its nature, L7-filter natively works only on Linux platforms and can only analyze traffic from a network interface. The only way to run it on previously-captured traffic is to replay that traffic (e.g. using tcpreplay [23]) on a network interface. Unfortunately such trick does not allow to work at high traffic rates ($\sim$ 1 Mbps), thus practically limiting its application to small traffic traces. By supporting the same technique under TIE we do not have these limitations anymore and we can run it both under Linux and FreeBSD.

L7-filter is an open-source project for Linux and it is available in two different versions: kernel and user-space. The original project was born in kernel space, where many functionalities are implemented by the Netfilter [24] framework, the same used by *iptables* to provide firewalling, NAT (Network Address Translation) and packet mangling under Linux. The user-space version, currently in a early stage of development, gets data through Netfilter's queues and implements connection tracking from scratch.

The L7-filter classification technique uses regular expressions. A regular expression (or regexp, or pattern) is a rule, in the form of a text string, describing set of strings. In general a regexp r matches a string s if s is in the set of strings described by r.

For instance, the regex:

```
typedef struct port_info {
    u_int16_t sport;    /* source port (key) */
    u_int16_t dport;    /* destination port (key) */
    u_int8_t proto;     /* protocol type (key) */
    u_int16_t app_id;   /* application ID */
    u_int8_t app_subid; /* application sub ID */
} port_info;
```

Fig. 7: TIE: element of the port information hash table.

```
^ssh-[12]\.[0-9]
```

matches the first characters of a SSH connection, where the initial "*ssh-*" string is followed by the version number. As specified in the pattern it could be $1.x$ or $2.x$, where x is a digit from 0 to 9.

L7-filter during its startup loads the application patterns from several text files with "pat" extension.

After loading signatures, L7-filter processes packets by collecting the payload of each session into an array, independently of its direction, and removing the null bytes. Removing null bytes is necessary to pattern matching, because the regular expression engine uses null-terminated strings. The matching process is triggered by the reception of a packet carrying payload and if no match is found the session is left unclassified. If after 10 packets the session can not be classified, then it will be set as *Unknown* and its subsequent packets are ignored.

To develop a TIE classification plugin implementing the same technique used by L7-filter (TIE-L7) we started from the latest code-repository checkout of the user-space version. It was necessary to adapt some aspects of the TIE platform to work with this plugin. First, we had to add to TIE an option to make several classification attempts for the same session, by modifying the Decision Combiner. Moreover, to let TIE work with the same definition of session, it was necessary to implement some heuristics to follow the state of TCP connections. We analyzed the heuristics implemented in the user-space version of L7-filter, that simply assume that a packet carrying the FIN flag determines the expiration of a TCP session, and added them as optional. During this study we also identified few bugs in the user-space version of the code that we reported to the developers. Moreover, as explained in Section IV-B, we added more heuristics for TCP connections that can be optionally activated.

The pattern matching routines did not need any change to be ported to TIE's classification plugin. However, in order to support the FreeBSD operating system we included the GNU pattern matching libraries into the plugin package, because the implementation of such libraries under several versions of this operating system is extremely slow.

Furthermore, in order to integrate into TIE the pattern files containing signatures used by L7-filter, it was necessary to port the L7-filter parser into the plugin and to associate each application to the corresponding TIE identifiers. To set such association, at start-up, TIE-L7 loads the signatures reading the list from a configuration file, which also contains associations between each application name and the corresponding $(AppID, SubID)$ pair.

Finally, to state the equivalence of TIE-L7 with the original L7-filter (the user-space version) we added to both software few routines to output debug information. Such output reports the list of the sessions detected and the corresponding classification result. We performed tests on several traffic traces, and after fixing problems related to small differences, we verified that TIE-L7 and L7-filter produced the same output.

VI. TIE and the Research Community

TIE is a community-oriented tool, that has been designed to allow the scientific community to easily develop real implementations of classification techniques to be evaluated by anyone on real (and live) traffic and fairly compared. However, besides community needs and deficient aspects of the state of art, during the design of TIE and its development, we have constantly payed attention to what the scientific community had already produced, both in terms of functionalities and data definitions/formats. Few examples follow:

- TIE uses the Libpcap library for live traffic capture and trace management, which is a *de facto* standard supported by most common operating systems. The vast majority of the traces made publicly available by the scientific community are in Libpcap (tcpdump) format, this makes them immediately usable by TIE.
- TIE supports different definitions of *sessions* according to those produced in literature.
- In the definition of classes and class IDs, we have carefully considered definitions already used by the most popular tools (e.g. CoralReef from CAIDA [5] and the Linux project L7-filter [7]). Moreover we have created a class hierarchy made of application groups, applications, and application sub-IDs, in order to represent the different types of classes considered in literature and to allow comparison even when they differ (e.g. approaches classifying applications against approaches classifying categories of applications).
- In the implementation of the first classification plugins we adopted definitions and algorithms widely used and accepted, as the CoralReef file of rules for port-application associations in the case of the Port-based classifier, and L7-filter algorithm and signatures in the case of the TIE-L7 classifier.

Furthermore, TIE was involved since its prototype stage into collaborative projects with other research groups. In particular, we cite: (i) an Italian research project, PRIN *RECIPE*, specifically focused on traffic classification and involving researchers from seven Italian universities. (ii) NETQOS, a European Specific Targeted Research Project (STREP) from the 5th call of IST FP6 framework, developing an autonomous policy-based QoS management approach for heterogeneous networks, in order to provide enhanced end-to-end QoS and efficient resource utilization. TIE has been successfully used in this project as an online traffic classifier interacting with other components of the NETQOS framework. (iii) TIE has been recognized as reference tool in the European COST Action IC0703 “Data Traffic Monitoring and Analysis (TMA): theory, techniques, tools and applications for the future networks” (shortly COST-TMA) [27] regarding all joint activities on traffic classification.

Finally, Table I summarizes TIE classification plugins that are both available or under development and highlights connections with the research community.

VII. Conclusion

In this paper we introduced a novel community-oriented software tool for traffic classification called TIE, supporting the fair evaluation and comparison of different techniques and fostering the sharing of common implementations and data. Moreover, TIE is thought as a multi-classifier system and its architecture is designed to allow online traffic classification. TIE will allow the experimental study of a number of hot topics in traffic classification, such as:

- *multi-classification*: we are working on the combination of multiple classification techniques with pluggable fusion strategies.
- *sharable data*: we are implementing algorithms to produce pre-labeled and anonymized traffic traces, which will allow the sharing of reference data for comparison and evaluation purposes.
- *privacy*: we are working on the design of lightweight approaches to payload inspection that are privacy-friendly and more suitable for online classification.
- *ground truth*: we are working on developing more accurate approaches for the creation of ground-truth reference data through the combination of multiple and novel techniques.
- *performance analysis*: disposing of multiple implementations of classification techniques on the same platform allows to fairly compare different techniques *on the field*. TIE will support the measurement of operating variable such as classification time, computational load, as well as memory footprint.

References

[1] Thomas Karagiannis, Andre Broido, Nevil Brownlee, KC Claffy, and Michalis Faloutsos. Is p2p dying or just hiding? In *IEEE Globecom*, 2004.

[2] Thomas Karagiannis, Konstantina Papagiannaki, and Michalis Faloutsos. Blinc: Multilevel traffic classification in the dark. In *ACM SIGCOMM*, August 2005.

[3] Tom Auld, Andrew W. Moore, and Stephen F. Gull. Bayesian neural networks for internet traffic classification. *IEEE Transactions on Neural Networks*, 18(1):223–239, January 2007.

[4] Nigel Williams, Sebastian Zander, and Grenville Armitage. A preliminary performance comparison of five machine learning algorithms for practical ip traffic flow classification. *ACM SIGCOMM CCR*, 36(5):7–15, October 2006.

[5] *CoralReef*. http://www.caida.org/tools/measurement/coralreef/.

[6] Vern Paxson. Bro: A system for detecting network intruders in real-time. In *Computer Networks*, pages 23–24, 1999.

[7] *L7-filter, Application Layer Packet Classifier for Linux*. http://l7-filter.sourceforge.net.

[8] Cisco Systems. *Blocking Peer-to-Peer File Sharing Programs with the PIX Firewall*. http://www.cisco.com/application/pdf/paws/42700/block_p2p_pix.pdf.

[9] *netAI: Network Traffic based Application Identification*. http://caia.swin.edu.au/urp/dstc/netai.

[10] *Tstat*. http://tstat.tlc.polito.it [November 2008].

[11] Dario Bonfiglio, Marco Mellia, Michela Meo, Dario Rossi, and Paolo Tofanelli. Revealing skype traffic: when randomness plays with you. In *SIGCOMM ’07: Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 37–48, New York, NY, USA, 2007. ACM.

[12] Luca Salgarelli, Francesco Gringoli, and Thomas Karagiannis. Comparing traffic classifiers. *SIGCOMM Comput. Commun. Rev.*, 37(3):65–68, 2007.

TABLE I: TIE classification plugins available and under development. The table highlights input from the community and joint activities.

Classification Plugin	Features based on	Classification approach	Status	Collaborations and contributions from the community
Port	Protocol ports	Port-based	Available	Developed by UNINA, signatures from CAIDA [5]
L7	Payload	Deep payload inspection	Available	Developed by UNINA, code and signatures from Linux L7-filter [7]
NBC	Payload	Lightweight Payload Inspection	To be released	Developed by UNINA
GMM-PS	First few packet sizes	Gaussian Mixture Models [16]	Under test	Developed by UNINA
HMM	Packet size and inter-packet time	Hidden Markov Models [25]	Under devel.	Development by UNINA
FPT	Packet size and inter-packet time	Statistical [21]	Under devel.	Joint work between UNINA and University of Brescia in the context of the RECIPE research project [26]
Joint	Packet size and inter-packet time	Nearest Neighbour	Under devel.	Joint work: UNINA, CAIDA, Seoul National University
GT	Information from Hosts	Ground-Truth	In early devel.	Joint work: University of Brescia, CAIDA, UNINA

[13] *Tcpdump and the Libpcap library*. http://www.tcpdump.org [November 2008].

[14] V. Jacobson S. McCanne. The bsd packet filter: A new architecture for userlevel packet capture. *Winter 1993 USENIX Conference*, pages 259–269, January 1993.

[15] Wei Li and Andrew W. Moore. A machine learning approach for efficient traffic classification. *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, October 2007.

[16] Laurent Bernaille, Renata Teixeira, and Kave Salamatian. Early application identification. In *ACM CoNEXT*, December 2006.

[17] Thomas H. Ptacek, Timothy N. Newsham, and Homer J. Simpson. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, 1998.

[18] Andrew Moore, Denis Zuev, and Michael Crogan. Discriminators for use in flow-based classification. Technical Report RR-05-13, Department of Computer Science, Queen Mary, University of London, 2005.

[19] Ludmila I. Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms*. Wiley-Interscience, 2004.

[20] Andrew Moore and Konstantina Papagiannaki. Toward the accurate identification of network applications. In *PAM*, April 2005.

[21] Manuel Crotti, Maurizio Dusi, Francesco Gringoli, and Luca Salgarelli. Traffic classification through simple statistical fingerprinting. *ACM SIGCOMM CCR*, 37(1):7–16, January 2007.

[22] Zhu Li, Ruixi Yuan, and Xiaohong Guan. Accurate classification of the internet traffic based on the svm method. In *ICC*, June 2007.

[23] *Tcpreplay*. http://tcpreplay.sourceforge.net [November 2008].

[24] *Netfilter/IPTables*. http://www.netfilter.org [November 2008].

[25] A. Pescapé P. Salvo Rossi A. Dainotti, W. de Donato. Classification of network traffic via packet-level hidden markov models. In *IEEE GLOBECOM 2008*, December 2008.

[26] *RECIPE (Robust and Efficient traffic Classification in IP nEtworks)*. http://recipe.dis.unina.it.

[27] *COST Action IC0703: Data Traffic Monitoring and Analysis (TMA): theory, techniques, tools and applications for the future networks*. http://www.cost-tma.eu.