TIE: A Community-Oriented Traffic Classification Platform

Alberto Dainotti, Walter de Donato, and Antonio Pescapé

University of Napoli "Federico II", Italy
{alberto,walter.dedonato,pescape}@unina.it

Abstract. The research on network traffic classification has recently become very active. The research community, moved by increasing difficulties in the automated identification of network traffic, started to investigate 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 have been 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.

1 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 in 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 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.

M. Papadopouli, P. Owezarski, and A. Pras (Eds.): TMA 2009, LNCS 5537, pp. 64–74, 2009.

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.

2 Operating Modes

Before describing the architecture and the functionalities we introduce the three operating modes of TIE. Their operation will be further detailed in the next sections.

– **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 needs information related to 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.

3 Architecture Overview and Functionalities

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 set of plugins dynamically loaded at run time. A collection of utilities is 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. Figure 1 shows the main blocks composing TIE.

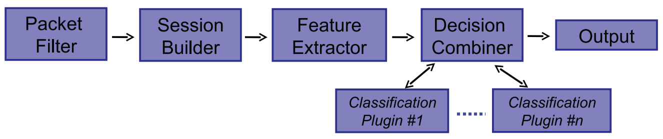

Fig. 1. TIE: main components involved in classification

3.1 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. Moreover, Libpcap allows to read packets from files in *tcpdump* format (a *de facto* standard [13]) rather than from network interfaces, without modifications to the application's code. This allows to easily write a single application able to work both in realtime and offline conditions.

By supporting the BPF syntax [14], Libpcap allows programmers to write applications that transparently support a rich set of constructs to build detailed packet filtering expressions for most network protocols. 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 first m packets, selecting traffic within a specified time range, and checking for headers integrity (TCP checksum, valid fields etc.).

3.2 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: 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 $\{SRC_{IP},\ SRC_{port},\ DEST\ _{P},\ DEST_{port},$ *transport-level protocol*$\}$ and an inactivity timeout, with a default value of 60 seconds.
- **biflow:** Defined by the tuple $\{SRC\ _{P},\ SRC_{port},\ DEST\ _{P},\ DEST_{port},$ *transport-level protocol*$\}$, where source and destination can be swapped, and the inactivity timeout is referred to packets in any direction.
- **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 (e.g. 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 (e.g. Telnet, SSH). For these reasons, we implemented heuristics to follow the state of TCP connections by looking at TCP flags that can be optionally activated:

- 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. 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 state machine may be integrated into TIE, however this would significantly increase computational complexity.

Some session types (i.e. *biflow* and *host*) contain traffic flowing in two opposite directions, which we call *upstream* and *downstream*. These 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 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.

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.

3.3 Feature Extraction

In order to classify sessions, TIE has to collect the features needed by the specific classification plugins activated. 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 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.).

3.4 Classification

TIE provides a multi-decisional engine made of a Decision Combiner (*DC* in the following) 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. 2. TIE: interface of classification plugins

Classification plugins have a standard interface, shown in Figure 2. To help plugin developers, a *dummy* plugin with detailed internal documentation is distributed with TIE. Moreover the other classification plugins distributed with TIE 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. The *DC* 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 *DC*. For each session, the *DC* 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 *DC* 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

```
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. 3. The class_output structure stores the output of a classification attempt

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 3.

To highlight the central role of the *DC* and how it is possible, with few functions and structures, to design flexible decision strategies, 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 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.
- **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, if a session is recognized as carrying encrypted traffic by a classification plugin, then the *DC* may start a classifier specifically designed for encrypted traffic.
- **When the final classification decision is taken.** The *DC* must decide when TIE has to assign a class to a 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 set.
- **How to combine the classification outputs from the classification plugins into the final decision.** The *DC* receives a *class_output* structure (Figure 3) 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*. 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. 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.

3.5 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 (*applications*) and propose such IDs as reference. Moreover, several approaches presented in literature classify sessions into classes that are groups of applications offering 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 classifying 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 an 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 two application databases, we then decided to create

```
#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. 4. TIE: definitions of application classes from the file *tie_apps.txt*

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: (i) an identifier, (ii) a human readable label, (iii) a group identifier. To properly define the application groups we started from the categories proposed by [20] and then we extended them by looking at those proposed by CoralReef [5] and L7-filter [7]. Moreover, to introduce a further level of granularity, for 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.). Figure 4 shows portions of the *tie_apps.txt* file. Each line defines one application identified by the pair $(AppID, SubID)$. 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: a 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 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 at which the classification of the session has been made. This is done 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.

4 Conclusion

In this paper we introduced a 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 to perform 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.

Acknowledgements

This work has been partially supported by he CONTENT EU Network of Excellence (IST-FP6-038423) and by the European Community's Seventh Framework Programme under Grant Agreement No. 216585 (INTERSECTION Project).

References

1. Karagiannis, T., Broido, A., Brownlee, N., Claffy, K.C., Faloutsos, M.: Is p2p dying or just hiding? In: IEEE Globecom (2004)
2. Karagiannis, T., Papagiannaki, K., Faloutsos, M.: Blinc: Multilevel traffic classification in the dark. In: ACM SIGCOMM (August 2005)
3. Auld, T., Moore, A.W., Gull, S.F.: Bayesian neural networks for internet traffic classification. IEEE Transactions on Neural Networks 18(1), 223–239 (2007)
4. Williams, N., Zander, S., Armitage, G.: A preliminary performance comparison of five machine learning algorithms for practical ip traffic flow classification. ACM SIGCOMM CCR 36(5), 7–15 (2006)
5. CoralReef, `http://www.caida.org/tools/measurement/coralreef/`
6. Paxson, V.: Bro: A system for detecting network intruders in real-time. In: Computer Networks, pp. 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 (November 2008), `http://tstat.tlc.polito.it`
11. Bonfiglio, D., Mellia, M., Meo, M., Rossi, D., Tofanelli, P.: Revealing skype traffic: when randomness plays with you. In: SIGCOMM 2007, pp. 37–48. ACM, New York (2007)
12. Salgarelli, L., Gringoli, F., Karagiannis, T.: Comparing traffic classifiers. SIGCOMM Comput. Commun. Rev. 37(3), 65–68 (2007)

13. Tcpdump and the Libpcap library (November 2008), `http://www.tcpdump.org`
14. Jacobson, V., McCanne, S.: The bsd packet filter: A new architecture for userlevel packet capture. In: Winter 1993 USENIX Conference, January 1993, pp. 259–269 (1993)
15. Li, W., Moore, A.W.: A machine learning approach for efficient traffic classification. In: IEEE MASCOTS (October 2007)
16. Bernaille, L., Teixeira, R., Salamatian, K.: Early application identification. In: ACM CoNEXT (December 2006)
17. Ptacek, T.H., Newsham, T.N.: Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report (1998)
18. Moore, A., Zuev, D., Crogan, M.: Discriminators for use in flow-based classification. Technical Report RR-05-13, Dept. of Computer Science, Queen Mary, University of London (2005)
19. Kuncheva, L.I.: Combining Pattern Classifiers: Methods and Algorithms. Wiley, Chichester (2004)
20. Moore, A., Papagiannaki, K.: Toward the accurate identification of network applications. In: Dovrolis, C. (ed.) PAM 2005. LNCS, vol. 3431, pp. 41–54. Springer, Heidelberg (2005)