A Tunnel-aware Language for Network Packet Filtering

A Tunnel-aware Language for Network Packet Filtering

Luigi Ciminiera, Marco Leogrande, Ju Liu, Fulvio Risso
Dipartimento di Automatica e Informatica
Politecnico di Torino
10129 Torino, Italy
Email: $first.$last@polito.it

Olivier Morandi
Laser S.r.l.
Via Nazionale per Donnas, 55
11026 Pont Saint Martin, AO, Italy
Email: olivier.morandi@laser-group.com

***Abstract*—While in computer networks the number of possible protocol encapsulations is growing day after day, network administrators face ever increasing difficulties in selecting accurately the traffic they need to inspect. This is mainly caused by the limited number of encapsulations supported by currently available tools and the difficulty to exactly specify which packets have to be analyzed, especially in presence of tunneled traffic. This paper presents a novel packet processing language that, besides Boolean filtering predicates, introduces special constructs for handling the more complex situations of tunneled and stacked encapsulations, giving the user a finer control over the semantics of a filtering expression. Even though this language is principally focused on packet filters, it is designed to support other advanced packet processing mechanisms such as traffic classification and field extraction.**

I. Introduction

Many networking tools based on protocol parsing (packet filters, firewalls, Intrusion Detection Systems, etc.) were developed in a time when the number of protocols was limited and the encapsulation relationships among them were rather simple.

Currently this assumption has become no longer valid. The explosion of the number of applications triggered a corresponding increase in the number of custom layer-7 protocols, each one implementing its own format to fulfill the application requirements. On the other hand, the number of possible protocol encapsulations in network traffic is growing day after day due to several reasons: the attempt to bypass application-layer constraints and, in general, to escape network restrictions (e.g., different application protocols transported in HTTP in order to bypass firewalls), or the necessity to establish Virtual Private Network sessions in many different environments and to transport unsupported traffic. Many of these solutions require the encapsulation of lower-layer protocols in other ones of the same level (as IPv6 in IPv4), or even in higher-layer ones (e.g. IP in UDP).

Although current packet filtering tools are able to support common conditional predicates (e.g, the presence of a protocol or a field with a given value), they fail when requested to select only specific types of encapsulation (e.g. IPv6 traffic encapsulated in IPv4). This is a problem for the aforementioned network-based tools, whose solution requires the addition of a module that is able to selectively inspect the network traffic before the actual processing; this strategy is, however, error-prone and adds computational overhead. Furthermore, the list of supported encapsulations is generally hardcoded in the packet filtering tools (therefore not easily expandable) and it is often limited with respect to tunneled protocols. This represents an additional limitation because the packet filter needs to be recompiled in order to extend its supported protocol set.

This paper proposes a new packet filtering language, called NetPFL *(Network Packet Filtering Language)*, which aims at overcoming the previous limitations. Among its main strengths, *(i)* it can handle complex situations of tunneled and stacked encapsulations, giving the user a finer tuned control over the semantics of a filtering expression; *(ii)* it supports several independent filters that can lead to multiple matches; *(iii)* it allows the user to choose, in an implementation-agnostic way, the action that should be performed upon receipt of each matching packet; finally, *(iv)* it is human-friendly, making it suitable for fast command-line processing. Furthermore, no protocol knowledge is embedded in NetPFL, facilitating its integration with any other language that provides protocol specification, e.g. external protocol databases that can be defined (and updated) independently from the packet filtering language. Particularly, our implementation associates the NetPFL to the NetPDL [1] language, which provides the protocol definition, enabling our implementation to support run-time updates of the protocol data set.

This paper is structured as follows. Section II presents the state of the art with regard to packet filter languages; Section III introduces the proposed language specification; Section IV evaluates our NetPFL implementation; Section V draws the conclusions.

II. Related Work

Current packet filtering solutions are based on different filtering languages that look extremely similar, as they are engineered with similar purposes in mind, but are often different with respect to the language details or the approach used to express Boolean conditions. A non-exhaustive listing of the most common packet filtering languages is depicted here.

libpcap [2] is probably the most famous packet capture (and filtering) library; it runs on top of many UNIX-based kernels (and Windows, in its WinPcap flavor [3]) and exposes to the filtering applications a set of primitives [4] that can be combined to fully express the desired capture syntax. Predicates operate on selected fields of some of the most common protocols (i.e. Ethernet, IEEE 802.11, IP, TCP, UDP and others) or on the length of the packet. **Wireshark** [5] is a popular packet sniffing and analysis application that, while relying on libpcap as primary packet filtering engine, uses its own syntax in post-processing mode. This language allows a broader set of filters to be expressed, both in terms of allowed predicates and in terms of protocol and fields supported; the official website states that, as of version 1.2.6, over 85000 protocol fields can be specified. Both libpcap and Wireshark, as many others, allow to define a filter that matches a selected number of protocols (e.g., `l2tp`), but fail when requested to express different conditions for the encapsulating protocols (e.g., `l2tp in ipv6`) and do not support matching against multiple filters. Furthermore, they do not have action capabilities and have static protocol descriptions hardwired in their code, making extensibility cumbersome.

The last problem is solved by the **NetPDL** [1] language, which describes protocol formats and encapsulation rules. However, this language is purely descriptive and it does not specify any primitive to define the actual filter, requiring another language for defining the filtering expression based on its extensible protocol description. **Binpac** [6] is similar to NetPDL since it focuses on protocol description, although its many primitives also enable the definition of generic actions. However, the language focuses on application-layer protocols, and it requires full programming, making it not suitable for fast, command-line interface commands.

While packet filtering is a fundamental part of most networking applications, often the resulting packet stream has to be further processed in order to complete the application's job. These applications often define their own language, including both filtering primitives and a specification of the actions to be applied to the resulting packet stream. Some well-known examples can be found in some popular Intrusion Detection Systems (IDS) such as **Snort** [7] or **Bro** [8], which allow only a handful of protocol fields to be inspected, focusing instead on the action to be performed or on the payload of the transport protocol. Still, primitives required to differentiate tunneling do not exist and the protocol set is hardcoded in the application, but in this case they are able to define some complex actions, such as inspect the payload, raise an alert, and more. In fact, their languages focus on the peculiar set of actions required by the application, with limited possibility to reuse (or extend) that language in case of a different application. Differently from the previous applications, IDSs are able to define multiple rules (i.e. filters) that can be active at the same time and that can lead to multiple matching; the receiving module is made aware of the filters that matched against each packet.

Similarly, also **Stream-SQL** [9] focuses on high-level actions, enabling sophisticated elaborations (e.g. grouping, counting, ordering, etc.) on a data live stream through a SQL-like syntax. However, this language does not include the traditional filtering capabilities operating on packets. Filtering primitives (i.e., the `SELECT` keyword) operate on a live stream that looks like a structured table, as in a traditional database, making this approach unfeasible for classical packet filtering. For the same reason, the number of protocols and fields identified is limited to those known by the engine that preprocesses the network traffic and creates the live stream in tabular format.

All the approaches presented above suffer of at least one of the five following problems:

1) **no tunneling support**: even if tunneling protocols are successfully recognized, multiple instances of the same protocol in the same packet cannot be treated separately; furthermore, the user cannot select precisely which encapsulations have to be considered when capturing packets;
2) **no multiple independent filters**: most of the languages do support multiple independent (and potentially overlapping) filters that can be satisfied at the same time, and do not allow to return the list of the matching filters to the application;
3) **limited actions and no extensibility**: each language aims at solving only the problem of the particular application and there is no provision of a generic action-based language that can support many applications;
4) **human-friendliness**: we want the language to be used on a command line tool, in order to be able to quickly react to changing network conditions without having to rely on complex building toolchains;
5) **hardwired protocol description**: the number of protocols recognized by each implementation can be increased only by editing the source code of the application and recompiling it.

NetPFL, as proposed in this paper, aims at solving the first four problems: appropriate (and human-friendly) primitives enable the identification and selection of each protocol in a tunnel, while keeping an high degree of flexibility. Furthermore some common actions have been defined, which can be further extended at will, and multiple independent filters are supported. With respect to the fifth problem, NetPFL is completely protocol-agnostic, as it has no *a priori* knowledge about the structure of the packets to be processed. NetPFL relies on other languages to describe protocol formats: in our implementation it has been associated with NetPDL, that, incidentally, was specifically designed to overcome exactly the problem of decoupling protocol descriptions from network tools.

III. Network Packet Filtering Language

This section presents the principles on which the NetPFL language[1] is based upon, by presenting at first an overview

[1]The complete NetPFL specification can be retrieved from http://nbee.org/download/netpfl-20100315.pdf.

of the language and then moving onto its most peculiar characteristics.

A. Operating Principles

NetPFL is defined as a rule-based language which follows a *filter-action-stream* model; this architecture seamlessly supports different applications, such as accepting a packet for a simple packet filter, extracting the actual values of a set of fields for a more advanced processing tool, and more. The basic syntax for a NetPFL rule is the following:

```
[<FilteringExpression>] [<Action>] [as
stream <StreamID>]
```

The rule is applied to every incoming packet with the following semantics:

When `FilteringExpression` *is true, perform* `Action`. *Then associate the current packet and the results of the action to the stream identified by* `StreamID`.

B. Filters

The filtering expression is an optional part of the statement and it is composed by a Boolean function, which can be either based on *(i)* the *presence* of certain protocols and fields (i.e. a filter is satisfied if the packet contains a specific header, possibly in a specific position of the header chain, or contains a field that may be optionally present in the header itself) or *(ii)* the *value* of some fields (i.e. a filter is specified as an expression based upon the value of one or more header fields). The filtering expression may be omitted; in this case the specified action will be applied to every incoming packet. Multiple conditions can be defined through the standard Boolean operators `and`, `or` and `not`.

If a packet matches a filtering condition, then the action is performed: this action could consist in simply returning the desired packet to the application; more details will be presented in Section III-C.

The filtering expression is based on tokens (protocols and protocol fields) that are not directly specified in the language: in fact, these tokens are described in a companion specification. Our implementation uses a database written in the NetPDL language, even though any kind of description language could be used for the same purpose, being even a plain text file listing all the tokens. In other words, the resolution of the protocols names and associated fields used in a NetPFL rule is transparent to the language: the only substantial requirement is that such identifiers correspond to valid protocols and fields within the protocol database in use.

C. Actions

Whenever a packet matches a filtering expression, the behavior that should be followed is described by the action component. NetPFL currently defines the following actions: *(i)* `returnpacket`, for simply accepting packets which satisfy the rules imposed by the `FilteringExpression`, *(ii)* `extractfields`, for extracting the values of an user-defined list of fields and *(iii)* `classify`, for deploying traffic classification mechanisms. `returnpacket` represents the default action, which is invoked when the `Action` keyword is missing.

While the behavior of the `returnpacket` action is self-explanatory, the other two actions may need additional details. In fact, `extractfields` allows the user to specify a list of data references and/or some additional information to be extracted and returned as metadata associated to the packet. In this way, it is possible to extract only the required information from the packet (for example the IP source address or the TCP destination port) and obtain a more compact data stream format. Therefore, the `extractfields` action is considerably suitable to be used in conjunction with a language such as the aforementioned Stream-SQL, since it can format the output stream accordingly. The `classify` action, instead, instructs the processing engine not to return the entire packet, but to use some fields as an entry for more advanced classification. For instance, `classify (tcp.sport)` will create a table that contains, for each value of the `tcp.sport` field, the number of bytes and packets associated to those values. Both filtering and classification are done in the processing engine and only the final result (i.e. the table containing classification results) will be returned to the user. The format of the returned data is implementation-dependent.

It is worthy to notice that the filtering component and the action component are two distinct but consecutive phases of the processing mechanism: the filtering phase returns a set of packets, which are then further processed in the action phase. This architecture enables an high degree of flexibility, since a specific action may be assigned to deal with each filtering condition. In addition, this modular scheme facilitates the extension of the language with more actions whenever new behaviors are required, e.g. the possibility to raise an alert when a specific condition occurs.

D. Streams

In NetPFL, the term "stream" is used to define the sequence of processed packets associated with the corresponding metadata: in other words, the concept of stream defines both the set of all the packets accepted by the filter and the set of all the tuples of fields extracted from consecutive packets. With the optional trailing component "`as stream <StreamID>`", the user can associate an identifier to each NetPFL statement. If multiple filters are being used, the user is able to distinguish which rule matched a certain packet; in addition, if more than one rule matches the same packet, the whole set of corresponding stream IDs would be returned to the user. The actual format of this mechanism is implementation-dependent: for example, it could be composed by a list of stream IDs or by a vector containing a bitmask identifying the associated streams.

Another advantage brought by the concept of stream is a more user-friendly management interface for the end user:

in fact, it should be possible to add a new stream, delete or replace an existing one, without interfering with the behavior of the other active streams. The implementation of NetPFL has to provide a number of primitives, such as `setstream()` and `deletestream()` functions, in order to add, replace or delete a stream. In general, the functions required to handle streams (i.e. to get results from multiple streams at once) are implementation-dependent.

E. Tunneling support

In order to correctly handle packets containing tunneled encapsulations, the user must be given the possibility to specify filtering conditions more restrictive than those defined in the previous section: for example, if tunneling is used, a packet may contain multiple headers belonging to the same protocol. This section will use the concept of *header instance*, which constitutes the basic building block of more complex structures that will be described in the following paragraphs.

While the simplest filtering conditions on protocols and fields, such as for example `ip` or `ip.src`, apply to any instance of the chosen protocol, the user may prefer to enforce more restrictive conditions on the filter; for this reason, the *header indexing* technique allows to specify which specific occurrence of the protocol header has to be analyzed. The syntax to be used is:

```
proto_id%n
```

where *n* is an integer number indicating the ordinal number of the occurrence of the `proto_id` protocol to be considered. For example, "`ip%2`" refers to the second instance of the `ip` protocol header and the resulting filter will match all the packets containing at least two instances of such header.

NetPFL provides some operators, called *protocol placeholders*, which can be used to create very flexible matching conditions: the *(i)* `any` keyword represents a wildcard matching any protocol defined in the protocol database in use, while the *(ii)* `layer` keyword followed by an integer `n` in the [2..5] range, matches any protocol included in the database whose "natural" layer is specified by `n`. The association of each protocol to the right level (e.g. `ip` is a layer 3 protocol) falls under the responsibility of the NetPFL implementation.

In order to describe situations where a particular header may occur a variable number of times in the packet, *repeat operators* are used; the paradigm and the syntax are borrowed from the symbols used in the regular expressions. As an example, we would use "`ip+`" if we would want to filter all the packets that contain one or more consecutive instances of the `ip` protocol, while we would use "`mpls*`" to match all the packets that contain zero or more consecutive instances of the `mpls` protocol.

F. Protocol chains

The NetPFL language includes two constructs, called *chaining operators*. These are defined by the keywords `in` and `notin`, which allow specifying the sequences of headers that a packet must contain in order to be selected.

The `in` operator allows defining a chain where the left-hand element is encapsulated within the right-hand one. Its dual is represented by the `notin` operator, which allows specifying a chain where the left-hand element is encapsulated in any header other than the one defined by the right-hand element. An example may be:

```
ip in vlan
```

where we would want to match all the packets containing an `ip` header encapsulated in a `vlan` one. Instead

```
tcp notin ip
```

would match all the packets containing a `tcp` header encapsulated in any header other than an `ip` one. A slightly more complex example is

```
tcp in any in ip
```

which, by using the `any` keyword, matches all the packets where `tcp` is encapsulated in any protocol, which however must be encapsulated in `ip`. Analogously,

```
ip in vlan+ in ethernet
```

requires `ip` to be encapsulated in at least one `vlan` header, transported within an `ethernet` frame.

G. Contexts

If we would prefer to deal with tunneling without having to define a specific protocol chain, the paradigm of *tunneling contexts* may come into help: even though contexts are less flexible than protocol chains, they are more powerful when writing some types of filters. The concept of *context* is fairly simple:

- The sequence of headers in a packet is divided in one or more consecutive contexts.
- The first context starts at the beginning of the packet. A new context starts whenever a tunnel is detected, i.e. each time the layer of a protocol is less or equal than the layer of the protocol that encapsulates it.

Figure 1 depicts an example of how a packet can be divided in various contexts: while the Ethernet header and the first IP header belong to the first context, the presence of a second IP header causes the beginning of a tunnel and, consequently, the beginning of a new context.

The main importance of contexts is that, in many cases, a set of conditions becomes interesting only when they are all verified in the same context. For example, a network administrator may want to intercept all the HTTP traffic coming from a given server: in this case, it isn't really important if such traffic is tunneled or not, as long as the source address of the IP instance directly carrying HTTP

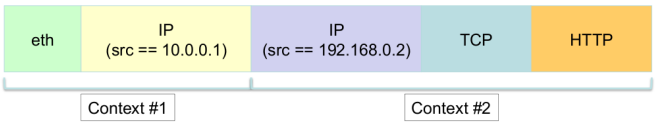

Figure 1. Context example.

is the desired one. The syntax that expresses a context is enclosed by < > symbols: the following filter matches all the packets carrying `http` traffic coming from `192.168.1.3`, either encapsulated or not:

```
<ip.src == 192.168.1.3 and http>
```

In case of a tunnel, the value of the external `ip` address does not affect the result of the filtering, because it is not part of the same context of the `http` predicate.

It is worthy to remember that some of the constructs (such as header indexing, protocol chains and contexts) that have been introduced to manage tunneled and stacked encapsulations, can be also used within an action. For instance, we can extract a field from all the IP header instances in case of a tunneled IP packet (e.g. `extractfields (ip.src)`, or selectively define the wanted instance (e.g. `extractfields (ip%2.src)`. Within the same protocol header, we can also select if we want to extract the first instance of a certain field, or the N-th instance or even all the instances of such a field. The same applies also for the `classify` action.

IV. Validation

The advantages of the NetPFL language have been validated by implementing its specification in the NetBee framework, which includes an experimental NetPFL compiler that creates run-time code for the NetVM [10] virtual machine, using the already mentioned NetPDL language as protocol database.

The architecture of the whole system is described in [11]. The NetPFL filter, coupled with the NetPDL protocol database, is given as input to a high-level compiler that generates NetIL code, a NetVM-specific assembly-like language. The generated code can be interpreted by the NetVM itself, for maximum compatibility, or compiled Just-In-Time if a back-end compiler is available for the target architecture (Intel x86, Octeon [12] and X11 [13] are currently supported)

Our experimental implementation supports most of the primitives defined in Section III, including tunneling support and header indexing. Although we did not consider performance as a main objective, we evaluated the outcome of different filters with and without tunneling support, as described in the next section.

A. Methodology

The aim of the following tests is to understand how different captures, filters and protocol descriptions potentially impact on the performance of our implementation.

The following packet captures were prepared: a reference one, composed by 47 packets, carrying data from an HTTP and a SSH session encapsulated in TCP packets; a more complex one, made of 78 packets coming from various application protocols, encapsulated in a IP-GRE-PPP-IP-TCP tunnel.

Three different NetPDL files were also written: a *minimal* database, including definitions for Ethernet, IP, TCP and UDP, without tunnel support; a *medium* one, that added GRE, PPP and tunneling encapsulations; a *complete* and very complex one, describing over 130 protocols and their full encapsulation relationships.

All the tests were performed on a Linux-based, Xeon dual-processor machine with 4 physical processing cores running at 3 GHz and provided with 4 GiB of RAM; the test machine was left otherwise unloaded and a single core was used. A benchmark script was deployed, giving the different traces and protocol databases as inputs, and the number of clock cycles needed to recognize the `tcp` protocol was measured, by using the `RDTSC` assembly instruction available on the x86 architecture. The result is the average value calculated after 1000 repetitions of the measurements, excluding the samples that are substantially different from the average value.

B. Tunneling verification

The first sequence of tests was focused on evaluating the correct behaviour of our tunneling-handling implementation: the different NetPDL databases were used in three different batches of tests to verify the ability to recognize an encapsulated protocol header. The results are provided in Table I.

NetPDL complexity	**Reference trace** (clock cycles/pkt)	**Tunneled trace** (clock cycles/pkt)
minimal	16	n/a
medium	57	94
complete	69	104

Table I
AVERAGE NUMBER OF CLOCK CYCLES NEEDED TO RECOGNIZE A PACKET CARRYING A *TCP* HEADER.

Note that these results represent only the performance of the prototypical implementation: since the main goal of this paper is to explain the characteristics of the NetPFL language, the performance estimate may be only approximate. In fact, it seems that the computational cost required to filter a tunneled protocol is linear with regard to the header length and not to the encapsulation complexity. This is a good result since it shows that, given an encapsulation ruleset, there is no increased difficulty in recognizing tunnels, other than the expected cost to process more protocols. In this example, the difference between the two columns can be explained by realizing that a normal TCP header begins at the 35th byte (note that the traces were captured on an Ethernet link), while the same header in a GRE-PPP tunnel starts at the 75th byte.

On the other hand, the results show also that the increased computational cost must be payed either if the incoming packet is tunneled or not. This is a direct consequence of the larger number of checks that should be performed to discover potential tunnels. However, it should be taken into account that this

implementation is prototypal and only the few optimizations described in [12] (compared to the thousands of algorithms available in commercial compilers) were implemented.

Finally, it is shown that the modular approach offered by an external protocol definition let the user selectively choose which tunnels have to be supported: this choice increases consistently the scalability of the system, as shown by the comparison between the second and third row in Table I.

C. Action verification

In order to verify the behaviour of the *filter-action-stream* mechanism, various tests have been performed on the actions that were available in the framework. In particular, a sequence of tests was run to compare the performance of the `extractfields` action, inserted within the NetPDL and NetPFL architecture, with the performance of a simple extraction function implemented in C. While the integrated approach used the string ``tcp extracfields(tcp.sport, tcp.dport)'' to perform both the filtering and the extraction, the latter approach used the filter ``tcp'' to match all the TCP traffic and then proceeded to extract the desired fields by manually editing the functions involved in the filtering. The results are shown in Table II.

NetPFL `extractfields` (clock cycles/pkt)	**Simple C extraction** (clock cycles/pkt)
44	47

Table II
AVERAGE NUMBER OF CLOCK CYCLES NEEDED TO EXTRACT THE COUPLE *(TCP source port, TCP destination port)* USING A MINIMAL NETPDL.

Looking at the results obtained, we may notice that the extractfields action slightly outperforms the manual field extraction: in addition, if the size of the NetPDL database increases, the integrated approach would still execute the filtering and the extraction in one single step, while the separated approach would have to analyze the protocol encapsulations twice, therefore doubling the computational effort. Moreover, the level of flexibility of the integrated approach is fairly superior, especially considering its easiness of use.

V. CONCLUSION

This paper presented NetPFL, a new packet filtering language, which can naturally support complex situations of tunneled and stacked encapsulations. While this feature is becoming more and more important, it is crucial especially in case of network security applications because of the necessity to inspect also tunneled traffic and/or to control precisely which encapsulations we are referring to.

Additionally, the *filter-action-stream* model allows to configure thoroughly the behavior of the filter, giving the user a more precise control over the dynamics of a filtering expression and extending the operations done (efficiently) in the packet filter without the necessity to deliver all the packets to the application. For example, this model supports several independent filters to be deployed on the same datastream, thus allowing multiple matching of different conditions; the user can also configure which action has to be taken whenever a filtering condition is satisfied.

Furthermore, another structural choice that enhanced the flexibility of the architecture is the separation between the filtering component and the protocol description one: in fact, while in our implementation NetPFL has been used in conjunction with NetPDL, any kind of protocol description language can be adapted to be deployed, since NetPFL does not contain in itself any knowledge regarding protocols, headers and fields.

Our tests show that an efficient implementation is possible and that the modularity offered by the combination between NetPFL and NetPDL allows a finer configuration with respect to supported tunnels.

Current NetPFL implementation is still partial and some primitives are not yet optimized; a more complete implementation is in progress.

ACKNOWLEDGMENT

The authors would like to thank Lorenzo De Carli, who took part in the early specification of the NetPFL language.

REFERENCES

[1] F. Risso, M. Baldi, NetPDL: An Extensible XML-based Language for Packet Header Description. *Computer Networks* (COMNET), Vol. 50, No. 5, Elsevier, pp. 688-706, April 2006.

[2] S. McCanne, V. Jacobson, The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the 1993 Winter USENIX Technical Conference*, San Diego, CA, Jan. 1993, USENIX.

[3] F. Risso, L. Degioanni, An Architecture for High Performance Network Analysis. In *Proceedings of the 6th IEEE Symposium on Computers and Communications (ISCC 2001)*, Hammamet (Tunisia), pp. 686-693, July 2001.

[4] The PCAP Library Man Page. Available at http://www.tcpdump.org/pcap3_man.html

[5] G. Combos, The Wireshark Network Protocol Analyzer. Available at http://www.wireshark.org/

[6] R. Pang, V. Paxson, R. Sommer, L. Peterson, Binpac: a yacc for writing application protocol parsers. In *Proceedings of the 6th ACM Internet Measurement Conference*, pp. 289-300, Rio de Janeiro, Brazil, October 2006.

[7] M Roesch, Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th Systems Administration Conference (LISA '99)*, pp. 229-238, Seattle, WA, November 1999.

[8] V. Paxson, Bro: A System for Detecting Network Intruders in Real-Time, *Computer Networks*, Vol. 31, No. 23-24, pp. 2435-2463, Elsevier, December 1999.

[9] StreamBase Systems, StreamSQL online documentation. Available at http://streambase.com/developers/docs/latest/streamsql/index.html, 2007.

[10] M.Baldi, F. Risso, Towards Effective Portability of Packet Handling Applications Across Heterogeneous Hardware Platforms, In *Proceedings of the 7th Annual International Working Conference on Active and Programmable Networks*, Sophia Antipolis, France (November 2005).

[11] O. Morandi, F. Risso, M. Baldi, A. Baldini, Enabling Flexible Packet Filtering Through Dynamic Code Generation. In *Proceedings of IEEE International Conference on Communications (ICC 2008)*, Beijing, China, pp. 5849-5856, May 2008.

[12] O. Morandi, F. Risso, S. Valenti, P. Veglia, Design and Implementation of a Framework for Creating Portable and Efficient Packet Processing Applications. In *Proceedings of the 7th ACM International Conference on Embedded Software (EMSOFT 2008)*, Atlanta, GA, pp. 237-244, October 2008.

[13] O. Morandi, F. Risso, P. Rolando, O. Hagsand, P. Ekdahl, Mapping Packet Processing Applications on a Systolic Array Network Processor. *IEEE International Workshop on High Performance Switching and Routing (HPSR 2008)*, Shanghai (China), pp. 213-220, May 2008.