Computer Networks 136 (2018) 51–67

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Enabling precise traffic filtering based on protocol encapsulation rules

Ivano Cerrato*, Fulvio Risso

Department of Control and Computer Engineering, Politecnico di Torino, Italy

ARTICLE INFO

Article history:
Received 22 June 2017
Revised 10 January 2018
Accepted 27 February 2018
Available online 28 February 2018

Keywords:
Packet filtering
Protocol encapsulations
Protocol encapsulation constraints
Construction algorithm
Augmented finite state automata
xpFSA
NetPFL

ABSTRACT

Current packet filters have a limited support for expressions based on *protocol encapsulation relationships* and some constraints are not supported at all, such as the value of the IP source address in the inner header of an IP-in-IP packet. This limitation may be critical for a wide range of packet filtering applications, as the number of possible encapsulations is steadily increasing and network operators cannot define exactly which packets they are interested in. This paper proposes a new formalism, called eXtended Finite State Automata with Predicates (*xpFSA*), that provides an efficient implementation of filtering expressions, supporting both constraints on protocol encapsulations and the composition of multiple filtering expressions. Furthermore, it defines a novel algorithm that can be used to automatically detect tunneled packets. Our algorithms are validated through a large set of tests assessing both the performance of the filtering generation process and the efficiency of the actual packet filtering code when dealing with real network packets.

1. Introduction

While protocol encapsulations were rather simple in the past (e.g., TCP/UDP in IP in Ethernet), new necessities, arising in particular from network virtualization, are rapidly increasing the complexity of the protocol stack. This impacts on the complexity of packet filters, which represent the basic building blocks for many applications such as firewalls and network monitors. In fact, while on the one hand packet filters should be able to capture all the traffic of interest (e.g., web traffic) independently from the actual encapsulations used at the lower layers (e.g., plain Ethernet or a tunnel transporting IPv6 traffic over IPv4 networks), on the other hand they should allow to finely select/filter only packets that include specific protocol encapsulations (e.g., PPP in GRE, TCP in the second IP header instance of the packet).

Traditional packet filters, which are based on the existence of some protocol and on the value of some protocol fields, do not allow such a precise selection of traffic according to the encapsulations found in packets. For example, they cannot specify the value of the IP source address in the inner header of an IP-in-IP packet.

The precise filtering of such traffic requires both a packet filtering language that allows to express conditions on the encapsulation relationships between protocols, and an efficient implementation of that language in order to cope with the speed of current networks. While the Network Packet Filtering Language (NetPFL) [1] already addresses the first point, its implementation is still partial and not optimized in case of complex protocol encapsulation rules [2].

Based on the above considerations, this paper brings the following contributions to packet filtering. First, it proposes the *eXtended Finite State Automata with Predicates (xpFSA)*, a new formalism to represent filtering expressions and that extends the pFSA (Finite State Automata with Predicates) packet filtering model [3]. Like its ancestor, xpFSA guarantees the optimal number of checks on packet fields in order to identify their possible match of the filtering expression, even in case of composition of multiple filters. In addition, it introduces counters and elementary operations that reduce the number of states of the automaton, which results in a more efficient generation of the executable code implementing the packet filter. Second, the paper defines an algorithm that transforms filtering expressions (potentially including complex protocol encapsulation constraints) into xpFSA, which completely replaces the automaton building process defined for pFSA and that cannot be used in case of complex encapsulation patterns. Third, it proposes a novel algorithm that can be used to automatically assign protocols to network layers, which is exploited to detect tunneled encapsulations.

In order to evaluate our algorithms and the filtering code generated from a xpFSA, we implemented them into the NetBee library [4]. Notably, our implementation does not require *a priori* protocols definition; in fact, it exploits a protocol database

* Corresponding author.
E-mail address: ivano.cerrato@polito.it (I. Cerrato).

https://doi.org/10.1016/j.comnet.2018.02.027
1389-1286/© 2018 Elsevier B.V. All rights reserved.

provided at run-time that can be easily extended or modified in order to recognize any new protocol and/or encapsulation, according to the properties of the NetPDL language [5]. In other words, our implementation can support both current and future protocols and encapsulations seamlessly, provided that the proper description is included in the protocol database.

This paper is structured as follows. Section 2 discusses the related works, while Section 3 summarizes the main characteristics of the NetPFL language and the pFSA packet filtering model. Section 4 presents the xpFSA model, while the algorithm to transform a NetPFL filtering expression into a xpFSA is detailed in Section 5. Section 6 shows the algorithm that automatically associates protocols to network layers. Section 7 provides an overview of the implementation; experimental results are then shown in Section 8, while Section 9 concludes the paper.

2. Related work

Despite the high number of publications on packet filters, at the best of our knowledge none of them proposes a solution able to handle filtering conditions with protocol encapsulation constraints. For example, neither libpcap [6], representing the foundation of many packet filtering tools (e.g., `tcpdump`, Wireshark), nor the Wireshark display filters [7] (which replace the basic filtering capabilities of libpcap when packets have to be shown on screen) support such filtering expressions. Instead, the Network Packet Filtering Language (NetPFL) [1] supports protocol encapsulation patterns in the language definition, but its implementation is partial and limited to traditional packet filters with simple encapsulation rules [2].

Traditional packet filters, such as the CMU/Standford Packet Filter [8], the BPF [6] and BPF+ [9], PathFinder [10] and the Dynamic Packet Filter (DPF) [11], focus more on the filtering architecture (a.k.a., virtual machine), leaving less attention to the programming abstraction. Moreover, they do not support constraints on protocol encapsulation patterns and rely on ad hoc optimizations often inspired by compiler-oriented techniques, which are then applied to the code to be executed.

To support filtering expressions including protocol encapsulation constraints, this paper proposes xpFSA, namely an extension of the pFSA packet filtering model that enables to reuse optimal composition rules and optimization techniques defined in the automata theory [12]. In fact, the idea of extending an FSA is not new when looking at the broader field of packet processing; for instance, xpFSA takes some inspiring idea from the following proposals, although none of them was designed (nor able) to satisfy our objectives, some not being able even to filter packets.

The eXtended Finite Automata (XFA) [13] formalism augments traditional FSA with a finite memory and generic executable code to manipulate this memory, which is oriented to improve efficiency of signature matching in network intrusion detection systems. Similar ideas can be found also in the Extended Finite State Automata (EFSA) [14], which extends traditional FSA with finite sets of variables in order to model fast intrusion detection and prevention systems. However, its design goals are rather different, as EFSA is used to monitor sequences of system calls, which also requires a completely different algorithm to build the automaton. Similar considerations hold also for pfsr [15], a predicate-augmented finite state recognizer that aims at simplifying the FSA used in natural language processing. Ruler [16] is a packet rewriter designed to anonimize traffic traces, which can also be used for packet filtering. It exploits a generalization of the FSA model called Tagged DFA [17] and uses variables to store the current position in the input string. FlowSifter [18] and COPY [19] extend context free grammars, regular grammars and automaton with predicates on transitions, variables and actions. Particularly, they define *Counting Regular Grammars* (CRG) [18] and *Distinguishable Counting Regular Grammars* (DCRG) [19] as extensions of regular grammars that use counters; albeit their theoretical degree of expressiveness is equivalent to our proposal, the implementation is rather different and targets a diverse use case. In fact, they aim at efficiently parsing application layer protocols (e.g., Facebook, Youtube) and extract fields of such protocols, while the goal of our work is to recognize packets satisfying constraints expressed on protocol encapsulations, with strong requirements in terms of real-time recomputation of the filtering code.

The Stateless FSA-based Packet Filter (SPAF) [20] model for packet filtering, which is the predecessor of pFSA and xpFSA, guarantees code optimality and safety, and it could be used to represent filtering expressions that include protocol encapsulation constraints. However, it is extremely slow in the automata generation phase because of the large number of generated states, and it is therefore suitable only for applications that can tolerate long filter generation time.

Finally, an early ancestor of the algorithm described in this paper has been presented in [2]; however, that algorithm does not support filters including the header indexing, the tunneling constraint and predicates on protocol fields (described in Section 3.1).

3. Background

3.1. Network Packet Filtering Language (NetPFL)

The NetPFL [1] is a declarative high-level language aimed at describing the conditions that a packet must satisfy in order to be accepted. Unlike other languages for packet filtering, NetPFL does not define any protocol header and encapsulation by itself, but it exploits definitions described externally, e.g., through the Network Packet Description Language (NetPDL) [5]. Moreover, NetPFL filtering expressions, or *header chains*, extend the traditional conditions based on the existence of some protocols and on the value of some protocol fields with conditions based on protocol encapsulation patterns, such as a specific chain of protocol headers.

This is achieved with the `in` and `notin` keywords, requiring respectively that, within a packet, the left-hand protocol is directly encapsulated into the right-hand one, or that the left-hand protocol is encapsulated in any protocol but the right-hand one. For instance, `tcp in {ip,ipv6}` matches packets having TCP directly encapsulated in IP or IPv6, while `tcp notin ip` accepts packets in which TCP is encapsulated in any protocol but IP. To define an encapsulation in which any protocol is valid, the literal `any` can be used; as an example, `tcp in any in ppp` is satisfied by packets having the TCP header encapsulated in any protocol, in turn encapsulated in PPP. Notably, the sequence of protocols specified in the filtering expression could start anywhere in the packet, therefore it could be preceded and followed by any protocol repeated an unspecified number of times.

Repetition operators describe conditions in which one or more protocols may occur a variable number of consecutive times in a certain position of the packet. In particular, “+” means one or more occurrences of the given protocol, “*” corresponds to zero or more, while “?” means zero or one. For example, the filter `ip in vlan* in ethernet` accepts the packets having IP encapsulated in zero or more consecutive VLAN headers, preceded by an Ethernet header.

More complex filters based on protocol encapsulations are available as well. For instance, `tcp.sport==80 in (ip.src!=10.0.0.1)+ in ethernet` matches packets having the TCP protocol encapsulated in a sequence of one or more consecutive IP headers, in turn encapsulated in Ethernet; furthermore, the TCP source port must be equal to 80, while