Demo: Implementing iptables using a programmable stateful data plane abstraction

Luca Petrucci[‡], Marco Bonola[‡], Salvatore Pontarelli[‡], Giuseppe Bianchi[‡], Roberto Bifulco[†]
[‡] CNIT/University of Rome Tor Vergata
[†] NEC Laboratories Europe

CCS Concepts

•Networks $\rightarrow$ Programming interfaces; Middle boxes / network appliances;

1. INTRODUCTION

Iptables is a well known Linux's user interface to control the `Netfilter` module, which is responsible for processing packets traversing the Linux's networking subsystem. In cooperation with the `conntrack` module, Netfilter supports a wide range of network functions such as: filtering, NAT, stateful firewall, load balancer, anomaly detection, etc.

Given the central role of the iptables' functions in the Linux networking subsystem, their implementation's packet forwarding performance is critical. For reference, a today's server is equipped with a couple of 10Gbps network interfaces, and 40Gbps interfaces are becoming common. Unfortunately, current general purpose systems' speed is not growing as fast as the network interfaces speed [1], therefore, providing such a packet forwarding throughput is a challenge.

In this work, we explore the feasibility of using programmable data plane abstractions to offload iptables operations from a server's CPU to a smart NIC. Given the dynamic nature of the iptables configurations and the need to support multiple applications at the same time, the selected abstraction (i) should provide the ability to perform runtime updates as well as (ii) support multiple concurrent functions.

Data plane abstractions. A Match-Action Table (MAT) abstraction, e.g., the one adopted by OpenFlow, would fulfill our requirements, since it provides runtime programmability of the forwarding tables while supporting forwarding entries belonging to different functions. Unfortunately, a typical MAT does not support the implementation of functions that require read/write operations of algorithmic state. In fact, MATs are already used by some smart NICs as a mean to support programmability, but they only provide programmable lookup operations over network packets [2].

Proposals such as OpenState[3] modify the OpenFlow's MAT abstraction to introduce support for algorithmic state read/write operations. Furthermore, recent evolutions of OpenState, namely OPP [4], add additional expressiveness to the abstraction, enabling the realization of complex algorithms. Considering that implementations of such abstractions for smart NICs are already available [5], we selected OPP as candidate abstraction for our iptables implementation.

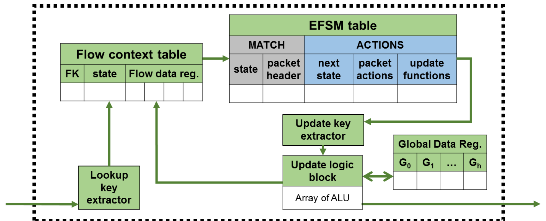


Figure 1: Simplified architecture of an OPP stage

2. MACHINE MODEL

We assume a smart NIC provides a machine model comprising a pipeline of stages, whose number depends by the particular smart NIC implementation. A stage can be either stateless, i.e., a typical OpenFlow-like MAT, or stateful. In this last case, the stage is actually an OPP stage (Fig. 2)[1].

When a packet enters an OPP stage, a *lookup key extractor* builds a key that uniquely identifies a flow context for such packet. The extractor is programmed at runtime by specifying a list of relevant header fields and packet's metadata (e.g., the TCP/UDP 4-tuple). The key is then used to extract the flow context from the *flow context table*. The context includes a state label s and an array of registers $\vec{R} = \{r_0, r_1, ..., r_{(k-1)}\}$. A new flow is assigned with a default context.

Then, the packet's header, metadata and state label s are passed to the *Extended Finite State Machine (EFSM) table*. Such table is a MAT that supports ternary matching on the just mentioned values. For each entry, a programmer can specify (i) a list of OpenFlow-like actions to be executed on the packet, (ii) the next state label s in which the flow context shall be set, and (iii) a list of instructions to update the registers $\vec{R}$. Furthermore, the update functions can also operate

[1]For brevity, here we describe a simplified version of OPP. Refer to [4] for a complete OPP presentation.

SOSR '17, April 3–4, 2017, Santa Clara, CA, USA
© 2016 ACM. ISBN 978-1-4503-4947-5/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3050220.3063773

on the global variables $\vec{G} = \{G_0, G_1, ..., G_{(h-1)}\}$. Since the variables $\vec{G}$ are global, their read and update operations happen atomically. The EFSM table describes the transitions of a state machine, hence the name.

Finally, the packet's header and metadata, the action to be applied, the update instructions and the new value of the state label are passed to the *update logic block*. Here, an array of Arithmetic and Logic Units (ALUs) performs the required update instructions to update the values stored in both the flow context registers $\vec{R}$ and global registers $\vec{G}$, using arithmetic functions. Such functions can range from simple integer sums, for instance to update the value of a register representing a packet or byte counter, to more complex ones, e.g., floating point processing, depending on the specific implementation and required performance. The result of this block is then used to update the flow context identified by the key generated by the *update key extractor*.

3. IPTABLES IMPLEMENTATION

We implemented the machine model both in software, extending OfSoftSwitch , and in hardware, using the NetFPGA SUME. Furthermore, we implemented a thin software layer that translates iptables rules into a set of entries for the data plane pipeline. The software layer is a regular OpenFlow controller extended to support OPP. In fact, we modify RYU and its OpenFlow implementation, adding new protocol messages to populate and inspect EFSM tables, flow context tables, etc.

The controller identifies the functional blocks for the implementation of an iptables rule and then writes the corresponding entries to implement them in the data plane's stages. E.g., the functional block that mimics the Linux's conntrack module uses a set of 7 EFSM table's entries in a single OPP stage. The combination of the these functional blocks finally provides the implementation of the iptables rules.

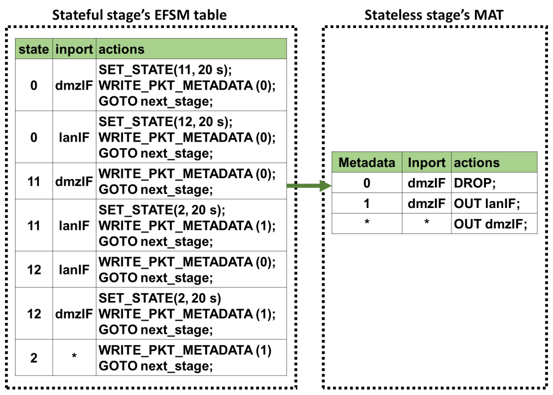
We realize 3 different network functions combining iptables' rules: a stateful firewall, a load balancer and a dynamic NAT. In total, we need 4 OPP stateful stages, plus a stateless one, to support the three functions altogether. Our current NetFPGA implementation can fit up to 6 stages and it is therefore able to fully support the mentioned use cases. For lack of space, we describe only a stateful firewall use case and redirect the interested reader to [6] for additional details.

Use case. Assume a stateful firewall allows a host in a DMZ to communicate with a host in a protected LAN only if the latter initiated the communication. In iptables, the connection tracking is realised by the following rule:

```
iptables -A FORWARD -i dmzIF -o lanIF -m state
                    --state ESTABLISHED -j ACCEPT
```

We realize the firewall using two stages, of which only one stateful. The stateful stage is configured with a bidirectional 4 tuple flow for both the update and lookup key extractors. Such a configuration provides the same flow context's key regardless of the order of the address and port fields. This stage's entries implement a connection tracking state machine, i.e., the Linux's conntrack function.

When at least a packet per direction is exchanged, the connection is established. Such information is passed to the next stage using the packet's metadata.


Stateful stage's EFSM table
Stateless stage's MAT

state	inport	actions
0	dmzIF	SET_STATE(11, 20 s); WRITE_PKT_METADATA (0); GOTO next_stage;
0	lanIF	SET_STATE(12, 20 s); WRITE_PKT_METADATA (0); GOTO next_stage;
11	dmzIF	WRITE_PKT_METADATA (0); GOTO next_stage;
11	lanIF	SET_STATE(2, 20 s); WRITE_PKT_METADATA (1); GOTO next_stage;
12	lanIF	WRITE_PKT_METADATA (0); GOTO next_stage;
12	dmzIF	SET_STATE(2, 20 s) WRITE_PKT_METADATA (1); GOTO next_stage;
2	*	WRITE_PKT_METADATA (1) GOTO next_stage;

Metadata	Inport	actions
0	dmzIF	DROP;
1	dmzIF	OUT lanIF;
*	*	OUT dmzIF;

Figure 2: Data plane's stages configuration

4. FUTURE WORK

Our current implementation supports all the iptables functions excluding those that perform operations on the packet's content[2]. When the smart NIC's data plane is used, the implemented functions can handle traffic at 40Gbps (NetFPGA SUME's line rate) with practically no server's CPU involvement. However, the hardware data plane is likely to have stringent scalability constraints e.g., on the number of entries that can be written to the stage's tables. Using the smart NIC's data plane as a cache for a software data plane running on the server's CPU could be a way to address the scalability issue. However, given the data plane statefullness, maintaining consistency between the the smart NIC's and the software data plane is far from being straightforward. Exploring these issues and possible solutions is part of our future work.

Acknowledgement: This work has been partly funded by the EU in the context of the "BEBA" (Grant Agreement: 644122) and "VirtuWind" (Grant Agreement: 671648) projects.

References

[1] N. Zilberman, P. M. Watts, et al. "Reconfigurable Network Systems and Software-Defined Networking". In: *Proceedings of the IEEE* 103.7, July 2015.

[2] E. J. Jackson, M. Walls, et al. "SoftFlow: A Middlebox Architecture for Open vSwitch". In: USENIX ATC'16.

[3] G. Bianchi, M. Bonola, et al. "OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch". In: *ACM SIGCOMM CCR* 44.2, Apr. 2014.

[4] G. Bianchi, M. Bonola, et al. "Open Packet Processor: a programmable architecture for wire speed platform-independent stateful in-network processing". In: *CoRR* abs/1605.01977, 2016.

[5] S. Pontarelli, M. Bonola, et al. "Stateful OpenFlow: Hardware proof of concept". In: HPSR '15.

[6] L. Petrucci, N. Bonelli, et al. "Towards a Stateful Forwarding Abstraction to Implement Scalable Network Functions in Software and Hardware". In: *CoRR* abs/1611.02853, 2016.

[7] R. Bifulco, J. Boite, et al. "Improving SDN with InSPired Switches". In: SOSR '16.

[2] We actually require the implementation of InSP[7] to support also packet generation use cases.