Policy Routing using Process-Level Identifiers

Oliver Michel, Eric Keller
University of Colorado Boulder
{oliver.michel, eric.keller}@colorado.edu

***Abstract*—Enforcing and routing based on network-wide policies remains a crucial challenge in the operation of large-scale enterprise and datacenter networks. As current dataplane devices solely rely on layer 2 – layer 4 identifiers to make forwarding decisions, there is no notion of the exact origin of a packet in terms of the sending user or process. In this paper we ask the question: *Can we go beyond the MAC?* That is, can fine-grained process-level information like user ids, process ids or a cryptographic hash of the sending executable be semantically used to make forwarding decisions within the network? Toward this goal, we present a system enabling such capabilities without the need for modification in applications or the operating system's networking stack. We implemented an early prototype leveraging the P4 technology for protocol-independent packet processing and forwarding in conjunction with on-board tools of the Linux operating system. We finally evaluate our system with regards to practicability and discuss the performance-behavior of our implementation.**

I. Introduction

The introduction of software-defined networking (SDN) led to a significant shift in the way today's computer networks are managed and operated [7], [10]. The SDN paradigm proposes a network architecture in which the control and forwarding planes are decoupled, offering logically centralized, flexible, and programmable network control. Using open interfaces to forwarding devices such as switches, the centralized control plane communicates to the forwarding functions.

One of the first practical instantiations of SDN was Ethane [5], an architecture allowing the enforcement of enterprise-wide security policies using a centralized policy store and forwarding devices implementing these policies in the data plane. While the research community subsequently presented more sophisticated models for enforcing policies using SDN architectures [13], [14], [16], most of these technologies are primarily network-oriented in the sense that the finest level of identifying a flow's origin is typically a combination of its MAC or IP source address together with information about the protocols (potentially incorrectly) inferred from TCP/UDP port numbers.

As these identifiers can easily be changed and in fact do not uniquely identify a communicating user or service pair, we believe that using more detailed and more accurate information about a particular flow is crucial for the task of implementing security policies in today's computer networks.

In this paper we ask: *Can we go beyond the MAC?* That is, can we identify network traffic not just based on network headers, but by all of the information that an operating system uses to distinguish running applications.

Instead of identifying a user or service solely by information available to forwarding devices within the network (*i.e.,* L2 - L4 header information), we propose an architecture where available process-level identifiers such as user ids, process ids, or host firewall marks (that all remain unused today) can be used to steer or block traffic within the network.

In this work we present PRPL[1], an architecture and system which allows making network routing and forwarding decisions based on fine-grained identifiers. This architecture allows packet processing on a *real* per-user or per-process granularity instead of mapping a user to a MAC address which can easily be changed and even more importantly is by no means a unique identifier for a user in a multi-user system.

In the process of designing this system several challenges emerged: Packets must be annotated with higher-level information at the origin host, the data plane must be able to see and understand the annotated finer-grained information about traffic, and finally packet annotations must be correct, valid and securely determined without trusting the end host. We address each of these challenges in this paper.

As a step towards the solution of the problem of annotating packets with finer-grained information and making decisions based on these annotations, we leverage the trend towards more programmable hardware in the context of software-defined networks [2]. While the most prominently used SDN control protocol OpenFlow [1] has a limited set of header-match fields [12], the community presented approaches that allow matching packets based on arbitrary bit patterns and custom header formats [3], [9], [11], [15]. In this work we are using the P4 technology [3] to match a custom flow identification header within the data plane.

II. Beyond the MAC: The need for finer grained control

Today's networking equipment has much more sophisticated responsibilities than delivering packets from a source to a destination. Middleboxes perform complex network functions such as address translation, traffic prioritization and optimization, load balancing, as well as intrusion detection to name a few. Without doing computationally expensive and performance-intensive deep packet inspection (DPI), these tasks are performed on a limited set of knowledge about a packet's origin and destination in terms of the actual user or service that injected the packet into the network or is expecting the packet.

[1]**P**olicy **R**outing using **P**rocess-**L**evel Identifiers — the paper title, non sensically pronounced purple

A. Limited information

This limited knowledge typically consists of layer 2 – layer 4 addresses (*i.e.,* TCP and UDP port numbers, IP addresses, and MAC addresses). These identifiers however do not necessarily uniquely identify a service or a user or may be forged. Furthermore, with the widespread use of virtualization in both datacenters and enterprise networks (*e.g.,* using terminals or thin clients) L2-L4 identifiers even further lose semantics. Besides the problem that multiple users may share the same address, there are various issues with identifiers at the different layers:

- TCP/UDP PORTS: Without using DPI, there is no way making sure that a packet destined for a well-known port is indeed carrying traffic of that service as any socket may bind to any transport-layer address.
- IP ADDRESSES: IP addresses may frequently change and are commonly translated into a different address space.
- MAC ADDRESSES: In virtualized environments MAC addresses can be arbitrarily set. Using them as an identifier for a user which previous approaches do, is not a safe assumption in respect to multi-user systems.
- LABEL SWITCHING IDENTIFIERS: Label switching tags (such as MPLS tags) are typically introduced within the network and are determined from standard L2-4 identifiers.

B. Fine-grained information

Compared to devices within the network, the hosts sending and receiving packets have access to a vast array of additional information about a packet's origin and its properties. We therefore investigate in this paper how such information can be used not only at the end hosts but also within the network for routing and forwarding purposes.

Modern operating systems provide advanced techniques to filter incoming and outgoing packets and apply custom actions before a packet even left the source host or reached the destination process. Linux provides such capabilities in the form of iptables and the netfilter system. Similar mechanisms exist for other operating systems, *e.g.,* the Microsoft Firewall Architecture. Using these tools, packets can be filtered using a wide array of parameters. Among them are the process id of the sending process, their owner with user and group id, the CPU which is handling a specific packet, the payload size of the packet, quota information, connection status or the control group of the sending process. Figure 1 depicts a selection of the various additional information available to the operating system in comparison to a typical forwarding model. It becomes apparent that this valuable additional information remains unused in today's network setups.

Additionally, from this knowledge we can infer further information about a process or user; examples are the number and list of open file descriptors or their CPU or memory usage. It is even possible to obtain a cryptographic hash from a processes executable in order to verify that packets originate from a specific version of the application or to prove that the process is indeed an instance of the expected application (as opposed to relying on the process name).

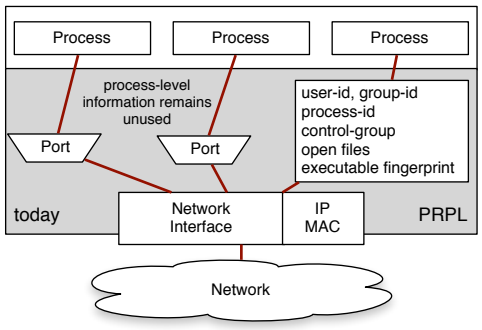

Fig. 1. Fine-grained process-level identifiers and information

C. Benefiting scenarios

We now briefly elaborate on a few example scenarios where extended information about a packet can be useful. Instead of presenting use-cases with their complete solutions, we rather like to give suggestions and share our (still limited) imaginative set of interesting applications.

1) Identifying user sessions: Enforcement of (security) policies remains a major management challenge in today's enterprise networks. Our system is a first step towards a higher level of granularity in defining and enforcing policies. While we realize that a user id or a process id is not necessarily unique within an administrative domain, the combination of multiple parameters (potentially mixed with hardware identifiers) can form a unique identifier for a user session.

2) Identifying software: By matching streams of packets to a process id, it is possible to define policy rules for certain services. Today, this is mostly performed by matching on TCP or UDP port numbers which however is not necessarily accurate as any service may bind to any port number. Today this problem would need to be solved using expensive deep packet inspection.

3) Isolating vulnerable software: Keeping software up to date and secure by applying security patches quickly remains a challenging task in large organizations. In addition to the previous example, a process id may be used to infer the version number of a particular software by either reading the version number explicitly or by taking a hash of the binary. Using PRPL, packets originating from a known to be vulnerable software may temporarily blocked or sent through additional inspection (*e.g.,* by an intrusion detection system) before a respective patch is released and installed.

4) Quality of Service: Another potentially interesting use-case is the enforcement of quality of service requirements or quota constraints in networks. Packets exceeding a certain quota on a host or originating from a bandwidth expensive service may for example be metered within the network. In the case of multi-homed hosts (*e.g.,* in datacenters) certain services may be sent over a different network using a centralized policy

that can easily be enabled or disabled in the field as opposed to per-host routing configuration.

5) Forensic analysis: Using fine-grained policies and assuming that certain statistics about packets matching a policy rule are logged, may be valuable for security audits or detailed forensic analysis of attacks or data breaches.

III. Related Work and enabling Technologies

A. Network policy enforcement

After one of the first centralized network-policy enforcement systems leveraging SDN technology Ethane [5], the community has further refined and optimized ideas of this early work. While the vast amount of work in this area focuses on middlebox-traversal policies relying on capabilities of specific middleboxes [6], [14], we would like to point out two papers that present approaches for centralized policy management enforced by various devices within a network and are as such more similar to our work. FortNox [13] introduces a security enforcement kernel on top of the NOX SDN controller, allowing for role-based authorization of OpenFlow applications and enforcement of security constraints. Merlin [16] proposes a centralized framework for network-wide enforcement of policies by automatically partitioning complex policies into components that can be placed on a variety of devices. However, while spanning the entire network, both systems rely on standard header fields.

B. More programmable hardware and protocol-independent forwarding

Traditional SDN technologies are designed around match and action patterns in the data plane based on packet header fields. The most prominent implementation of this architecture is OpenFlow [1]. The current version of OpenFlow (1.4) supports 41 different match fields [12]. Still, most hardware implementations of the OpenFlow specification support anything between 12 and approximately 20 header fields. To our knowledge there is no hardware implementation of the full OpenFlow 1.4 specification today. With this trend of slow adoption, there is (on top of technical challenges) a demand for a paradigm shift. Instead of continuously extending the OpenFlow protocol, it should be possible to match packets on arbitrary bit patterns. As this is a relatively trivial task in software, it is extremely complicated at high packet rates in hardware and at scale.

Recent research exploring *more programmable hardware* has shown that this vision of flexible packet forwarding is not too far fetched and can actually be achieved at terabit speeds using custom and specialized ASIC designs [2], [8], [11]. As these first hardware designs emerge, relatively early work on programming abstractions has recently been presented. Apart from [15], which proposes this concept in the domain of network processors, P4 [3] is the most prominent example of work in this area as it aims to cover the entire processing range from ASICs, over FPGAs, NPUs to CPUs using a common language interface. Using the P4 language, an abstract forwarding model can be described which defines how a packet is traversing a data plane device, which bit ranges need to be extracted and matched and how a packet is initially parsed and further processed. This abstract forwarding model can then be compiled using a switch-vendor specific compiler to a target-dependent version which then provides platform-compatible description of the processing flow.

IV. Architecture

The main goal of PRPL is to process streams of packets in the dataplane using the currently unused fine-grained additional information about applications that an operating system has. We define a stream as a sequence of related packets, and can be the traditional 5-tuple flow, or all packets matching a particular policy rule (or any other grouping). To do so, as shown in figure 2, the PRPL architecture has three main components.

Within end-hosts, packets are mapped to policy rules and prepared for the data plane by annotating them with a unique token (section IV-A). Secondly, packets are processed and handled within the network through data plane devices that match our custom tokens (section IV-B). Finally, a control layer consisting of a logically centralized policy controller and store responds to rule requests, configures the data plane as well as the host-internal filtering and classification mechanisms using a custom control protocol (section IV-C). This is akin to a typical reactive SDN approach.

The general architecture of our system is depicted in figure 2. We now elaborate on each of the three major building blocks.

A. Annotating packet streams

In PRPL, we introduce a new header between a packet's L2 and L3 headers to represent process-level information. With the variety of possible information a packet may be annotated with (as explained in section II-B), it is inherent that all these parameters can not be directly encoded in a custom header. This is for two reasons: The header would have to be long to fit the variety of fields and secondly the system would be harder to extend as the header format would have to be modified when introducing new parameters.

Each packet sent by an application needs to be first classified using the combination of (fine-grained) policy rule parameters and then tagged with the respective token. Upon matching a policy specification, packets are marked with a locally unique identifier. We currently do not support overlapping policies where a given packet could match multiple policy specifications and leave this issue open for future research.

Whenever a new token is required, the agent requests a new token from the token authority in the central controller in a reactive manner. A new token is then cached in the agent to save unnecessary communication to the controller and latency overhead for subsequent packets of a particular stream or flow. Finally, the PRPL header containing the token is inserted between the L2 and L3 headers of the packet and the packet is sent out to the network.

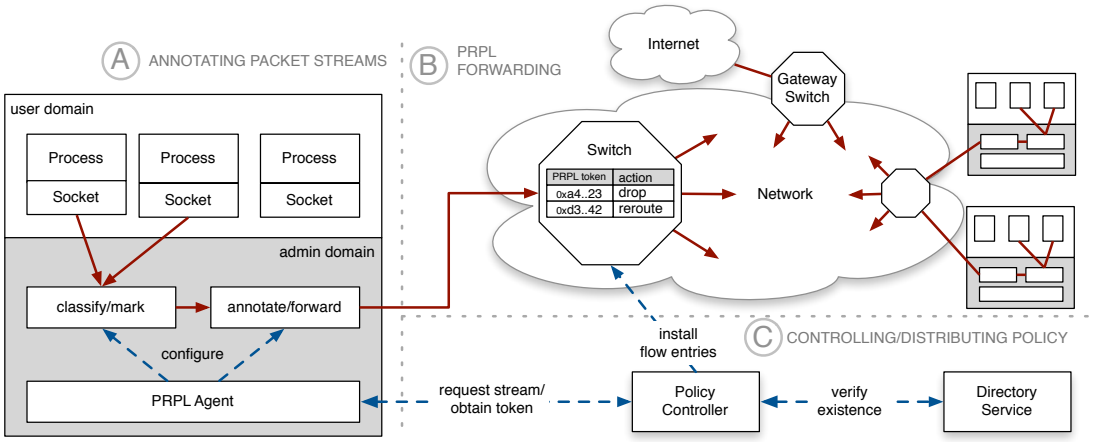

Fig. 2. System architecture

Inherently, this reactive approach comes with a latency overhead when waiting for a new token and copying the data for embedding of the additional header. We however argue that this latency overhead is negligible as the controller is located in the same administrative domain and thus most likely in the same (local area) network. We leave the discussion of implications of such an architecture in a wide-area setting for future research.

B. PRPL Forwarding

Packet forwarding in PRPL is based on the inserted PRPL header. Clearly, this header is not a standard header and thus cannot be parsed by today's rigid forwarding devices. However, with the recent trend of *more programmable hardware*, which we elaborate more on in section III-B, and new programming abstractions that make it possible to use this new flexibility, we believe that in the near future packet forwarding can be done on arbitrary bit patterns, which can be changed in the field (*i.e.,* not requiring expensive new hardware like today). A recent proposal toward this goal is P4 [3], which we leverage in this work.

In a limited way PRPL could still be deployed without protocol-independent forwarding mechanisms. This however would require misusing one of the standard header fields (*e.g.,* the IP source field) to encode custom tokens. At the expense of more state within the network, the use of network address translation and SDN techniques would still allow correct packet routing.

Using the protocol-independent forwarding techniques, we can define a parser that expects our custom header after stripping the L2 encapsulation from the packet. From this header, the token (being located at a specific bit-range) can then be extracted and used as a match field in a similar manner like traditional identifiers. The token can then trigger forwarding decisions on top of standard match/action tables with the policy token or flow token being the lookup key.

In our current design we require all packets within an administrative domain using our system to carry a token. Still, an empty (all zeroes) token may denote that no policy rule applies to the packet and the policy routing mechanisms should be bypassed. Finally, packets coming from outside of the network (*e.g.,* the Internet) must also be assigned a token in order to ensure compatibility with the network's forwarding mechanisms. As the simplest solution we envision adding a default token that designates a packet as external to each packet. This would make it possible to treat the packet as untrusted. Of course, finer-grained tokens (which however in this case would rely on standard L2-L4 header fields) can be used as well.

In an alternative (hybrid) approach, a custom EtherType could denote packets carrying the extra header. This would make it possible for annotated and unannotated packets to use the same forwarding infrastructure alongside.

C. Controlling and distributing policy

Our design uses a centralized controller managing the distribution of policies with their unique tokens, approving requests, and configuring forwarding state in the data plane.

The controller provides a management interface using which new policies may be defined and distributed to the agents running on end-hosts, which then configure the local state and intra-host classification and annotation mechanisms. The classification part is typically implemented using operating-system tools (*e.g.,* iptables in Linux). By configuring these tools through a centralized controller, we provide a control interface not only to networking functions but also for host configuration.

The controller maintains the list of distributed tokens and their respective policy rules or flow information ensuring uniqueness of tokens within the administrative domain. In a reactive manner the controller provides agents with new tokens upon an approved request for a flow belonging to a policy or a stream's first packet.

Before such a new token is given out to an agent, the controller is also responsible for setting up the required

forwarding state in the network (*i.e.,* a match and action pair) in order to actually enable enforcement of the policy.

Instead of solely relying on the policies configured within the controller, we also envision an interface to directory services like LDAP or ActiveDirectory. This would enable more secure verification of a user session by checking if a particular user actually exists and is indeed currently logged into a specific machine.

In order to implement a split between the user and administrative domains, our system currently leverages the concept of superuser privileges. That is, the per-host policy agent as well as virtual interfaces are owned by the superuser ensuring users cannot bypass the traffic filtering process. A problem however arises when users need to run services that require well-known reserved ports. However, for solving this problem Linux provides the privbind mechanism to specify policies on a per-host basis allowing a user to bind a socket to a reserved port. Akin to this, similar techniques exist to give users extended privileges on a constrained set of subsystems or commands without the need of giving full superuser privileges.

V. Prototype Implementation

We implemented a prototype of PRPL to test and evaluate enforcement of basic network policies. The host agent which uses pre-configured tokens is implemented in C, while the forwarding logic is implemented in P4 code. For testing purposes we are using the P4 behavioral model which is a software implementation of a P4 switch. We use this software-switch in conjunction with Mininet. The software switch is exposing an Apache Thrift RPC API which allows the insertion or deletion of forwarding rules at runtime. Our Mininet testbed setup consists of three end-hosts connected to the P4 switch. This setup including all custom routing and packet filtering rules using iptables runs on a Ubuntu 14.04 Linux machine.

A. Mapping packets to policy rules

For the basic set of mapping functionality we solely rely on standard tools shipping with Linux. As previously mentioned, we leverage iptables as a first layer for filtering packets based on one or a combination of the various available match parameters. A typical rule to match a user's id looks like this: `iptables -I OUTPUT -t mangle -m owner --owner-uid 1042 -j MARK set-mark 0x1`. Based on internal firewall marks, packets are then redirected to a custom routing table using rules set up through the `ip rule` command. This routing table has a single (default) route to the particular virtual interface of the policy. Figure 3 illustrates the different rules and table entries needed for this setup.

The PRPL agent is implemented in about 200 lines of C. It is mainly a proof of concept for the header and token insertion mechanisms. The program maintains a list of file descriptors of virtual interfaces and uses epoll for multiplexing. Based on the virtual interface a packet is read from, a different (currently statically defined) token is assigned to the packet before it is being sent out to the network using a raw socket. We leave the full implementation of the policy controller and custom control protocol open for future work.

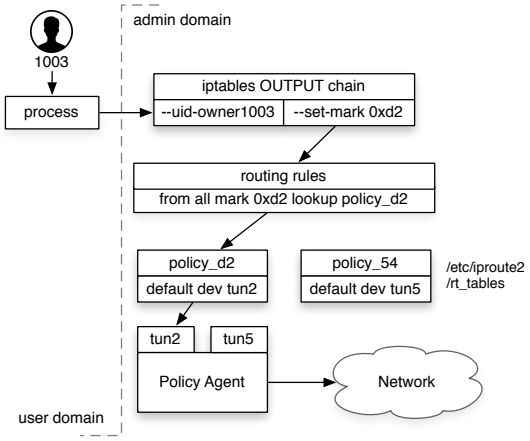

Fig. 3. Detailed packet mapping flow within a participating host

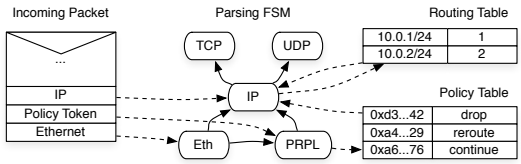

Fig. 4. Schema of data plane implementation in P4

B. Making forwarding decisions

The relatively simple data plane is implemented on top of an example static router implementation in P4. The data plane currently supports dropping a packet or rewriting the destination IP address based on a policy. As a default action, the packet is being handed to the normal routing stack. We leverage a custom P4 table to match the token and take one of the three (above described) actions based on the matching table entry. This table is the first layer of processing a packet traverses in our switch implementation. In order to do this, we had to modify the initial parsing state machine to first extract the policy token before taking any other actions. On the last hop of a packet, the PRPL header is removed before the packet is sent out and received by an end host.

VI. Evaluation

A. Performance

Since in our current implementation of PRPL every packet being sent out to the network needs to be processed in user-space and annotated with a token, the PRPL tunnel client may become a performance bottleneck. Every packet requires two additional context switches as well as the time required to copy the packet into a new buffer that contains the token. We see an increase in end-to-end transmission delay in the order of one to two milliseconds which we believe is negligible. Based on these preliminary measurements the main overhead here comes from copying the packet data into a new buffer. Our

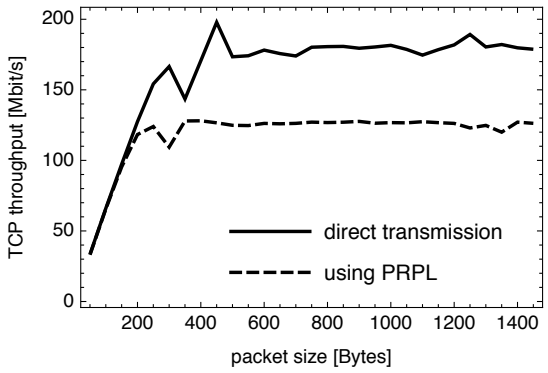

Fig. 5. TCP throughput as a function of packet size

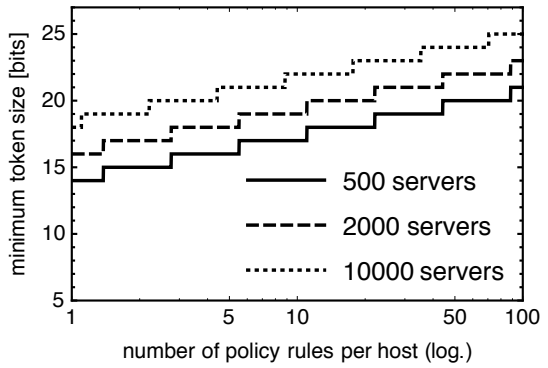

Fig. 6. Minimum required token size as a function of policy rules per host

throughput measurements using iperf show that there is little to no decrease in throughput performance for small packet sizes. However, at larger packet sizes, the maximum throughput we were able to achieve in our Mininet setup was in the order of 110 - 130 Mbit/s (see figure 5).

B. Token size

Our current token size of 32 bits allows for 2^{32} different, unique tokens. We now evaluate if this number is sufficient even for large setups. The following discussion is based on medians from the *PRV2* datacenter from Benson et.al. [4]. Assuming a setup with 48 servers per top-of-rack (ToR) switch, and a median of 1140 active flows per ToR switch at any given time, each server handles about 23.75 active flows at any given time. If we further assume that each 5-tuple flow matching a certain policy rule is annotated with a flow- and rule-unique token, the number of bits required in the header for accommodating all unique streams, is simply given by $\lceil \log_2(a \times n \times m) \rceil$, with a being the number of active flows per host, n being the number of hosts within the administrative domain, and m being the number of policy rules per host. Figure 6 shows a plot of this expression as a function of m for different network sizes with $a = 23.75$ on logarithmic scale. We see that even for large networks ($\geq$ 10000 hosts) a token size of 32 bits is absolutely sufficient. For smaller networks, even a token size of only 16 bits may be adequate.

VII. Conclusion

In this paper we introduced the concept and architecture of PRPL — Policy Routing using Process-Level Identifiers. Our contributions are threefold: We make a case for why network management can greatly benefit from fine-grained process-level information and elaborate on several example use cases. Secondly, we present a system architecture that enables packet processing and forwarding based on this process-level information without requiring any modifications of programs or the operating system's networking stack. Finally, we implemented a prototype of our system using the P4 system for protocol-independent forwarding. To our knowledge this is one of the first practical instantiations of P4 in an academic paper.

VIII. Acknowledgements

This work was supported in part by NSF NeTS award number 1320389.

References

[1] OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM CCR*, 38(2), 2008.

[2] Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference*, SIGCOMM '13. ACM, 2013.

[3] P4: Programming Protocol-independent Packet Processors. *ACM SIGCOMM CCR*, 44(3), 2014.

[4] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10. ACM, 2010.

[5] M. Casado, M. Freedman, and J. Pettit. Ethane: Taking control of the enterprise. *ACM SIGCOMM CCR*, 37(4), 2007.

[6] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul. Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, pages 543–546, 2014.

[7] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4D approach to network control and management. *ACM SIGCOMM CCR*, 35, 2005.

[8] Intel. Intel Ethernet Switch Silicon FM6000. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ethernet-switch-fm6000-sdn-paper.pdf.

[9] L. Jose, L. Yan, G. Varghese, and N. McKeown. Compiling Packet Programs to Reconfigurable Switches. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15. USENIX Association, 2015.

[10] H. Kim and N. Feamster. Improving network management with software defined networking. *Communications Magazine, IEEE*, 51(2), 2013.

[11] C. Kozanitis, J. Huber, S. Singh, and G. Varghese. Leaping Multiple Headers in a Single Bound: Wire-speed Parsing Using the Kangaroo System. In *Proceedings of the 29th Conference on Information Communications*, INFOCOM'10. IEEE Press, 2010.

[12] ONF. OpenFlow 1.4 Switch Specification, 2013.

[13] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu. A security enforcement kernel for OpenFlow networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12. ACM, 2012.

[14] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13. ACM, 2013.

[15] H. Song. Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13. ACM, 2013.

[16] R. Soul, R. K. E. G, and N. Foster. Managing the Network with Merlin. In *Proceedings of the 12th ACM Workshop on Hot Topics in Networks (HotNets-XII)*. ACM, 2013.