PNPL: Simplifying Programming for Protocol-Oblivious SDN Networks

Xiaodong Wang[a], Ye Tian[a,*], Min Zhao[a], Mingzheng Li[a], Lei Mei[a], Xinming Zhang[a]

[a] *Anhui Key Laboratory on High-Performance Computing*
School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui, 230026, China

Abstract

Protocol-Oblivious Forwarding (POF) is a groundbreaking technology that enables a protocol-independent data plane for the future Software-Defined Networking (SDN). Compared to OpenFlow and P4, POF provides more generality and flexibility in the data plane, but at a cost of additional complexity in the control plane. To overcome such complexity, in this paper we present *PNPL*, the first control plane programming framework over the POF SDN data plane. PNPL provides an easy-to-use programming paradigm that includes a *header specification language* and a set of *protocol-agnostic programming APIs*. With PNPL, a programmer can arbitrarily define network protocols, and compose network policies over the self-defined protocols with high-level abstractions. PNPL's *runtime system* takes the user program as input, automatically produces and maintains forwarding pipelines in POF switches. The pipeline efficiently parses the self-defined protocol headers and enforces the programmer's network policy. We have implemented a PNPL prototype, and experiment with existing and novel protocols and a number of network applications. We show that PNPL can effectively facilitate network innovations by simplifying programming over novel protocols, producing and maintaining forwarding pipelines of high quality; compared to P4, PNPL doesn't require a device configuration phase, and improves network performance by significantly reducing the packet parsing overhead.

Keywords: Protocol-Oblivious Forwarding (POF), Software-Defined Networking (SDN), Network programming

1. Introduction

Early Software-Defined Networking (SDN) programming languages [1][2][3][4][5][6][7][8][9] are based on OpenFlow [10][11], which supports only limited network protocols. Recent progresses such as P4 [12][13], BPFabric [14], and Protocol-Oblivious Forwarding (POF) [15][16][17] have proposed protocol-independent data planes. In this paper, we focus on POF, a groundbreaking technology that seeks to shape the data plane of the future SDN.

Similar to a computer architecture's instruction set (e.g., RISC or CISC), POF defines a concise set of protocol-independent *instructions*, and with the instructions, a POF switch can be programmed to support arbitrary protocols and a wide range of network functionalities. A POF switch doesn't have any protocol-specific knowledge, but matches packet fields with their {*offset*, *length*} tuples in forwarding rules. By completely

*Corresponding author
Email address: yetian@ustc.edu.cn (Ye Tian)

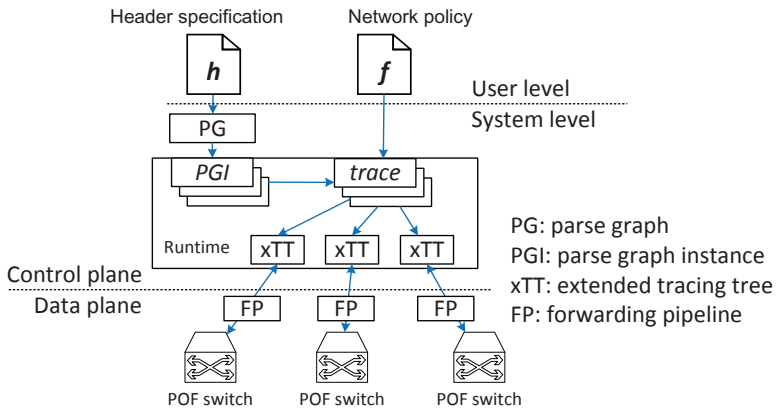


Figure 1: PNPL architecture.

decoupling control plane and data plane, POF enables the data plane infrastructures to evolve independently from specific network protocols, thus can support new protocols and services of the future Internet.

As a data plane technology, POF provides more generality and flexibility comparing with OpenFlow and P4, however, as we will see in this paper, the benefits come at a cost of additional complexity in programming, as a programmer is forced to parse packets in bits from the control plane. We believe that for high-level network programming, such packet parsing complexity should be hidden from the programmer, who only needs to focus on his network policy. In addition, as POF introduces advanced features such as dynamic multi-table pipeline and inter-table metadata, the programming language should also provide easy-to-use abstractions to exploit these features.

In this paper, we present *PNPL*, the first SDN programming framework that allows user to program high-level network policies over the POF data plane. PNPL provides two programming interfaces: a *header specification language* that allows a user to arbitrarily define network protocols with complicated header structures, and a set of *protocol-agnostic programming APIs*, with which the programmer can compose his network policy over the self-defined network protocols.

PNPL has a *runtime system* that is responsible for generating, deploying, and maintaining forwarding pipelines in POF switches. The runtime system parses the `PacketIn` packet reported by a POF switch according to the header specification, and executes the network policy program to reach a policy decision. More importantly, the runtime system traces the packet parsing and policy execution steps, and use them to handle the subsequent `PacketIn` events.

The architecture of PNPL is demonstrated in Figure 1. PNPL transforms the header specification into a static *parse graph* (*PG*) [18]. To cope with the dynamic nature of network packets and policy execution flows, when a "policy miss" `PacketIn` event occurs, that is, the policy decision for the `PacketIn` packet is not traced by the runtime system, the packet is firstly applied to PG to derive a dynamic *parse graph instance* (*PGI*), which is a subgraph of the PG and records the parsing results of the packet. Then PNPL's runtime system invokes the policy program upon the PGI, logs packet and metadata field accesses as a *trace*, and incrementally constructs an *extended tracing tree* (*xTT*). An xTT not only traces policy execution flows, but also traces packet

parsing steps as well. Finally, the runtime system produces *forwarding pipelines* (*FPs*) from the xTT, which contains multiple stages of rules for parsing network packets and enforcing the network policy, and installs them in POF switches.

Comparing with OpenFlow-based SDN programming languages (e.g., [1][2][7]), PNPL provides a protocol-independent SDN programming paradigm, which facilitates researchers to innovate with novel network protocols. PNPL is also very different from P4 [12][13], as the latter is a device configuration language for configuring individual P4 switches, while PNPL is a framework at the control plane for programming the entire POF data plane. A PNPL programmer doesn't need to configure individual POF switches explicitly, but only to describe the protocol header formats and network-wide forwarding policy, while the high-performance runtime system automatically generates pipelines for each of the individual POF switches.

We have implemented a PNPL prototype [19] and assess it with experiments. We find that the PNPL program achieves comparable performance to the equivalent programs manually implemented on mainstream OpenFlow controllers; more importantly, with PNPL, we can easily program a wide range of existing and novel protocols and network applications, and PNPL produces FPs of high qualities; finally, comparing with P4, we find that PNPL improves network performance by significantly reducing the packet parsing overhead.

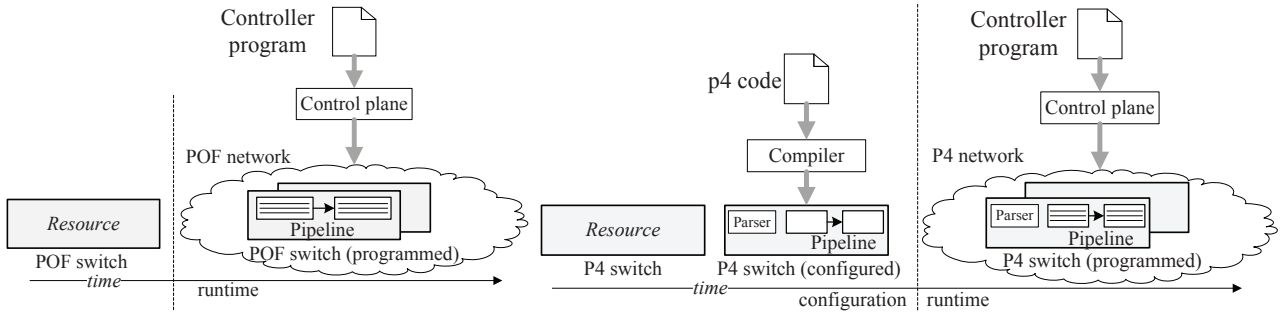
The remainder of this paper is organized as follows. Section 2 discusses the related works; we introduce the background and present our motivation in section 3; section 4 presents PNPL's programming model, and we describe its runtime system in section 5; section 6 presents and evaluates the PNPL prototype; finally, we conclude this work in section 7.

2. Related Work

Programmable protocol-independent data plane. The P4 project [12][13] proposes a language to program the data plane of programmable packet processors. P4 has two phases, in the configuration phase, a programmer specifies the headers, parsers, actions, tables and control flows with the P4 language; the P4 compiler compiles the target-independent program to a variety of target switches in an offline way. At run time, a controller uses the API (e.g., OpenFlow) generated by the compiler to populate the flow table on the configured target switch.

Although POF shares many common features with P4, they have different goals. POF seeks to provide the runtime programmability to an SDN data plane, so that the POF switches in a network can be dynamically programmed to support various network protocols and network services at run time. For serving this purpose, POF defines a unified set of forwarding instructions on general-purposed POF-compatible switches, and allow a programmer to use the instructions to program the data plane at run time. As we will see in the next section, comparing with P4, POF has a simple and general-purpose data plane, but shifts the complexity of packet parsing to the control plane. One objective of this work is to simplify such complexity.

BPFabric [14] is a recently proposed SDN data plane architecture that is platform, protocol, and language-independent. BPFabric is centered around eBPF [20], which provides a protocol and platform-independent instruction set for packet filtering. BPFabric is highly flexible, and with it, a wide range of network services such as statistic gathering and reporting, network telemetry, and anomaly detection can be delivered. BPFabric



(a) POF (b) P4

Figure 2: Comparison of POF and P4 operational environments.

has yet been implemented on hardware, and as a software approach in essence, it is unknown whether eBPF can sustain a line speed of packet processing up to hundreds of Gbps for backbone networks.

SDN programming languages. In recent years, a number of high-level SDN programming languages have been proposed. Pyretic [2][3] and Maple [1] employ an imperative paradigm that enables programmers to directly manipulate the execution flows; on the other hand, declarative SDN programming languages use many different paradigms, such as SQL-like queries, finite-state machines, and graphs, to describe network policies. Examples include Frenetic [4], Merlin [5], FatTire [6], Kinetic [7], NetEgg [8], PGA [9], etc. Some SDN languages exploit the advanced features such as multi-table pipeline, branching, and switch-level state-keeping provided by the next generation SDN switch interfaces [12][15][21]. For example, Concurrent NetCore [22] provides a language for a programmer to specify routing policies and graphs of processing tables at a same time; SNAP [23] exploits the persistent state support from switches and provides a stateful model to manage the distributed states in a centralized way.

In this work we present PNPL, the first control plane programming framework over the POF SDN data plane. PNPL differs from the previous works in two aspects. First, comparing with the protocol dependent SDN programming languages (e.g., the OpenFlow-based languages), PNPL provides a protocol-agnostic programming paradigm, which facilitates network innovations by enabling a programmer to arbitrarily define network protocols and describe the network policies upon the self-defined protocols. Second, as we will see in the next section, comparing with the P4 language, which configures offline P4 switches to support specific network protocols, PNPL provides runtime programmability, with which a programmer can deploy self-defined protocols and enforce his network policies on POF networks flexibly at run time.

3. Background and Motivation

3.1. POF Introduction

POF proposes that the future SDN's data plane elements should play a similar role as CPU in computer by providing a concise set of protocol oblivious *instructions*. The proposed POF instructions seek to cover all the network data plane capabilities, including packet and metadata editing, packet forwarding, table entry manipulation, etc. POF inherits OpenFlow's communication paradigms between control plane and data plane.

As demonstrated in Figure 2(a), a POF switch doesn't offer any functionality until it connects to the control plane, which programs the switch to provide specific network functionality. A runtime POF switch can be viewed as a pipeline containing multiple stages of tables with "match + instruction" rules. POF defines a set of controller-to-switch *commands* that enable a controller to dictate pipeline structure, allocate resources, and deploy forwarding rules in POF switches at run time.

As its name suggests, a POF switch doesn't have any protocol-specific knowledge such as packet formats, therefore it handles incoming packets very differently from existing SDN data plane technologies. Taking OpenFlow as an example, in an OpenFlow switch, an incoming packet is firstly parsed by a *frontend parser*, which knows all the packet formats the switch can handle and parses the incoming packet comprehensively. In a POF switch, however, there is no frontend parser. POF adopts *on-demand parsing*, where a packet field is parsed only when the switch is programmed to have a rule that matches the field with its offset and length.

More specifically, for each incoming packet, a POF switch keeps a *packet offset* (denoted as *p-offset*) cursor to point to the position where the match begins in the current table. The cursor can be moved forward and backward using the `MOVE_PACKET_OFFSET` instruction, with positive or negative offset value as the instruction parameter. To express a match key, POF employs an {*r-offset*, *length*} tuple, where *r-offset* indicates the relative offset from the *p-offset* cursor, and *length* is the bits that should be included in the key starting from the (*p-offset* + *r-offset*) position within the packet. For example, for an IPv4 header, suppose that the *p-offset* is pointing to the start position of the header, then the source and destination address fields can be expressed as $\{12B, 4B\}$ and $\{16B, 4B\}$ respectively.

POF supports a *dynamic multi-table pipeline*. In POF, a controller program can dynamically create a flow table and append it to the pipeline, using the `TABLE_MOD` command. For creating a new table, the control plane must specify the table's *matching fields*, which are the data pieces in packet (and its associated metadata) that are supposed to be matched in the table, with their offsets and lengths. To move a packet (together with its metadata) from one table to the next one in pipeline, POF defines the `GOTO_TABLE` instruction, and the `MOVE_PACKET_OFFSET` instruction is usually executed with `GOTO_TABLE`, to move the *p-offset* cursor to the new position within the packet where match begins in the next table. A match key of a data piece in metadata is also expressed using an {*offset*, *length*} tuple. A detailed description of the POF instructions can be found in [16], and a POF pipeline example is given in Section 3.3.

Since its introduction, POF has attracted extensive attentions from the industry and the research community. Huawei has opensourced a POF controller and software switch, and implemented NPU-based POF switches in its NE40E and NE5000E backbone routers [15]. The European Telecommunications Standards Institute (ETSI) considers POF as a candidate technology for the 5G Network Function Visualization (NFV) [24], and the International Telecommunication Union (ITU-T) considers POF as a promising direction for the IMT-2020 network [25].

3.2. Comparing with P4

P4 [12][13] is another protocol independent SDN data plane technology. Unlike POF that centers around instructions, P4 defines a language for configuring a clean-slate P4 switch. More specifically, an operator defines

protocol header formats, parser specification, match-action table specification, and control flow that indicates the table execution sequence in a P4 program, and the P4 compiler compiles the program into configurations in different target specific forms. As we can see in Figure 2(b), after the configuration, a P4 switch becomes a specialized switch, which contains a protocol-specific frontend parser, and its pipeline structure is also fixed. Note that P4 is not a control plane programming language. As one can see in Figure 2(b), after configuration, a P4 switch still needs to connect to a specific controller (such as an OpenFlow controller if the switch is configured to support OpenFlow), and populates its tables with the rules from the controller program.

P4 has been very successful in recent years by supporting a wide range of target switches, from ASIC and FPGA switches to Linux-based software ones. However, by carefully comparing POF and P4, we find that POF has its own merits. First, POF doesn't have a configuration phase, and all capabilities of a POF switch can be determined by the control plane program at run time. Second, POF allows an on-demand parsing scheme, where a packet field will be parsed in a POF switch only when it is necessary; on the other hand, in P4, the frontend parser comprehensively parses all the fields regardless of whether or not they are of interest [26]. Third, POF is more flexible by allowing a dynamic pipeline, where a flow table can be created and appended to the pipeline at run time; on the other hand, after the configuration, the pipeline structure of a P4 switch can not be changed anymore.

From the above discussion, we can see that as a data plane technology, POF provides more generality compared to OpenFlow, and is more flexible than P4. However, these benefits come at a cost of additional complexities in the control plane. In the following, we discuss these complexities and analyze the inherent challenges in programming a POF network.

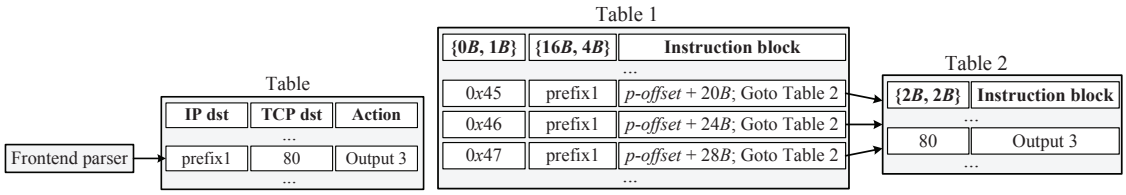
3.3. Challenges in POF Programming

Using {*offset*, *length*} tuples as match keys requires that, when deploying a forwarding rule, the programmer must assign them with correct values. However, a programmer at the control plane can not anticipate how the packets on the data plane will look like at the run time, therefore has difficulties in determining the right values for the tuples and instruction parameters when composing the forwarding rules. In the following, we discuss the reasons that cause the programming challenges.

3.3.1. Variable-length header

The first challenge is that some headers have variable formats and lengths. For example, both IPv4 and TCP headers may contain options. To parse a packet with a variable-length header, a specialized frontend parser needs to extract the "header length" field, such as IPv4's `ihl` or TCP's `offset` field, and decides the actual packet length.

However, in POF programming, a programmer will have difficulties when assigning values to the {*r-offset*, *length*} tuples, as packets may have various-length headers at run time. For example, consider a single-table scenario, where a programmer wishes to match the TCP destination port. If the TCP segment is carried within an IPv4 datagram without options in the header, the tuple is $\{36B, 2B\}$ (assuming *p-offset* $= 0$), but if the IP header has a 4-byte option, the tuple for the TCP destination port field should be $\{40B, 2B\}$, and the tuple



(a) OpenFlow/P4 table (b) POF pipeline

Figure 3: Comparison between an OpenFlow/P4 table and a POF pipeline implementing an L3+L4 rule for forwarding packets with variable-length IPv4 headers.

becomes $\{44B, 2B\}$ when the IP header contains an 8-byte option, $\cdots$. The problem is, since a programmer at the control plane can not anticipate whether an IP header contains an option, and if contained, how long the option field is, he can not decide the tuple value when composing a match key for the TCP destination port.

3.3.2. Variable header sequence

The second challenge is that packets may have variable header sequences. For example, an Ethernet header may be followed by either an IPv4 or an IPv6 header, or there would be up to two VLAN headers or up to eight MPLS headers between Ethernet and IP. For handling such a variable header sequence, a specialized frontend parser needs to extract current header's "next-layer protocol" field, such as Ethernet's `ethertype` or IPv4's `proto` field, to determine the next-layer header. However, with such a variable header sequence, it is a tedious job for a programmer to assign values to the $\{r\text{-}offset, length\}$ tuples from the control plane, as they vary with packets of different header sequences at run time. For example, to match the TCP destination port, the programmer must enumerate all the possible cases that the packet carries VLAN or MPLS headers, and has IPv4 or IPv6 as its L3 protocol.

3.3.3. Multi-table pipeline

In the previous discussion, we assume a single-table scenario, nevertheless, the challenges introduced by variable-length headers and variable header sequences also exist in multi-table pipelines.

Consider an example as in Figure 3, which presents an OpenFlow/P4 table and a POF pipeline for implementing an L3+L4 combined forwarding policy respectively. Figure 3(a) demonstrates the OpenFlow/P4 table[1], which simply matches the destination IP address and TCP port, as values of these fields have already been extracted by switch's frontend parser, and executes the output action with one single rule. However, in the POF pipeline as in Figure 3(b), the first table examines IPv4's `ihl` field at $\{0B, 1B\}$ to decide if there is an option, and before the first table moves the packet to the second table, it needs to execute the `MOVE_PACKET_OFFSET` instruction with the right parameter values to move the *p-offset* cursor to the starting position of the next-layer protocol (i.e., TCP) header, according to the match results at $\{0B, 1B\}$. As a consequence, a programmer must install multiple rules in the pipeline's first table, for the cases when the packet encountered at run time contains no IP option, a 4-byte option, an 8-byte option, $\cdots$.

[1]We assume that the P4 switch has already been configured to recognize OpenFlow-supported protocols.

Variable header sequences also introduce complexity. Consider implementing an L2+L3 forwarding policy with a POF pipeline. The L2 table needs to examine Ethernet's `ethertype` field, and employs multiple rules to move the packets to different tables for next-state processing, under the cases that the packet contains various numbers of VLAN or MPLS headers after Ethernet. Note that all the complexities arise because when programming at the control plane, the programmer can not anticipate the lengths and sequences of the headers in packets that the switches are going to handle on the data plane, therefore cannot decide the right tuple or parameter values in the forwarding rules.

From the above discussion we can see that POF has its own merits regarding the data plane generality and flexibility, but it also introduces considerable complexities to the control plane, making programming tedious and error-prone. It is desirable for a POF programming framework to manage these complexities in an efficient and automated way, so that a programmer can focus on network policies rather than on the protocol-specific parsing details.

4. Programming Model

In this section, we describe PNPL's programming model, and demonstrate how a user defines packet headers and composes a network policy over the self-defined protocols.

4.1. Header Specification

PNPL presents a *header specification language*, which provides a high-level abstraction other than {*offset*, *length*} tuples for specifying complicated packet header formats. In the following, we demonstrate an example. A formal description of the language syntax can be found in our technical report [27].

```
header Ethernet
    fields
        _mac_dst : 48;
        mac_src : 48;
        _ethertype : 16;
    next select (ethertype)
        case 0x0800 : IPv4;
        case 0x9100 : VLAN;
header VLAN
    ...
header IPv4
    fields
        _ver_ihl : 8;
        ...
        _proto : 8;
        ...
        option : *;
```

```
    length: (ver_ihl & 0x0F) << 2;
    next select (proto)
        case 0x06 : TCP;
        case 0x11 : UDP; ...
start Ethernet; // packet starts with Ethernet
```

As demonstrated in the example, a header specification consists of a sequence of `header` definitions. Each header has a `fields` statement, which declares an ordered list of fields with their names and lengths (in bits) in the header. Note that when a header contains a variable-length field, such as the `option` in IPv4, we use a "$*$" to indicate that the field's actual length should be inferred at run time. If a field is supposed to be matched in a flow table, we add "_" in front of it, and refer to the field as a *matching field* of the header.

Optionally, a header contains a `length` statement. When defining a header of variable length, the `length` statement specifies which field contains the actual length of the header, and how it is derived from the field value. For example, in IPv4, the header length should be derived using `(ver_ihl & 0x0F)` $\ll$ `2`. The `next` statement indicates the next-layer header, and if there are multiple choices, the `select` statement specifies the current header's "next-layer protocol" field, such as Ethernet's `ethertype`, and the field's possible values are enumerated in a `case` statement to indicate different next-layer headers. Finally, the `start` statement is used to specify with which header the parsing starts.

Note that PNPL does not require a header defined in the specification to exactly correspond to a protocol header. In essence, a header only specifies a sequence of fields that the programmer wishes to match in one flow table, and for each header declared, PNPL creates a flow table in the switch pipeline. For example, in `l2-learning` [28], if a programmer wishes to match the destination MAC address in one table for forwarding, and match the source MAC address in another table for address learning, then two headers, one containing `mac_dst`, and the other containing `mac_src` and `ethertype`, should be defined.

Finally, although the header specification language looks similar to the header definition part of the P4 language, they have different roles. PNPL's header specification is not used to configure a switch's frontend parser in an offline way, and as we will see in subsequent sections, it is a part of the controller program which is used to help produce forwarding pipelines for switches at run time.

4.2. APIs and Network Policy

PNPL provides a set of *protocol-agnostic programming APIs* for users to compose network policies. PNPL defines over 20 APIs. The APIs such as `read_packet` and `test_equal` read and make assertions on a packet field respectively. For supporting metadata, PNPL provides the `declare_metadata` API for declaring a data piece in metadata, the `write_metadata` API for writing value to metadata, and similar to the packet access APIs, the APIs such as `read_metadata` and `test_equal_metadata` read and make assertions on a metadata data piece. For packet parsing, PNPL defines the `apply_PG` API that applies an unparsed `PacketIn` packet to the parse graph (PG), and derives a parse graph instance (PGI); and the `search_header` API is provided to search the PGI and locate the specified header. A completed description of the APIs can be found in [27].

```
0   Path * f (Packet *pkt, struct map * env) {
1     pgi = apply_PG(pkt);
2     declare_metadata(pgi, "DstMAC", 48);
3     write_metadata(pgi, "DstMAC", "mac_dst");
4     if (search_header(pgi , "IPv4", {})) {
5       ipSrc = read_packet(pgi , "ip_src");
6       if (Lookup(LegitimateIPs , ipSrc )) {
7         if (search_header(pgi, "TCP", {"DstMAC"})) {
8           if (test_equal(pgi , "tcp_dport", 80)) {
9             macDst = read_metadata(pgi , "DstMAC");
10            return CalcPath(env, macDst); } } } }
11    return EmptyPath(env); }
```

Figure 4: Example network policy composed with the APIs.

A PNPL programmer composes a network policy in an ***f*** function. The function is invoked on a "policy miss" `PacketIn` event. When a non-empty path is returned from the ***f*** function, PNPL establishes a network flow for allowing the matched packets along the path, but when an empty path is returned, it means that the matched packets should be dropped. In Figure 4, we use an example firewall policy to illustrate how to compose a network policy with PNPL. The example policy is to allow the HTTP traffics originated from legitimate source IP addresses, while prevent all the other traffics. Note that the ***f*** function takes two arguments: *pkt* is the `PackeIn` packet, and *env* is a pointer for pointing to the environmental variables such as network topology, parse graph, etc. The program first calls `apply_PG` to parse the unparsed *pkt* with the PG and obtains a PGI (line 1); it then allocates a 6-byte data piece in metadata, and names it as "DstMAC" (line 2). The data piece will be used for storing the destination MAC address.

After packet parsing and metadata allocation, the program writes the destination MAC address to metadata at "DstMAC" (line 3), it then calls `search_header` to skip one or more headers until an IPv4 header is encountered, and examines whether the source IP is legitimate (line 4-6). Note that the variable *LegitimateIPs* in line 6 is a pre-configured dictionary for tracking all the legitimate IP addresses on the network, and Lookup() is a user-define function to lookup *ipSrc* in the *LegitimateIPs* dictionary. If the packet's source IP address is legitimate, the program skips to the TCP header (line 7), examines whether the destination port is 80, extracts the destination MAC from metadata, and calculates a path to it using the user-defined CalcPath() function (line 8-10). Otherwise, the user-defined function EmptyPath() is called to return an empty path (line 11). When a path is returned, PNPL generates pipelines that implement the network policy as described in the ***f*** function, and installs them on each switch along the path to establish the network flow.

We explain `search_header` in more details. Note that the API has three arguments, and the third argument is a list of {*offset*, *length*} tuples that describes data pieces in metadata. The metadata data pieces should be matched together with the matching fields of the specified header in the flow table that corresponds to the header. For example, in line 7, when `search_header` returns `TRUE`, a flow table corresponding to TCP will be created and appended to the pipeline, which matches the TCP destination port and the "DstMAC" tuple in metadata.

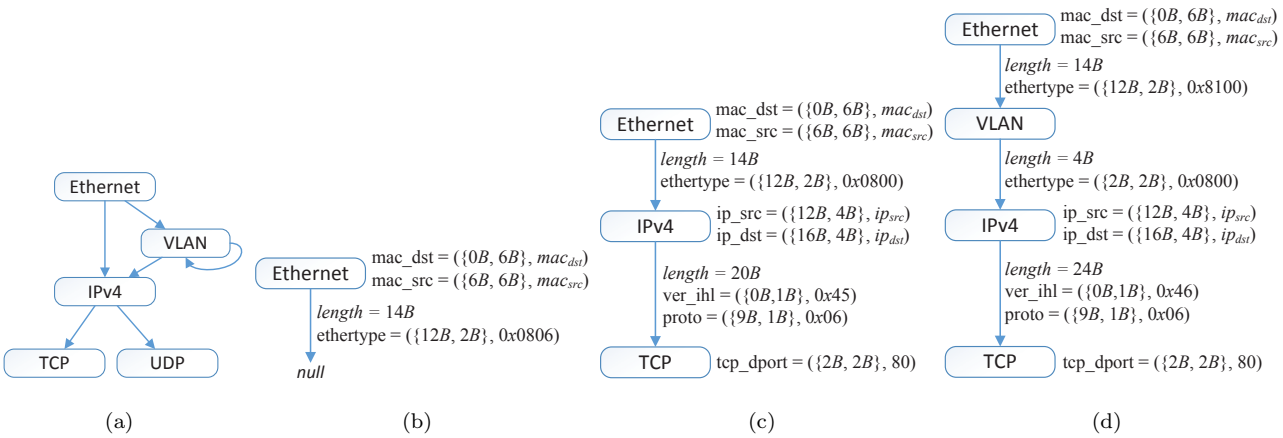


Figure 5: (a) PG derived from header specification in Section 4.1 and (b-d) three PGIs obtained by parsing three `PacketIn` packets.

From the above example we can see that comparing with existing SDN languages, programming network policies with the PNPL APIs has several advantages. First, the APIs are protocol-agnostic, while all the protocol-specific information, such as header and field names, are passed as API arguments. Second, in addition to packet data access, PNPL also provides APIs for programmer to manipulate the metadata resource, which significantly improves capability of the SDN program [28]. Finally, a programmer doesn't need to compose any code with low-level {*offset*, *length*} tuples, but use the two high-level APIs, `apply_PG` and `search_header`, to handle the packet parsing issues efficiently.

5. Runtime System

After a programmer specifies the network protocol with the header specification and describes his network policy in the ***f*** function, PNPL's runtime system takes them as input, produces and deploys pipelines that enable the user-defined protocols and network policy on the POF data plane. In this section, we describe the key components and methodologies in PNPL's runtime system.

The general idea of PNPL's runtime system is inspired by Maple [1], a programming framework over OpenFlow v1.0 networks. However, in addition to trace policy execution flows as in Maple, PNPL introduces PG and PGI for handling arbitrarily defined protocols, and extends Maple for tracing packet parsing steps and metadata manipulations. Unlike Maple that only generates rules for a single OpenFlow v1.0 table, PNPL's runtime system produces and maintains pipelines that consist of multiple stages of flow tables, and populates them with forwarding rules derived from the ***f*** function.

5.1. Parse Graph and Parse Graph Instance

Unlike OpenFlow, a packet reported by a POF switch in the `PacketIn` message to the control plane is unparsed. PNPL parses `PacketIn` packet with the user-defined header specification, and uses the parsing result to assist POF switches to parse the subsequent packets on the data plane.

After loading a header specification, PNPL derives a *parse graph* (*PG*) from it. A PG is a directed acyclic graph $G = (V, E)$, where each node $v \in V$ represents a declared header, and each directed edge $e \in E$ from

Algorithm 1: ParsePGI(G, pkt)

Input : PG $G = (V, E)$ and `PacketIn` packet pkt

Output : PGI $G_i = (V_i, E_i)$

```
1  G_i ← null ;
2  v ← G's source node;
3  while v ≠ null do
4      Create v_i from v and add v_i to G_i;
5      foreach field f in v_i do
6          v_i.[f] = (tuple, value) in pkt;
7      if v ≠ G's sink node then
8          Select next-layer header node v' ∈ G according to pkt;
9          Find e from v to v' in G;
10         Create e_i from e and add e_i to G_i;
11         e_i.length ← v_i's actual header length in pkt;
12         if v_i's "header length" field f' exists in pkt then
13             e_i.[f'] ← v_i.[f'];
14         if v_i's "next-layer protocol" field f'' exists in pkt then
15             e_i.[f''] ← v_i.[f''];
16     v ← v';
17 return G_i
```

one node to another indicates a "next-layer protocol" relation between the two corresponding headers. For example, the PG derived from the specification in Section 4.1 can be found in Figure 5(a).

PG is a static data structure. To cope with the dynamic nature of network packets in the wild, PNPL creates a data object named *parse graph instance* (*PGI*) for each "policy miss" `PacketIn` packet. More specifically, when a `PacketIn` packet is reported, it is firstly applied to the xTT to see if the policy decision on such a packet has already been traced (Section 5.3). On "policy miss", the unparsed packet is handled by the ***f*** function, and is applied to the PG in the `apply_PG` API to obtain a PGI, which is a subgraph of the PG and presents the packet parsing result, and the ***f*** function accesses the PGI in the subsequent API calls to reach a policy decision.

In Algorithm 1, we present the ParsePGI algorithm employed by `apply_PG` for parsing a `PacketIn` packet to a PGI. Basically, the algorithm traverses the PG from the source node according to the packet content, and returns the traversal path as a PGI, denoted as $G_i = (V_i, E_i)$. Inherited from the PG, each node $v_i \in V_i$ on a PGI corresponds to a header appeared in the packet, and the node keeps a dictionary of {*r-offset*, *length*} tuples and values for all the matching fields appeared in the header. For each directed edge $e_i \in E_i$ originating from a header, it records the header's actual length, and if the header has a "header length" or a "next-layer protocol" field, it also keeps the fields' tuples and values as its attributes.

In Figure 5(b-d), we demonstrate three PGIs parsed from three different `PacketIn` packets with the PG in Figure 5(a). The first PGI is the result of parsing an ARP packet. The second and third packets both have IPv4 and TCP headers, but the third packet contains a VLAN header, and has a 4-byte option in IPv4.

After a PGI is obtained, PNPL maintains a cursor to point to its current header node. Initially, the cursor points to the source node. When a `search_header` API is called in the $\boldsymbol{f}$ function, the cursor is moved along the edges until a node corresponding to the specified header is encountered, or the program reaches to the end of the graph. The API returns a `BOOL` value to indicate whether the specified header is located or not. For example, the API call `search_header(pgi, "IPv4", {})` in line 4 of the example $\boldsymbol{f}$ function in Section 4.2 returns `FALSE` with the PGI in Figure 5(b), but returns `TRUE` with the PGIs in Figure 5(c) and (d).

5.2. Trace

Each time an $\boldsymbol{f}$ function is invoked, PNPL records its execution details as a *trace*. In PNPL, a trace generated from an $\boldsymbol{f}$ invocation depends on two factors: the API execution sequence that represents the policy execution flow, and the PGI accessed by the APIs that indicates how the packet should be parsed.

To illustrate trace generating, we consider invoking the example $\boldsymbol{f}$ function in Section 4.2 to handle two `PacketIn` packets, which are parsed by `apply_PG` to the PGIs as in Figure 5(c) and (d) respectively. Table 1 lists the $\boldsymbol{f}$ function's API execution sequences on the two `PacketIn` events and the corresponding traces. Note that the two invocations have a same API execution sequence, but generate different traces labeled as "trace1" and "trace2" respectively, as each $\boldsymbol{f}$ function execution accesses a different PGI.

Table 1: Traces generated from the example $\boldsymbol{f}$ function executions with different PGIs.

API calls	**trace1**	**trace2**
write_metadata	write_metadata: $\{0B, 6B\}@m \leftarrow \{0B, 6B\}@p$	write_metadata: $\{0B, 6B\}@m \leftarrow \{0B, 6B\}@p$
search_header	read_packet: $\{12B, 2B\}@p = 0x0800$ next_table: *p-offset*$+14B$, goto IPv4	read_packet: $\{12B, 2B\}@p = 0x8100$ next_table: *p-offset*$+14B$, goto VLAN read_packet: $\{2B, 2B\}@p = 0x0800$ next_table: *p-offset*$+4B$, goto IPv4
read_packet	read_packet: $\{12B, 4B\}@p = ip_{src}$	read_packet: $\{12B, 4B\}@p = ip_{src}$
search_header	read_packet: $\{0B, 1B\}@p = 0x45$ read_packet: $\{9B, 1B\}@p = 0x06$ next_table: *p-offset*$+20B$, goto (TCP+$\{0B, 6B\}@m$)	read_packet: $\{0B, 1B\}@p = 0x46$ read_packet: $\{9B, 1B\}@p = 0x06$ next_table: *p-offset*$+24B$, goto (TCP+$\{0B, 6B\}@m$)
test_equal	test_equal: $(\{2B, 2B\}@p == 80) = true$	test_equal: $(\{2B, 2B\}@p == 80) = true$
read_metadata	read_metadata: $\{0B, 6B\}@m = mac_{dst}$	read_metadata: $\{0B, 6B\}@m = mac_{dst}$

A trace in PNPL contains execution details of three categories of APIs. The first category is the packet field manipulation and assertion APIs, such as `read_packet` and `test_equal`. For such an API, PNPL records the tuple of the relevant packet field, as well as the value or the assertion result. For example, the trace line

`read_packet:` $\{12B, 4B\}@p = ip_{src}$

generated from the API execution `read_packet(pgi, "ip_src")` in Table 1 indicates that the tuple of the source IP address field is $\{12B, 4B\}@p$, where '$@p$' means that it is a packet field rather than a data piece in metadata (which ends with '$@m$'), and the field value is ip_{src}.

The second category is the metadata manipulation and assertion APIs, such as `read_metadata` and `write_metadata`. Similar to the first category, the runtime system traces the metadata tuple and execution result, and for `read_metadata`, it also logs source of the data that is written to metadata. For example, the line

`write_metadata:` $\{0B, 6B\}@m \leftarrow \{0B, 6B\}@p$

generated from the API call `write_metadata(pgi, "DstMAC", "mac_dst")` in Table 1 indicates that the packet's destination MAC address at $\{0B, 6B\}@p$, is copied to $\{0B, 6B\}@m$ in metadata. Note that `declare_metadata` sequentially allocates space in metadata, and $\{0B, 6B\}@m$ is the first available data piece allocated.

The last category contains only one API, `search_header`. Recall that when the API is called, it searches the PGI and skips one or more headers. For each skipped header, PNPL's runtime system calls an internal function named `next_table`, which suggests that the packet should be directed to the flow table in the pipeline that corresponds to the skipped header. Moreover, if the current header has a "header length" field or a "next-layer protocol" field, PNPL inserts a `read_packet` trace to log the tuple and value of the corresponding field before `next_table`. For example, for the API execution `search_header(pgi, "IPv4", {})` on the PGI in Figure 5(c), a `read_packet` line is inserted to read the `ethertype` field before the `next_table` line as

`read_packet:` $\{12B, 2B\}@p = 0x0800$
`next_table:` *p-offset*+$14B$, `goto IPv4`

in trace1, while for the PGI in Figure 5(d), the generated trace2 becomes

`read_packet:` $\{12B, 2B\}@p = 0x8100$
`next_table:` *p-offset*+$14B$, `goto VLAN`
`read_packet:` $\{2B, 2B\}@p = 0x0800$
`next_table:` *p-offset*+$4B$, `goto IPv4`

since two headers, Ethernet and VLAN, are skipped by `search_header` in this case.

5.3. Extended Trace Tree (xTT)

PNPL uses the traces from the $\boldsymbol{f}$ function executions to incrementally construct a data structure named *extended trace tree (xTT)*, and generates forwarding pipelines from it on POF switches. Here, we first give a formal definition of xTT.

Definition 1 (xTT)**.** An extended trace tree (xTT) is a rooted tree where each node t has an attribute $type_t$, whose value is one of **L** (leaf), **V** (value), **T** (testify), **N** (next table), **WM** (write metadata), or **E** (empty), such that:

1. If $type_t = \mathbf{L}$, then t has an op_t attribute, which ranges over possible operations that can be done on the packet, such as forwarding/dropping the packet, adding/removing a header or a field of the packet, or modifying a packet field. This node represents the behavior of the $\boldsymbol{f}$ function that returns op_t without further processing the packet.
2. If $type_t = \mathbf{V}$, then t has a $tuple_t$ attribute and a $subtree_t$ attribute, where $tuple_t$ can be either a packet or a metadata tuple, and $subtree_t$ is an associative array such that $subtree_t[val]$ is a trace tree for value $val \in keys(subtree_t)$. This node represents the behavior of the $\boldsymbol{f}$ function that, if a given PGI i has a value val for $tuple_t$, i.e., $i.[tuple_t] == val$, then it continues with $subtree_t[val]$.
3. If $type_t = \mathbf{T}$, then t has a $tuple_t$ attribute and a $value_t$ attribute, where $tuple_t$ can be either a packet or a metadata tuple, and two subtree attributes $subtree_+$ and $subtree_-$. This node reflects the behavior that tests the assertion $i.[tuple_t] == value_t$ of a given PGI i, then branches to $subtree_+$ if true, and $subtree_-$ otherwise.
4. If $type_t = \mathbf{N}$, then t has an $offset_t$ attribute, a $header_t$ attribute, a $tuple\_list_t$ attribute, and a subtree attribute $subtree_t$. This node represents the behavior that moves the *p-offset* cursor with $offset_t$, forwards the packet to the next-stage flow table corresponding to $header_t$, and continues with $subtree_t$. In case that the table does not exist, it is created on demand, with all the matching fields of $header_t$, as well as the metadata tuples in $tuple\_list_t$, as the table's matching fields.
5. If $type_t = \mathbf{WM}$, then t has a dst_t attribute for a metadata tuple, a src_t attribute, which can be either a packet tuple, or a data value, and a subtree attribute $subtree_t$. This node reflects the behavior that writes the value at tuple src_t in the packet to a data piece at dst_t in the metadata, or directly write value src_t to dst_t, and continues with $subtree_t$.
6. If $type_t = \mathbf{E}$, then t has no attribute. This node represents an unknown behavior.

Note that in the above definition, the notion of **L**, **T**, **V**, and **E** nodes is inherited from Maple, but we have extended the **T** and **V** nodes to cope with the self-defined protocols and metadata accesses. We introduce the **WM** node for tracing metadata write operations, and the **N** node for producing POF's multi-table forwarding pipeline, with each table matching a header specified in the header specification. Our extension greatly enhanced the trace tree model in Maple, as the latter works on the protocol-dependent OpenFlow v1.0 data plane that have only one single flow table.

An xTT is constructed from traces in an incremental way. For building an xTT, we can directly apply Maple's approach, i.e., the AUGMENTTT algorithm in [1], as a new branch is only introduced at the **T** or **V** node. A network policy starts with an empty trace tree containing only an **E** node. After collecting a new trace, PNPL's runtime system finds the appropriate **T** or **V** node on the xTT, and augments it with the new branch from the trace.

In Figure 6, we present an xTT example. The xTT is formed by applying five `PacketIn` packets to the example $\boldsymbol{f}$ function in Section 4.2. Among the packets, three are parsed into the PGIs as in Figure 5(b)-(d), and the other two have same header structures as in Figure 5(c) and (d), but have non-80 TCP destination ports. In the figure, we label each node on the xTT with a unique ID, such as L_1, N_1, ..., and group the nodes

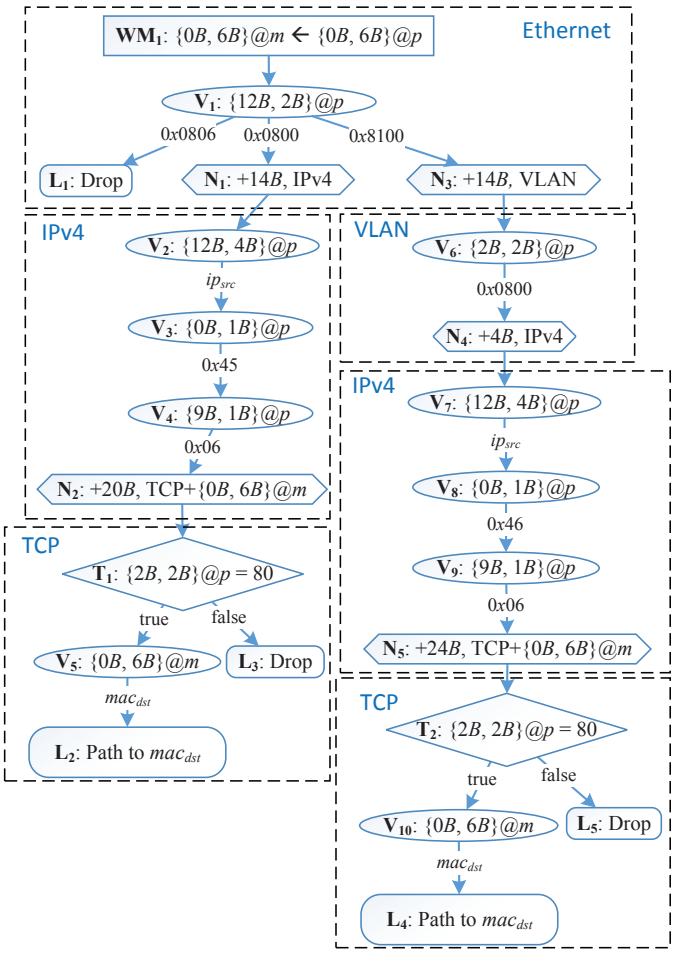


Figure 6: xTT formed by applying a number of `PacketIn` packets to the example $\boldsymbol{f}$ function in Section 4.2.

Algorithm 2: SEARCHXTT(t, pkt)

Input : xTT root node t and `PacketIn` packet pkt

Output : op_t attribute of the **L** node returned from or *null*

```
1  i=ParsePGI(G, pkt);
2  while true do
3      if type_t = E then
4          return null;
5      else if type_t = L then
6          return op_t;
7      else if type_t = V ∧ i.tuple_t ∈ keys(subtree_t) then
8          t ← subtree_t[i.tuple_t];
9      else if type_t = V ∧ i.tuple_t ∉ keys(subtree_t) then
10         return null;
11     else if type_t = T ∧ i.tuple_t = value_t then
12         t ← subtree_+;
13     else if type_t = T ∧ i.tuple_t ≠ value_t then
14         t ← subtree_−;
15     else if type_t = N ∨ type_t = WM then
16         t ← subtree_t;
```

that handle a same header in a dashed box, which we refer to as a *partial tree* (PT). For example, in Figure 6 we have one PT for Ethernet and VLAN, and two PTs for IPv4 and TCP, respectively.

From the figure we can see that, an xTT contains the nodes corresponding to the API calls in the $\boldsymbol{f}$ function, such as V_2 and V_5 for reading source IP address and metadata. The xTT also contains the nodes that match the "header length" and "next-layer protocol" fields for packet parsing, such as V_1 for reading the `ethernet` field in the Ethernet PT, and V_3 and V_4 for reading `ver_ihl` and `proto` in the IPv4 PTs, as accesses to these fields have been recorded in the trace (see Table 1).

When a `PacketIn` packet is reported to the control plane, PNPL searches it in the constructed xTT, using the SearchxTT algorithm as presented in Algorithm 2, which simply traverses the tree from the root node with a deep-first manner. If the xTT has already traced all the steps of applying user's network policy upon the packet, searching the xTT returns from an **L** node that contains the policy decision on the packet in its op_t attribute; otherwise, the algorithm returns *null*, and invokes the $\boldsymbol{f}$ function to handle the untraced `PacketIn` packet. For example, when searching the xTT in Figure 6 with packets corresponding to the PGIs as in Figure 5(b)-(d), the xTT reads the value at $\{12B, 2B\}@p$, which is `ethertype`, on node V_1, and reaches to L_1, L_2, or L_4 eventually; however, when a packet with a new `ethertype` value is encountered, searching the xTT returns *null*, and PNPL invokes the $\boldsymbol{f}$ function, which augments the xTT for including the policy decision on the packets with the new `ethertype`.

Overall, we conclude that an xTT represents all the packet parsing steps, execution flows, and corresponding policy decisions that have been made by the $\boldsymbol{f}$ function. Formally, for the xTT constructed, we have:

Theorem 1 (xTT correctness)**.** Let t be the result of augmenting an empty tree with the traces formed by applying the $\boldsymbol{f}$ function to `PacketIn` packets $pkt_1, \cdots, pkt_n$, then for $\forall pkt \in \{pkt_1, \cdots, pkt_n\}$, SearchxTT$(t, pkt)$ should return from an **L** node t_l, such that $t_l.op_t = \boldsymbol{f}(pkt)$.

Proof. We prove the theorem by contradiction. Assume that there exists a `PacketIn` packet $pkt_i \in \{pkt_1, \cdots, pkt_n\}$, and SearchxTT$(t, pkt_i)$ either returns *null* or returns from an **L** node t_l such that $t_l.opt_t \neq \boldsymbol{f}(pkt_i)$. For the former case, if we invoke the $\boldsymbol{f}$ function on pkt_i, and augment t from the generated trace, we can obtain a new xTT t', such that SearchxTT(t', pkt_i) returns from an **L** node t'_l that traces the policy decision on pkt_i, i.e., $t'_l.op_{t'} = \boldsymbol{f}(pkt_i)$. However, since t is the result of augmenting an empty tree with the traces formed by applying the $\boldsymbol{f}$ function to $\{pkt_1, \cdots, pkt_n\}$, which includes pkt_i, then we have $t = t'$, which contradicts our initial assumption that SearchxTT$(t, pkt) = null$.

For the latter case that SearchxTT(t, pkt_i) reaches an **L** node t_l with $t_l.op_t \neq \boldsymbol{f}(pkt_i)$,. Suppose that t_l is augmented by packet pkt_j, i.e., $t_l.op_t = \boldsymbol{f}(pkt_j)$. Since SearchxTT takes different branches only at the **T** and **V** nodes based on the tuple values examined at these nodes, we can see that pkt_i and pkt_j should have the same tuple values accessed and testified by the $\boldsymbol{f}$ function, thus reach to a same policy decision. In other words, $\boldsymbol{f}(pkt_i) = \boldsymbol{f}(pkt_j)$, which contradicts our initial assumption that $t_l.op_t \neq \boldsymbol{f}(pkt_i)$. □

Algorithm 3: BUILDFP(t)

Input : xTT t, PG G

Output : FP

```
1  Algorithm BUILDFP(t)
2      h ← G's source node header;
3      FTtables[h] =TABLE_MOD(h.match_fields);
4      FTtables[h].p ← 0;                                          /* priority */
5      BUILD(t, h, any, null);
6      return;
7  Procedure EMITRULE(h, m, op)
8      FLOW_MOD(FTtables[h], FTtables[h].p, m, op);
9      FTtables[h].p ← FTtables[h].p + 1;
10 Procedure BUILD(t, h, m, op)
11     if type_t = L then
12         EMITRULE(h, m, op ⋄ op_t);
13     else if type_t = WM then
14         op_t = op ⋄ SET_FIELD(dst_t, src_t);
15         BUILD(subtree_t, h, m, op_t);
16     else if type_t = V then
17         for val ∈ keys(subtree_t) do
18             m_t = m ∧ (data@tuple_t == val);
19             BUILD(subtree_t[val], h, m_t, op) ;
20     else if type_t = T then
21         BUILD(subtree_−, h, m, op);
22         m_t = m ∧ (data@tuple_t == value_t);
23         EMITRULE(h, m_t, OUTPUT(Controller));
24         BUILD(subtree_+, h, m_t, op);
25     else if type_t = N then
26         if FTtables[header_t] doesn't exist then
27             FTtables[header_t] =TABLE_MOD(tuple_list_t ∪ header_t.match_fields);
28             FTtables[header_t].p ← 0;
29         op_t = op ⋄ MOVE_PACKET_OFFSET(offset_t) ⋄ GOTO_TABLE(FTtables[header_t]);
30         EMITRULE(h, m, op_t);
31         BUILD(subtree_t, header_t, any, null);
```

5.4. Forwarding Pipeline Generating

In each POF switch along the forwarding path, PNPL produces and maintains a *forwarding pipeline* (*FP*) generated from the xTT. In general, an FP consists of multiple *flow tables* (*FTs*), each can be viewed as a collection of (*priority, match, operation*) rules, where *priority* is a numerical priority, *match* is a combination of one or more {*tuple*, *value*} pairs, and *operation* is a concatenation of one or more instructions executed by the POF switch on the matched packets.

Unlike P4, which determines a fixed pipeline structure during the configuration phase, PNPL dynamically creates FTs and appends them to the FP at run time. As previously introduced, for enabling a dynamic pipeline, the POF introduces the `TABLE_MOD` command for the control plane to create a logical FT on a specific POF switch, and provides the `GOTO_TABLE` instruction to direct a packet to a specific FT for the next-stage processing.

We present the BuildFP algorithm for generating the FP from the xTT in Algorithm 3. BuildFP traverses the xTT from the root and, during the traversal, the algorithm outputs two types of controller-to-switch commands: The first is the `TABLE_MOD` command that instructs the switch to create a new FT, using the matching fields of the specified header and the supplied metadata tuples as the table's matching fields. The second is the `FLOW_MOD` command for deploying a {*priority*, *match*, *operation*} rule to a given FT.

The algorithm maintains an associative array $FTtables$ to map a protocol header to an FT in the pipeline. The main part of the algorithm is a recursive Build procedure. Build takes four parameters, namely t, h, m, and op, in which t is the xTT node that the algorithm is currently visiting, h is the header such that $FTtables[h]$ is the FT that the algorithm currently writes rules to, m and op are the match conditions and the operations on the packet respectively.

Initially, the algorithm creates the first FT for the header corresponding to the source node of the PG (line 2-3), and Build starts to traverse the xTT from the root node, with $m = any$ and $op = null$, indicating to match any packets and have no operations on the matched packets (line 5).

Build processes a node t according to its type. For an **L** node, Build concatenates the passed op with the node's op_t attribute, and emits a rule with the passed match m and the concatenated operation $op \diamond op_t$ (line 12). Note that by letting $op = op_1 \diamond op_2$, a POF switch executes the instructions in op_1 and op_2 sequentially. For a **WM** node, Build concatenates the passed op with a `SET_FIELD` instruction, which writes data at src_t to dst_t in metadata, and proceeds with $subtree_t$ (line 14-15).

For a node of type **V**, Build combines the passed m with the node's match condition, i.e. $m \wedge (data@tuple_t == val)$, and proceeds with $subtree_t[val]$ (line 18-19). Note that by $m = m_1 \wedge m_2$, we mean the intersection of the two match conditions. For a **T** node, the algorithm emits a *barrier rule* to separate the rules generated from the positive branch and the negative one as in Maple (line 23).

Finally, for an **N** node, which is responsible for constructing the FP, one naive way is to create a new FT on each **N** node. However, such an approach will create many FTs for the same header, leading to a flow table explosion. In our approach, a `TABLE_MOD` command is issued to create $FTtables[header_t]$ for the next header $header_t$ only when such a table does not exist (line 26-28), thus ensures one FT for each header in the FP. For the

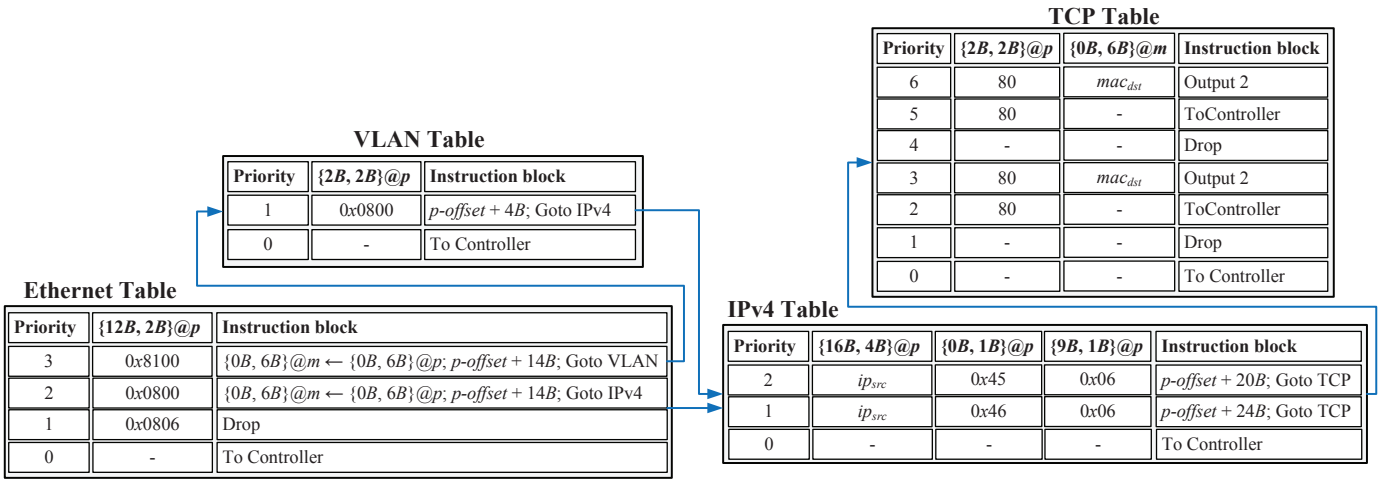


Figure 7: FP generated from the xTT in Figure 6 by BuildFP.

current table $FTtables[h]$, Build concatenates op with the `MOVE_PACKET_OFFSET` and `GOTO_TABLE` instructions to move the $p - offset$ cursor, and direct packets to $FTtables[header_t]$ (line 29-30). Build proceeds with $subtree_t$ on the next table $FTtables[header_t]$, and initializes with $m = any$ and $op = null$ (line 31). Note that inherited from Maple, in each FT, the rule priority is incremented by one each time a new rule is installed, as by doing so, each FT rule has a priority identical to the order in which the rule is added to the FT, so that a newly installed rule can overturn an old one if there is a conflict [1].

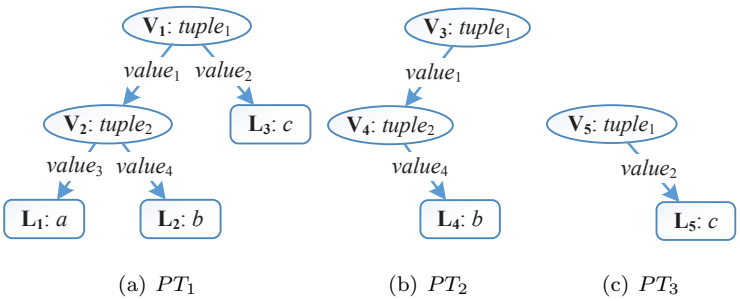
Figure 7 presents the FP generated by applying BuildFP on the xTT in Figure 6. Note that when created, each FT contains a default rule of the lowest priority to report any packet to the control plane. From the figure we can see that BuildFP constructs an FP containing multiple FTs, with each corresponding to a header, and directing packets to appropriate next-stage FTs according to their header sequences.

From the algorithm, we can see that a rule emitted by a node on an xTT only accumulates the match conditions and concatenates the instructions passed from the ancestor nodes in the same PT. In other words, modifying a PT only influences its associated FT without changing other FTs. Another observation is that the FP generated by BuildFP parses the packets incrementally, with each FT matching only the useful fields of its corresponding header on demand, thus reducing the parsing overhead comparing with the comprehensive parsing performed by the frontend parser.

Finally, for the FP generated, we have:

Theorem 2 (FP correctness)**.** *The generated FP encodes the same packet parsing and network policy functionality as in the xTT.*

Proof. For any packet, say pkt, suppose it is handled by a sequence of rules R in the FP. Now if we search the xTT with pkt using the SearchxTT algorithm, the packet will traverse a path P on the xTT and return from an **L** node t_l. By comparing the BuildFP algorithm with the SearchxTT algorithm, we can see that they have the same deep-first node traversal structure, therefore the rules in sequence R for processing pkt is sequentially generated exactly by the nodes on path P. Since the BuildFP algorithm emits rules with



(a) PT_1 (b) PT_2 (c) PT_3

Figure 8: Redundant PTs.

forwarding operations on pkt only on **L** node, and the other nodes either accumulate match conditions and operations, pass them to the downstream nodes, or emit rules for going to another table within the FP, and from Theorem 1, we know that $\boldsymbol{f}(pkt) = t_l.op_t$, therefore the rule sequence R correctly handles pkt as the $\boldsymbol{f}$ function. □

5.5. Optimization

Rule optimization is critical for SDN, as the resources for storing and executing rules in a commodity SDN switch are limited. In this subsection, we seek to optimize the rules produced by Algorithm 3.

Since PNPL traces packet parsing steps in xTT, it is possible that a same policy decision may be traced by multiple branches on the xTT, with only the difference on how the packets should be parsed. Moreover, the branches tracing a same policy decision may have identical structures in the partial trees (PTs), thus produce same rules to the corresponding FTs, resulting in rule redundancy. Formally, suppose an $\boldsymbol{f}$ function makes policy decisions based on packet fields f_1, f_2, $\cdots$, f_n that belongs to headers h_1, h_2, $\cdots$, h_k. If two `PacketIn` packets pkt_1 and pkt_2 have identical values on f_1, f_2, $\cdots$, f_n, then the $\boldsymbol{f}$ function should have a same policy decision on them. Now suppose pkt_1 has at least one header, say h_a, that pkt_2 doesn't have, that is, $h_a \notin \{h_1, h_2, \cdots, h_k\}$, then the trace generated from the $\boldsymbol{f}$ function on pkt_1 will contain the lines for parsing and skipping h_a, while the trace generated on pkt_2 will not have such lines. As a consequence, the two traces augment the xTT with two branches that separate at the "next-layer protocol" field of the header preceding to h_a, but they may have identical structures in the PTs corresponding to the headers subsequent to h_a in their branches. For example, in Figure 6, the two branches ending at L_2 and L_4 encode a same policy decision (that is, planning a path for a TCP flow to port 80 from a legitimate source), they have identical TCP PTs, but separate at the V_1 node, as one path is augmented by a `PacketIn` packet that contains a VLAN header, while the other doesn't. From Figure 7, we can see that Algorithm 3 produces rules that have identical contents in the TCP table (i.e., rule 1 and 4, rule 2 and 5, rule 3 and 6) from the two branches, and obviously, half of the rules are redundant.

We propose the following methodology for eliminating such redundancy. Each time the xTT is augmented, we compare each pair of the PTs, say PT_1 and PT_2, that correspond to a same header, and decide whether they produce redundant rules. More specifically, if all the branches to the **L** and **N** nodes within PT_2 can be found in PT_1, we say that PT_2 is redundant with PT_1. When generating a FP, if a rule-emitting node t belongs

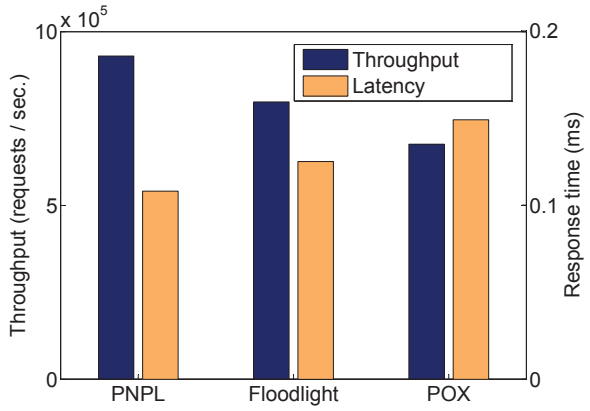


Figure 9: Performance comparison of PNPL prototype, Floodlight, and POX.

to a redundant PT, it is skipped without emitting any rule.

We use an example to show the effectiveness of our proposed methodology. Consider three PTs on a same xTT as in Figure 8. After executing Algorithm 3, we have a set of rules generated from the three PTs as follows:

`5;` $(tuple_1 : value_1)$ `&` $(tuple_2 : value_3)$`;` a
`4;` $(tuple_1 : value_1)$ `&` $(tuple_2 : value_4)$`;` b
`3;` $(tuple_1 : value_2)$`;` c
`2;` $(tuple_1 : value_1)$ `&` $(tuple_2 : value_4)$`;` b
`1;` $(tuple_1 : value_2)$`;` c

However, by examining the PTs, we can see that PT_2 and PT_3 are redundant with PT_1, thus they can be skipped without emitting any rules. After the optimization, we have only three rules emitted from PT_1 as follows:

`3;` $(tuple_1 : value_1)$ `&` $(tuple_2 : value_3)$`;` a
`2;` $(tuple_1 : value_1)$ `&` $(tuple_2 : value_4)$`;` b
`1;` $(tuple_1 : value_2)$`;` c

As for the xTT in Figure 6, after identifying the redundant PT for TCP, we shall have only three rules written to the TCP table.

6. Evaluation

6.1. Prototype and Performance

We have implemented a PNPL prototype with 15,000 C/C++ LOC [19], and run on a server with Intel Xeon E5 2.2 GHz CPU and 64G RAM. We evaluate performance of the PNPL prototype in handling `PacketIn` packets. In the first experiment, we employ a variant of Cbench [29], which is a switch emulator for benchmarking SDN controllers, to emulate `PacketIn` packets from 8 switches. We use `l2-learning` as the policy program. We measure the throughput and latency of handling `PacketIn` packets by the prototype with Cbench, and

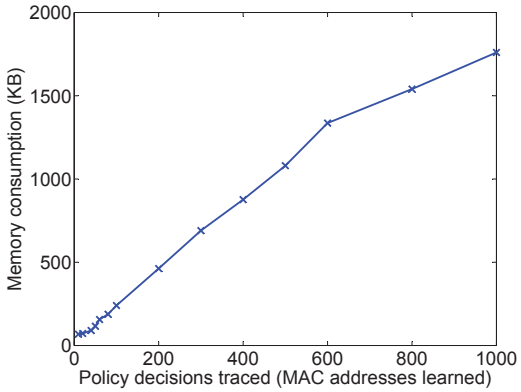


Figure 10: Memory consumption of the PNPL prototype when tracing various number of policy decisions by `l2-learning`.

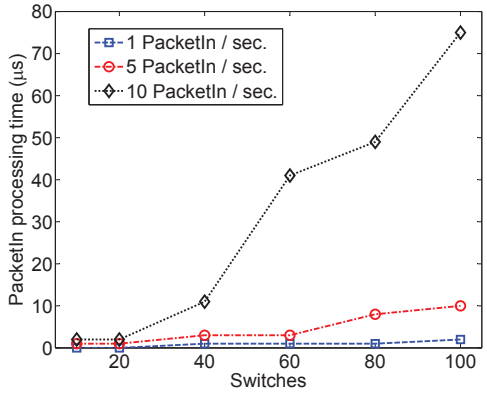


Figure 11: `PacketIn` processing time under various workloads.

compare with the mainstream OpenFlow controllers of Floodlight [30] and POX [31]. Figure 9 presents the results. From the figure, we can see that our prototype has a decent performance, which is slightly better than POX and FloodLight. There are two reasons for PNPL's better performance: First, PNPL maintains an xTT that traces all the previous policy decisions in memory, so when a `PacketIn` packet is reported from a switch, PNPL handles it with the xTT without invoking the user program; on POX and FloodLight, however, the policy programs are explicitly invoked to handle the `PacketIn` packets, and they are executed slower compared to directly searching the xTT in memory. Second, our PNPL prototype is implemented with C/C++, which is generally faster than Python and Java, with which POX and FloodLight are implemented respectively.

Since PNPL maintains xTT in memory to trace all the previous policy decisions, and an xTT is augmented each time a new decision is made, in the next experiment, we investigate how much memory PNPL consumes for maintaining the xTT. We use `l2-learning` as the policy program, and employ Cbench to emulate up to 100 switches, with each switch reporting up to 10 `PacketIn` packets. On receiving each `PacketIn` packet, PNPL invokes the policy program, reaches a decision, and augments the xTT with a new branch. In Figure 10, we present the memory consumptions of the PNPL process observed using the `pmap` command, under the conditions that various number of policy decisions have been traced in xTT. From the figure we can see that, even tracing $1,000$ policy decisions (i.e., the xTT has $1,000$ branches ending at different **L** nodes), the PNPL process doesn't impose a heavy overhead by consuming less than 2MB memory.

In addition to memory consumption, another concern is how fast an xTT can be searched by Algorithm 2 and traversed by Algorithm 3 when handling a large number of `PacketIn` packets. To investigate the xTT searching performance, we first augment an xTT to trace $1,000$ policy decisions as in the previous experiment, then we emulate $10 \sim 100$ POF switches, with each switch reporting 1/5/10 `PacketIn` packets per second. Note that the policy decision for each `PacketIn` packet has already been traced by the xTT, therefore PNPL doesn't invoke the `l2-learning` policy program. We embedded codes in the PNPL prototype to record how long it takes to handle a `PacketIn` packet under various workloads. Figure 11 presents the results. From the figure we can see that even under the workload of $1,000$ `PacketIn`/second, the prototype handles a `PacketIn` packet in dozens of microseconds; however, the processing time obviously increases with the workload.

In summary, our observations in this subsection suggest that the PNPL prototype has a decent performance compared to the mainstream OpenFlow controllers. Searching xTT is efficient, and maintaining it doesn't imposed a large overhead regarding the memory consumption. PNPL can be further improved in two ways in future: In case that an xTT becomes very large, we can introduce tree invalidation API to remove part of an xTT as in Maple. For example, the programmer can remove all the branches on the xTT that contain instructions on a certain switch that no longer exists. For handling extremely large `PacketIn` overhead, parallel computing techniques on many-core platforms can be exploited.

6.2. Use Cases and Pipeline Quality

In this subsection, we implement a number of use case network policies with PNPL, and examine the FPs that are installed by PNPL to POF switches. We seek to answer three questions: 1) Will PNPL produce many additional rules for packet parsing? 2) Will PNPL efficiently support novel network mechanisms with self-defined protocols? 3) Will PNPL fully exploit the multi-table pipeline and metadata resources for avoiding flow explosion? Codes of the PNPL use case policy programs can be found in our technical report [27].

6.2.1. Firewall policies

In our first experiment, we select eight representative ebtables [32] and iptables [33] firewall policies, which cover all the protocol layers from L2 to L5, and implement them with PNPL. For each policy, we examine how the PNPL-produced FPs are organized, and how many tables and flow entries are installed. We also implement the firewall polices on an OpenFlow network, and compare the PNPL-produced FPs with the flow tables in the OpenFlow switch. We use the Huawei NE40E router [34] to serve as the POF data plane. The OpenFlow experiment is carried out in Mininet [35], and we use POX to compose the OpenFlow firewall rules for comparison.

For each firewall policy implemented, we employ the Ostinato traffic generator [36] to send 100 packets to the switch. The packets invoke the policy program to make 100 different policy decisions, and moreover, the packets have variable-length headers and variable header sequences, thus demand the POF switch to parse them differently. More specifically, we randomly select 30% packets to have one VLAN header, and 10% packets to have two VLAN headers; we also let 20% of the IPv4 headers to have a 4-byte option, 15% headers to have an 8-byte option, and 5% to have a 12-byte option. Note that with such a setting, even for the packets sharing a

Table 2: Comparison of PNPL generated FPs and OpenFlow rules for enforcing firewall policies.

Firewall policy	Policy decisions	FP generated by PNPL						OpenFlow rules
		FT_1	FT_2	FT_3	FT_4	FT_5	FT_6	
MAC NAT	100	100	-	-	-	-	-	100
Blocking specific IP addresses	100	2	2	1	100	-	-	100
Forwarding IP traffics from specific MAC addresses	100	2	2	1	100	-	-	100
Associating IP addresses to MAC addresses	100	2	2	1	100	-	-	100
Allowing Pings from outside	100	2	2	1	4	125	-	125
Allowing outbound DNS queries	100	2	2	1	4	1	125	125
Disallowing outgoing mails	100	2	2	1	4	100	-	100
Allowing SSH from specific IP addresses	100	2	2	1	4	125	-	125

same policy decision, they may have $3 \times 4 = 12$ different appearances with 0, 1, or 2 VLAN headers and a 0-, 4-, 8-, or 12-byte `option` field in the IPv4 header, and are required to be handled differently.

Table 2 presents a summary of the FPs generated by PNPL for enforcing the firewall policies, and we also list the OpenFlow rules for implementing the same policies for comparison. In the following, we describe the PNPL-produced FPs for each of the firewall policies in details:

1. **NAT MAC**: This policy re-writes the source MAC address, and forwards packets based on the destination MAC address. The pipeline has only one FT: FT_1 contains 100 rules for modifying the Ethernet header's `mac_src` field, and forwarding 100 packets according to their destination MAC addresses.
2. **Blocking specific IP addresses**: In this policy, packets with specific source IP addresses shall be dropped. The pipeline contains 4 stages of FTs: FT_1 matches the Ethernet header's `ethertype` field, and directs packets to either FT_2 or FT_4 under the cases that the next header is VLAN or IPv4 respectively; similarly, FT_2 matches the first VLAN header's `ethertype` field, and directs packets to FT_3 or FT_4 under the cases that the next header is VLAN or IPv4 respectively; FT_3 matches the inner VLAN header's `ethertype` field, and directs packets to FT_4; finally, FT_4 matches the IPv4 header's `ip_src` field, and makes the policy decisions on blocking or not for 100 packets with 100 different source IP addresses.
3. **Forwarding IP traffics from specific MAC addresses**: In this policy, only the packets from specific source MAC addresses will be forwarded. The pipeline contains 4 stages of FTs: FT_1 writes the `mac_src` field to metadata, matches the Ethernet's `ethertype` field, and directs packets to either FT_2 or FT_4 under the cases that the next header is VLAN or IPv4 respectively; FT_2 and FT_3 handle the outer and inner VLAN headers as in the previous example; finally, FT_4 matches the metadata data piece that keeps the source MAC address as well as the IPv4 header's `ip_dst` field, and makes the policy decisions on forwarding or not for 100 packets with 100 different source MAC address and destination IP address combinations.
4. **Associating IP addresses to MAC addresses**: this policy checks whether a packet's source IP address is associated with a right source MAC address. The pipeline contains 4 stages of FTs: FT_1, FT_2, and FT_3 are same as in the previous example for writing the source MAC address to metadata and handling

the outer and inner VLAN headers; FT_4 matches the metadata data piece that keeps the source MAC address as well as the IPv4 header's `ip_src` field, and makes the policy decisions on forwarding or not for 100 packets with 100 different source MAC and source IP address combinations.

5. **Allowing Pings from outside**: this policy allows Ping from outside IP addresses. The pipeline contains 5 stages of FTs: FT_1, FT_2, and FT_3 are same as in the "blocking specific IP addresses" example for handling the Ethernet, outer and inner VLAN headers; FT_4 writes the IPv4 header's `ip_src` field to metadata, matches the `proto` field, move the *p-offset* cursor to the next header using the `MOVE_PACKET_OFFSET` instruction with 4 different offset values, under the cases that the IPv4 header contains a 0-, 4-, 8- or 12-byte `option` field, and directs packets to FT_5; finally, FT_5 matches the metadata data piece that keeps the source IP address as well as the ICMP header's `icmptype` field, and makes the policy decisions on allowing Ping or not for 100 Ping probe packets from 100 different source IP addresses. Note that for allowing Ping from a source, two flow entries need to be installed for allowing the Ping probe and response in both directions[2].
6. **Allowing outbound DNS queries**: This policy allows DNS queries to outside IP addresses. The pipeline contains 6 stages of FTs: FT_1, FT_2, FT_3, and FT_4 are same as in the previous example, except that FT_4 writes the IPv4 header's `ip_dst` field to metadata; FT_5 matches the UDP header's `udp_dport` field and directs packets to FT_6; finally, FT_6 matches the metadata data piece that keeps the destination IP address as well as the DNS header's `dns_flag` field, and makes the policy decisions on allowing the DNS query or not for 100 DNS query packets to 100 different destination IP addresses. Note that for allowing a DNS service, two flow entries need to be installed for allowing the DNS query and reply in both directions[3].
7. **Disallowing outgoing mails**: This policy drops SMTP packets sent to outside IP addresses. The pipeline contains 5 stages of FTs: FT_1, FT_2, FT_3, and FT_4 are same as in the previous example for handling the Ethernet, outer and inner VLAN, and IPv4 headers, and writing the destination IP address to metadata; FT_5 matches the metadata data piece that keeps the destination IP address as well as the TCP header's `tcp_dport` field, and makes the policy decisions on dropping the packet or not for 100 SMTP packets to 100 different destination IP addresses.
8. **Allowing SSH from specific IP addresses**: This policy allows SSH connections from specific source IP addresses. The pipeline contains 5 stages of FTs: FT_1, FT_2, FT_3, and FT_4 are same as in the previous example for handling the Ethernet, outer and inner VLAN, and IPv4 headers, and writing the destination IP address to metadata; FT_5 matches the metadata data piece that keeps the destination IP address as well as the TCP header's `tcp_dport` field, and makes the policy decisions on allowing the SSH TCP connection or not for 100 SSH TCP SYN segments to 100 different destination IP addresses. Note that

[2]In our experiment, 75 rules are installed for dropping the Ping probes, and 25 pairs of rules are installed for allowing the Ping probes and responses in FT_5.

[3]In our experiment, 75 rules are installed for dropping the DNS queries, and 25 pairs of rules are installed for allowing the DNS queries and replies in FT_6.

Table 3: Summaries of PNPL-produced FPs for source routing and OpenFlow rules produced by `l2-learning` policy.

	FP produced by PNPL for source routing on POF		OpenFlow rules
	FT_1 (Ethernet)	FT_2 (SRP)	by `l2-learning`
Core	1	4	84
Aggregation	4	2	48
Edge	32	2	58

for allowing an SSH connection, two flow entries need to be installed for allowing the SSH TCP segments in both directions[4].

From Table 2 and the above description, we can see that PNPL automatically produces multi-table pipelines. We note that within an FP, only the last FT makes policy decisions, and the other preceding FTs contain the rules for parsing the packet headers of different protocol layers. For example, the "allowing outbound DNS queries" policy program produces an FP consisting of up to six FTs, as the switches need to match the `dns_flag` field in the L5 DNS protocol to make the policy decision. Compared to the OpenFlow rules, PNPL produces a little more forwarding rules on POF switches, as the additional rules are used for packet parsing. However, as we can see in the table, the numbers of the additional packet parsing rules are limited, as PNPL only parses the necessary packet fields on demand. For example, for the "allowing outbound DNS queries" policy, only 10 rules are installed in FT_1-FT_4 for parsing the necessary fields in the Ethernet, outer and inner VLAN, IPv4 and UDP headers. In other words, as a protocol agnostic programming framework, PNPL doesn't produce many packet parsing rules and impose a significant additional overhead regarding the rule space on the POF data plane.

6.2.2. Source routing

One benefit of programming over self-defined protocols is to enable novel network mechanisms. For example, it is well-known that the conventional L2 forwarding and L3 routing mechanisms do not scale for a datacenter network, while source routing is considered as a promising solution [37][38]. For enabling source routing, people either modify the semantics of the exiting protocol's header field (e.g., Portland [37]) or introduce brand new architecture and protocol stack (e.g., Sourcey [38]). Inspired by Portland, in its position paper [12], P4 demonstrates an example named *mTag*, in which a 5-field mTag header is added after Ethernet for encoding the location information for enabling a Portland-like source routing service.

In this experiment, we use PNPL to introduce a self-defined protocol named *SRP*, and implement a Portland-like souring routing mechanism on a datacenter network. We organize the datacenter network as a FatTree [39] with $k = 4$ pods, and emulate it using an extended Mininet with the POF software switch module.

SRP serves as a 2.5-layer protocol after Ethernet, and it contains four fields. The first three fields are identical to the three parts of Portland's pseudo MAC address that encodes the destination host's location

[4]In our experiment, 75 rules are installed for blocking the connections, and 25 pairs of rules are installed for allowing the connections in FT_5.

information. More specifically, the header contains an 8-bit `srp_pod` field that specifies the pod in which the destination host resides, an 8-bit `srp_pos` field that indicates the position of the edge switch that the destination host connects to, and a 16-bit `srp_port` field that shows the switch port connecting to the destination host. The last `ethertype` field is for indicating the next layer protocol. The SRP header specification is displayed as the following.

```
header SRP
    fields
        _srp_pod : 8;
        _srp_pos : 8;
        _srp_port : 16;
        _ethertype : 16;
    select (ethertype);
        ....
```

We compose the source routing mechanism as in Portland with PNPL, and examine the FPs that are installed by PNPL to the edge, aggregate, and core switches in the FatTree datacenter network. PNPL produces FPs containing two FTs on all the POF switches, however, different types of switches have different behaviors. In edge switch, FT_1 adds the SRP header to the upward packets according to their destination MAC addresses, forwards them based on `InPort`, and directs the downward packets to FT_2; FT_2 matches the SRP header's `srp_port` field to forward the downward packets to their destination hosts, and removes the SRP header before forwarding them out. In aggregate switch, FT_1 forwards the upward packets according to `InPort`, and directs the downward packets to FT_2, while FT_2 matches the SRP header's `srp_pos` field to forward the packets to the right edge switches. In core switch, FT_1 forwards the packets to FT_2, and FT_2 matches the SRP header's `srp_pod` field to forward the packets to the aggregate switches in their destination pods. Table 3 lists a summary of the FPs produced by PNPL, from which we can see that, with the self-defined SRP protocol allowed by PNPL and POF, source routing can be efficiently enabled on the datacenter network, and each switch contains only limited number of rules.

We also simulate a FatTree datacenter network with OpenFlow switches in Mininet, and apply `l2-learning` on POX as its policy program. We use the `pingall` command to trigger POX to install rules on OpenFlow switches for allowing any pair of hosts to communicate. Since FatTree contains loops, we run the `spanning-tree` component on POX with `l2-learning`. In the last column of Table 3, we show the OpenFlow flow entries installed on different types of switches in the FatTree network. We can see that compared to `l2-learning`, the source routing mechanism enabled by SRP can greatly reduce the number of the forwarding rules deployed to the data plane, thanks to the programmability provided by POF and PNPL.

Table 4: Comparison of PNPL generated FPs and OpenFlow rules for implementing VRF.

	VRF by PNPL		VRF with
	Ethernet FT	IPv4 FT	OpenFlow
Rules installed	100	500	5,000
Entries evicted	0	0	3,976

6.2.3. Virtual routing and forwarding

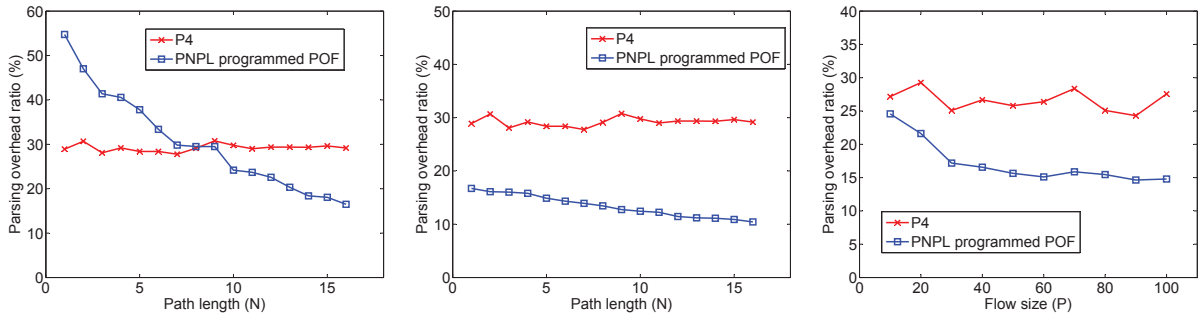
We use the virtual routing and forwarding (VRF) example as in [28] to demonstrate the benefits of metadata programming enabled by PNPL.

In the experiment we consider a layer-2 network with N mobile hosts, which are internally grouped into M virtual LANs based on their MAC addresses (however, packets from the hosts do not carry VLAN tags). The network connects to the Internet through a gateway VRF router. The router maintains M IPv4 routing table instances, one for each VLAN, and each routing table has K entries. With a simple analysis, we can see that if the VRF router is implemented with a single-table OpenFlow switch, a total number of $(N \times K)$ rules are required, as each flow entry matches the host's source MAC address and the destination IP prefix, and there are $(N \times K)$ combinations. Since the number of the IP prefixes K in a routing table is large, if the number of the hosts N is big, flow explosion occurs. But with a POF switch, the VRF router can be implemented as a FP consisting of two FTs: the first Ethernet FT matches the source MAC address and writes the assigned VLAN ID to metadata, which contains only N flow entries; the second IPv4 FT matches the VLAN ID in metadata and the destination IP address to forward the packet, and since there are M different VLANs and K different IP prefixes, the FT contains $(M \times K)$ rules. Overall, the pipeline requires $(N + M \times K)$ rules. Since the number of the VLANs M are much smaller than the number of the hosts N, i.e., $M \ll N$, so $(N + M \times K) \ll (N \times K)$, in other words, the FP requires much fewer rules than the single-table for implementing VRF.

We compose VRF with PNPL, and compare the produced FP with an OpenFlow flow table that enforces the same functionality. In our experiment we let $N = 100$, $M = 10$, and $K = 50$, and for both switches, the maximum number of flow entries is limited to $1,024$. Table 4 lists the rules installed on both switches and the rules that are evicted because of flow space limitations. We can see that the OpenFlow switch suffers a flow explosion problem by evicting as many as $3,976$ rules, while the POF switch can accommodate all the rules as much fewer rules are required. From the experiment we can see that PNPL enables users to fully exploit the benefits from the multi-table pipeline and the inter-table metadata resource.

6.3. Comparing with P4

One big difference between a PNPL-programmed POF network and a P4 network is how packets are parsed. In P4, each switch contains a frontend parser that is configured by the P4 language to provide a comprehensive parsing for all the incoming packets. One the other hand, when a POF switch encounters a packet that it doesn't know how to handle, the packet is reported to PNPL with a `PacketIn` message. PNPL parses the `PacketIn` packet, augments the xTT with a new branch, and installs rules that contain the tuples and values as match keys for parsing the packet on all the POF switches along the forwarding path, and all the subsequent



(a) $P = 1$ (b) $P = 100$ (c) $N = 4$

Figure 12: Comparison of packet parsing overheads of P4 and PNPL-programmed POF networks, under various flow size P and forwarding path length N.

packets will be parsed without the control plane involvement. Moreover, instead of comprehensive parsing, the PNPL-produced FPs parse packets in an on-demand way by only matching the fields that are of interest, thus should be more efficient.

In this subsection, we seek to compare the packet parsing overheads of a PNPL-programmed POF network and a P4 network. For implementing a P4 network, we use the `bmv2` P4 software switch and the `simple router` P4 code from the P4 project [40]. The code simply programs a P4 switch to recognize Ethernet and IPv4 headers and perform the LPM IP forwarding. Our experiment network topology is a chain of N P4 switches, where each switch forwards IP packets to the next hop. We create the same topology using POF switches, and compose a PNPL program to implement the same network protocol and forwarding policy as in the P4 network.

Before presenting the experiment results, we first analyze the components of the packet processing time in both networks. First for the P4 network, as each packet is comprehensively parsed by the frontend parser of each P4 switch, the total time for packet parsing can be expressed as

$$T_{P4}^{parse} = t_{P4}^{parse} \times N \times P \tag{1}$$

where t_{P4}^{parse} is the time for a P4 switch's frontend parser to parse a packet, P is number of the packets in a flow (i.e., the flow size), and N is the number of the switches along the forwarding path (i.e., the path length). Besides packet parsing, each P4 switch also matches every packet's IP destination address in an LPM way and forwards them, therefore the total node processing time for transmitting P packets along an N-hop path is

$$T_{P4} = T_{P4}^{parse} + t_{P4}^{LPM} \times N \times P \tag{2}$$

where t_{P4}^{LPM} is the time of the LPM matching for one packet in a P4 switch. Obviously, the ratio of the packet parsing time in the entire node processing time can be expressed as

$$R_{P4} = \frac{T_{P4}^{parse}}{T_{P4}} = \frac{t_{P4}^{parse}}{t_{P4}^{parse} + t_{P4}^{LPM}} \tag{3}$$

On the PNPL-programmed POF network, the first packet is sent to the control plane and handled by PNPL, and we denote the control plane parsing time as t_{ctl}^{parse}. For the subsequent packets, as PNPL has produced

and deployed FPs to the POF switches, they are only parsed by the POF switches in an on-demand way, and in our case, a POF switch only matches the `EtherType` field and directs the packet to the next table for LPM matching. We use t_{POF}^{parse} to denote such parsing overhead per switch per packet. Combining the two overheads, for the PNPL-programmed POF network, the total parsing time is

$$T_{PNPL}^{parse} = t_{ctl}^{parse} + t_{POF}^{parse} \times N \times P \tag{4}$$

Similar to the analysis for the P4 network, each POF switch also matches every packet's destination IP address in an LPM way, thus the ratio of the packet parsing time in the entire node processing time can be expressed as

$$R_{PNPL} = \frac{T_{PNPL}^{parse}}{T_{PNPL}^{parse} + t_{POF}^{LPM} \times N \times P} \tag{5}$$

where t_{POF}^{LPM} is the time of the LPM matching for one packet in a POF switch.

We insert codes in the PNPL prototype as well as the P4 and POF software switches to measure t_{ctl}^{parse}, t_{P4}^{parse}, and t_{POF}^{parse} respectively. For obtaining the packet parsing overhead ratios of R_{P4} and R_{PNPL}, we measure the end-to-end delay for transmitting P packets along the N-hop path, and approximate it as the overall node processing time. We vary the path length N and the flow size P in each experiment, and present the results in Figure 12. From the figures, we can see that the packet parsing overhead ratio of the P4 network is nearly constant, but for the PNPL-programmed POF network, the ratio decreases as the path length and flow size increases. The observation is easy to understand, as from Equation (3), each packet must be comprehensively parsed by the frontend parser in each switch along the path on the P4 network, thus the ratio of the packet parsing overhead is constant as the flow size and path length vary. However, on the PNPL-programmed POF network, only the first packet of a flow is parsed by PNPL on the control plane, and each POF switch only parses the subsequent packets in an on-demand way, incurring much lower overhead compared to P4's comprehensive parsing. From Equation (5) we can see that, as the path length and flow size increases, parsing the first packet by PNPL can benefit more switches on the forwarding path by assisting them to parse more subsequent packets in the flow, therefore the parsing overhead ratio decreases as the path length and flow size increase. Note that here we compare the ratios rather than the absolute values of the packet parsing overheads. In fact in our experiment, t_{P4} is orders of magnitude longer than t_{POF}, and the PNPL-programmed POF network achieves a much lower end-to-end delay compared to the P4 network. Our observation shows that when compared to P4, PNPL enables an on-demand parsing scheme that can significantly reduce the network's parsing overhead and improve its performance.

In summary, our evaluation in this section show that our PNPL prototype has decent performance; a wide range of network applications can be implemented with PNPL, and the produced FPs are of high quality; finally, comparing with P4, PNPL can significantly reduce the packet parsing overhead and improve the forwarding performance.

7. Conclusion

In this paper, we presented PNPL, a framework that simplifies SDN programming over POF networks. PNPL supports a P4-like header specification language, and provides high-level protocol-agnostic abstractions for composing network policies over POF networks. We developed novel methodologies for automatically producing and maintaining forwarding pipelines that incrementally parse network packets with variable-length headers and variable header sequences, and enforce users' network policies. We prototyped and evaluated PNPL with a wide range of protocol headers and network applications, and find that PNPL fully realizes POF benefits, the forwarding pipelines generated by PNPL are correct and of high quality, and the on-demand parsing enabled by PNPL significantly reduces the parsing overhead.

Acknowledgements

This work was supported in part by the National Natural Science Foundation of China under Grants 61672486 and 61672485, in part by the subtask of New Generation Broadband Wireless Mobile Communication Network Major Project in China under Grant 2017ZX03001019-004, and in part by the Anhui Provincial Natural Science Foundation under Grant 1608085MF126.

References

[1] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, P. Hudak, Maple: Simplifying SDN programming using algorithmic policies, in: Proc. of SIGCOMM'13, Hong Kong, China, 2013.

[2] J. Reich, C. Monsanto, N. Foster, J. Rexford, D. Walker, Modular SDN programming with Pyretic, USENIX 38 (5) (2013) 1–7.

[3] C. Monsanto, J. Reich, N. Foster, J. Rexford, D. Walker, Composing software-defined networks, in: Proc. of NSDI'13, Lombard, IL, USA, 2013.

[4] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, D. Walker, Frenetic: A network programming language, in: Proc. of ICFP'11, Tokyo, Japan, 2011.

[5] R. Soule, S. Basu, R. Kleinberg, E. G. Sirer, N. Foster, Managing the network with Merlin, in: Proc. of HotNets'13, College Park, MD, USA, 2013.

[6] M. Reitblatt, M. Canini, A. Guha, N. Foster, FatTire: Declarative fault tolerance for software-defined networks, in: Proc. of HotSDN'13, Hong Kong, China, 2013.

[7] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, Kinetic: Verifiable dynamic network control, in: Proc. of NSDI'15, Oakland, CA, USA, 2015.

[8] Y. Yuan, D. Lin, R. Alur, B. T. Loo, Scenario-based programming for SDN policies, in: Proc. of ACM CoNEXT'15, Heidelberg, Germany, 2015.

[9] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, Y. Zhang, PGA: Using graphs to express and automatically reconcile network policies, in: Proc. of ACM SIGCOMM'15, London, UK, 2015.

[10] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, OpenFlow: Enabling innovation in campus networks, ACM SIGCOMM CCR 38 (2) (2008) 69–74.

[11] OpenFlow, `https://www.opennetworking.org/sdn-resources/openflow`, accessed: Mar. 24, 2017.

[12] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, D. Walker, P4: Programming protocol-independent packet processors, ACM SIGCOMM CCR 44 (3) (2014) 87–95.

[13] P4, `http://p4.org/`, accessed: Mar. 24, 2017.

[14] S. Jouet, D. P. Pezaros, BPFabric: Data plane programmability for software defined networks, in: Proc. of ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'17), Beijing, China, 2017.

[15] H. Song, Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane, in: Proc. of HotSDN'13, Hong Kong, China, 2013.

[16] J. Yu, X. Wang, J. Song, Y. Zheng, H. Song, Forwarding programming in protocol-oblivious instruction set, in: Proc. of ICNP'14, Raleigh, NC, USA, 2014.

[17] H. Song, J. Gong, H. Chen, J. Dustzadeh, Unified POF programming for diversified SDN data plane devices, in: Proc. of International Conference on Networking and Services (ICNS'15), Rome, Italy, 2015.

[18] G. Gibb, G. Varghese, M. Horowitz, N. McKeown, Design principles for packet parsers, in: Proc. of ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'13), San Jose, CA, USA, 2013.

[19] PNPL prototype, `https://gitlab.com/nhpcc416/PNPL`, accessed: Mar. 10, 2018.

[20] Linux Socket Filtering aka Berkeley Packet Filter (BPF), `https://www.kernel.org/doc/Documentation/networking/filter.txt`, accessed: Jul. 22, 2018.

[21] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, M. Casado, The design and implementation of Open vSwitch, in: Proc. of NSDI'15, Oakland, CA, USA, 2015.

[22] C. Schlesinger, M. Greenberg, D. Walker, Concurrent NetCore: From policies to pipelines, in: Proc. of ICFP'14, Gothenburg, Sweden, 2014.

[23] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, D. Walker, SNAP: Stateful network-wide abstractions for packet processing, in: Proc. of ACM SIGCOMM'16, Florianopolis, Brazil, 2016.

[24] ETSI, Network functions virtualisation (NFV); ecosystem; report on SDN usage in NFV architectural framework, Standard ETSI GS NFV-EVE 005 V1.1.1, European Telecommunications Standards Institute (Dec. 2015).

[25] ITU-T, FG IMT-2020: Report on standards gap analysis, Standard TD 208 (PLEN/13), International Telecommunication Union (Dec. 2015).

[26] D. Hancock, J. van der Merwe, HyPer4: Using P4 to virtualize the programmable data plane, in: Proc. of CoNEXT'16, Irvine, CA, USA, 2016.

[27] X. Wang, Y. Tian, M. Zhao, M. Li, L. Mei, X. Zhang, PNPL: Simplifying programming for protocol-oblivious SDN networks, available at `http://staff.ustc.edu.cn/~yetian/pub/PNPL.pdf` (Mar. 2018).

[28] The benefits of multiple flow tables and TTPs, Tech. Rep. ONF TR-510, Open Networking Foundation (Feb. 2015).

[29] Cbench, `https://github.com/intracom-telecom-sdn/mtcbench`, accessed: Mar. 24, 2017.

[30] Floodlight, `http://www.projectfloodlight.org/floodlight/`, accessed: Mar. 24, 2017.

[31] The POX controller, `https://github.com/noxrepo/pox`, accessed: Mar. 24, 2017.

[32] Linux Ethernet bridging firewalling, `http://ebtables.netfilter.org/`, accessed: Mar. 24, 2017.

[33] The iptables project, `http://www.netfilter.org/projects/iptables/`, accessed: Mar. 24, 2017.

[34] Huawei, NetEngine40E universal service router, `https://e.huawei.com/en/products/enterprise-networking/routers/ne/ne40e`, accessed: Jul. 22, 2018.

[35] Mininet, `http://mininet.org/`, accessed: Mar. 24, 2017.

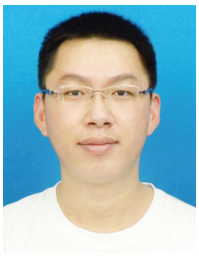
[36] Ostinato network traffic generator, `http://ostinato.org/`, accessed: Mar. 24, 2017.

[37] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, A. Vahdat, Portland: A scalable fault-tolerant layer 2 data center network fabric, in: Proc. of ACM SIGCOMM'09, Barcelona, Spain, 2009.

[38] X. Jin, N. Farrington, J. Rexford, Your data center switch is trying too hard, in: Proc. of ACM Symposium on SDN Research (SOSR'16), Santa Clara, CA, USA, 2016.

[39] M. Al-Fares, A. Loukissas, A. Vahdat, A scalable, commodity data center network architecture, in: Proc. of ACM SIGCOMM'08, Seattle, WA, USA, 2008.

[40] A. Bas, S. Fowler, S. Smolka, Y. Tseng, p4language, `https://github.com/p4lang`, accessed: Oct. 12, 2017.



Xiaodong Wang is a Ph.D. candidate at the School of Computer Science and Technology, University of Science and Technology of China, Hefei, China. His research is focused on the SDN-based technology for the future Internet.

Ye Tian received the Ph.D. degree from the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, in 2007. He joined the University of Science and Technology of China (USTC) in 2008 and is currently an Associate Professor with the School of Computer Science and Technology, USTC. His research interests include Software-Defined Networking, future Internet architectures, Internet measurement and performance evaluation. He has published over 60 papers and co-authored a research monograph published by Springer. He is the winner of the Wilkes Best Paper Award of the Oxford Computer Journal in 2016. He is currently serving as an Associate Editor of the Springer Frontiers of Computer Science Journal.

Min Zhao is a master candidate at the School of Computer Science and Technology, University of Science and Technology of China, Hefei, China. Her research is focused on Software-Defined Networking.

Mingzheng Li is a master candidate at the School of Computer Science and Technology, University of Science and Technology of China, Hefei, China. His research is focused on the future Internet technology.

Lei Mei is a master candidate at the School of Computer Science and Technology, University of Science and Technology of China, Hefei, China. His research is focused on the SDN-based technology for the future Internet.

Xinming Zhang received the Ph.D. degree in computer science and technology from the University of Science and Technology of China (USTC), Hefei, China, in 2001. Since 2002, he has been with the faculty of the USTC, where he is currently a professor with the School of Computer Science and Technology. He is also with the National Mobile Communications Research Laboratory, Southeast University, Nanjing, China. From September 2005 to August 2006, he was a visiting professor with the Department of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology, Daejeon, Korea. His research is focused on wireless networks. He has published more than 70 papers in networking.