POMP: Protocol Oblivious SDN Programming with Automatic Multi-Table Pipelining

Chunhui He
School of Computer Science and Technology
University of Science and Technology of China
hchunhui@mail.ustc.edu.cn

Xinyu Feng
State Key Laboratory for Novel Software Technology
Nanjing University
xyfeng@nju.edu.cn

***Abstract*—SDN programming has been challenging because programmers have to not only implement the control logic, but also handle low-level details such as the generation of flow tables and the communication between the controller and switches. New generation of SDN with protocol oblivious forwarding and multi-table pipelining introduces even more low-level details to consider.**

We propose POMP, the first SDN programming environment supporting both *protocol oblivious forwarding* and *automatic multi-table pipelining*. POMP applies the static taint analysis technique to automatically infer compact and efficient multi-table pipelines from a data-plane agnostic network policy written by the programmer. The runtime system tracks the execution of the network policy, and automatically generates table entries. POMP also introduces a novel notion of *dependent labels* in the taint analysis, which, combined with the runtime information of the network policy, can further reduce the number of table entries. Like P4, POMP supports protocol-oblivious programming by providing a network protocol specification language. Parsers of packets can be automatically generated based on the protocol specification. POMP supports two main emerging SDN platforms, POF and P4, therefore network policies written in POMP are portable over any switches supporting POF or P4.

I. Introduction

Software-Defined Networking (SDN) is a network architecture that decouples control and forwarding. Distributed switches are managed by a logically-centralized controller, which can be implemented by software through SDN programming.

OpenFlow is the first SDN standard, but it only supports a predefined fixed set of networking protocols, and allows only a fixed *single* flow table. New generation of SDN platforms [1], [2], [3], [4] offer two new flexible features at the data-plane, *i.e.* multi-table pipelining and protocol-oblivious forwarding. The former allows multiple flow tables on switches to form a forwarding pipeline, which can be customized by users. The latter supports customized packets for new protocols.

SDN programming has been challenging because the programmer has to not only implement the network policy (*i.e.* the control logic), but also handle low-level data-plane details. Specifically, one faces the following challenges:

1) to manually translate high-level network policies to flow table entries. It is error-prone, and makes network policies hard to write and read [5].
2) to design flow table layout and forwarding pipelines for new generation SDN to achieve compact flow tables and efficient forwarding. As we explain in Sec. III, different forwarding pipelines may generate flow tables with significantly different sizes.
3) to implement parsers for packets with new header fields to fit in the underlying protocol oblivious forwarding mechanism. For instance, one needs to generate the `(offset, length)` tuples required by POF [2] to locate packet fields.

There have been many languages proposed to simplify SDN programming, but none of them address all the problems above. Languages such as NetKAT [6], NetCore [7] and Maple [5] try to address the first problem, but they are designed for Openflow and do not support multi-table pipelining and protocol-oblivious forwarding. P4 [1] provides a header specification language for protocol independence. Programmers can write a specification of the header format, from which P4 generate parsers automatically. But P4 is more of a low-level switch configuration language, and is not suitable for high-level controller programming.

We propose POMP, a high-level programming environment to simplify SDN programming. Following the idea of Maple [5], POMP provides a set of APIs and a runtime system. Programmers can use the APIs to describe the *network policy* by writing an algorithmic sequential program with a high-level centralized view of the network environment. This high-level network policy is data-plane agnostic. It is the runtime system that tracks the execution of the network policy and generates flow table entries automatically. POMP also makes significant extensions to the above ideas in Maple to support automatic multi-table pipelining and protocol oblivious programming.[1] As far as we know, POMP is the first high-level programming environment that solves all the aforementioned problems. Our work on POMP makes the following new contributions:

- We apply the static taint analysis technique [8] to analyse the network policy's dependence over the fields of packets. Based on the fine-grained dependence relation we automatically generate compact and efficient multi-table pipelines. The analysis and the pipeline generation

Corresponding author: Xinyu Feng. This work is supported in part by grants from National Natural Science Foundation of China (NSFC) under Grant Nos. 61379039 and 61632005.

[1]The name POMP highlights the two key features of our work, *i.e.* protocol-oblivious programming and multi-table pipelining.

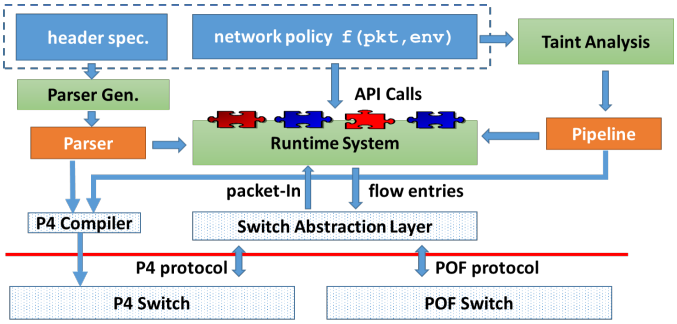

Fig. 1. Structure of POMP

is done statically before the deployment of the controller, therefore it does not introduce any runtime overhead.

- We extend the taint analysis with a novel notion of *dependent labels*, which further refines the dependence relation and allows it to be conditional upon the `if` statement branches taken at runtime. Combining the dependent labels with the runtime execution traces of the network policy, the runtime system can further reduce the number of flow table entries. Our experiments show that, for learning switches, POMP generates flow tables with up to 47x fewer table entries and being 137x faster than the traditional single flow table (as generated in Maple).
- Inspired by P4, POMP incorporates a packet header specification language, with which programmers can specify the format of packet headers. Based on the specification we can automatically generate parsers for packets. This makes POMP a protocol oblivious programming environment. Our runtime system can support two main emerging SDN platforms: POF and P4. It makes network policies written in our language portable over any SDN chipsets and software switches that support POF or P4.

In the remaining part of the paper, we give a system overview and introduce a running example in Sec. II. We present the taint analysis and the pipeline generation in Sec. III, and propose the dependent labels and the runtime generation of table entries in Sec. IV. We then introduce the packet parsing for protocol oblivious programming in Sec. V. We show evaluation results in Sec. VI. Finally we discuss related work in Sec. VII and conclude in Sec. VIII.

II. Overview

In this section we first give an overview of the system structure of POMP. Then we introduce learning switches as a running example used in the following sections.

A. System structure

Figure 1 shows the system structure. Following Maple, POMP allows programmers to implement the network policy as an algorithmic sequential program with a centralized view of the network environment. The network policy is implemented as a function `f(pkt, env)`. It is a *C program* with API calls of the POMP library.

From programmers' point of view, `f(pkt, env)` is invoked upon the arrival of *every* packet `pkt` in each data-plane switch. `f` also takes a *centralized view* of the network environment `env`. It calculates the routing path and returns the *port number* to forward the packet. The return value can also be 0 or negative, which means to drop the packet or to broadcast it, respectively. `f` may also call POMP APIs to modify the global environment or to modify fields of packets.

What really happens is that the controller (the runtime system) invokes `f(pkt, env)` when it receives a PacketIn message, and passes the packet to `f`. The runtime then monitors the execution of `f` and generates an execution trace, which records the sequence of API calls made by `f` to read or update packet header fields or the environment data. Based on the dependence between actions and values of packet fields reflected in the trace, the runtime automatically generates flow table entries and install them on switches.

So far the ideas all come from Maple, but Maple does not support multi-table pipelining and protocol-oblivious forwarding. As we demonstrate in Sec. III, the dependence derived by Maple from the execution trace is way too imprecise and leads to flow tables with $O(n^2)$ forwarding rules for learning switches with n hosts, even though $O(n)$ rules are sufficient.

POMP introduces a static taint analysis phase to analyse the code of `f`, from which it infers the fine-grained dependence between actions and packet fields. Based on the fine-grained dependence, it automatically generates a multi-table pipeline. The runtime takes the multi-table pipeline into account and generates table entries accordingly.

To support protocol-oblivious programming, we follow the ideas in P4 and ask programmers to provide a header format specification written in a specification language. Then the parser generator automatically generates parsers for packets. POMP supports both POF and P4. For POF, the runtime queries the parser to map the string field names to the `(offset, length)` tuples. For P4, the generated parser and forwarding pipeline are fed into the P4 compiler to configure the switch. We also introduce a switch abstraction layer, which accepts P4 or POF messages separately and then converts them into a uniform format for the runtime system.

B. Learning Switches: a running example

We use *learning switches* to demonstrate how the system works. It is also used as a running example to illustrate the key ideas in the following sections. As shown in Fig. 1, our system takes as inputs a protocol specification and a network policy `f`, which are shown in Figs. 2 and 3 respectively. (Ignore the comments in Fig. 3, which shows the result of taint analysis and is explained in Sec. III).

The protocol specification describes the `eth` and `ipv4` protocols. Each is defined by a `header` block. For each `header` block, there are the `fields` section that describes the fields of the protocol, and the `next` section that describes the inner protocols. For example, the `eth` protocol has three fields: 48 bits `dst`, 48 bits `src` and 16 bits `type`. Depending on the values of `type`, the protocol follows the `eth` is `ipv4`

```
header eth {
  fields {
    dst : 48;
    src : 48;
    type : 16;
  }
  next select (type) {
    case 0x0800: ipv4;
    case 0x86dd: ipv6;
    ...
  }
}

header ipv4 {
  fields {
    ...
    ttl : 8;
    ...
  }

  ...
}

start eth;
```

Fig. 2. Header specification

or `ipv6`. The `start` clause at the end indicates the name of the outmost protocol.

The network policy `f` is written in C with POMP primitives. As shown in Fig. 3, the `f` for learning switches first reads the ingress port, the source address, the destination address and the type of the packet. It learns the source host, and remembers the ingress port in `mac2port` (line 6). Then it looks up `mac2port` to find the port for destination (line 7) and set the return value to the port number (line 9). If the result is 0, it means the port is unknown, so we set the return value to negative (line 11) to *broadcast* the packet.

For IPv4 packets, we also decrease the `ttl` (line 17) if it is greater than 0. Otherwise we drop the packet by setting the return value to 0.

Data-plane agnostic and protocol-oblivious network policy. Note that `f` does not describe forwarding pipelines and forwarding rules, which are automatically generated. Also it uses strings as field names to access packet header fields (*e.g.*, `read_packet(pkt, "eth.dst")`). Parsing is done automatically to map field names to their offsets in packets.

III. Pipeline Generation

Following the idea of Maple by recording the execution traces of `f`, we can derive the actions and their dependence over packet header fields, and generate a *single* flow table. Fig. 4 shows the flow table for learning switches.

The match fields of the flow table come from `read_inport()` and `read_packet()` in the network policy `f`. There are two possible actions of the flow table. The first is to modify the `ttl` field, and then to forward the packet. The second is to drop the packet.

However, such a flow table may cause unnecessarily large number of table entries. To see the problem, suppose there are n hosts in the network. When a host h_i sends a packet to h_j, the controller has to install a flow entry that matches "inport" (represented as "in_p" in the table), "src" and "dst". This results in $O(n^2)$ entries of the flow table.

One may have noticed that, since the switch forwards packets only based on the destination address, there is no need to enumerate all combinations of "src" and "dst". An apparent optimization might be omitting the match fields "src" and "inport" by filling in "*" in the table. But this is *incorrect* because we do need to match the exact values of "src" and "inport" — we rely on a mismatch to detect a new ("src", "inport") combination, which leads to a packet-in message sending to the controller, so that the controller can update the environment `mac2port` (line 6 in Fig. 3).

But still we do not need to exhaustively enumerate all the combination of "src" and "dst" (leading to $O(n^2)$ forwarding rules). Learning the topology needs to match "inport" and "src" but not "dst", and the forwarding needs to match "dst" (and "ttl" for IPv4 packets) but not "src". Each of the two functionalities requires $O(n)$ rules respectively. Therefore we could greatly reduce the number of forwarding rules (*i.e.* flow table entries) if we can have multiple flow tables, each corresponds to one independent functionality only.

To achieve this, we not only need the support of multi-table forwarding pipelines on switches (which is available in the emerging SDN platforms), but also need to have more fine-grained dependence relation between the functionality and the packet header fields. Since Maple only tracks the execution trace of `f`, it lets an action (*e.g.* the return of the value `r` at line 23) depend on *all* earlier actions that read packets or environments (*e.g.* the read of "src" at line 4). This is overly conservative and leads to imprecise dependence relation.

As one of the major contributions of the work, we use static taint analysis [8] to infer more fine-grained dependence relation. The analysis takes the function `f` as input and tracks the information flow from packets and environments to the actions that perform the functionalities (*e.g.* `mod_env()` at line 6), without actually running `f`. The result can be used to design multi-table forwarding pipelines.

A. Taint analysis

Taint analysis computes the information flow from sources to sinks. In our settings, the sources are the ingress port, packets' header fields, and environments. The sinks are the operations that output information, including the return of routing decisions, and the update of packets and environments. Since all accesses of packets and environments must be made through POMP APIs, they can be easily recognized in the code. In Table I, we list the APIs that obtain information from the sources. We also assign labels to identify each source. Note that for `test_equal(pkt, fld, v)` (which tests the equality between `fld` and `v`), we label the source with `test(fld)` instead of `fld`. The latter means we need the exact value of the field `fld`, while the former means we only care about certain property of `fld`. They indicate different number of entries in the flow table. Distinguishing them allows us to generate more compact forwarding pipelines, which we explain below. The sinks are the API calls of `mod_packet(pkt, fld, v)` and `mod_env(env, var, key, v)`, and also the command `return r`.

The process of the taint analysis propagates labels from sources to sinks. Our algorithm is mostly standard [8] except the extension with *dependent labels*. Here we only introduce the rough idea of taint analysis based on the learning switches example. Dependent labels are introduced later in Sec. IV.

```
1  f(pkt, env) {
2    inport = read_inport(pkt);                        // inport <- {inport}
3    dst = read_packet(pkt, "eth.dst");                // dst <- {dst}
4    src = read_packet(pkt, "eth.src");                // src <- {src}
5
6    mod_env(env, "mac2port", src, inport);            // mod_env@6 <- {src, inport}
7    port = read_env(env, "mac2port", dst);            // port <- {dst, env(mac2port)}
8    if (port != 0) {                                  // branch@8 <- {dst, env(mac2port)}
9      r = port;                                       // r <- {dst, env(mac2port)}
10   } else {
11     r = -inport;                                    // r <- {dst, env(mac2port), inport}
12   }
13                                                     // r <- {dst, env(mac2port), inport}
14   if(test_equal(pkt, "eth.type", 0x800)) {          // branch@14 <- {test(type)}
15     ttl = read_packet(pkt, "eth.ipv4.ttl");         // ttl <- {test(type), ttl}
16     if (!test_equal(pkt, "eth.ipv4.ttl", 1)) {      // branch@16 <- {test(ttl)}
17       mod_packet(pkt, "eth.ipv4.ttl", ttl - 1);     // mod_packet@17 <- {test(type), ttl}
18     } else {
19       r = 0;                                        // r <- {test(type), test(ttl)}
20     }
21   }
22                                  // r <- {dst, env(mac2port), inport, test(type), test(ttl)}
23   return r;                      // return@23 <- {dst, env(mac2port), inport, test(type), test(ttl)}
24 }
```

Fig. 3. Taint analysis for learning switches in POMP

priority	in_p	src	dst	type	ttl	action
1	2	h_2	h_1	0x800	64	MOD_FLD(ttl,63); OUT[1]
1	1	h_1	h_2	0x800	64	MOD_FLD(ttl,63); OUT[2]
1	1	h_1	h_3	0x800	1	DROP
...	...	...	...	...	...	

Fig. 4. Flow table layout of learning switches

TABLE I
SOURCES AND SINKS

Source	Label
read_inport(pkt)	inport
read_packet(pkt, fld)	fld
test_equal(pkt, fld, v)	test(fld)
read_env(env, var, key)	env(var)

(a) Sources and Labels

Sink
mod_packet(pkt, fld, v)
mod_env(env, var, key, v)
return r

(b) Sinks

Comments in Fig. 3 demonstrates how we trace the information flow. Every variable is assigned to a set of labels, recording the information flowing into the variable.

For assignment statements, the label set of the variable on the left hand side is the union of the label sets of the variables on the right and the set of sources accessed. For example, the variable `dst` is assigned the label set {`dst`} at line 7 (packet field prefix omitted in the label set to avoid clutter), and `port` at line 4 is assigned {`env(mac2port), dst`}, which is obtained by {`env(mac2port)`} $\cup$ **label**(`dst`).

The label set of a sink is the union of the label set of its arguments. The `mod_env()` at line 6 is a sink. Its label set is the union of the label sets of `src` and `inport`.

For `if` statements, we analyse both branches and then merge the results. As an example, the label set for `r` at line 13 is the union of the sets at lines 9 and 11 respectively. When analysing each branch, we also need to consider the label set of the boolean branch condition of the `if` statements, because of the implicit information flow caused by control dependence. Therefore the label set of variable `r` is the *union* of the label sets of both `inport` and `port`.

B. Xgraphs and Pipeline Generation

Although the taint analysis generates dependence for each sink (at lines 6, 17 and 23), we also need to maintain the control flow to decide the order of these actions in the forwarding pipeline. We let the taint analysis generate Xgraph, an intermediate representation of both the control flow and the dependence (*i.e.* label sets). The Xgraph for our example is shown in Fig. 5. There are two types of nodes. The square node represents an action corresponding to a sink. It records the name and the line number of the action (*e.g.* `mod_env@6` in the first node), and the corresponding label set (*e.g.* {`src`, `inport`} in the first node). Each diamond node represents a branch. It records the label set of the branch expression and the line number of the branch. The edges in the Xgraph represent the control flow.

Given the Xgraph, we can do a "node to node" translation to generate the multi-stage forwarding pipeline. Fig. 6 shows the pipeline generated from Fig. 5. For each square node in the Xgraph, we generate a flow table for the functionality. The match fields of the flow table are the packet header fields and the ingress port in the dependence set of the corresponding Xgraph node. The action is translated from the corresponding controller action in the Xgraph node, but not necessarily the same. For example, we translate the controller action `mod_env()` in Fig. 5 into a `[GOTO]` action, which does nothing and jumps to the next flow table on the pipeline. As we explained before, the only effect of this table is to generate

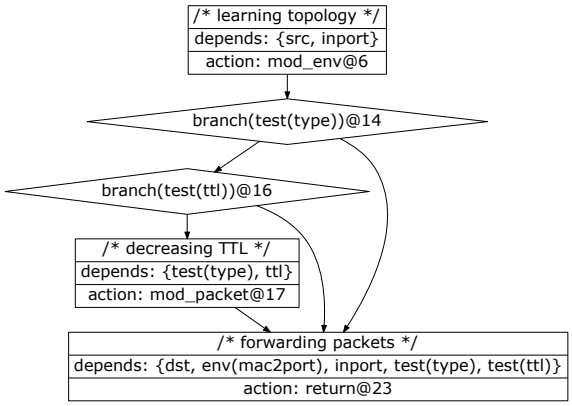

Fig. 5. The Xgraph for the example

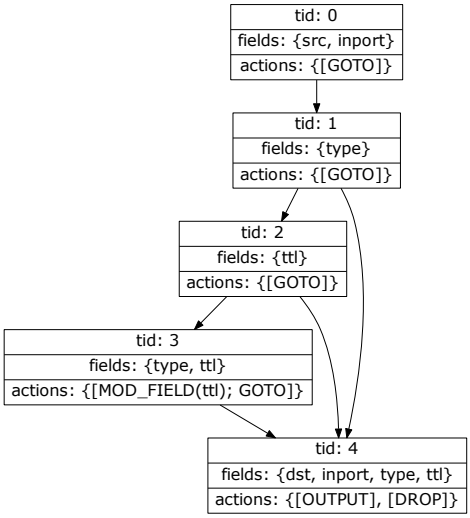

Fig. 6. Multi-stage pipeline

mismatches (and the corresponding PacketIn messages) for fresh ("src", "inport") pairs and to let the controller to receive the packet and execute the `mod_env()` action. The controller action `mod_packet()` is translated to the flow table action `[MOD_FLD; GOTO]`, which modifies the corresponding packet field and then jumps to the next table. The `return` action in the Xgraph is translated into a corresponding `OUT` or `DROP` flow table action.

We also generate a flow table for each diamond node. The match fields are the fields that the branch depends on. The action of the table is `[GOTO]`. The flow table can jump to different flow tables based on the values of the match fields.

Because we generate a separate flow table for each independent functionality, and our taint-analysis generates more fine-grained dependence relation than Maple, we can avoid enumerating values of unnecessary combination of match fields. For learning switches, we can avoid the $O(n^2)$ table entries in Fig. 4. The total number of possible entries of the pipeline in Fig. 6 is $O(n)$.

C. Optimizations

The generated pipeline can be further optimized. From Fig. 6 we can see the match fields in different flow tables have overlaps, which may cause these fields to be matched multiple times. If we can merge some of these tables into one, we can reduce the redundancy. Another advantage for the merge is to reduce the length of the pipeline, so that the execution time is reduced accordingly. However, we have to be careful with the merge to avoid unnecessary combination of different match fields, as shown in Fig. 4.

We do the optimization by first trying to merge the Xgraph nodes. We consider two situations. In the first situation, we merge adjacent square nodes n_1 and n_2 if the dependence set of one is a subset of the other, *i.e.* $n_1.depends \sqsubseteq n_2.depends$ or $n_2.depends \sqsubseteq n_1.depends$, where

$$L_1 \sqsubseteq L_2 \quad \stackrel{\text{def}}{=} \quad \forall l_1 \in L_1.\, \exists l_2 \in L_2.\, l_1 \sqsubseteq l_2$$
$$l_1 \sqsubseteq l_2 \quad \stackrel{\text{def}}{=} \quad l_1 = l_2 \vee (\exists \texttt{fld}.\, l_1 = \texttt{test(fld)} \wedge l_2 = \texttt{fld}).$$

As we explain before, since the label `test(fld)` refers to certain properties of the value of the field only, while `fld` relies on the exact value, the former contains less information than the latter and thus we let `test(fld)` $\sqsubseteq$ `fld`. The new node after the merge simply contains the union of the dependence set and the union of the actions in n_1 and n_2.

In the second situation, we merge the diamond node b and the square nodes n_1 and n_2 in its branches, if each branch has at most one square node (there could be an empty branch with no nodes, as shown in Fig. 5), and the dependence sets of these two branches overlap. More formally, we do the merge if $n_1.depends \sqsubseteq n_2.depends$ or $n_2.depends \sqsubseteq n_1.depends$, (if there is an empty branch, we can view it as a dummy node with empty dependence set). Then we merge the three nodes (b, n_1 and n_2) into n, where $n.depends = n_1.depends \cup n_2.depends \cup b.depends$. The action set of n is also a union of those of n_1 and n_2. For example, in Fig. 5 the action of decreasing TTL at line 17 can share a flow table with the sibling empty branch, so that we can merge the square node with its parent diamond node.

We repeat the above processes until there are no more nodes that can be merged. For the Xgraph in Fig. 5, we eventually merge the two diamond nodes and the square node for decreasing TTL. The new Xgraph has three square nodes only. The merge does not increase the number of table entries, because it does not introduce new combination of fields.

Then we translate the merged Xgraph into a forwarding pipeline. For the Xgraph in Fig. 5, after merging we get an optimized pipeline of flow tables shown in Fig. 7. In addition to the table layout, we also show some table entries to help understanding how this pipeline works (the generation of table entries is explained in the next section). Table 0 depends only on "inport" and "src", and the action is to jump to Table 1 directly. Table 1 depends on the exact values of "ttl", decrements it (if the value is greater than 1) and jumps to Table 2, which then either drops the packet or send it out depending on whether "ttl" is 1 or not.

Flow Table 0:

priority	in_p	src	action
1	1	h_1	`GOTO(1)`
1	2	h_2	`GOTO(1)`
...	...	...	...

Flow Table 1:

priority	type	ttl	action
1	0x800	64	`MOD_FLD(ttl,63); GOTO(2)`
...	...	...	...
1	0x800	1	`GOTO(2)`

Flow Table 2:

priority	in_p	dst	type	ttl	action
5	1	h_2	0x800	1	`DROP`
3	2	h_1	0x800	*	`OUT[1]`
3	2	h_3	0x800	*	`OUT[1,3,4...]`
...	...	...	...	...	...

Fig. 7. Optimized flow table layout of learning switches

However, Flow Table 2 may still contain more table entries than needed. When "ttl" is 1, the packet is always dropped, and the action is independent of "inport" and "dst". So ideally in the first row of the table, the first two columns should be "*" instead of the exact values. Unfortunately we cannot achieve this effect because the label set generated by the taint analysis for the return value `r` is not precise enough — at the end of the `if` statement we always merge the label sets of the two branches, leading to a conservative upper bound of the set of labels that may affect the value of `r`. Our idea to solve this problem is to introduce *dependent labels* to generate more precise dependence relation, which allows us to fill "*" into the table to further reduce the size of tables. We explain the details in the next section.

IV. Table Entry Generation and Dependent Labels

Following the idea of Maple, our system automatically populates flow table entries by discovering reusable network policy executions. When `f` is invoked, the runtime system tracks its execution, and records the trace consisting of events accessing packets and the environment data. In the leftmost column of Table II, we show a trace generated when `f` for learning switches is invoked with an IPv4 packet from h_1 to h_2, and the field of "ttl" is 1 (so the packet is dropped).

Maple derives the dependence between actions (such as dropping the packet) and the values of packet fields, from which the corresponding flow table entries are generated for the switches. (Maple actually also maintains historical execution traces, and merges all the traces into a trace tree. We follow the same mechanism, but omit the details of trace trees here to simplify the presentation.) This simple approach works for a monolithic flow table with large amount of unnecessary table entries, but it cannot be applied directly in our setting with multi-table pipelines. Since the linear-timed execution trace contains *all* the events produced by `f`, how do we fit it into multiple flow tables?

To solve this problem, we need to derive sub-traces from the global trace and assign a sub-trace to each flow table, from which we generate the corresponding table entries. During this process, we also take advantage of the runtime values to further refine the conservative dependence relation generated by the static taint analysis, as we pointed out at the end of Sec. III. This is achieved by extending the taint analysis algorithm with the more refined *dependent labels*. Below we explain in detail the dependent labels and the use of it together with runtime information to generate compact table entries.

TABLE II
Example of an execution trace

trace	sub-trace 2	instrumentation
read_inport(pkt) == 1;	$\surd$*	
read_packet(pkt, "dst") == h2;	$\surd$*	
read_packet(pkt, "src") == h1;		
mod_env(env, "mac2port", h1, 1);		
read_env(env, "mac2port", h2) == 0;	$\surd$*	
		branch(8, false);
test_equal(pkt, "type", 0x800) == true	$\surd$	
		branch(14, true);
		branch(16, false);
test_equal(pkt, "ttl", 1) == true	$\surd$	
return 0;	$\surd$	

A. Splitting the trace

With multi-table pipelines, we need to derive sub-traces from a full execution trace for individual flow tables. Given the Xgraph and the corresponding pipeline generated by the taint analysis, we can simply extract the events that correspond to those in the dependence set and the set of actions in each Xgraph node. For instance, from the last square node in Fig. 5 (which corresponds to the layout of Flow Table 2 in Fig. 7), we can easily find all the relevant events in the trace shown in Table II. We mark these events with $\surd$ (ignore the "*" for now), which form the sub-trace for Flow Table 2. With this sub-trace we can generate the first row in the Flow Table 2.

However, when "ttl" is 1, we drop the packet regardless of the values of "inport" and "dst", and the result of `read_env`. Therefore the three events marked with "$\surd$*" in Table II should be removed from the sub-trace, so that we can fill "*" in the corresponding matching fields ("inport" and "dst") of the flow table entry. As we explained at the end of Sec. III, the conservative label set for the return value `r` should be blamed for the inclusion of the three events in the sub-trace. Knowing the runtime value of "ttl", we should be able to infer that the action in this case only depends on the testing of "type" and "ttl". But how do we take advantage of the runtime value to generate the more precise dependence relation?

B. Dependent labels

To solve the problem above, we extend the labeling system in the taint analysis with dependent labels. Labels and label sets are defined below:

(*Labels*) l ::= `fld` | `test(fld)` | ... | $L_{@N}\,?\,L_1 : L_2$
(*LabSets*) L ::= $\{l_1, \ldots, l_n\}$

What's new here is the dependent label in the form of $L_{@N}\,?\,L_1 : L_2$. It is used to label variables at the end of the statement **if**(B) C_1 **else** C_2, starting at line number N in the code. L is the label set for B, and L_1 and L_2 are the label sets assigned to the variables in the two branches respectively. As

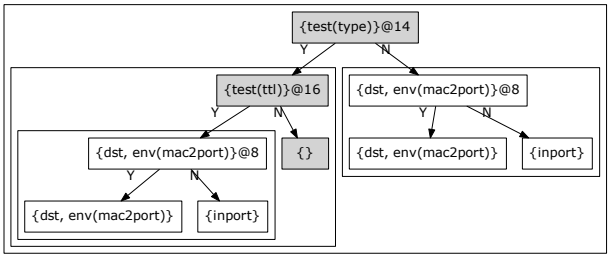

Fig. 8. Dependent label of `r`

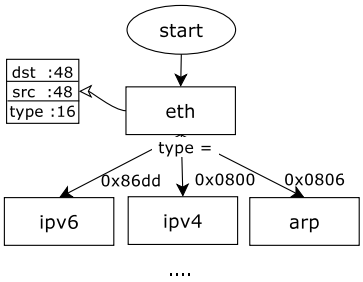

Fig. 9. A parse graph

we explain in Sec. III-A, the standard taint analysis algorithm merges the sets L_1 and L_2 of the two branches when we reach the end of the `if` statement. We generate the dependent label instead, which is a more refined view, saying the label set depends on the value of the branch condition B containing information in L. If B holds the label set is L_1, otherwise L_2.

As an example, the `if` statement at line 8 in Fig. 3 assigns different values to `r`. We define the label set of `r` as $L_{r1} = \{l_{r1}\}$, where l_{r1} is a dependent label defined as $L_{\texttt{port}_{@8}} ? L_{\texttt{port}} : \{\texttt{inport}\}$, and $L_{\texttt{port}}$ is the label set for `port` at line 8, *i.e.* $L_{\texttt{port}} = \{\texttt{dst, env(mac2port)}\}$. Similarly, the `if` statement at line 16 leaves `r` untouched in one branch, and assigns 0 to it in the other. So the label set of `r` is either the same as before (*i.e.* L_{r1}), or empty set (if it is reset to 0, which contains no source information). We define the label set as $L_{r2} = \{l_{r2}\}$, where the dependent label l_{r2} is defined as $\{\texttt{test(ttl)}\}_{@16} ? L_{r1} : \{\}$. We can also give an intuitive graphic view of dependent labels. Fig. 8 shows the dependent label assigned to `r` at the end of the `if` statement at line 14 (which is also the final label for `r`).

In the learning switches example, the extension of taint analysis with dependent labels affects only the labeling of `r`, whose value depends on the `if` statement branches taken at runtime. Although it gives us a more refined view of information flow, the dependent label does not affect our algorithm to generate pipelines shown in Sec. III because statically we do not know which branch will be taken and have to maintain a conservative view as before. The generated pipeline is the same as we have shown before.

However, we can know which branch is taken at runtime, then we can derive more refined label sets from the dependent label. This means we can find more precise sub-traces for different flow tables. However, to take advantage of dependent labels, we need to instrument the code and the traces to remember which `if` statement branch is taken at runtime.

C. Code and trace instrumentation

We instrument the `if` statement in the code of the network policy `f` to generate events showing which branch is taken at runtime. For the example execution trace shown in Table II, we have three more events after the instrumentation, shown in the right column.

With this instrumented trace and the dependent labels, we can derive more compact sub-traces. In the learning switches example, the label set for the `return` command is the same with the set for the variable `r`, as shown in Fig. 8. From the instrumented trace we know the branches taken for the `if` statements at lines 14 and 16, therefore we know the relevant labels are only those shown on the grey path in Fig. 8. This allows us to remove the events labeled with "*" in the middle column of Table II. With this more compact sub-trace, in Flow Table 2 of Fig. 7 we can fill "*" into the first two fields of the first row, which results in not only compact flow tables, but also much less mismatches and PacketIn messages.

V. Packet Parsing

POMP supports protocol oblivious programming. It uses dotted paths of packet header fields to access packets. For instance, the API call of `read_packet(pkt, "eth.ipv4.ttl")` in Fig. 3 accesses the "ttl" field of the ipv4 packet. In the dotted string path, "eth" and "ipv4" represent protocol names and the last substring "ttl" is the field name. The string path can be arbitrary for any user customized protocols. It is not limited to a predefined set of protocols.

To achieve this, the programmers provide protocol specifications, from which a parse graph is generated. The parse graph serves both as a P4 parser, which can be compiled and deployed on P4 switches, or as an input to the runtime systems to parsing packets at the control plane.

A. Protocol specification

As shown in the learning switches example in Fig. 2, a protocol specification consists of a set of header definitions and a starting protocol declaration. A header definition has the following parts.

- **header name:** name of the protocol, *e.g.* `eth` and `ipv4`.
- **fields:** the layout of the header, consisting of a sequence of field names and the corresponding lengths in bits.
- **next headers:** the name of the next header, which could be conditional upon the value of certain field. For instance, if the `type` of `eth` is `0x800`, the next header is `ipv4`; if it is `0x86dd`, the next header is `ipv6`.

The starting protocol declaration shows which protocol is the outmost one, *i.e.* the one that needs to be parsed first. It is `eth` in our example.

B. Parsing graphs and packet parsing

From the protocol specification one can easily derive a parse graph as in P4. As shown in Fig. 9, the parse graph is a directed graph, where nodes represent protocols declared in the specification, and edges represent the ordering of protocols. The edges are generated from the `next` block of the specification, and the labels represents the corresponding values of the field that one uses to determine the next protocol. The start node is specified by the `start` clause.

Both the runtime system of POMP in the control plane and the data plane may use the parse graph for packet parsing.

1) Parsing in the control plane: The POMP runtime takes the parse graph and a raw packet as input, and outputs a map from field names to `(offset, length)` tuples, which can be used to extract values of fields from the raw packet. As explained before, we use dotted notations (*e.g.* "eth.ipv4.ttl") for field names to avoid name clash. The parsing starts with the start node and traverses the parse graph. During the process it also maintains a current pointer pointing to the raw packet, which is used to extract the value of certain field (*e.g.* `type`) to decide the next node to transit to.

The parsing process is invoked on receiving a `PacketIn` message. In the network policy `f(pkt, env)`, when packet fields are accessed using string names, we lookup the map generated by the parser to find the corresponding `(offset, length)` tuples, using which we can locate the field and do the read/write of the field.

Since the switch of POF uses `(offset, length)` tuples in the flow table to access packet fields, the runtime system also needs to translate field names into `(offset, length)` when installing flow table entries for POF switches.

2) Parsing in the data plane: POF and P4 uses different strategies to achieve protocol independence. In POF, no parser is required in the data plane. We use `(offset, length)` generated by the control plane parser to specify packet fields.

On the other hand, our parsing graph also serves as a P4 parser, which can be fed into the P4 compiler to generate a data plane parser. All incoming packets must be parsed at the data plane before being sent to the pipeline.

VI. Evaluation

We compare the effects of different techniques we use, and demonstrate that POMP generates high-quality forwarding pipelines. The evaluation setting is the following:

- Hardware: Intel Core I7 4712MQ 2.3GHz, 8G RAM.
- Network Emulator: Mininet [9].
- Switch: P4 bmv2 [10] / Open vSwitch [11] 2.6.2.
- Topology: Single 10 port switch with n hosts.

To show the effects of different techniques, POMP provides three tunable options:

- "multi" / "single": Enabling or disabling the generation of multi-table pipelines. The "single" option turns off the taint analysis and generates a single monolithic flow table.
- "opt" / "nonopt": Enabling or disabling the pipeline optimization shown in Sec. III-C.
- "dep" / "nondep": Enabling or disabling the use of dependent labels explained in Sec. IV-B.

We run the learning switches example using different options and different n (*i.e.* the number of hosts). We record the number of flow tables. Then we use the "pingall" test provided by Mininet to send packets among hosts. We record the total number of flow table entries and number of packet-in events generated during the test, and the execution time.

A. The quality of pipelines

We use the number of flow tables and total number of table entries to evaluate the quality of pipelines. The result is shown in Table III, from which we can see:

- The total number of entries grows quadratically in single-table, while it grows linearly in multi-table pipelines.
- POMP under the multi-opt-dep setting uses up to 47x fewer number of entries than single-table.
- Turning on "opt" reduces the number of tables from 5 to 3, while the number of entries is about the same. This is because the reduced flow tables (with tid 1 and 2 in Fig. 6) contain only 3 entries in total.
- Dependent labels (turning on "dep") save 66% - 80% entries, comparing the last two rows of Table III.

B. Performance

We use the execution time and the number of PacketIn events to evaluate the performance. The result is shown in Tables IV and V. We also compare our learning switches with the built-in learning functionality of Open vSwitch, a production quality soft switch. Its profile is shown as "ovsk".

From Tables IV and V we can see:

- Multi-table pipeline causes significantly less PacketIn messages than single table, and is up to 134x faster.
- Multi-table pipeline is 20% - 40% slower than "ovsk". This is unsurprising because "ovsk" is a pure data plane application, and there is no controller-switch communication (*i.e.* no PacketIn messages).
- Dependent labels help reducing the number of PacketIn events, thus improve the performance.

Note that turning on "opt" alone is not helpful to reduce the number of PacketIn messages. Since the pingall time is spent mostly by PacketIn messages, "opt" introduces no obvious decrease of pingall time either.

VII. Related Work

In Sec. I we have pointed out the limitations of many representative languages, including NetKAT [6], NetCore [7], Maple [5] and P4 [1]. Here we compare our work with some more SDN programming languages.

Magellan [12] also derives and populates multi-table pipelines from algorithmic policies, but it takes very different approaches from ours. It starts from extremely fine-grained pipelines by treating every statement as a flow table. Then it recognizes the so called compact operations and merges their corresponding flow tables with others, so that the number of flow tables is reduced without significantly increasing the

TABLE III
THE QUALITY OF PIPELINE

profile	# tables	# entries					
		n = 10	n = 20	n = 30	n = 40	n = 50	n = 100
single	1	121	441	961	1681	2601	10201
multi-nonopt-dep	5	43	63	83	103	123	223
multi-opt-nondep	3	120	232	350	460	570	1120
multi-opt-dep	3	40	60	80	100	120	220

TABLE IV
PERFORMANCE: PINGALL TIME

profile	pingall time (s)					
	n=10	n=20	n=30	n=40	n=50	n=100
single	0.60	6.1	28	78	181	2831
ovsk	0.10	0.60	1.3	2.4	3.7	15
multi-nonopt-dep	0.20	0.78	1.8	3.1	4.9	20
multi-opt-nondep	0.54	2.1	4.5	7.8	12	47
multi-opt-dep	0.19	0.76	1.7	3.1	4.8	19

TABLE V
PERFORMANCE: NUMBER OF PACKETIN

profile	# packet in					
	n=10	n=20	n=30	n=40	n=50	n=100
single	117	423	939	1643	2549	10099
multi-nonopt-dep	46	74	79	92	111	223
multi-opt-nondep	124	258	351	452	562	1134
multi-opt-dep	47	72	80	93	112	221

number of table entries. Also, unlike POMP that generates flow table entries reactively at runtime, Magellan generates table entries proactively by enumerating all possible executions of the network policy before its deployment. Moreover, Magellan provides no mechanisms for protocol oblivious programming.

Concurrent NetCore [13] extends NetCore [7] to support multi-tables, but it requires the programmer to manually define the layout of flow tables instead of generating it automatically from high-level policies as we do in POMP.

Stateful NetKAT [14] and SNAP [15] extend NetKAT [6] and NetCore [7] respectively to support persistent states in emerging SDN data plane. They are for quite different purposes (to take advantage of the switch-local computation power) and do not handle multi-table pipelines.

P4 Runtime [16] provides a way for control plane to control P4 switches. The programmer can use P4 Runtime APIs to install flow table entries at runtime. In POMP, the programmer doesn't use P4 Runtime APIs. The switch abstraction layer (see Fig. 1) accepts entries generated by the runtime of POMP, then calls P4 Runtime APIs to install entries for P4 switches.

VIII. CONCLUSION

POMP is a general purpose SDN programming environment supporting both protocol oblivious programming and automatic multi-table pipelining. It extends the C language with a set of APIs and a runtime system to allow programmers to write algorithmic, data-plane agnostic network policy for SDN programming. It then applies static taint analysis techniques to automatically infers compact and efficient multi-table pipelines from the high-level network policy. It also introduces a novel notion of dependent labels to refine the label sets generated by the taint analysis. The dependent labels are used at the runtime for flow table entry generation to further reduce the number of table entries. The evaluation results show that POMP generates high-quality forwarding pipelines. POMP pipeline for learning switches uses up to 47x fewer number of table entries, and is up to 137x faster than single table.

Note that, although the learning switches example is used throughout the paper to demonstrate the key ideas, POMP is a general purpose language for SDN programming. In addition to these APIs used in learning switches, there are more provided, including those for environment management. The set of POMP APIs is as expressive as those in Maple.

REFERENCES

[1] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[2] H. Song, "Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane," in *Proc. of HotSDN*, 2013, pp. 127–132.

[3] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *Proc. of SIGCOMM*, 2013, pp. 99–110.

[4] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford, "Pisces: A programmable, protocol-independent software switch," in *Proc. of SIGCOMM*, 2016, pp. 525–538.

[5] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak, "Maple: Simplifying SDN programming using algorithmic policies," in *Proc. of SIGCOMM*, 2013, pp. 87–98.

[6] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "Netkat: Semantic foundations for networks," in *Proc. of POPL*, 2014, pp. 113–126.

[7] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," in *Proc. of POPL*, 2012, pp. 217–230.

[8] S. Hunt and D. Sands, "On flow-sensitive security types," in *Prof. of POPL*, 2006, pp. 79–90.

[9] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proc. of Hotnets-IX*, 2010, pp. 19:1–19:6.

[10] "P4 behavioral model," https://github.com/p4lang/behavioral-model, accessed: 2017-07-31.

[11] "Open vSwitch," http://openvswitch.org/, accessed: 2017-07-31.

[12] A. Voellmy, S. Chen, X. Wang, and Y. R. Yang, "Magellan: Generating multi-table datapath from datapath oblivious algorithmic SDN policies," in *Proc. of SIGCOMM*, 2016, pp. 593–594.

[13] C. Schlesinger, M. Greenberg, and D. Walker, "Concurrent netcore: From policies to pipelines," in *Proc. of ICFP*, 2014, pp. 11–24.

[14] J. McClurg, H. Hojjat, N. Foster, and P. Černý, "Event-driven network programming," in *Proc. of PLDI*, 2016, pp. 369–385.

[15] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "Snap: Stateful network-wide abstractions for packet processing," in *Proc. of SIGCOMM*, 2016, pp. 29–43.

[16] "P4 runtime," https://p4.org/p4-runtime/, accessed: 2017-12-27.