# HIGH-LEVEL ABSTRACTIONS FOR PROGRAMMING NETWORK POLICIES

Yifei Yuan

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial
Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2016

Rajeev Alur
Zisman Family Professor of Computer and Information Science
Supervisor of Dissertation


Boon Thau Loo
Associate Professor of Computer and Information Science
Co-Supervisor of Dissertation


Lyle Ungar
Professor of Computer and Information Science
Graduate Group Chairperson

Dissertation Committee

Andreas Haeberlen, Associate Professor of Computer and Information Science (Chair)

Jonathan M. Smith, Professor of Computer and Information Science

Insup Lee, Cecilia Fitler Moore Professor of Computer and Information Science

Sanjit A. Seshia, Associate Professor of Electrical Engineering and Computer Sciences,
University of California, Berkeley

HIGH-LEVEL ABSTRACTIONS FOR PROGRAMMING NETWORK POLICIES

COPYRIGHT

2016

Yifei Yuan

*To my parents and wife.*

# Acknowledgments

First of all, I would like to thank my advisors Professor Rajeev Alur and Professor Boon Thau Loo. Rajeev provided me with well-framed training in formal methods and Boon introduced me into the area of computer networks. With their mentoring, I was able to find this dissertation topic and complete this work. I also learned from Rajeev and Boon essential academic skills; both of them provided me with constructive advice on shaping my research ideas, solving the research problems and presenting the results.

I would also like to thank my dissertation committee, including Professor Andreas Haeberlen, Profesor Jonathan M. Smith, Professor Insup Lee and Professor Sanjit A. Seshia. They offered me insightful comments and detailed feedbacks on the dissertation, which helped me significantly on improving it.

I also want to give credit to my collaborators who have helped me on this dissertation work. Dong Lin, Shanni Xi, Harsh Verma and Siri Anil worked with me on the NetEgg project, and Dong Lin, Ankit Mishra, Sajal Marwaha collaborated with me on the NetQRE project. Their contribution was remarkably valuable in the implementation and the experiments of both projects.

In addition, I feel grateful for my family's unconditional support on my career pursuit. Their love always encourages me and inspires me in my life. I specially want to thank my wife with whom I spent an enjoyable and enriched time during my Ph.D. years at Penn.

Moreover, I am thankful for the friends I made throughout these years at Penn: Mingchen Zhao, Zhepeng Yan, Ang Chen, Chen Chen, Loris D'Antoni, Mukund Raghothaman, Abhishek Udupa, Salar Moarref, Nimit Singhania and many others. In particular, I should thank Ang Chen for his help on submitting the hard copy of this dissertation for me while I am not able to do this in person.

ABSTRACT

HIGH-LEVEL ABSTRACTIONS FOR PROGRAMMING NETWORK POLICIES

Yifei Yuan

Rajeev Alur

Boon Thau Loo

The emergence of network programmability enabled by innovations such as active networking, SDN and NFV offers tremendous flexibility to program network policies. However, it also poses a new demand to network operators on programming network policies. The motivation of this dissertation is to study the feasibility of using high-level abstractions to simplify the programming of network policies.

First, we propose scenario-based programming, a framework that allows network operators to program stateful network policies by describing example behaviors in representative scenarios. Given these scenarios, our scenario-based programming tool NetEgg automatically infers the controller state that needs to be maintained along with the rules to process network events and update state. The NetEgg interpreter can execute the generated policy implementation on top of a centralized controller, but also automatically infers flow-table rules that can be pushed to switches to improve throughput. We study a range of policies considered in the literature and report our experience regarding specifying these policies using scenarios. We evaluate NetEgg based on the computational requirements of our synthesis algorithm as well as the overhead introduced by the generated policy implementation. Our results show that our synthesis algorithm can generate policy implementations in seconds, and the automatically generated policy implementations have performance comparable to their hand-crafted implementations. Our preliminary user study results show that NetEgg was able to reduce the programming time of the policies we studied.

Second, we propose NetQRE, a high-level declarative language for programming quantitative network policies that require monitoring a stream of network packets. Based on a novel theoretical foundation of parameterized quantitative regular expressions, NetQRE integrates regular-expression-like pattern matching at flow-level as well as application-level

payloads with aggregation operations such as sum and average counts. We describe a compiler for NetQRE that automatically generates an efficient implementation from the specification in NetQRE. Our evaluation results demonstrate that NetQRE is expressive to specify a wide range of quantitative network policies that cannot be naturally specified in other systems. The performance of the generated implementations is comparable with that of the manually-optimized low-level code. NetQRE can be deployed in different settings. Our proof-of-concept deployment shows that NetQRE can provide timely enforcement of quantitative network policies.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Recent years have seen the emergence of programmability in computer networks enabled by a wave of revolutionary innovations, including active networking [85], SDN [65], and NFV [30]. For example, software-defined networking (SDN) [65] promises rapid service provisioning and agile network management by decoupling the control plane and the data plane in networks. A logically centralized controller in SDN offers a global view of the network and application programming interfaces (API) to enforce a variaty of network policies. Furthermore, network functionality virtualization (NFV) [30] aims at shifting network functions from a range of dedicated hardware to software deployed on general-purpous computers. By programing network functions in software, NFV promises low cost and high flexibily in deploying and managing network services. Not even mentioning previous proposed active networking

This new wave of innovations offers network operators tremendous flexibility to program custom network policies, ranging from traffic steering [14, 46, 90] to dynamic access control [55, 71], from dynamic service chaining [39, 74] to network function migration [42].

While network programmability promises the aforementioned benefits, it poses a higher (if not totally new) demand to network operators on programming. In order to thoroughly understand the innovation trend and the challenges that come with it, in the fall of 2015 we conducted informal interviews with 101 network operators, architects and engineers from universities, telcos, service providers and router vendors, in the NSF I-Corps program [7]. We made two observations from the interviews. First, more than 80% interviewees agreed

that network operators will need to program networks in the future. Second, they might lack the required programming skills to program reliable networks. Our observations are consistent with the online discussions we studied [10, 24, 25], which suggest that programming is becoming an essential skill set for network operators, and it requires network operators to have a deeper understanding on network programming.

To ease the job of programming network policies, in this disseratation, we study the feasibility of using high-level abstractions for programming a range of network policies.

First, we consider stateful network policies which make decisions on each packet based on the state it maintains and also update the state accordingly. As an example, consider the following policy.

**Example 1 (Learning switch).** Consider the example where a network operator wants to program a learning switch policy supporting migration of hosts on top of the controller for a single switch. The learning switch learns incoming ports for hosts. For an incoming packet, if the destination MAC address is learned, it sends this packet out to the port associated with the destination MAC address; otherwise it floods the packet. To support migration of hosts, the learning switch needs to remember the latest incoming port of a host.

To implement this policy, a network operator first has to manually maintain the state indicating whether a port has been learned for a MAC address and also the mapping between the hosts and incoming ports. Second, the network operator has to implement the logic to update the state as each packet arrives. Third, in order to improve the throughput, the network operator may need to manually update the flow table rules on the switch as the state changes, which is nontrivial and error-prone [88]. As a result, it may be difficult and time-consuming to implement this policy (see Section 2.6 and 2.7).

On the other hand, illustrating how the policy behaves seems much simpler and more intuitive. For example, one can describe the behavior of this policy in a scenario using an example trace of packets together with the actions that should be applied to each packet in the trace. An interesting question is that can we *synthesize* the desired policy implementation automatically from the example behaviors.

Second, we consider the network policies which may involve quantitative metrics additionally. As an example, consider the following policy.

**Example 2 (Voice-over-IP monitoring).** Consider the policy where an enterprise mon-

itors the usage of Voice-over-IP (VoIP) for each user, and alert the user whose usage is significantly higher than the average usage over all users. We use the Session Initiation Protocol (SIP) used in VoIP applications as our example. Typically, a call based on SIP consists of three phases, *init*, *call*, and *end*. In the *init* phase, the caller and the callee exchange messages (e.g. call ID, user name, the connection channel) in order to set up the call session. Once established, the *call* phase allows VoIP data to be transmitted between the caller and callee using the channel defined in the first phase. Finally, either side can end the call as shown in the *end* phase.

To analyze the SIP call in the midst of network traffic from all types of protocols, the protocol analyzer is required to (1) identify all SIP traffic traversing the network, (2) separate the SIP traffic into different sessions for different caller/callee pairs, (3) group packets within each session into phases (init, call, and end), and then perform a count of bytes within the second (call) phase. Again, the network operator may face a set of difficulties in manually maintaining the state (e.g. the state of SIP, the VoIP traffic count of each call) and implementing the logic to update the state.

Unlike the previous example, it is not clear how to describe this policy using its example behaviors. Instead, we explore the feasibility of specifying the policy in a high-level declarative fashion. The challenge is to design a reasonably expressive specification language to capture a wide range of such quantitative policies, together with the compilation techniques in order to compile the specifications into efficient implementations.

Figure 1.1 illustrates the overview of our framework. To program a network policy, the network operator simply *describes* the network policy in a declarative way using high-level abstractions. The network operator does not need to implement low-level details of the policy, such as what state to maintain and how to update it. Given the high-level description of the network policy, a tool (e.g. a compiler, a synthesizer) automatically generates the correponding low-level policy implementation that can run efficiently on the network. As part of this generation, the tool automatically infers the state that needs to be maintained from the high-level description, and generates the code that is necessary to update the state.

In particular, this dissertation proposes the following two high-level programming abstractions.
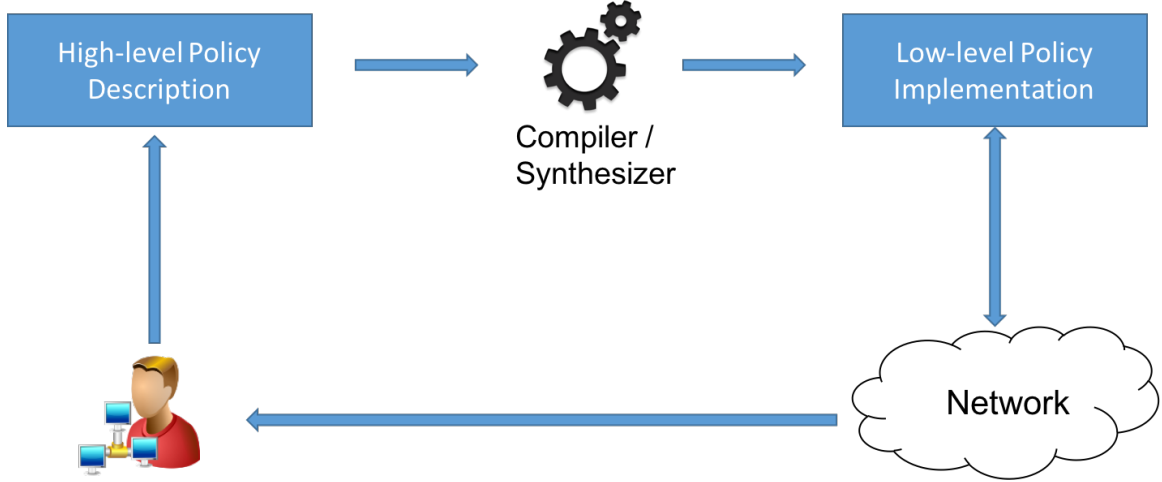
Figure 1.1: High-level programming framework.

- First, we propose scenario-based programming, a framework that allows network operators to program stateful SDN policies, such as the one described in Example 1, by describing its *example behaviors* in representative scenarios.

  We have developed the NetEgg tool, including a synthesis algorithm and an interpreter for executing policies. Given the scenarios as input, our synthesizer automatically generates a controller program that is consistent with example behaviors, including inferring the state that needs to implement the network policy, and rules for processing packets and updating the state. The interpreter executes the generated policy program for incoming network events on the controller, as well as infers rules that can be pushed onto switches.

  We validate the NetEgg tool by synthesizing SDN programs that use the POX controller directly from examples. Our tool is agnostic to the choice of SDN controllers, and can also be used in non-SDN settings. We demonstrate that using our approach, we are able to synthesize a range of network policies using a small number of examples in seconds. The synthesized controller program has low performance overhead and achieves comparable performance to equivalent imperative programs implemented manually in POX. Moreover, the example scenarios are concise compared to equivalent imperative programs. Our preliminary user study results show that NetEgg is intuitive to learn and use, and can reduce the programming time of network policies.

- Second, we propose NetQRE, a specification language and toolkit for quantitative network policies, such as the example policy in Example 2.

  NetQRE is declarative language for monitoring network packet streams based on a novel theoretical foundation of *parameterized quantitative regular expressions* (PQRE). PQRE extends prior work on QRE [18], which provides a formal foundation of combining traditional regular expressions with numerical computations. NetQRE integrates regular-expression-like pattern matching at flow-level and application-level payloads with aggregation operations such as sum and average counts.

  We developed a compiler that can automatically generate efficient NetQRE implementations. As part of the compilation process, the compiler automatically infers the state that needs to be maintained for NetQRE programs, and optimizes the compiled imperative code. A NetQRE runtime then executes the generated code efficiently.

  We have developed a NetQRE prototype which we evaluate over a range of quantitative network policies. As a proof-of-concept deployment, we consider an evaluation scenario where the NetQRE runtime taps into mirrored traffic within SDN switches, and is integrated with an SDN controller. Our evaluation results demonstrate that NetQRE can express a wide range of quantitative network policies that cannot be naturally specified in other systems, results in performance that is comparable with optimized manually-written low-level code and is significantly more efficient compared to monitoring solutions based on Bro [73] and OpenSketch [91].

In the rest of the dissertation, we first elaborate NetEgg in Chapter 2, followed by the description of NetQRE in Chapter 3. We discuss related work in Chapter 4. Finally, we conclude the dissertation and discuss potential research directions related to this topic in Chapter 5.

# Chapter 2

# NetEgg: Scenario-based Programming for SDN Policies

In this chapter, we decribe the tool NetEgg. NetEgg is a scenario-based programming framework that allows network operators to program network policies by describing representative example behaviors. Given these scenarios, the synthesis algorithm automatically infers the controller state that needs to be maintained along with the rules to process network events and update state. The NetEgg policy interpreter can execute the generated policy implementation on top of a centralized controller, but also automatically infers flow-table rules that can be pushed to switches to improve throughput.

The rest of this chapter is organized as follows. First, we provide an overview of NetEgg in Section 2.1. We use the learning switch policy as an example to illustrate the use of NetEgg. Second, in Section 2.2 we describe the scenario-based programming framework, which allows network operators to specify network policies using example behaviors in scenarios. Third, in Section 2.3 we describe the synthesizing algorithm that takes the scenarios as input, and automatically generates a controller program that is consistent with example behaviors, including inferring the state that is needed to implement the network policy and rules for processing packets and updating states. Section 2.4 discusses how the interpreter executes the generated policy program for incoming network events on the controller, as well as infers rules that can be pushed onto switches. We show how to use NetEgg to program a wide range of policies in Section 2.5. In Section 2.6, we present our evaluation results,

6

which show that the synthesized policy implementation has low performance overhead and achieves comparable performance to equivalent imperative programs implemented manually in POX. We present our preliminary user study result in Section 2.7. Finally, we end this chapter with a discussion in Section 2.8.

This chapter is based on the work orginially published in [93, 94].

## 2.1 Overview of NetEgg

Figure 2.1 provides a high-level overview of NetEgg. The network operator describes example behaviors about the desired network policy in representative scenarios to NetEgg. These scenarios can be expressed either in network timing diagrams using the graphic user interface (see Figure 2.17), or in a configuration language, which we will describe in Section 2.2. Given these scenarios, NetEgg first checks whether there exist conflicts among the scenarios. If two scenarios conflict with one another, NetEgg displays the conflict to the network operator. After the operator resolves all conflicts, NetEgg tries to generate a policy described in the scenarios.

The generated policy uses a set of *state tables* and a *policy table*. State tables maintain the state used to remember the history of a policy execution. The policy table dictates the actions for processing incoming network events and updates to state tables for various cases.

When executing the policy, the interpreter, sitting on top of the controller, looks up the policy table for incoming network events (e.g. packetin, connectionup and other events), which will determine state table updates and actions to be applied to the network events. Moreover, NetEgg automatically infers rule updates to the data plane from current state of the policy execution, thus reducing controller overhead and network delay.

While NetEgg is general to handle any network events, we focus on packetin events in this paper in order to simplify our presentation.

### 2.1.1 Illustrative Example

To illustrate the use of NetEgg, let us revisit the learning switch example introduced in Chapter 1. Recall that a network operator wants to program a learning switch policy
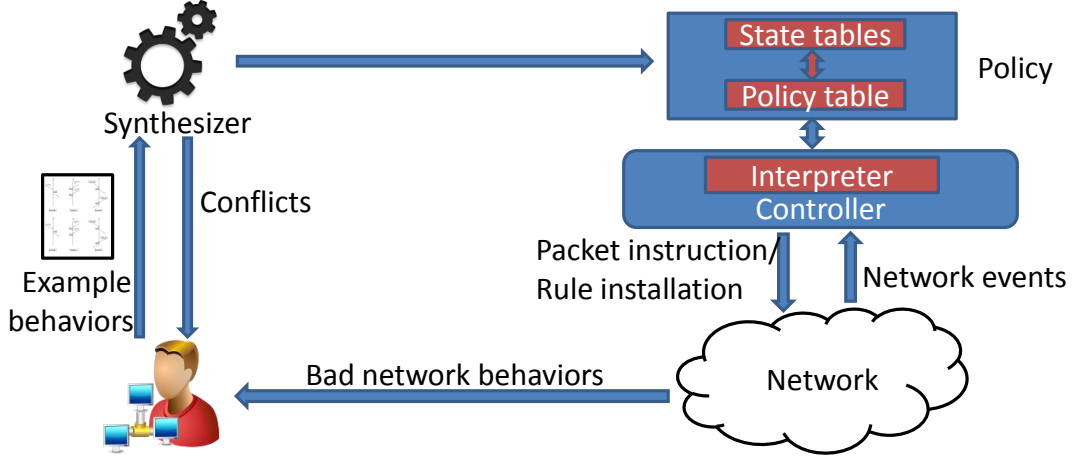
Figure 2.1: NetEgg architecture.

supporting migration of hosts on top of the controller for a single switch. The learning switch learns incoming ports for hosts. For an incoming packet, if the destination MAC address is learned, it sends this packet out to the port associated with the destination MAC address; otherwise it floods the packet. To support migration of hosts, the learning switch needs to remember the latest incoming port of a host.

To program the policy, the network operator simply describes example behaviors of the policy in representative scenarios, in the form of network timing diagrams. Figure 2.2 shows a scenario described by network operators.
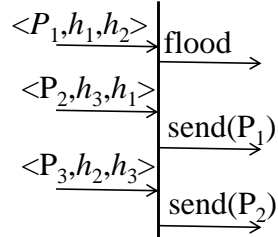


Scenario 1

Figure 2.2: A scenario describing the learning switch. In the scenario, a packet is denoted by a 3-tuple: ⟨incoming port, source MAC, destination MAC⟩.

**The scenario.** In this scenario, the network operator describes example behaviors of

the policy using three packets. The first packet arriving on port $P_1$ with source MAC address $h_1$ and destination MAC address $h_2$ is flooded by the switch, since no port has been learned for $h_2$. The second packet from $h_3$ to $h_1$ should be sent directly to the port $P_1$, according to the port learned from the first packet. The third packet from $h_2$ to $h_3$ should be sent to the port $P_2$, since the second packet indicts that $h_3$ is associated with port $P_2$. Note that instead of using real port numbers and MAC addresses in the packet, the network operator uses *variables* for each field. The variables stand for a variety of concrete values.

Given this scenario, NetEgg automatically synthesizes the desired program. The synthesized program can be executed on the SDN controller, as well as install flow table rules onto switches. As part of the program generation, NetEgg automatically generates the data structures and code necessary to implement the policy.

Network operators may further test the synthesized program using existing verification and testing techniques, and refine the program if needed. As part of refinement, network operators simply illustrate new scenarios (e.g. obtained from counter examples) to NetEgg, and NetEgg automates the refinement by synthesizing a new program from the new set of scenarios. We will demonstrate more use cases in Section 2.5.

Using the learning switch example, in the following subsections, we first describe the state tables that are generated by our tool, before describing the policy.

### 2.1.2 State Tables

In our example, the learning switch needs to remember whether a port is learned for a MAC address, and if learned, which port is associated with the MAC address. Hence, the generated policy maintains a state table $ST$, which stores a state and a value (in this case a port number) for each MAC address. An example of the snapshot of $ST$ is shown below.

| MAC | state | value |
|-----|-------|-------|
| A | 1 | 2 |
| OP... | ... | ... |

Our tool automatically derives the facts that for a given MAC address *macaddr*, the state of *macaddr* in $ST$ is either 0 or 1, indicating that the port associated with *macaddr* is unknown yet, or learned, respectively. $ST$ also stores a port for MAC addresses with state

1. Initially, the program assigns all states in the table to be 0. The program accounts for two cases: 1) When the destination port is unknown, it floods the packet through all ports; 2) When the incoming packet's destination port is known, it sends the packet out through the port associated with the destination MAC address. In both cases, the state associated with the source MAC address is set to be 1, and the incoming port for the source MAC address is remembered.

### 2.1.3 Policy Tables

The state table is manipulated by rules implementing the desired policy. These rules are captured in a policy table, as shown in Table 2.1 for the learning switch example. We delay the discussion of its generation to Section 2.3.

| match | test | actions | update |
|-------|------|---------|--------|
| * | $ST(\text{dstmac}).\text{state}=0$ | flood | $ST(\text{srcmac}):=(1, \text{port})$ |
| * | $ST(\text{dstmac}).\text{state}=1$ | send($ST(\text{dstmac}).\text{value}$) | $ST(\text{srcmac}):=(1, \text{port})$ |

Table 2.1: The policy table for the learning switch.

The policy table contains two *rules*, represented as the two rows in the table, corresponding to the two cases in the program described above. Every rule has four components: match, test, actions and update. The match specifies the packet fields and corresponding values that a packet should match. In this example, no matches need to be specified and we use $*$ to denote the wildcard. The test is a conjunction of checks, each of which checks whether the state associated with some fields in a state table equals a certain value. For example, the test in the second rule has one check $ST(\text{dstmac}).\text{state}=1$, which checks whether the state associated with the dstmac address of the packet is 1 in $ST$. The actions define the actions that are applied to matched packets. In this example, the action in the first rule floods the matched packet to all ports and the action send($ST(\text{dstmac}).\text{value}$) in the second rule first reads the value (in this case, the port) stored in $ST$ for the dstmac address of the matched packet, and sends the packet to that port. The update is a sequence of writes, each of which changes the state and value associated with some fields in a state table to certain values. For example, the write $ST(\text{srcmac}):=(1,\text{port})$ changes the state associated with the srcmac address of the packet to 1 in $ST$, and stores the value associated with the srcmac

10

address of the packet to the port of it.

## 2.1.4   Interpreter

The interpreter processes incoming packets at the controller using the policy table. The pseudocode of the interpreter is shown in Figure 2.3. The interpreter matches each incoming packet against each rule in the policy table in order. A rule is matched, if the packet fields match the match and all checks in the test of the rule are satisfied. The first matched rule applies actions to the packet, and state tables are updated according to the update of the rule. Moreover, NetEgg automatically infers the rules that can be installed on the data plane from the latest configuration of state tables. The corresponding function is update_flowtable in the pseudocode. We will describe policy execution in more detail in Section 2.4.

> **Input**: a packet $p$
> **for** $i = 1$ to $n$ **do**
>  **if** rule $r_i$ matches $p$ **then**
>   execute the actions and update of rule $r_i$ on $p$
>   update_flowtable($p$)
>   return
>  **end if**
> **end for**
> apply default actions to $p$

Figure 2.3: The interpreter pseudocode.

**Example.** Figure 2.4 shows an illustrative execution for the incoming packet trace in subfigure (a). Since the purpose of this example is to illustrate how a policy table is executed, we assume that every packet is processed on the controller. Initially, all states in the state table $ST$ are 0, and all values are $\perp$, meaning unknown, as shown in subfigure (b). The first packet $p_1$ is matched against each rule in Table 2.1 in order at the controller. The first matched rule is the first rule, since $p_1$ matches the match ($*$) and the state of the field dstmac of $p_1$ in $ST$ is 0, satisfying the check ($ST$(dstmac).state=0) in test of the rule. Therefore, the rule applies the action which instructs the switch to flood $p_1$, and updates the state table as in subfigure (c). The second packet $p_2$ in the trace matches the second rule in the policy table, since the state of its dstmac is 1. The program sends $p_2$ out to port

2, which is stored in the state table associated with MAC address A. Applying the update of the rule, we get the state table as in subfigure (d). The third packet $p_3$ matches the second rule in the policy table, and the updated state table remains the same and thus not shown here. The last packet $p_4$ suggests that the host with MAC address A has migrated to port 3. This packet matches the second rule in the policy table and gets sent to port 1. Subfigure (e) shows the state table after applying the update.

$p_1$:<port=2, srcmac=A, dstmac=B>
$p_2$:<port=1, srcmac=B, dstmac=A>
$p_3$:<port=2, srcmac=A, dstmac=B>
$p_4$:<port=3, srcmac=A, dstmac=B>

(a) An example packet trace

| MAC | state | value |
|-----|-------|-------|
| A | 0 | ⊥ |
| B | 0 | ⊥ |

(b) The initial state table

| MAC | state | value |
|-----|-------|-------|
| A | 1 | 2 |
| B | 0 | ⊥ |

(c) The state table after $p_1$

| MAC | state | value |
|-----|-------|-------|
| A | 1 | 2 |
| B | 1 | 1 |

(d) The state table after $p_2$

| MAC | state | value |
|-----|-------|-------|
| A | 1 | 3 |
| B | 1 | 1 |

(e) The state table after $p_4$

Figure 2.4: An illustrative execution.

## 2.2   NetEgg Model

In this section, we first describe the scenario-based programming model of NetEgg, and explain how this model allows the operator to describe example network behaviors in representative scenarios. Second, we define the policy model, which includes the model of state tables and policy tables. We will show how to generate a policy from scenarios in the next section.

### 2.2.1   Scenario-based Programming Framework

NetEgg provides a configuration language for expressing network timing diagrams. In this language, variables and fields of packets are typed. Examples of base types we use are bool, PORT, IP_ADDR(set of IP addresses), MAC_ADDR (set of MAC addresses). A packet-type consists of a list of names along with their types for each field in the packet. In our example,

$$
\begin{array}{lllll}
\text{Packet-type} & \text{P} & ::= & \langle \text{f}_1 : \text{T}_1, .., \text{f}_k : \text{T}_k \rangle \\
\text{Symbolic packet} & sp & ::= & \langle sv_1, .., sv_k \rangle \\
\text{Action} & a & ::= & \text{drop}|\text{flood}|\text{send(port)} \\
& & & |\text{modify(f},v)|.. \\
\text{Event} & e & ::= & sp \Rightarrow [a_1, .., a_i] \\
\text{Scenario} & sc & ::= & [e_1, .., e_j] \\
\text{Program} & \text{prog} & ::= & \{sc_1, .., sc_n\}
\end{array}
$$

Figure 2.5: Scenario-based programming model.

the packet-type consists of three fields and is given by $\langle$port : `PORT`, srcmac : `MAC_ADDR`, dstmac : `MAC_ADDR` $\rangle$.

A (concrete) packet specifies a value for each field of type corresponding to that field. A symbolic value of a type `T` is either a concrete value of type `T`, or a variable $x$ of type `T`. A symbolic packet specifies a symbolic value for each field.

We use `Act` to denote the set of action primitives for processing packets. For the action primitives with parameters, the user can use either concrete values or variables of the corresponding type.

In NetEgg, we provide a library that supports standard packet fields and actions such as drop, flood, send(port) (send to a port), modify(f,$v$) (modify the value of field f to $v$). Our tool also supports user-defined packet-type using customized field names and types, as well as user-defined actions. One can generalize it by providing handlers for user-defined fields and action primitives.

An *event* is a pair of a symbolic packet $sp$ and a list of actions $[a_1, ..a_l]$, denoted as $sp \Rightarrow [a_1, .., a_l]$. A *scenario* is a finite sequence of events. A scenario-based program is a finite set of scenarios.

With the notation, the scenario of Figure 2.2 can be written as in Figure 2.6.

$$
\begin{array}{l}
P_1,\ h_1,\ h_2 \Rightarrow \text{flood} \\
P_2,\ h_3,\ h_1 \Rightarrow \text{send}(P_1) \\
P_3,\ h_2,\ h_3 \Rightarrow \text{send}(P_2)
\end{array}
$$

Figure 2.6: The scenario corresponding to Figure 2.2 for the learning switch example.

A scenario is *concrete* if all the symbolic packets and actions appearing in the scenario have only concrete values. A scenario-based program with symbolic scenarios can be viewed as a short-hand for a set of concrete scenarios. This set is obtained by replacing each variable

by every possible value of the corresponding type with the following requirements. First, a variable can only take values that have not appeared in the scenario-based program. Second, if the same variable appears in multiple symbolic packets and actions in the program, then it gets replaced by the same value. Third, different variables in a program get replaced by different values. Thus, the symbolic scenario of Figure 2.2 corresponds to $\Pi_{i=0,1,2}(n-i)(l-i)$ concrete scenarios if the type MAC_ADDR and PORT contain $n$ and $l$ distinct values, respectively.

The language itself is simple and can be viewed more as a configuration language rather than a general-purpose programming language. We also build a visual tool that takes as input scenarios drawn as actual network timing diagrams, and generate the configuration (see Figure 2.17).

## 2.2.2   Policy Model

A policy consists of a *policy table* along with *state tables* that store the history of policy execution.

**State Tables.** A state table is a key-value map that maintains states and values for relevant fields. Let $\mathtt{T}_{i_j}$ be some base type appearing in the packet-type, $S$ be a state set with finitely many states, and the packet-type be $\langle \mathrm{f}_1 : \mathtt{T}_1,..,\mathrm{f}_k : \mathtt{T}_k \rangle$. A $d$-dimensional state table $ST$ stores a state in $S$ and a value of type $\mathtt{T}_{i_{d+1}}$, for all keys of type $\mathtt{T}_{i_1} \times .. \times \mathtt{T}_{i_d}$

The operations we allow on a state table are *reads*, *checks* and *writes*.

Let $ST$ be a state table of type $\mathtt{T}_1 \times .. \times \mathtt{T}_d \to S \times \mathtt{T}_{d+1}$, let $\mathrm{f}_1,..\mathrm{f}_d$ be field names of type $\mathtt{T}_1,..,\mathtt{T}_d$, respectively. A read of $ST$ indexes some entry in $ST$, and is of the form $ST(\mathrm{f}_1,..,\mathrm{f}_d)$. A check of $ST$ checks whether the state associated with some key is a particular state. Syntactically, it is a pair of a read and a state, written as $ST(\mathrm{f}_1,..,\mathrm{f}_d).\mathrm{state}{=}\mathtt{s}$, where $\mathtt{s} \in S$ is a state. In our example, $ST(\mathrm{dstmac}).\mathrm{state}{=}\mathtt{0}$ is a check with the field dstmac. In contrast to a check, a write of a state table changes the state along with the value associated with some key. A write of $ST$ is of the form $ST(\mathrm{f}_1,..,\mathrm{f}_d){:=}(\mathrm{sv},\mathrm{fv})$. Here, sv is either a state, or - representing no change. fv is either a concrete value of type $\mathtt{T}_{d+1}$, - representing no change, or a field name of type $\mathtt{T}_{d+1}$. In our example, $ST(\mathrm{srcmac}){:=}(\mathtt{1},\mathrm{port})$ is a write of $ST$ with the field srcmac.

We use the term configurations for the snapshots of state tables. For example, the

14

initial configuration of the state table in our example maps every MAC address to $(0,\perp)$ as shown in figure 2.4(b). Here, we use $\perp$ to represent the fact that no value is stored. A read $ST(f_1,..,f_d)$ for a packet $p$ at a configuration $c$ returns the state-value pair stored in $ST$ for the key $(p.f_1,..,p.f_d)$ at $c$. We use $ST(f_1,..,f_d)$.state and $ST(f_1,..,f_d)$.value to denote the state and value in the returned pair. A check $ST(f_1,..,f_d)$.state=s is true for a packet $p$ at a configuration $c$ if the state read from $ST$ at the configuration $c$ is s. In the example in Figure 2.4, $ST$(dstmac).state=0 is true for $p_1$ at the initial configuration (subfigure (b)) of $ST$. A write $ST(f_1,..,f_d):=$(sv,fv) for a packet $p$ writes the state-value pair to the corresponding entry indexed by the read. Note that if sv(fv, resp.) is -, the write does not write any state(value, resp.) to $ST$, and if fv specifies a field name, the value of $p$.fv should be written.

**Policy Tables.** Given a set of state tables $\mathcal{T}$, a rule $r$ based on $\mathcal{T}$ has four components, namely, match, test, actions and update. match is of the form $\langle f_1=v_1,..,f_k=v_k \rangle$, where $f_i$ is a name of a packet field, and $v_i$ is a concrete value or a wildcard. A packet $p$ matches $\langle f_1=v_1,..,f_k=v_k \rangle$ iff $v_i$ is a wildcard, or $p.f_i = v_i$ for all $i = 1$ to $k$. The actions is a list of actions using action primitives in `Act`. In the case where an action primitive accepts parameters, the parameters can be concrete values or values read from state tables in $\mathcal{T}$ using reads. test is a conjunction of checks and update is a sequence of writes, where each check/write is of some state table in $\mathcal{T}$. As an example, last two rows in Table 2.1 are two rules. A *policy table* based on $\mathcal{T}$ is an ordered list of rules, and every rule is based on $\mathcal{T}$.

A configuration $\mathcal{C}$ of a policy consists of all the configurations of each state table in $\mathcal{T}$, on which the policy table is based. A packet $p$ matches a rule at a configuration $\mathcal{C}$ iff $p$ matches match and every check in test is true for $p$ at the corresponding configuration in $\mathcal{C}$. Suppose the first matched rule for a packet $p$ at a configuration $\mathcal{C}$ is $r$. Then actions of $r$ will be executed on $p$ and every write in update of $r$ will be executed. We denote the execution for packet $p$ as $\mathcal{C} \xrightarrow{p/as}_{PT} \mathcal{C}'$, with $\mathcal{C}'$ the new configuration, and $as$ the actions applied to $p$.

## 2.3    Policy Generation

Given a set of scenarios describing a policy, our synthesizer first checks if there are conflicts among the scenarios. If there are no conflicts existing in the scenario, NetEgg tries to generate a policy consistent with all scenarios.

In this section, we first describe the objective policies NetEgg aims to generate. Then we describe how to detect conflicts in the scenarios. Finally, we present the synthesis algorithm for scenarios without conflicts in steps.

### 2.3.1    The Policy Learning Problem

First, we note that, since the input scenarios describe the behaviors of the desired policy in representative scenarios, the generated policy should be *consistent* with all the behaviors described in all scenarios.

**Definition 1** (Consistency). *Given a concrete scenario $SC = [sp_1 \Rightarrow as_1, .., sp_k \Rightarrow as_k]$, a policy table PT is* consistent *with SC iff $\mathcal{C}_{i-1} \xrightarrow{sp_i/as_i}_{PT} \mathcal{C}_i$ for $i = 1, .., k$, where $\mathcal{C}_0$ is the initial configuration in which every state table maps every key to the initial state 0 and a value of $\perp$. A policy table is consistent with a scenario-based program, iff it is consistent with all the concrete scenarios represented by the scenario-based program.*

As an example, the policy given in Table 2.1 is consistent with the scenario in Figure 2.2. However, the following policy is not consistent with the scenario, since it floods the third packet in the scenario instead of sending it to $P_2$.

| match | test | actions | update |
|---|---|---|---|
| * | $ST(\text{dstmac}).\text{state}=0$ | flood | $ST(\text{srcmac}):=(1, \text{port})$ |
| * | $ST(\text{dstmac}).\text{state}=1$ | send($ST(\text{dstmac}).\text{value}$) | $ST(\text{srcmac}):=(\text{-}, \text{-})$ |

Table 2.2: An inconsistent policy table.

In addition to consistency, NetEgg also aims to generate a generalized policy from input scenarios. For this, we aim at generating a consistent policy with minimal number of rules. To see how this heuristic can help to generate a general policy, let us consider the 3-rule policy table in Table 2.3. It can be verified that the policy is consistent with the scenario in Figure 2.2. However, this policy overfits the input scenario and will not generalize to a

fourth packet such as $\langle P_1, h_4, h_2 \rangle$, because this packet would be flooded by the policy. On the other hand, the desired policy in Table 2.1 only uses two rules, and can handle the fourth packet mentioned above correctly.

| match | test | actions | update |
|-------|------|---------|--------|
| * | $ST$(dstmac).state=0 | flood | $ST$(srcmac):=(1, port) |
| * | $ST$(dstmac).state=1 | send($ST$(dstmac).value) | $ST$(srcmac):=(2, port) |
| * | $ST$(dstmac).state=2 | send($ST$(dstmac).value) | $ST$(srcmac):=(-, -) |

Table 2.3: A consistent yet restrictive policy table.

We summarize the major computational problem as the following *policy learning problem*.

**Policy learning problem.** Given scenarios $SC_1, ..., SC_n$, the policy learning problem seeks a set of state tables $\mathcal{T}$ and a policy table $PT$ based on $\mathcal{T}$, such that (1) $PT$ is consistent with all scenarios $SC_i$. (2) $PT$ has the smallest number of rules among all consistent policy tables.

### 2.3.2 Conflict Detection

Two concrete scenarios conflict with one another if they describe different behaviors for a same sequence of packets. Thus, there is no policy consistent with a scenario-based program if there exist two conflicting concrete scenarios represented by the scenario-based program.

In this subsection, we describe how NetEgg detects conflicts in the input scenario-based program. In particular, our conflict detection algorithm checks whether there exist two concrete scenarios represented by the scenario-based program which conflict with one another. The conflict detection accounts for two cases. First, the two concrete scenarios are represented by a single symbolic scenario in the scenario-based program. Second, the two concrete scenarios are represented by two symbolic scenarios in the scenario-based program.

For the first case, let us consider two concrete conflicting scenarios $S_1$ and $S_2$, and suppose they are obtained from the symbolic scenario $SC$. Let us further suppose that $S_1$ and $S_2$ first differ at the $i$-th event. That is, 1) all events before the $i$-th one are the same in both $S_1$ and $S_2$; 2) the $i$-th event $p_1 \Rightarrow as_1$ in $S_1$ and the $i$-th event $p_2 \Rightarrow as_2$ in $S_2$ describe the same packet (i.e. $p_1 = p_2$), but different actions (i.e. $as_1 \neq as_2$). In particular,

there must exist a position $k$, such that the $k$-th action $a_1^k$ in $as_1$ is different from the $k$-th action $a_2^k$ in $as_2$. Since both $S_1$ and $S_2$ are obtained from $SC$ by instantiating the variables in $SC$, both $a_1^k$ and $a_2^k$ must have the same action primitive, but different parameters that are instantiated from a variable $x$ in $SC$. Therefore, it must be the case that $x$ have not appeared in any symbolic packets $sp_j$ in $SC$, for all $j \leq i$. Here, $sp_j$ is the symbolic packet in the $j$-th event in $SC$. Otherwise, $S_1$ and $S_2$ describe different packets at event $j$ for some $j \leq i$.

To summarize the above idea, Algorithm 1 shows the algorithm to detect conflicts for this case.

---

**Algorithm 1** detect_conflicts_case1

---

1: **for** each scenario $SC$ **do**
2:   **for** $i = 1$ to $|SC|$ **do**
3:     let $sp_i \Rightarrow as_i$ be the $i$-th event of $SC$
4:     **if** $as_i$ contains a variable that does not appear in any of $sp_j$ for $j \leq i$ **then**
5:       **return** conflict found in scenario $SC$ at event $i$
6:     **end if**
7:   **end for**
8: **end for**
9: **return** no conflict

---

Similarly, for the second case, the algorithm checks whether two symbolic scenarios describe conflicting behaviors for a same sequence of packets.

Algorithm 2 describes the algorithm for detecting conflicts for the second case. For every pair of scenarios $SC_1$ and $SC_2$ in the scenario-based program (line 1), the algorithm checks whether there exist conflicts for the sequence of packets $p_1...p_i$ in the two scenarios (line 4-21). The algorithm first pre-processes the variables in the two scenarios (line 2) by replacing each variable $x$ in $SC_1$ using a new variable $x_1$. This pre-process accounts for the fact that the variable $x$ in $SC_1$ can be replaced by a value that is different from the value used in $SC_2$. The algorithm further maintains a constraint $C$ which is used for checking whether the sequence of packets $p_1...p_i$ exists. Initially, $C$ contains the constraints indicating that all variables are different from one another in a scenario, and also different from all concrete values appeared in the scenario (line 3). The checking process is conducted incrementally. In each iteration in the for loop at line 4, the algorithm checks whether the $i$-th symbolic packet in $SC_1$ can represent a packet that can also be represented by the $i$-th

**Algorithm 2** detect_conflicts_case2

---

1: **for** each pair of scenarios $(SC_1, SC_2)$ **do**
2:    replace each variable $x$ by $x_1$ in $SC_1$ and $x_2$ in $SC_2$
3:    initialize the constraint $C$
4:    **for** $i = 1$ to $\min(|SC_1|,|SC_2|)$ **do**
5:        let $sp_i^j \Rightarrow as_i^j$ be the $i$-th event of $SC_j$, $j = 1, 2$
6:        **for all** field $f$ in the packet **do**
7:            $C = C \wedge \{sv_1 = sv_2\}$, $sv_j$ is the symbolic value of field $f$ in $sp_i^j$
8:        **end for**
9:        **if** $C$ is satisfiable **then**
10:            **if** 1) $|as_i^1| \neq |as_i^2|$ or 2) from some $k$, $as_i^1[k]$ and $as_i^2[k]$ have different primitives or different number of parameters **then**
11:                **return**  conflict found in $SC_1$ and $SC_2$ at event $i$
12:            **end if**
13:            **for** each symbolic value $sv_1$ appeared in $as_i^1$ and the symbolic value $sv_2$ appeared in $as_i^2$ at the same position **do**
14:                **if** $C \wedge \{sv_1 \neq sv_2\}$ is satisfiable **then**
15:                    **return**  conflict found in $SC_1$ and $SC_2$ at event $i$
16:                **end if**
17:            **end for**
18:        **else**
19:            no conflict in $SC_1$ and $SC_2$; continue for the next pair of scenarios
20:        **end if**
21:    **end for**
22: **end for**
23: **return**  no conflict

---

symbolic packet in $SC_2$. Therefore, the algorithm needs to check for each field whether the symbolic values of that field can be the same in the two symbolic packets by adding the corresponding constraint to $C$ (line 7). If $C$ becomes unsatisfiable, we conclude that the packet sequence $p_1...p_i$ cannot exist, therefore the algorithm reports no conflicts in $SC_1$ and $SC_2$, and continues the checking for the next pair of scenarios (line 19). However, if $C$ is satisfiable, such a packet sequence exists, thus the algorithm further checks whether $SC_1$ and $SC_2$ may contain different actions for it (line 9-18). $SC_1$ and $SC_2$ may conflict with each other in two cases. First, the action lists have different length in two scenarios, or they contains different action primitives at some position, or some actions at a position accepts different number of parameters (line 10-11). Second, the variables used as parameters at some position in the action lists can represent different values (line 13-17). For the second case, it suffices for the algorithm to check whether $C$ together with the corresponding

constraint is satisfiable (line 14).

In the following sections, we describe the synthesis algorithm for scenarios without conflicts. We break our synthesis algorithm into steps, and discuss each step separately.

### 2.3.3 Policies without tests

First, let us consider the simplest case where the desired policy table does not have tests. In this case, each rule in the policy table only consists of a match and actions. Thus, it suffices to generate an ordered list of matches, together with the corresponding list of actions.

**Generate ordered match list.** We first describe the algorithm which generates a list of matches from the input scenarios, shown in Algorithm 3. As defined in our scenario-based programming model, a symbolic packet represents a set of concrete packets, which is obtained by replacing symbolic values by concrete field values. Therefore, Algorithm 3 generates a match for each symbolic packet in the scenarios, by replacing symbolic values by $*$ (line 3). Moreover, to ensure that the generated match does not mismatch unrepresented packets, the algorithm inserts the generated match to the list, such that no match under it is completely covered (line 4). Note that for two generated matches which are overlapping with each other, we can order either one above the other. We will explain this through an example later in this subsection.

---

**Algorithm 3** generate_ordered_match_list($[SC_i]$)

---

1: $L = \emptyset$
2: **for all** packet $sp = \langle f_i = v_i \rangle$ in every scenario $SC_i$ **do**
3:     let $m = \langle f_i = m_i \rangle$, where $m_i = v_i$ if $v_i$ is a concrete value else $*$
4:     insert $m$ to $L$
5: **end for**
6: **return** $L$

---

**Search for actions.** With the ordered list of matches, it is straightforward to search for the actions for every match in the list: we can simply search the first matched match in the list for each symbolic packet and check consistency between the actions with the packet and actions with the match. If actions for the match are not set, we can set them to the actions associate with the symbolic packet. A consistent policy table is returned when all actions in every symbolic events are consistent.

**Examples.** Consider a scenario describing a firewall, shown in Figure 2.7(a). Here a

$$A, ip1 \Rightarrow \text{send}(1)$$
$$ip2, B \Rightarrow \text{send}(2)$$
$$A, C \Rightarrow \text{drop}$$

(a) A scenario for the policy.

| match | actions |
|---|---|
| srcip=$A$, dstip=$C$ | drop |
| srcip=$A$, dstip=$*$ | send(1) |
| srcip=$*$, dstip=$B$ | send(2) |

(b) The policy table.

Figure 2.7: A firewall example.

symbolic packet is expressed by $\langle$srcip, dstip$\rangle$. The generated policy table is shown in Figure 2.7(b). The match column of the policy table is generated by Algorithm 3. We note that if the second rule is swapped with the third rule in the policy table, the resulted policy table is still consistent with the scenario. The reason is that by definition the variables $ip1$ and $ip2$ only represent values other than $A, B, C$. Thus, the packet $\langle A, B \rangle$ is not represented by any symbolic packets in the scenario.

### 2.3.4 Policies with tests

Now we consider synthesizing policies that use tests. As an example, consider the scenario for the learning switch in Figure 2.2. One can verify that a consistent policy table for the scenario requires tests. In fact, the only match in the match list generated by Algorithm 3 is $\langle * \rangle$, and every packet in the scenario matches it. However, their actions differ from each other. Therefore, in addition to the only match $\langle * \rangle$, a consistent policy table must have tests. In this subsection, we will assume that all tests in the policy table are given, and show the algorithm to synthesize the policy. We will relax this assumption in the next subsection.

**Sketch.** Suppose the tests used in the policy for learning switch are $ST(\text{dstmac}).\text{state}=0$ and $ST(\text{dstmac}).\text{state}=1$. Composing the match and test, we know that the policy table has the form as shown in Table 2.4.

| match | test | actions | update |
|---|---|---|---|
| $*$ | $ST(\text{dstmac}).\text{state}=0$ | $ax_1$ | $ST(\text{srcmac}):=(sx_1,\ ux_1)$ <br> $ST(\text{dstmac}):=(sx_2,\ ux_2)$ |
| $*$ | $ST(\text{dstmac}).\text{state}=1$ | $ax_2$ | $ST(\text{srcmac}):=(sx_3,\ ux_3)$ <br> $ST(\text{dstmac}):=(sx_4,\ ux_4)$ |

Table 2.4: A policy table sketch for the learning switch.

Table 2.4 is a symbolic representation of policy tables. It represents actions and update

in rules using variables. We call such symbolic representations *policy table sketches* (or *sketches* in short). In this sketch, we derive all possible writes in update from the state tables that are used in each rule's tests. In this example, since the only state table accepts keys of type MAC_ADDR, the state table can be updated using either srcmac or the dstmac of a packet. Therefore, we have two writes per rule in its update, and $sx$'s range over all states (in this example, $\{0,1\}$) plus a special symbol - meaning no update, and $ux$'s range over all field names plus -. For ease of presentation, we intentionally ignore potential overwrittings due to the two writes in each rule's update. In practice, we need to consider other orders of writes as well. We use variables $ax$'s to represent possible actions for each rule. In this example, $ax$'s can range from $\{\text{flood}, \text{send}(ST(\text{srcmac})), \text{send}(ST(\text{dstmac}))\}$, which are obtained from actions appearing in the scenario. To distinguish variables appearing in sketches from variables appearing in the scenarios, we will call these variables appearing in a sketch as sketch variables.

Algorithm 4 shows the algorithm of generating sketches given the match list $L$ and tests. The set $C$ contains all possible reads to the state tables appearing in $TESTS$ (line 1-line 4), and this set is used to derive writes in each rule (line 7). The sketch is generated by composing each match and test in $TESTS$ (line 5-line 6), and every rule has all possible writes derived from reads in $C$ (line 7).

---

**Algorithm 4** generate_sketch($L$, $TESTS$)

---

1: let $C = \emptyset$
2: **for all** state table $ST$ appearing in $TESTS$ **do**
3:     add all reads $ST(\text{f}_1,..,\text{f}_k)$ to $C$, where $\text{f}_i$ is a field name of the corresponding type.
4: **end for**
5: **for all** match $match$ in $L$ in order **do**
6:     **for all** test $test$ in $TESTS$ **do**
7:         construct a rule $r = (match, test, ax, update)$, where $ax$ is a new variable for actions, and $update = [read_i{:=}(sx_i,ux_i)], \forall read_i \in C$, $sx_i$, $ux_i$ fresh variables.
8:         add rule $r$ to $sketch$
9:     **end for**
10: **end for**

---

**Search for sketch variables.** Using the sketch, we can search concrete values for sketch variables, with the goal that the obtained policy table is consistent with all scenarios. To search for a consistent policy table, we perform a simple backtracking search algorithm over

all sketch variables. The algorithm is shown in Algorithm 5.

---

**Algorithm 5** search_sketch($[SC_i], sketch$)

---

1:  $stack = []$
2:  **for all** scenarios $SC_i$ **do**
3:      **for all** events $e_j$ in $SC_i$ **do**
4:          let $r$ be the first matching rule
5:          initialize actions of $r$ and push the sketch variable to $stack$, if $r$'s actions are not set
6:          **if** actions of $r_i$ are consistent **then**
7:              initialize any sketch variables in $r$'s update and push them to $stack$, if they are not assigned values yet
8:              apply update of $r$
9:          **else**
10:             backtrack($stack$)
11:             **return** FAILED **if** $stack$ is empty **else** restart from line 2
12:         **end if**
13:     **end for**
14: **end for**
15: **return** $sketch$

---

The algorithm maintains a stack of sketch variables together with the values assigned to them. Whenever a sketch variable is assigned a value, it ensures that the sketch variable is pushed to the stack (line 5, 7). For each symbolic event in every scenario, the algorithm checks consistency of the first matching rule's actions (line 6). Whenever inconsistency encountered (line 9), it performs standard backtracking procedure on the stack (line 10-11); and when it is consistent, the algorithm executes the update of the rule (line 8) and carry on to the next symbolic event.

### 2.3.5   Putting It Together

Now we describe the overall synthesis algorithm (Algorithm 6), using the procedures described in previous subsections. At a high level, the synthesis algorithm enumerates sketches by increasing the number of rules, in order to generate a consistent policy table using as few rules as possible. For each sketch, it invokes Algorithm 5 to search for a consistent policy using the sketch.

**Example.** We use the learning switch example in Figure 2.2 to illustrate how Algorithm 6 works. For the scenario in Figure 2.2, the set $A$ contains $2^{|fields|} = 2^3 = 8$ possible reads,

**Algorithm 6** synthesize($\{SC_i\}$)

---

1: $L =$ generate_ordered_match _list($[SC_i]$)
2: let $A$ contain all possible reads
3: **for all** $(c, \mathtt{m})$ with ascending order of $\mathtt{m}^c$ **do**
4:   **for all** subset $B \subset A$ with size $c$ **do**
5:     $TESTS = \{\bigwedge_i read_i.state{=}v_i | \forall read_i \in B, \forall v_i \in [0,..,\mathtt{m\text{-}1}]\}$
6:     generate_sketch($L, TESTS$)
7:     **if** search_sketch($sketch$) returns a consistent policy table **then**
8:       **return** the policy table
9:     **end if**
10:   **end for**
11: **end for**

---

each of which corresponds to a combination of field names, and a state table with the corresponding type. Because two state tables with the same type can be merged into a single state table with larger states, we only construct one state table per type. Then the algorithm constructs a sketch, where each rule has $c$ checks, and its state ranges from $\mathtt{0}$ to $\mathtt{m\text{-}1}$. Thus the generated sketch has $\mathtt{m}^c|L|$ rules. Note that, when $\mathtt{m}$ is $\mathtt{1}$ or $c$ is $0$, the generated sketch does not use tests essentially, hence enumeration of the pair $(c, \mathtt{m})$ can start from $(1, 2)$. When picking reads from $A$ (line 4), we pick reads with high dimension first. As an example for the learning switch, the first generated set $B$ with size 1 in line 4 is $\{ST(\text{port,srcmac,dstmac})\}$, followed by other reads with dimension two.

### 2.3.6 Additional Heuristics

In addition to the basic synthesis algorithm described above, the synthesizer has implemented other heuristics.

**Lazy initialization.** Algorithm 5 initializes sketch variables and pushes them to the stack as soon as applying update of the matching rule. This eager initialization could push irrelevant sketch variables to the stack and increase the search depth. For example, the variables $sx_2, ux_2$ in Table 2.4 are not used when checking consistency for any symbolic packet in Figure 2.2, and hence irrelevant to the consistency checking. Thus, the synthesizer takes a lazy initialization heuristic. That is, only when an uninitialized sketch variable is read from state tables, the synthesis algorithm initializes it and pushes it to the stack.

**Post processing.** After synthesizing a consistent policy, the synthesizer applies additional

post processing to the policy table in order to simplify the policy table. These includes: (1) If a rule in the policy table is not matched by any symbolic packet in the input scenarios, this rule can be removed; (2) The synthesizer removes `writes` in each rule's `update`, if they do not change the state table; (3) When multiple rules can be merged into one without causing inconsistency, the synthesizer will merge these rules.

## 2.4   Policy Execution

Given the synthesized policy, our tool uses the interpreter to process packets on the controller. As described in Section 3.3, the interpreter simply iterates through all rules in the policy table and picks the first matched rule for the incoming packet. Then it updates all state tables based on the `update` of the matched rule, and instructs the switch to apply the action of the rule to the packet.

While processing packets on the controller is sufficient for executing the policy, it is not practically efficient and degrades the performance of the network. In this section, we show how the tool infers flow table rules which can be installed onto switches, thus reducing the overhead of controller and delay of packet delivery.

Our key observation is the following theorem.

**Theorem 1.** *A packet can be handled on switches if and only if handling this packet on the controller does not change any state tables.*

Indeed, if a packet $p$ is handled on switches, the controller will not be aware of the packet and thus the state tables remain unchanged. On the other hand, if $p$ is sent to the controller for execution and the updated state tables remain the same as before, we know handling $p$ on switches would not affect future packets execution. Therefore, it is sufficient and necessary to install rules on switches for the packets whose execution will not change current configuration of state tables.

Based on this observation, we have implemented a reactive installation approach which installs flow table rules that only match necessary fields. Moreover, to keep the installed rules up to date, we update installed rules when the policy configuration changes, and remove invalid rules on switches. Note that, one can also infer flow table rules in a proactive

way based on this observation. We leave the implementation of proactive approaches to future work.

---

**Algorithm 7** update_flowtable($p$)

---
1: let rule $r$ be the matched rule for $p$ in the policy table
2: **if** $r$ does not update state tables, or the updated state tables remain unchanged **then**
3:     $match \leftarrow \langle f_{i_1}=p.f_{i_1},..,f_{i_k}=p.f_{i_k}\rangle$, for all field $f_{i_j}$ appearing in the policy table
4:     add $match \rightarrow r$.actions to the flow table, if the actions $a_j$ applied to $p$ by $r$ is supported by the switch
5: **end if**
6: **for all** installed rule $match' \rightarrow [a'_1,..,a'_l]$ in the flow table **do**
7:     let $p'$ be a packet matches $match'$
8:     let rule $r$ be the matched rule in the policy table for $p'$
9:     **if** $r$ does not update state tables or the updated state tables remain unchanged **then**
10:         update the installed rule to $match' \rightarrow r$.actions, if the actions $a_j$ applied to $p$ by $r$ is supported by the switch
11:     **else**
12:         remove the installed rule from the flow table
13:     **end if**
14: **end for**

---

Algorithm 7 shows the installation strategy. First the algorithm checks whether the matched rule $r$ for $p$ will change the configuration of state tables. The rule $r$ will not change the configuration, if $r$ does not have writes, or the updated states and values remain the same as the old ones (line 2). If executing $p$ would not change the configuration, the algorithm installs a flow table rule $match \rightarrow [a_1,..,a_l]$ onto the switch, where $match$ specifies the values for fields related to the policy, and $a_j$'s are the actions that should be applied to $p$ (line 3-4). The algorithm also needs to check whether previously installed rules are still correct. For this, the algorithm repeats a similar process for each installed rule (line 6-14).

**Example.** Revisit the example run in Figure 2.4. By the interpreter's algorithm shown in Figure 2.3, the first packet is processed on the controller, and the state table is updated to the one shown in subfigure (c). Applying Algorithm 7, the matching rule $r$ for $p_1$ would be the first rule in the policy table shown in Table 2.1. Since port 2 is already remembered for the srcmac A, $r$ would not change the state table. Therefore, a flow table rule $fr_1 = \langle$port=2,srcmac=A,dstmac=B$\rangle \rightarrow$flood, which matches the port, srcmac and dstmac of $p_1$ is pushed down to the switch. After processing the second packet $p_2$, the state table is updated as in subfigure (d) and a flow table rule matching $p_2$ can be pushed

26

down. Moreover, the algorithm checks the installed flow table rule $fr_1$. Since now $p_1$ would match the second rule in the policy table, and the applied action to $p_1$ is different from the installed flow table rule, the action of $fr_1$ is updated to send(1).

## 2.5 Use Cases

In this section, we demonstrate scenario-based programming for four policies. For each policy, we will show the packet-type we use, the scenarios that can be used to synthesize the desired policy, and the policy table generated from the scenarios. To this end, we manually validate that the synthesized policy is the correct policy. One can also formally verify the correctness of the generated policy against logical specifications using control plane verification tools such as Vericon [20] and Nice [27]. We plan to explore light-weight verification tools for the custom policy abstraction in the future.

### 2.5.1 Learning Switch

First, we revisit our motivating example. Recall that we can program the learning switch application for a single switch using a scenario in Figure 2.2. Now we show how to adapt the scenario to program the learning switch for a network. That is, the policy needs to maintain the port of each switch for hosts. To program this policy, we need a field specifying which switch the packet is located. Therefore, we use the packet-type $\langle$ switch : SWITCH, port : PORT, srcmac : MAC_ADDR, dstmac : MAC_ADDR $\rangle$. For the scenario, we simply add the switch field to each symbolic packet in the scenario in Figure 2.2. This modified scenario suffices for NetEgg to synthesize the network-wide learning switch policy. The scenario and synthesized policy table is shown in Figure 2.8 and Table 2.5.

> scenario 1:
> $s_1,P_1,h_1,h_2 \Rightarrow$ flood
> $s_1,P_2,h_3,h_1 \Rightarrow$ send($P_1$)
> $s_1,P_3,h_2,h_3 \Rightarrow$ send($P_2$)

Figure 2.8: Scenario-based program for the learning switch.

| match | test | actions | update |
|-------|------|---------|--------|
| * | $ST$(switch,dstmac).state=0 | flood | $ST$(switch,srcmac) :=(1, port) |
| * | $ST$(switch,dstmac).state=1 | send($ST$(switch, dstmac).value) | $ST$(switch,srcmac) :=(1, port) |

Table 2.5: The policy table for the learning switch.

## 2.5.2 Stateful Firewall

Now, we show how to use scenarios to program stateful firewall policies inductively.

**First firewall.** First, we consider a stateful firewall which protects hosts connecting to port 1 by blocking untrusted traffic from port 2. The firewall should allow all outbound packets from port 1, and only allow inbound packets from port 2 if the sender of the packet has received packets from the receiver before. For this policy, we use the packet-type ⟨port:PORT, srcip:IP_ADDR, dstip:IP_ADDR⟩. We start by giving two of the most intuitive scenarios shown in Figure 2.9. In the first scenario, the switch blocks the traffic from port 2, and the second scenario demonstrates the case where the firewall allows the traffic from port 2. It turns out that these two scenarios are sufficient to generate the desired policy, shown in Table 2.6.

scenario 1:    scenario 2:
$2,h_2,h_1 \Rightarrow$ drop    $1,h_1,h_2 \Rightarrow$ send(2)
    $2,h_2,h_1 \Rightarrow$ send(1)

Figure 2.9: Scenario-based program for the first stateful firewall.

| match | test | actions | update |
|-------|------|---------|--------|
| port=1 | True | send(2) | $ST$(dstip,srcip):=(1, -) |
| port=2 | $ST$(srcip,dstip).state=0 | drop | - |
| port=2 | $ST$(srcip,dstip).state=1 | send(1) | - |

Table 2.6:   The policy table for the first stateful firewall.

**Second firewall.** Now suppose we want to specify a policy such that it allows inbound traffic if the sender has received packets from any protected hosts before. One may notice that the policy should maintain a state for each host, instead of a pair of hosts. Using the scenario-based programming, we can simply adapt scenarios from Figure 2.9 and change the dstip of the second packet in scenario 2, as following:

modified scenario 2:
1,$h_1$,$h_2$⇒ send(2)
2,$h_2$,$h_3$⇒ send(1)

Figure 2.10: The modified scenario for the second stateful firewall.

The synthesized policy maintains a 1-dimension state table, and is shown in Table 2.7.

| match | test | actions | update |
|---|---|---|---|
| port=1 | True | send(2) | $ST$(dstip):=(1, -) |
| port=2 | $ST$(srcip).state=0 | drop | - |
| port=2 | $ST$(srcip).state=1 | send(1) | - |

Table 2.7: The policy table for the second stateful firewall.

**Third firewall.** While we mostly focus on packetin events, NetEgg can be generalized to handle arbitrary events. In this use case, we will demonstrate how to use fields in symbolic packets to handle user-defined network events. Suppose we want to further implement a policy such that inbound traffic is allowed until a timeout event indicates the sender expires. For the policy, we need to handle a timeout event, and the expired host ip specified in the event. We can use a packet-type ⟨event:`EVENT`, eventip:`IP_ADDR`, srcip:`IP_ADDR`, dstip:`IP_ADDR`⟩. Here, the field named event specifies the type of the network event, and the field named eventip specifies the expired host. These two fields are set by the corresponding field handlers. For this policy, we can add one more scenario exhibiting the behavior of timeout, as in Figure 2.11. The first symbolic packet is similar to above, but since this is a packetin event, its eventip field is not applicable (we use - to denote its value). The second symbolic packet is the timeout event, which specifies that host $h_2$ is expired. Since the controller does not need to apply any actions to this event, we use nop for its action. The third packet from host $h_2$ now gets dropped. Scenario 1 and Scenario 2 can be adapted similarly from Figure 2.9 and Figure 2.10 respectively.

scenario 3:
packetin,-,1,$h_1$,$h_2$⇒ send(2)
timeout,$h_2$,-,-⇒ nop
packetin,-,2,$h_2$,$h_3$⇒ drop

Figure 2.11: The added scenario for the third stateful firewall.

Given the three scenarios, the desired policy can be synthesized, as in Table 2.8.

| match | test | actions | update |
|---|---|---|---|
| event=packetin, port=1 | True | send(2) | $ST$(dstip):=(1, -) |
| event=packetin, port=2 | $ST$(srcip).state=0 | drop | - |
| event=packetin, port=2 | $ST$(srcip).state=1 | send(1) | - |
| event=timeout | True | nop | $ST$(eventip):=(0,-) |

Table 2.8: The policy table for the third stateful firewall.

### 2.5.3 TCP Firewall

In this use case, we use scenarios to program the TCP firewall that tracks the state transition of TCP handshake protocol, and only allows packets that follow the protocol. We use the packet-type that contains 5 fields: ⟨flag:TCP_FLAG, srcip:IP_ADDR, dstip: IP_ADDR, srcport: TCP_PORT, dstport: TCP_PORT⟩.

We first specify two scenarios describing two allowed packet traces by the TCP firewall in Figure 2.12. A trivial policy which allows all packets would be generated. Next, we add two scenarios describing packets which should be denied by the firewall. Checking the policy, we find an undesired behavior of the generated policy, which allows the second packet in scenario 5. We add the correct behavior as in scenario 5, and the synthesizer generates the desired policy. The generated policy table is shown in Table 2.9, and the state table maintains states for each tuple of srcip,dstip,srcport and dstport.

### 2.5.4 ARP Proxy

In this use case, we use user-defined action primitives to program the ARP proxy in scenarios. An ARP proxy caches MAC addresses associated with IP addresses, and responds to ARP requests when the requested MAC is known. We use the packet-type ⟨ srcmac:MAC_ADDR, arpop:ARP_OP, srcip:IP_ADDR, dstip:IP_ADDR⟩ to specify this use case. The first scenario we provide is similar to the scenario for the learning switch example. In addition, we provide another scenario which describes learning srcmac from ARP reply

scenario 1:
SYN,$ip_1$,$ip_2$,$port_1$,$port_2$$\Rightarrow$ allow
ACK,$ip_2$,$ip_1$,$port_2$,$port_1$$\Rightarrow$ allow

scenario 2:
SYN,$ip_1$,$ip_2$,$port_1$,$port_2$$\Rightarrow$ allow
SYNACK,$ip_2$,$ip_1$,$port_2$,$port_1$$\Rightarrow$ allow
ACK,$ip_1$,$ip_2$,$port_1$,$port_2$$\Rightarrow$ allow

scenario 3:
ACK,$ip_1$,$ip_2$,$port_1$,$port_2$$\Rightarrow$ deny

scenario 4:
SYNACK,$ip_2$,$ip_1$,$port_2$,$port_1$$\Rightarrow$ deny

scenario 5:
SYN,$ip_1$,$ip_2$,$port_1$,$port_2$$\Rightarrow$ allow
ACK,$ip_1$,$ip_2$,$port_1$,$port_2$$\Rightarrow$ deny

Figure 2.12: Scenario-based program for the TCP firewall.

| match | test | actions | update |
|---|---|---|---|
| flag=SYN | True | allow | $ST$(dstip,dstport,srcip, srcport):=(1,-) |
| flag=SYNACK | $ST$(srcip,srcport, dstip, dstport).state=0 | deny | - |
| flag=SYNACK | $ST$(srcip,srcport, dstip, dstport).state=1 | allow | $ST$(dstip,dstport,srcip, srcport):=(1,-) |
| flag=ACK | $ST$(srcip,srcport, dstip, dstport).state=0 | deny | - |
| flag=ACK | $ST$(srcip,srcport,dstip, dstport).state=1 | allow | - |

Table 2.9: The policy table for the TCP firewall.

messages. Note that in the scenarios, we use the user-defined action primitive reply, which should construct an ARP reply message with the requested MAC address.

scenario 1:
$h_1$,request,$ip_1$,$ip_2 \Rightarrow$ flood
$h_3$,request,$ip_3$,$ip_1 \Rightarrow$ reply($h_1$)
$h_4$,request,$ip_4$,$ip_3 \Rightarrow$ reply($h_3$)

scenario 2:
$h_2$,reply,$ip_2$,$ip_1 \Rightarrow$ flood
$h_3$,request,$ip_3$,$ip_2 \Rightarrow$ reply($h_2$)

Figure 2.13: Scenario-based program for the ARP proxy.

| match | test | actions | update |
|---|---|---|---|
| arpop= request | $ST$(dstip).state=0 | flood | $ST$(srcip) :=(1,srcmac) |
| arpop=request | $ST$(dstip).state=1 | reply($ST$(dstip).value) | $ST$(srcip) :=(1,srcmac) |
| arpop=reply | True | flood | $ST$(srcip) :=(1,srcmac) |

Table 2.10:   The policy table for the ARP proxy.

## 2.6    Evaluation

We have developed a prototype of NetEgg written in Python. We evaluate NetEgg along two dimensions: (1) the feasibility of NetEgg in its ability to implement a range of SDN policies [20, 55, 68], (2) the performance and overhead of the synthesized policies, and finally, (3) correctness of the flow table rule installation strategy.

### 2.6.1    Feasibility

We explore two aspects of feasibility of NetEgg. First, is the policy generation process efficient in terms of execution time? Second, is NetEgg easy to use, in terms of the number of input scenarios required and lines of configuration code?

Table 2.11 summarizes our findings in terms of execution time and scenario size. We report the total number of events in the scenarios used to program each policy, and the number of scenarios. We also report the computation time of the synthesizer to generate the policy from scenarios.

We make the observation that most of the scenarios are expressed in less than 5 events, with some outliers requiring up to 10 events. NetEgg is also efficient, and in all examples, requires no more than 402 ms.

We further perform a "code size" comparison, by counting the number of lines in each

| | #EV | #SC | Time | Description |
|---|---|---|---|---|
| maclearner1 | 3 | 1 | 11 ms | The illustrative example. |
| maclearner2 | 3 | 1 | 15 ms | Section 2.5.1. |
| auth | 3 | 2 | 13 ms | Deny traffic from hosts unless the hosts are authenticated (adapted from [3]). |
| gardenwall | 5 | 3 | 52 ms | Deny, allow or redirect traffic from a host based on its state (adapted from [3]). |
| ids | 3 | 2 | 15 ms | Deny traffic from infected hosts detected by IDS (adapted from [3]). |
| monitor | 3 | 2 | 13 ms | Monitor a host's traffic based on external input events (adapted from [3]). |
| ratelimiter | 10 | 5 | 147 ms | Forward flows using different links based external rate limiting event (adapted from [3]). |
| serverlb | 7 | 3 | 143 ms | Forward traffic from hosts to a single port or forward traffic of each host to a pre-chosen port (input as an external event) for load balancing (adapted from [3]). |
| stateful firewall1 | 3 | 2 | 12 ms | The first use case in Section 2.5.2. |
| stateful firewall2 | 3 | 2 | 16 ms | The second use case in Section 2.5.2. |
| stateful firewall3 | 6 | 3 | 107 ms | The third use case in Section 2.5.2. |
| trafficlb | 7 | 3 | 402 ms | Similar to serverlb, but considers traffic from source-destination pairs (adapted from [3]). |
| ucap | 3 | 2 | 13 ms | Block traffic from hosts based on the user's input (adapted from [3]). |
| vmprov | 3 | 2 | 24 ms | Forward a host's flow to a primary or a backup server (adapted from [3]). |
| TCP firewall | 9 | 5 | 64 ms | Section 2.5.3. |
| ARP proxy | 5 | 2 | 49 ms | Section 2.5.4. |

Table 2.11: Network policies generated from scenarios. #SC is the number of scenarios used to synthesize the policy, #EV is the total number of events in scenarios, Time is the running time of the synthesizer.

of our NetEgg example configurations, and compare with corresponding policies that we implemented in Pyretic and POX. Table 2.12 summarizes our results. We observe that NetEgg is more concise, achieving a 4× and 10× reduction in code size compared with Pyretic and POX.

|  | NetEgg | Pyretic | POX |
|---|---|---|---|
| maclearner2 | 3 | 17 | 29 |
| stateful firewall1 | 3 | 21 | 58 |
| TCP firewall | 9 | 24 | 68 |

Table 2.12: Lines of code to implement policies in different programming abstractions. We report the total number of events in scenarios for NetEgg, lines of code for Pyretic and POX implementations.

### 2.6.2 Performance Overhead

NetEgg uses the policy table as the policy abstraction, and a generic interpreter to execute the policy table. Unlike hand-crafted implementations which can be customized to policies, generic execution of our abstraction of policies may incur additional overhead. We evaluate the generic execution engine of NetEgg using a combination of targeted benchmarks and end-to-end evaluation.

**Cbench Evaluation**

We first use the Cbench [86] tool to evaluate the response time of three policy implementations.

**Experiments.** We emulate one switch in Cbench, which sends one packet-in request to the controller as soon as it receives a reply for last sent request. The response time corresponds to the time between sending out a request and receiving its reply, which hence includes the execution time of policy implementations. For comparison, we also evaluate the policies' implementations in POX.

**Results.** Figure 2.14 shows the response time for the policy implementations in POX and NetEgg. We note that in all cases, the differences in response times between the POX and NetEgg versions are within 12%. In the case of MAC learning and stateful firewall, the differences are negligible (<1%). We observe that the response time between implementations in POX and NetEgg is comparable, which suggests our policy abstraction incurs reasonably small overhead on execution.
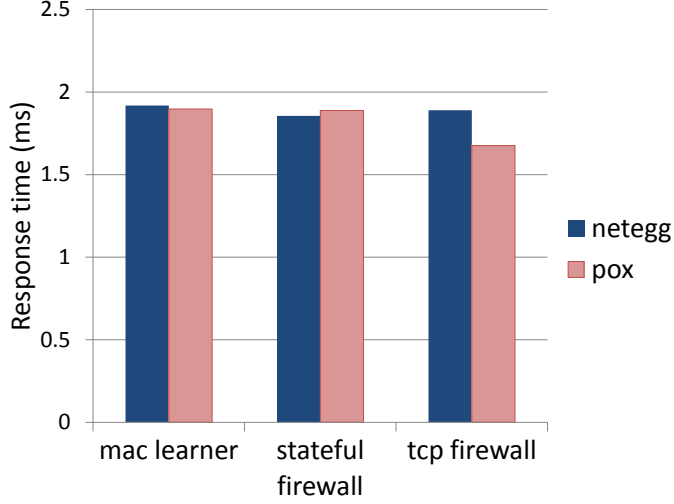
Figure 2.14: Response time for POX and NetEgg implementations.

**End-to-end Performance**

Our next set of experiments aim to validate that the synthesized implementation closely matches the hand-crafted implementation on end-to-end performance for network applications such as HTTP.

**Experiments.** We emulate a fattree topology [15] in Mininet, which consists of 20 switches and 16 hosts. We setup a HTTP server on one host, and run httperf on all other hosts as clients. Httperf sends HTTP requests from the clients to the server, and measures the HTTP connection time for each request, which is the time between a TCP connection is initiated and it is closed. We run httperf with different rates of sending requests, and the same number of connections (e.g. at rate 5 request/second, httperf issues 5 requests per client). Each run starts from the initial network state. On the controller side, we run the MAC learner policy using two implementations: POX and NetEgg.

**Results.** Figure 2.15 reports the average connection time over all 15 clients. The x-axis is the rate of HTTP requests issued by the clients. As expected, the connection time under the NetEgg implementation matches closely to that under hand-crafted POX implementation. These results suggest our synthesized implementation is able to achieve comparable end-to-end performance as hand-crafted implementations. This also further verifies that execution of our policy abstraction incurs small overhead, and our flow table

rule installation is efficient.
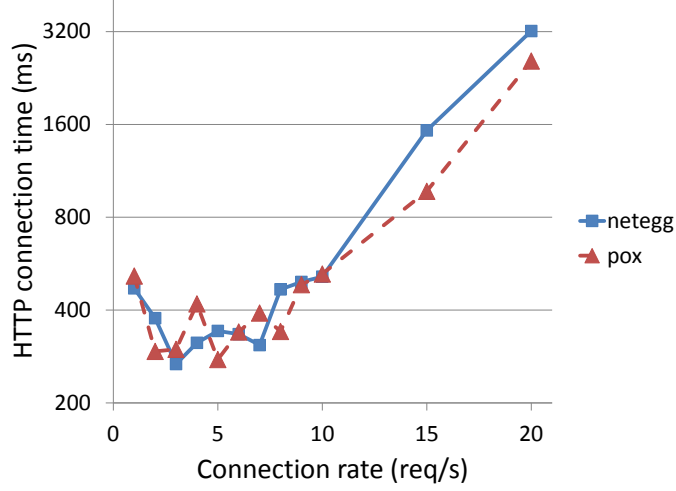


Figure 2.15: HTTP connection time.

### 2.6.3  Rule Installation

To achieve realistic performance, our interpreter infers and installs flow table rules. We validate the correctness of our rule installation strategy using emulation-based experiments.

**Experiments.** We run the synthesized MAC learner policy on the controller, and emulate a simple topology with a single switch connected with 300 hosts in Mininet [59]. We partition these hosts into two groups, with 150 hosts per group. Every host in a group sends 100 ping messages to another host in the other group with 1 message per second. For comparison, we run the set of experiments under two settings, one with flow table installation and one without.

**Results.** We plot the average RTT for all ping messages over time in Figure 2.16. The red line corresponds to the policy implementation without installing flow table rules. This implementation has a high RTT consistently over time, due to the fact that every packet is sent to the controller. The blue line corresponds to the case with installation. We observe that only the first message experiences high latency, and subsequent messages has significantly smaller RTT below 0.1 ms. This fact suggests that our installation strategy is able to infer flow table rules from the first incoming packet-in event, and correctly install

the rules onto the switch. Hence, subsequent packets are all processed by the switch.
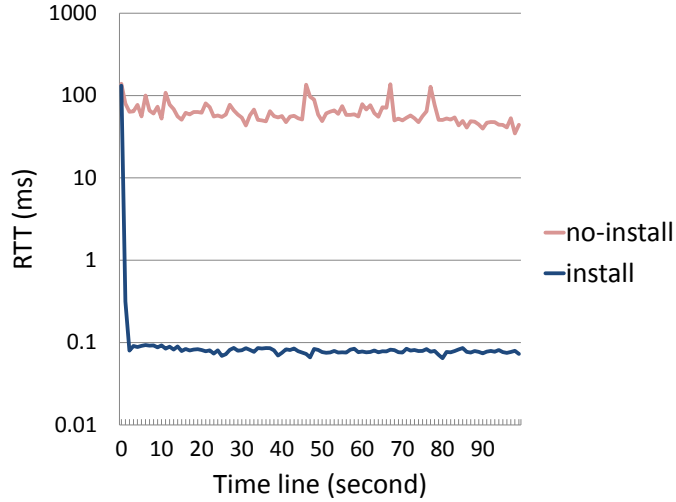


Figure 2.16: Effects of flow table rule installation.

## 2.7    User Study

To understand the feasibility of the proposed approach in practice, we conducted a user study involving 12 master and PhD students, all majored in Computer and Information Science, in the fall semester of 2015. In this section, we describe the main results we found in the user study.

### 2.7.1    Setup

In this user study, we asked the students to program three SDN policies: the stateful firewall, the learning switch, and the TCP firewall. All students were required to program these policies in both POX and NetEgg. For students who may not know how to program SDN using POX, we refer them to the online resources on programming in POX [5, 6, 8, 11, 59], and they needed to learn the use of POX. We developed a web-based GUI for NetEgg. Figure 2.17 shows a snapshot of the GUI. This GUI allowed the students to draw the scenarios, and then presented the generated policy table in a visual way. We prepared a webpage of basic instructions on NetEgg for the students' reference.
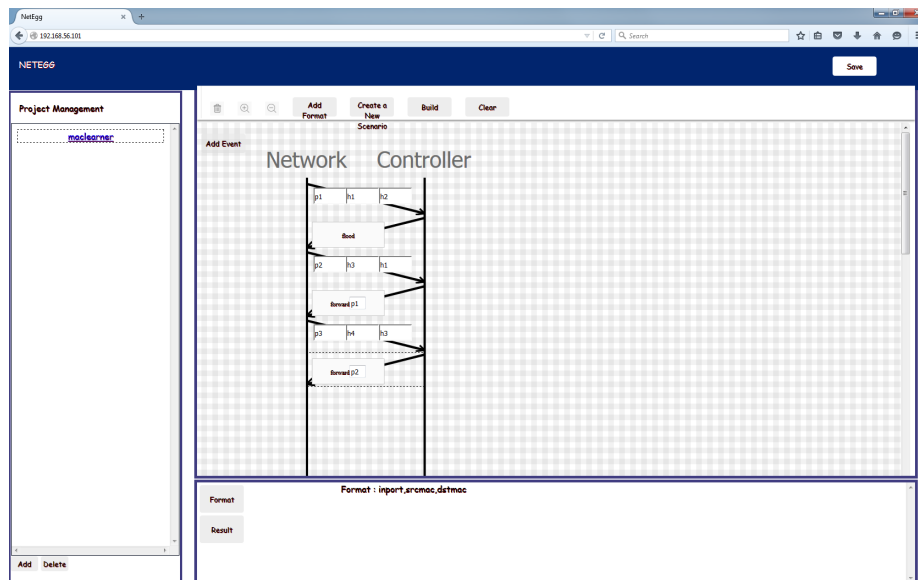
Figure 2.17: Web-based GUI of NetEgg.

To quantify the feasibility of NetEgg, we considered two major objectives. First, we measured how much time the students spent on each programming assignment using both POX and NetEgg. For this measurement, we asked all students to time the programming time of each programming assignment by themselves, and submit it along with the assignments. Second, we measured how intuitive of NetEgg the students found compared to POX. For this metric, we asked the students to report how much time they spent in learning the programming of POX and NetEgg. Moreover, we asked the students to rate the "intuitiveness" of POX and NetEgg after all programming assignments, from score 0 (least intuitive) to 10 (most intuitive).

### 2.7.2 Results

**Programming time**

Figure 2.18 shows the average programming time of the three policies in POX and NetEgg.

We make the following two observations.

First, NetEgg may reduce the programming time for some policies. Specifically, we observed that NetEgg was able to reduce the programming time of the first two network policies by 50% and 72%, respectively. To understand why NetEgg was able to achieve
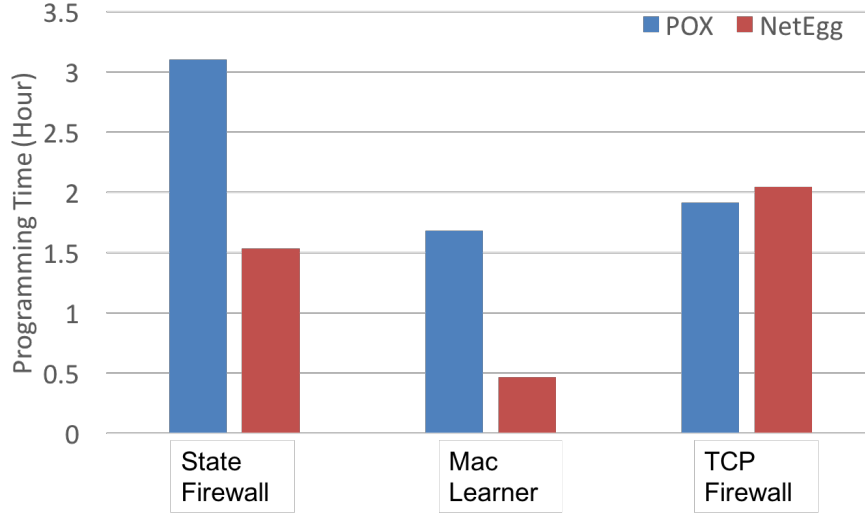
Figure 2.18: Programming time comparison.

this reasonably good reduction in programming time, we further analyzed the programs the students wrote. We found that the scenario-based programming approach offered by NetEgg was able to reduce the manual efforts of writing low-level code. For example, the students wrote from 29 lines to 150 lines of code for the stateful firewall policy, while they used no more than 3 scenarios to generate the policy. One of the reasons why the students needed to write considerably more code in POX is that they needed to handle low-level semantics offered by POX as well as to come up with imperative code in order to implement the policy, which could be simplified by NetEgg by allowing them to simply provide the behaviors of the policy.

Second, NetEgg may result in more programming time on complicated policies. In particular, the students spent 6% more time in programming the TCP firewall on average compared to the programming time in POX. This is perhaps not surprising, given that our heuristics aim at generating the smallest policy that is consistent with the scenarios. Therefore, as commented by the students on the TCP firewall policy, they spent more time in coming up with the right scenarios to tweak the generated policy table, even when they knew what the correct policy table should be. This observation may suggest a plausible improvement to NetEgg, which is to allow the users to provide additional information (e.g. the state tables used in the policy) to the synthesizer in order to accelerate the programming

process.

**Intuitiveness**

We first report the time the students spent on learning how to program in POX and NetEgg. Most students indicated that they spent several hours in learning POX, while they spent less than one hour in learning NetEgg. In particular, 4 out of the 12 students reported the estimated number of hours they spent in learning. On average, the learning time of POX is about 2.6 hours, and that of NetEgg is about 0.9 hours. Note that all students were majored in Computer Science, and had good programming background. We believe that the reduction of learning time would be greater for the users who may not be trained in programming. This result suggests that NetEgg comes with a much smoother learning curve compared to traditional programming approaches such as POX.

Second, we present the "score of intuitiveness" reported by the students. 11 students reported their scores for POX and NetEgg, with 0 being the least intuitive, and 10 being the most intuitive. Figure 2.19 shows the scores by the 11 students.
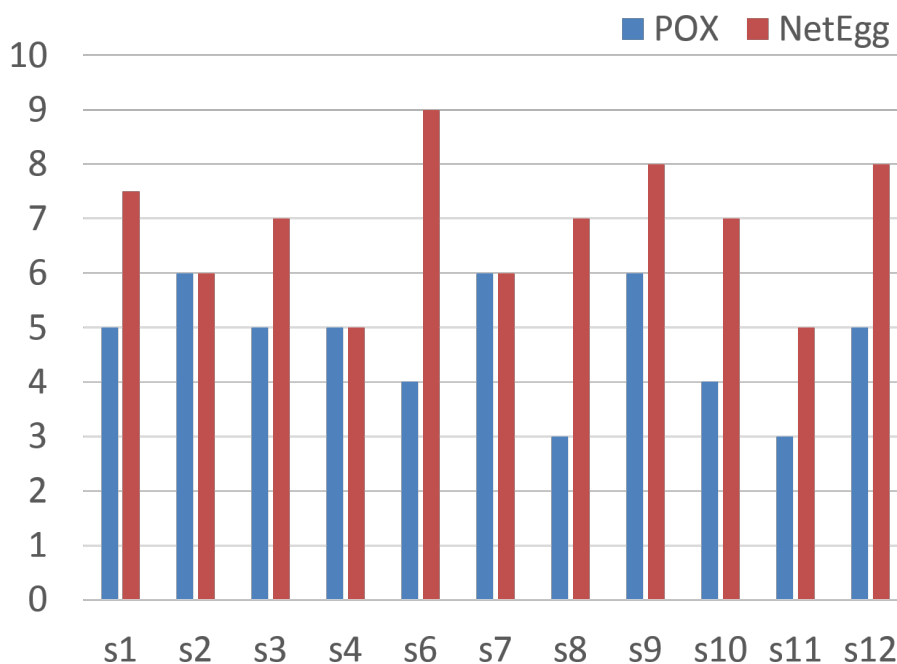


Figure 2.19: The "Score of intuitiveness" rate.

The average score of POX is 4.7, while the average score of NetEgg is 6.9. Furthermore, every student rated NetEgg higher than POX. This result suggests NetEgg may be more intuitive than traditional programming approaches.

The results of the preliminary user study seem promising. The user study suggests that NetEgg was able to reduce the programming time for 2 out of 3 policies in our user study. Furthermore, NetEgg was shown to be a more intuitive programming approach.

## 2.8   Discussion

**Policy Correctness.** NetEgg generates network policies directly from example behaviors, which may not cover all possible behaviors of the desired policies. Therefore, while NetEgg aims at generating a policy which is consistent with all scenarios, NetEgg does not guarantee the correctness of the generated policy.

To debug/verify the generated policy, there are other orthogonal debugging and verification tools. For example, one may use ATPG [95] and BUZZ [38] (which is based on KLEE [26]) to generate test cases for the policy; Vericon [20] and Nice [27] can be used to verify the correctness of the generated policy.

As one possible future work, it is interesting to explore the feasibility of programming network policies in a hybrid programming framework similar to the Sketch [80] framework, in order to ensure the correctness of the generated policy. For example, the user may also provide the synthesizer the information such as the state tables and the number of states that should be used, in addition to the example behaviors.

**Comparison with CEGIS.** In NetEgg, we use customized heuristics for the policy learning problem, which can also be solved using other program synthesis techniques. As a comparison, we also implemented a counter-example guided inductive synthesis (CEGIS) approach [80] using the solver Z3 [35]. We observe that our customized algorithm works more efficiently. In particular, the CEGIS approach used 371 ms, 536 ms and 281 ms for the learning switch, stateful firewall and ARP proxy policy, while our customized heuristics used 11 ms, 12 ms and 49 ms respectively.

**Limitations.** NetEgg aims at generating stateful policies which can be expressed by our policy tables. Therefore, NetEgg is not good for policies that are hard to be expressed

41

by a policy table. For example, policies such as traffic engineering often require solving optimization problems, which cannot be synthesized by NetEgg.

# Chapter 3

# NetQRE: Specification and Implementation for Quantitative Network Policies

Network management today often requires dynamic updates in response to traffic engineering and security events. For example, in data centers, heavy hitters [16, 91] need to be detected in real-time and bandwidth limits may be imposed on them. Within an enterprise network, users can be rate-limited if they exceed quotas on their application usage. Network traffic anomalies that are detected require immediate mitigation strategies to block potential security attacks [41].

Decisions for such updates are based on *quantitative network policies* that capture logic based on a variety of network and application-level performance metrics. Quantitative network policies involve two aspects: (1) monitoring a variety of network and application-level performance metrics, and (2) real-time network configuration updates in response to monitoring results in order to meet performance and security goals.

Today, there are several point-solutions to support certain aspects of quantitative network policies. However, they suffer from one or more of the following limitations. First, a large majority of network measurement tools focus on flow-level measurements [32, 36, 37, 89, 91, 92] that do not capture application-level or session-level semantics. This precludes a range of policies that are application-dependent, for example, rate-limiting users based

on their VoIP call usage. Second, these tools tend to provide ad-hoc solutions that are difficult to generalize or customize. Recently, programming frameworks for software-defined networks (SDN) have been proposed, such as Frenetic [40], Pyretic [67], FlowLog [72]. However, they do not support integration with queries beyond basic flow-level counters, and there is no language support to capture quantitative policies at the application or session level.

To address the above limitations, in the chapter we present NetQRE, a practical tool aimed at simplifying the specification and implementation of quantitative network policies. Our proposal is based on the observation that traffic patterns such as a TCP connection and a application-level session can be specified using regular expressions, which are an abstraction that programmers find natural to use, as shown by its usage in network management, for example, application-level packet classification [2], signature-based attack detection [4, 76]. While preserving the simplicity of regular expressions, our language extends regular expressions in support for quantitative network policies and thus allows programmers to reason about interactions across packets, and express a variety of quantitative network policies that span multiple packets grouped into flows and application-level sessions.

We organize the rest of this chapter as follows. First, we offer an overview of NetQRE in Section 3.1. To illustrate the features of NetQRE, we provide 3 example programs written in NetQRE. Second, we present the design of the NetQRE language in Section 3.2. We describe the high-level constructs in NetQRE for monitoring packet streams. In Section 3.3, we describe a range of quantitative network policies, and demonstrate how to specify them using NetQRE. We then describe the compilation algorithm for NetQRE in Section 3.4, in order to compile specifications in NetQRE to efficient low-level code. We finally describe the implementation of our NetQRE system in Section 3.5 and the evaluation results in Section 3.6.

## 3.1 Overview

Figure 3.1 shows the architecture of NetQRE. To program a quantitative network policy, a programmer views the input as the entire stream of packets, instead of single incoming packets. The programmer then simply specifies this policy on the input stream in a declar-
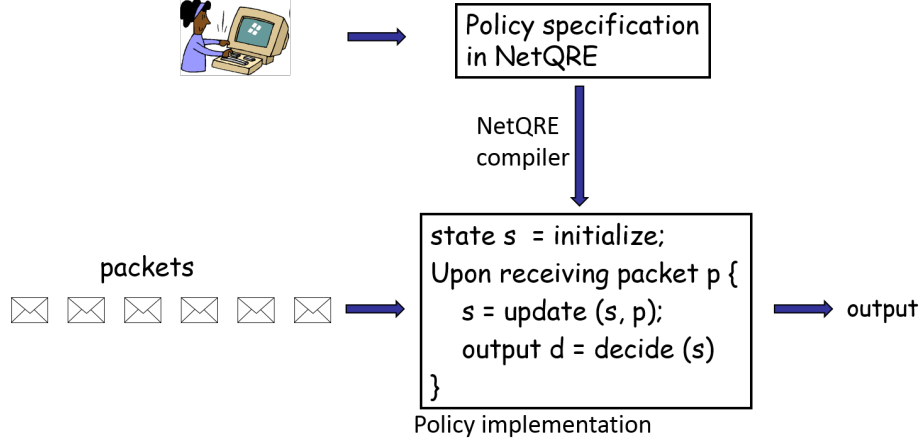
Figure 3.1: NetQRE architecture.

ative fashion. The NetQRE compiler automatically generates efficient low-level imperative code that implements the specification. At runtime, each incoming packet is parsed, and headers are extracted into a format that can be referenced within each NetQRE program. The execution on the packet involves updating any state in the NetQRE runtime, followed by executing the actual program itself to generate output. The NetQRE tool can be deployed in different settings, for example, a tap on a SPAN port analyzing mirrored traffic, an inline solution, or running in the cloud as a virtualized middlebox. Our language design and compiler is agnostic to the deployment setting.

### 3.1.1 Examples

We present three example NetQRE programs to highlight some of its key features. We start with a simple example of packet counting, before progressing to more complex ones. The actual language definition and more complex use cases will be presented in Section 3.2 and 3.3 respectively.

**Example 1.** In our first example, we consider a simple example of counting the number of packets seen so far. In an imperative language, one needs to maintain a counter that remembers the number of all packets, and increment the counter as every packet comes in. To program in NetQRE, one can simply iterate the stream of packets, and count the number in a declarative way as shown below:

```
sfun int count = iter(/./?1, sum);
```

This program defines a *stream function*, as specified by the keyword `sfun`. This implicitly defines an input stream which is the stream of packets. This function outputs a value of type `int`, and `count` is the name of the stream function. There is no explicit argument specified in the declaration of `count`. Its body is an expression followed by '='.

To understand the definition of the function, we first explain the expression `/./`. This expression defines an enhanced regular expression which we detail in Section 3.2. In this regular expression, `.` is a predicate that matches every single packet. A successful match will return a value of `1`.

The outer function `iter` has two arguments, with the first being the function described above, and the second an aggregation operator `sum`. The `iter` function splits each input stream into multiple *sessions*, such that applying its first argument to every session results in a valid return value, and then sums up all the return values. In this case, since the inner function returns a valid value `1` when reading a single packet, the `iter` function iterates all packets in the stream and applies the inner function. In other words, the whole function counts the number of packets in the stream.

**Example 2.** Our second example computes the average number of packets in a TCP connection between the source IP-port pair (`1.2.3.4`, `80`) and an arbitrary destination IP-port pair (`y`, `p`). For ease of exposition, we assume that all connections start with a SYN packet and end with a FIN packet in this example. We defer more complex examples involving sequence numbers to later in Sec. 3.3.

```
sfun int avg_conn_len(IP y, PORT p) =
    filter_tcp(srcip='1.2.3.4', srcport=80, dstip=y, dstport=p
        ) >>
    iter(tcp_conn_len, avg);
```

This function uses two expressions separated by the *stream composition* operator `>>`. At a high level, this function first filters all TCP packets with corresponding IPs and ports from the stream using the function `filter_tcp`, and applies the second expression to the TCP packet stream. The second `iter` expression iterates through all connections in the stream

and counts the number of packets in every connection using the function `tcp_conn_len`, and returns the averaged value.

The function `tcp_conn_len` calculates the number of packets transmitted in each TCP connection. The regular expression `[syn=1][fin=0]*[fin=1]` specifies a TCP connection pattern, by our assumption for this example. This pattern matches a stream if it starts with a SYN packet, followed by a sequence of non-FIN packets and ends with a FIN packet. For a stream that matches this pattern, the function returns the number of packets in the stream, by using the previously defined function `count`.

```
sfun int tcp_conn_len =
  /[syn =1][fin =0]*[fin =1]/? count ;
```

**Example 3.** In our third example, we show how to use previously defined functions and *aggregation expressions* in NetQRE to compute the largest possible average connection length among all destination IP-port pairs (y, p). The program has just a single expression, which calls `avg_conn_len` on all IP-port pairs, and returns the maximal among all return values from the calls, as shown below.

```
sfun int max_conn_len =
  max{avg_conn_len(y, p) | IP y, PORT p};
```

This example is significantly more complex to write in an imperative language. A typical implementation needs to maintain a state and a counter for each destination IP-port pair, in order to remember the state of the connection as well as the running count of the number of packet in the connection. In order to compute the average length, the program needs to additionally maintain the number of connections. Every time a connection ends, the imperative program needs to update maximum length over all destination IP-port pairs. However, with NetQRE, all functionalities can be implemented in an intuitive and concise way as illustrated above. The complexities of maintaining states and optimization of the implementation is abstracted away and handled automatically by our compiler.

## 3.2 The NetQRE Language

In NetQRE, a packet is modeled as a sequence of bytes, and we use parsing functions to extract information from the packet. Common parsing functions include `srcip` which returns the source IP of the packet, `srcport` (source port), `syn` (SYN bit), `data` (bytes in payload), and `time` (the time stamp on receipt of the packet). The parsing functions can be customized by the user, for example, to extract application-level headers.

Values, variables and functions in NetQRE are typed. NetQRE offers basic types, such as `int`, `bool`, `string`, as well as a set of domain-specific types, such as `IP` (IP addresses), `Port` (TCP and UDP ports), `packet` (all packets), and `action`. The `action` type consists of pre-defined functions that either generate alerts or send updates to switches. NetQRE also provides high-level types such as `Conn` (tuple of source IP-port and destination IP-port), which is used for TCP and UDP connections.

NetQRE offers a convenient way to write *stream functions* to process packet streams. A stream function takes as input a stream of packets, and produces as output values (e.g. monitoring results to an application)/ actions (e.g. alerts to the controller) and packets (e.g. packets filtered from an application). A stream function can be specified as below.

```
sfun type func_name(type var) = exp;
```

The stream function declaration includes the keyword `sfun`, returned type of the stream function, followed by the name of the function. Any other arguments are specified following the function name. The body of each stream function (right-hand-side to '=') consists of *expressions* which are used to specify the functionalities of the stream functions.

Figure 3.2 shows a summary of the syntax of NetQRE expressions. Expressions in stream functions are based on the theoretical foundations of *quantitative regular expressions* (QRE) [18], a novel proposal that integrates regular expressions with numerical computations. In the rest of this section, we describe the features of NetQRE expressions in stages. We first introduce how to use an extension to regular expressions to detect patterns of the input stream. Second, we discuss how to associate values and actions for the stream. Finally, we describe a set of high-level operations that allow modular programming of stream functions.

| Predicate | $P$ | $::=$ . $\|$ [field = value] |
|---|---|---|
| | | $\|$ [field = variable] |
| | | $\|$ $P\&P$ $\|$ $(P$ '$\|$|' $P)$ $\|$ $!P$ |
| Regular Exp. | $re$ | $::= P$ $\|$ $re$ $re$ $\|$ $re$ * $\|$ $re$ '$\|$|' $re$ |
| Conditional | $cond$ | $::= exp$ ? $exp$ $\|$ $exp$ ? $exp$ : $exp$ |
| Aggregation | $agg$ | $::=$ aggop $\{$ $exp$ $\|$ type variable $\}$ |
| | aggop | $::=$ **sum** $\|$ **avg** $\|$ **max** $\|$ **min** |
| Split | $split$ | $::=$ **split**$(exp$ , $exp$, aggop$)$ |
| Iteration | $iter$ | $::=$ **iter**$(exp$, aggop$)$ |
| Composition | $comp$ | $::= exp$ >> $exp$ |
| Expression | $exp$ | $::=$ value $\|$ action |
| | | $\|$ $exp$ op $exp$ $\|$ op $exp$ |
| | | $\|$ $re$ $\|$ $cond$ $\|$ $agg$ $\|$ $split$ $\|$ $iter$ $\|$ $comp$ |

Figure 3.2: Syntax of NetQRE expressions.

### 3.2.1 Pattern Matching over Streams

The basic feature of NetQRE is to detect the patterns of the input packet stream. As the basic building block, NetQRE uses an extension of regular expressions, to which we refer as *parameterized symbolic regular expressions* (PSRE), for pattern matching over the input stream. Regular expressions (RE) are widely used for pattern matching over strings (sequences of bytes). Typically, a RE uses a fixed finite alphabet, and the atoms in a RE are symbols in the alphabet. In NetQRE, a PSRE generalizes a RE in the two ways.

First, the atoms in a PSRE are predicates over packets instead of single symbols, which allows a PSRE to handle very large and potentially infinite alphabet. This property makes PSRE an appealing fit for network policies, since the space of packets is often very large and typically one is only interested in the values of some fields in a packet (e.g. the source and destination) instead of the whole packet itself.

Second, PSRE allows the use of *parameters* to represent unknown values in the predicate. With this generalization, a PSRE can detect a variety of patterns using different instantiation of the parameters. As a result, it allows the programmer to specify applications such as counting the number of distinct IP addresses appeared in the stream, which is hard, if not impossible, to be specified without parameters, because no concrete predicates can be defined without knowing those IP addresses at runtime.

**Predicate.** A basic packet predicate `[f = v]` checks whether the value in the field `f` of a

packet is v. As an example, the predicate `[srcip='1.0.0.1']` matches all packets whose source IP address is `1.0.0.1`. As an example of the use of parameters, `[srcip=x]` defines a function from the domain of `x` (i.e. IP addresses in this case) to a concrete predicate over packets. We also use `.` to denote the predicate that matches all packets. Predicates can be composed using standard boolean combinations. NetQRE also provides handy macros for widely used predicates. For example, `is_tcp(c)` is a short-hand for the predicate that matches a TCP packet in the connection `c`.

**PSRE.** Like REs, the basic operations for PSREs are concatenation, union and Kleene star. For example, `/[syn=1][syn=0]*/` matches a stream of packets where the first packet is a SYN packet followed by a sequence of non-SYN packets (including 0 non-SYN packets). Note that, syntactically, predicates are enclosed in square brackets and a PSRE is enclosed in a pair of slashes in NetQRE, and `*` is the Kleene star operator. To illustrate the use of parameters, consider the example to count distinct source IP addresses in the stream. The core PSRE to implement this example is the function `exist(x)` which checks whether a source IP `x` appeared in the stream. The function can be specified using the PSRE as below.

```
sfun bool exist(IP x) = /.*[srcip=x].*/;
```

We defer the discussion of the full expression for this example in Section 3.2.5.

Semantically, a PSRE defines a function that maps the values of its parameters and the input stream to a boolean value. Given an instantiation of all parameters in a PSRE using concrete values, the PSRE accepts a regular set of streams, which defines the domain of the PSRE under the instantiation.

### 3.2.2  Conditional Expressions

Given the pattern matching capability for the input stream, it is natural to use conditional expressions to incorporate PSRE with other values and actions in order to assign cost/generate actions to the stream.

In NetQRE, a conditional expression has the form $exp? \, exp_1$, where, $exp$ is an expression that returns boolean values such as a PSRE defined above, and $exp_1$ is an expression

in NetQRE. A stream evaluated to `true` by $exp$ will be applied to $exp_1$, otherwise this expression is not defined and returns `undef`.

For example, the expression `/./?1` returns `1` if the stream matches the PSRE `/./` (i.e. the stream consists of a single packet), otherwise it returns `undef`; the expression `/.*/? size(last)` returns the size of the last packet in the stream. The keyword `last` denotes the last received packet in the input stream. As another example, `(count>k)?alert` returns an action `alert` when the total number of packets in the stream is larger than `k`. Here, `count` is a stream function that counts the number of packets in the stream, and `alert` is a pre-defined action to generate an alert event, for example, to a controller node.

A conditional expression can also be specified as $exp?\ exp_1\colon exp_2$, which applies $exp_2$ to the stream if $exp$ is not satisfied. To ensure the consistency, $exp_1$ and $exp_2$ should return the same type for the stream. As an example, the expression `/[srcip='1.0.0.1']/?1:0` returns `1` if the stream contains a single packet with source IP `1.0.0.1`, and it returns `0` if the stream contains multiple packets, or the only packet in the stream does not have the source IP.

Given concrete values of the parameters of a conditional expression $exp?\ exp_1$, the domain of it is defined as $domain(exp) \bigcap domain(exp_1)$. For the ternary conditional expressions $exp?\ exp_1\colon exp_2$, its domain is defined as $(domain(exp) \bigcap domain(exp_1)) \bigcup (\overline{domain(exp)} \bigcap domain(exp_2))$.

### 3.2.3   Stream Split

In many network applications, the input stream consists of multiple phases. For example, a VoIP call splits into three phases as shown in the motivating example. It is convenient to handle different phases of the stream separately, and combine the results of each phase in a modular way. Following the operators defined in Quantitative Regular Expressions [18], NetQRE uses the `split` operator to split the stream and compose two stream functions.

A stream split expression has the form `split(f,g,aggop)`, where `f` and `g` are two NetQRE expressions, and `aggop` is an aggregation operator such as `sum`, `avg` (average), `max`, and `min`.

Given concrete values of all parameters, and an input stream $\rho$, a `split` function splits

$\rho$ into two substreams $\rho_1$ and $\rho_2$, such that f is defined on the first substream $\rho_1$ and g is defined on $\rho_2$. f and g are applied respectively to the two substreams, and the returned values are aggregated using aggop, as the return value of split(f,g,aggop). The split operation is a natural quantitative generalization of the concatenation operation in regular expressions, and Figure 3.3 illustrates how a split expression works.
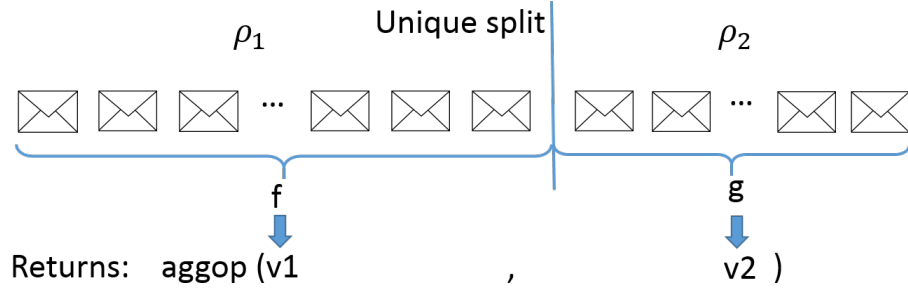


Figure 3.3: Illustration of split.

Note that, in order to return a unique value for a split expression, it is required that the splitting is unambiguous. That is, for all input streams and all values of parameters in the expression, there is at most one way to split stream, such that $\rho_1$ and $\rho_2$ are in the domain of f and g, respectively. The property of unambiguity can be checked efficiently at compile time [18]. When no unambiguous splitting is possible, the split expression returns undef for the input stream.

As an example of split, consider the example of counting the number of packets since the last SYN packet in the stream. Naturally, one can split the stream into two substreams separated by the last SYN packet, and count the number of packets in the second substream, as shown in the following expression.

```
split ( any ?0 ,  last_syn ? count ,  sum )
```

Here, any is the PSRE /.*/ that matches any packet streams, and last_syn is the PSRE /[syn=1][syn=0]*/ that matches a stream starting with a SYN packet and followed by non-SYN packets. Putting together, the split expression splits the input stream before the last appearance of a SYN packet. Note that the first substream is assigned the value 0, while the second substream is applied to the count function which is first introduced in Section 3.1.1.

### 3.2.4 Stream Iteration

A network application often requires to iterate over the input stream. For example, counting the number of packets in the stream requires to iterate over all single packets.

Similar to the `split` expression, NetQRE offers the `iter` operator to iterate over the stream. An `iter` expression is of the form `iter(f,aggop)`, where `f` is an expression in NetQRE and `aggop` is an aggregation operator. The `iter` expression splits the input stream into multiple substreams, such that `f` is defined on each substream. It then iterates through all substreams and evaluates `f` on each substream, and finally aggregates the return value on each substream using the aggregation operator. The `iter` operator is a quantitative generalization of the Kleene star operation, and Figure 3.4 illustrates this process. Again, the splitting is required to be unambiguous for all input streams to `f`.
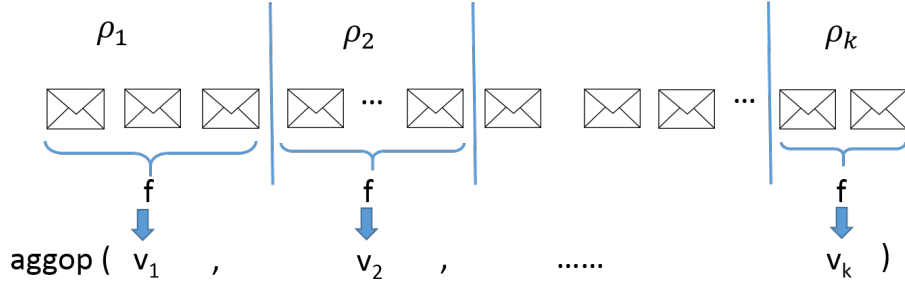


Figure 3.4: Illustration of `iter`.

As an example, the function `count` that counts the number of packets in the stream

```
sfun int count = iter(/./?1, sum)
```

splits the stream into single packets, and the inner expression returns `1` for each single packet. As a result, the whole `iter` expression counts how many packets in the stream.

### 3.2.5 Aggregation over Parameters

Aggregation is a common feature that many policies share. For example, computing the average flow size needs to aggregate the average size across multiple flows. NetQRE offers high-level aggregation expressions to aggregate functions over parameters.

Aggregation expressions are of the form `aggop{f | T x}`, where `x` is a parameter, with its type `T`, and `f` is an expression where the parameter `x` appears in it. Intuitively, the aggre-

gation expression goes through all possible values of x, and evaluate f using the substituted value for x, and finally aggregates all valid returned values using the aggregation operator aggop. Given concrete values for parameters in the aggregation expression ( note that x is not considered as a parameter in the aggregation expression), it is required that f has the same domain across all values for x, which is defined as the domain of the aggregation expression. Revisiting the example of counting distinct source IP addresses in a stream, we can use the expression sum{exist(x)?1:0 | IP x}. Note that the domain of the expression exist(x)?1:0 is all the streams, and thus the aggregation expression above is defined on all input streams.

### 3.2.6 Stream Composition

Typically the input stream consists of packets from multiple sources and destinations. Oftentimes, the programmer only wants to handle a stream from a particular source, for example. NetQRE offers the stream composition operator >> that allows to preprocess the stream before applying another stream function.

Stream composition expressions are of the form f >> g where f and g are two stream functions. The stream composition allows the processing of a stream using the first stream function f repeatedly on every prefix of the stream, and the returned outputs yield a new stream that is then piped as the input stream to a second stream function g. For example, if the stream contains two packets $(P_1, P_2)$, f is first applied to $P_1$ as a single-packet stream, and then $(P_1, P_2)$. The output of f on the two substreams is then piped to g.

Stream composition is useful when the stream of packets need to be preprocessed in order to filter related packets for further processing. For example, counting the number of all packets in a TCP connection c can be specified using filter_tcp(c)>> count, which first filters all TCP packets in the connection c using the function filter_tcp, and then counts the filtered stream using count. The function filter_tcp is defined as follows:

```
sfun packet filter_tcp(Conn c) =
  /.*[is_tcp(c)]/?last;
```

Filter functions are convenient and used extensively in our use cases. As a short-hand, NetQRE uses filter(p) for the filter function that filters packets satisfying the predicate

`p`. For example, the above filter function can be abbreviated as `filter(is_tcp(c))`.

Stream composition can also be used to filter packets according to timestamps. NetQRE builds in two filter functions based on timestamps. The `recent(t)` function filters the stream in the recent `t` seconds, and the `every(t)` function periodically filters the stream in every `t` seconds. For example, `recent(5)>>count` counts the number of packets in the recent 5 seconds.

## 3.3   Use Cases

We provide use cases, ranging from flow-level traffic measurements, to complex examples involving application-level content analysis and dynamic updates.

### 3.3.1   Flow-level Traffic Measurements

We highlight three measurement tasks that have been proposed recently as a means to do flow scheduling and attack detection.

**Heavy hitter.** Heavy hitters [37] are flows that consume bandwidth larger than a threshold `T`. A key step to identify a heavy hitter is to count the size of a flow. In this example, we consider a flow as a source-destination pair. A natural way to specify this functionality is to first filter all packets from a source `x` to a destination `y`, and then count the size of the filtered stream.

```
sfun int hh(IP x, IP y) =
  filter(srcip=x, dstip=y) >> count_size;
```

`hh` first filters packets based on the source and destinations IP, and second, the filtered stream is piped into `count_size` which counts the size of all packets in the stream. Due to space limits, we do not show the definition of `count_size` which is similar to that of `count`.

Second, to alert a heavy hitter in real time to the controller, we can use the following program.

```
sfun action alert_hh =
  (hh(last.srcip, last.dstip)>T) ?
  alert(last.srcip, last.dstip);
```

The function `alert_hh` checks for every newly received packet (i.e. `last` in the stream) that whether its flow reaches the threshold `T`, and issues an `alert` correspondingly. By further applying the time-based filtering `every(5)>>alert_hh`, one can detect heavy hitters in every 5 seconds.

**Super spreader.** A super spreader [91] is the host that contacts more than $k$ distinct destinations during a time interval. Like the use case of heavy hitter detection, a key step is to count how many distinct destinations a source `x` contacted. The following NetQRE program does this counting.

```
sfun int ss(IP x) =
  sum{ exist(x, y)?1:0 | IP y};
```

The function `exist(x,y)` checks whether the source-destination pair `(x,y)` appeared in the stream. The `ss` function uses an aggregation function `sum` to aggregate all possible destinations, and thus gives the total number of distinct destination addresses `x` contacted.

Similar to heavy hitter detection, we can use stream composition to filter the input stream based on time as well as traffic types.

**Entropy Estimation.** This case measures the entropy of the traffic [58], defined as $-\sum_i f_i \log f_i$, where $f_i$ is the frequency of the appearance of an IP address. We first need to count the number of packets from a source `x`, and then compute the frequency as follows.

```
sfun int num_pkts_from(IP x) =
  filter(srcip=x) >> count;
sfun int freq(IP x) =
  num_pkts_from(x) / count;
```

Note that the function `count` in `num_pkts_from` only counts the number of packets from source `x`, while it counts the number of packets in the entire stream in the function `freq`.

Now the entropy is computed following its definition, as in the following program.

```
sfun int entropy =
  -sum{freq(x)*log(freq(x)) | IP x};
```

### 3.3.2 TCP State Monitoring

We next showcase applications that rely on monitoring the states within a TCP flow.

**SYN flood attacks.** In this example, NetQRE is used to detect SYN flood attacks by counting the number of incomplete TCP handshakes in a time interval. We consider an incomplete TCP handshake as a packet trace consisting of a SYN packet and a SYNACK packet, with corresponding sequence number and acknowledge number, but does not have a further ACK packet to complete the handshake. As the key step, the following program counts the number of incomplete handshakes, assuming that the input stream consists of TCP packets between the same source and destination.

```
sfun int bad_tcp_pat(int x, int y) =
    concat(syn(x), synack(y,x+1), no_ack(y+1));
sfun int incomplete_handshake_num =
    sum{bad_tcp_pat(x,y)?1 | int x, int y};
```

The function `bad_tcp_pat` specifies the pattern of an incomplete handshake: there is a SYN packet with a sequence number `x` in the stream, followed by a SYNACK packet with acknowledge number `x+1` and sequence number `y`, and then followed by packets that do not include an ACK with acknowledge number `y+1` to complete the handshake. The `sum` aggregation in `incomplete_handshake` sums up the number of appearances of such patterns for all `x` and `y`. Using the stream composition of filtering functions based on time and packet type, the complete function can be specified as follows:

```
sfun int syn_flood(Conn c) =
  recent(5) >>
  filter_tcp(c) >>
  incomplete_handshake_num >T?block(c.srcip);
```

**Completed flows.** Our next example counts the number of legitimate flows that are completed, delineated by a SYN at the beginning, and ending with a FIN. Note that the last `iter` uses the regular expression which matches a stream ending with a SYN and a FIN packets, and no SYN-FIN pairs appear before. Therefore, the `iter` expression splits the entire (filtered) stream into sessions where each session contains exactly one complete

flow.

```
sfun int count_flow(Conn c) =
    filter_tcp(c) >>
    filter(flag=SYN || flag=FIN) >>
    iter(/[fin=1]*[syn=1]*[syn=1][fin=1]/?1, sum);
```

### 3.3.3 Application-level Monitoring

Our final example revisits the Voice-over-IP (VoIP) usage use case.

Let us first focus on the function to monitor the usage of a VoIP call based on a SIP connection `sip_conn` and media connection `m_conn`. Recall that a VoIP call consists of three phases; and a modular programming way is to handle the three phases using three stream functions, and then compose them using the `split` operator. The function is shown below.

```
sfun int usage_per_call(Conn sip_conn, Conn m_conn, string
    user, string id) =
  split(init(id,user,m_conn)?0,
        call(m_conn)?count_size,
        end(id)?0, sum);
```

Here, `init`, `call` and `end` are simply the PSREs that capture the patterns of each phase. Note that the `init` and the `end` phase return a value 0, and only the usage of the call phase is counted, as required.

Next, a programmer can view the traffic using the SIP connection `sip_conn` and the media connection `m_conn` as a sequence of calls. Using the `iter` operator, the programmer can easily aggregate the usage of all calls in the traffic, as shown below.

```
sfun int usage_per_conn(Conn sip_conn, Conn m_conn, string
    user, string id) =
  filter_sip(sip_conn, m_conn, user, id) >>
  iter(split(any?0, usage_per_call(sip_conn, m_conn, user,
      id), sum),sum);
```

Note that we first filter all traffic that belongs to `sip_conn` or `m_conn`, in order to get the stream of interest. This filtering may result in non-VoIP traffic between two VoIP calls, and thus we simply compose the PSRE `any` with `usage_per_call` inside the `iter` expression, to handle any traffic between two calls.

Now we can aggregate the usage for each user across multiple connections and calls using the aggregation operation, and further compute the average usage over all users, as shown below.

```
sfun int usage_per_user(string user) =
  sum{usage_per_conn(sip_conn, m_conn, user, id) | Conn
      sip_conn, Conn m_conn, string id};


sfun int average_usage =
  avg{usage_per_user(u) | string u};
```

Suppose we need to send an alert to the controller if a user's usage is larger than five times of average usage, we can simply specify the policy as below.

```
sfun action alert_high_usage(string user) =
  (usage_per_user(user) > 5*average_usage) ? alert(user);
```

## 3.4   Compilation Algorithm

We next describe how to compile a NetQRE program into an efficient executable that can run with low memory footprint. Typically the stream of packets is very large, thus it is not feasible to store the entire stream of packets. Therefore, the compiled program should evaluate *incrementally* on each single incoming packet without storing the history of the stream. To achieve this goal, we have to address the following challenges. First, the compiled program needs to maintain parameters in NetQRE in a succinct way. Second, in order to implement the semantics of `split` and `iter`, these operations have to be performed in a single online pass over streaming packets. Lastly, the compiled program needs to handle aggregation expressions over parameters.

### 3.4.1 Compilation of PSRE

First, we describe the algorithm for compiling a PSRE, as the basic building block for stream functions. Similar to traditional RE, a PSRE can be translated to an equivalent finite state machine, which we refer to as *parameterized symbolic automaton* (PSA). For example, consider the following PSRE `/.*[srcip=x].*/`. Intuitively this PSRE checks whether there exists a packet with source IP `x` in the stream. Its corresponding PSA is shown in Figure 3.5.
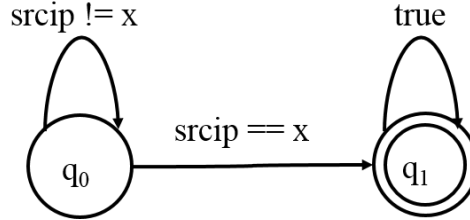


Figure 3.5: An example PSA.

However, PSA cannot be directly updated due to its uninstantiated parameters. To illustrate this challenge, consider a naive solution which is to instantiate the parameters using all possible values, and maintain all the instantiated state machines, namely symbolic automata (SA), for each instantiation of the parameters. When reading an input packet, we update all the instantiated SA in the standard way. However, it is not hard to note that this approach is not feasible due to the large space of parameters. As an example, the PSA in Figure 3.5 with a single IP parameter has up to $2^{32}$ instantiated symbolic automata. Whereas at runtime, the function only needs to store the source addresses that appeared in the stream, which can be significantly less than $2^{32}$.

To address this challenge, we propose a new algorithm that updates PSA on-demand. As the high-level idea, suppose we instantiated a PSA to a set of SA using all possible instantiations. Notice that even though there is a large space of instantiated SA, however, many of them will keep the same states at runtime given a stream of packets. As an example, consider feeding a first packet with the source IP `1.0.0.1` to all SA instantiated from the PSA in Figure 3.5. It is easy to see that the only SA that will transition to state $q_1$ is the one where `x` is instantiated by `10.0.0.1`; and all other SA will stay at $q_0$.

Therefore, the compiled program at runtime maintains *guarded states* for the PSA, where a guarded state is pair $(s, \phi)$. Here, $\phi$ is a predicate of the parameters in the PSA (which we will refer as a guard), and $s$ is a state in PSA. The pair $(s, \phi)$ means that when reading the input stream, all instantiated SA will be at state $s$, if they are instantiated by the values satisfying $\phi$. For example, initially there is only one guarded state maintained for the PSA in Figure 3.5, which is $(q_0, \texttt{true})$, meaning that all instantiated SA is at the initial state $q_0$ of the PSA.

To update the PSA, the key step is to update each guarded state dynamically when reading packets from the stream. The updating algorithm for a guarded state is shown in Algorithm 8. This algorithm iterates through all transitions from the state $s$, and for the

---

**Algorithm 8** update_psa$((s, \phi),$ packet)

1: **for all** transitions originated from $s$ **do**
2:   let $t$ be the destinate state, and $P$ be the parameterized predicate
3:   let $T$ be the guard instantiated from $P$ using the packet
4:   let $\phi' = \phi \wedge T$
5:   emit $(t, \phi')$ if $\phi \neq \texttt{false}$
6: **end for**

---

predicate (with parameters) $P$ on the transition, it instantiates $P$ using the current packet it receives in the stream. This step simply replaces the function name in $P$ by the return value of it on the packet. The obtained predicate $T$ is a predicate over the parameters in the PSA. Finally, the algorithm emits a new pair $(t, \phi')$ if $\phi'$ is not $\texttt{false}$. Intuitively, this step accounts the fact that the instantiated SA under $\phi'$ will transition to the next state $t$.

Using Algorithm 8, the overall updating algorithm is straightforward. Initially, the compiled program only contains a guarded state $(q_0, \texttt{true})$, where $q_0$ is the initial state of the PSA. Every time reading a new packet from the stream, we call Algorithm 8 on every guarded state, and the new guarded states consists of all the ones emitted from Algorithm 8. To check the state given concrete values for parameters, we simply go through all maintained guarded states, and return the state if the guard is satisfied.

**Example.** Using the example in Figure 3.5, we illustrate the execution of Algorithm 8 given the pair $(q_0, \texttt{true})$ and the packet with source IP $\texttt{10.0.0.1}$. Consider the transition from $q_0$ to $q_1$. By substituting $\texttt{srcip}$ in the predicate with the source IP of the packet, we get the guard $T$ which is $\texttt{x=10.0.0.1}$. Since $\phi$ is true now, $\phi'$ is simply $\texttt{x=10.0.0.1}$. At the

end, the algorithm will emit the pair $(q_1,$ `x=10.0.0.1`$)$. Similarly, for the other transition the algorithm emit the pair $(q_0,$ `x!=10.0.0.1`$)$. Therefore, there are two new guarded states, namely, $(q_1,$ `x=10.0.0.1`$)$ and $(q_0,$ `x!=10.0.0.1`$)$. The updated guarded states account for the fact that `10.0.0.1` has appeared and thus the corresponding instantiated SA transitioned to state $q_1$.

### 3.4.2 Compilation of `split`

We next discuss how to compile a `split` expression which is of the form `split(f, g, aggop)`. First, we highlight the key insight from [18] to compile `split` without parameters, and then describe how to generalize this idea to compile `split` with parameters.

**Without parameters.** The challenge of compiling `split` is to split the stream dynamically at runtime without revisiting the entire history of the stream. For example, in the `split` example in Section 3.2.3, we need to split the stream at the last appearance of the SYN packet. A natural way to implement the semantics of `split` is to maintain all cases to split the stream. For example, Figure 3.6 shows two cases to split the stream consisting of a non-SYN and a SYN packet at runtime for the example in Section 3.2.3. Moreover, the following two observations ensure that the compiled code only uses a constant space. First, each case can be represented using a triple $(s_f, s_g, F)$, where $F$ is a flag indicating whether the stream is split, and $s_f(s_g,$ resp.) is the state of $f(g,$ resp.) on evaluating the prefix(suffix, resp.) of the stream. Second, the number of maintained cases is bounded by a constant only related to $g$ at any time point.



Figure 3.6: Example run of `split`.

**With parameters.** Similar to PSRE compilation, in the general case, we need to maintain guarded cases. That is, we maintain a set of $(T, \phi)$ pairs, where $T$ is a triple $(s_f, s_g, F)$ corresponding to a case as defined above, and $\phi$ is a guard. It means that if the parameters are instantiated by values satisfying $\phi$, there is a case of splitting the stream, which can be

represented as $T$.

Algorithm 9 shows the algorithm to update a pair $(T, \phi)$. The algorithm feeds the input

---
**Algorithm 9** update_split$((T, \phi)$, packet)

---
1: suppose $T = (s_f, s_g, F)$
2: **if** $F$ is `false` **then**
3:    **for all** emitted $(s'_f, \phi')$ from update_f$((s_f, \phi)$, packet) **do**
4:      **if** `f` is defined on $s'_f$ **then**
5:         emit $((s'_f, s^0_g, \text{true}), \phi')$
6:      **end if**
7:      emit $((s'_f, s_g, \text{false}), \phi')$
8:    **end for**
9: **else**
10:    **for all** emitted $(s'_g, \phi')$ from update_g$((s_g, \phi)$, packet) **do**
11:      emit $((s_f, s'_g, \text{true}), \phi')$
12:    **end for**
13: **end if**

---

packet to `f` or `g` corresponding to whether the stream has been split as indicated by $F$. For example, consider the else branch from line 10 to 12 in the algorithm. In the branch, we need to update $(s_g, \phi)$ using `g`'s updating procedure, which may emit a set of guarded states of `g`. Correspondingly, the guard needs to be refined.

Given concrete values of parameters, evaluating a `split` expression is as follows. First, we check if there exist a guarded case $((s_f, s_g, F), \phi)$ such that $\phi$ is satisfied and both `f` and `g` are defined on the states in the case, then we evaluate the expression on the case. Otherwise, the expression is not defined.

### 3.4.3 Compilation of `iter`

Recall that an `iter(f, aggop)` expression needs to split the stream into multiple substreams, such that on each substream, `f` is defined. We first discuss the compilation in the case without parameters, and then discuss the compilation algorithm for the case with parameter.

**Without parameters.** Similar to the compilation of `split`, the state of the expression to maintain is a set of pairs corresponding to all cases to split the stream, each of which consists of $(val, s_f)$, where $val$ is the running aggregated value, and $s_f$ is a state of `f`. Initially, the state is $(v_0, s^0_f)$, where $v_0$ is an initial value based on the aggregation operator

aggop. For example, $v_0$ is 0 when the aggregation operator is sum; and $v_0$ is a pair (0,0) if aggop is avg, where the first 0 indicates the running sum, and the second 0 indicates the running count of the number of elements. At a high level, to update a pair $(val, s_f)$, every time receiving a packet, we need to update $s_f$ by f's updating function. When $s_f$ reaches a defined state (i.e. f evaluates to a defined value on $s_f$), f is evaluated and we add a new pair $(val', s_f^0)$, where $val'$ is new aggregated value using $val$ and the evaluated value from f using aggop, and $s_f^0$ is the initial state of f. To evaluate the iter function, we go through all pairs and pick the one $(val, s_f)$ where $s_f$ is defined, and output the aggregated value.

**With parameters.** Similarly, for an iter expression with parameters, the guarded case is a pair $(T, \phi)$, where $T$ is a pair $(v, s_f)$, and $\phi$ is a predicate of its parameters. Initially the guarded case is $((v_0, s_f^0), \text{true})$. Algorithm 10 shows the algorithm for updating a guarded case.

---

**Algorithm 10** update_iter$((T, \phi)$, packet)

---

1: suppose $T = (v, s_f)$
2: **for all** emitted $(s_f', \phi')$ from update_f$((s_f, \phi)$,packet) **do**
3:     **if** $s_f'$ is defined **then**
4:         let $v'$ be the evaluated value of f on $s_f'$
5:         emit $((v'', s_f^0), \phi')$, where $v''$ is the aggregated value of $v$ and $v'$ using aggop
6:     **end if**
7:     emit $((v, s_f'), \phi')$
8: **end for**

---

**Example.** We illustrate how to update the following expression iter(/[srcip!=x]*[srcip=x]/?1,sum) on a 3-packet stream, where each packet has source IP address A, B, A, respectively. The state machine for the inner PSRE is shown in Figure 3.7(a), Initially, there is one guarded case $((0,q_0),\text{true})$, standing for the fact that for all values of x, the aggregated sum is 0, and the state for the inner expression is $q_0$. Reading the first packet, we need to update $(q_0,\text{true})$ using Algorithm 8, which emits $(q_0,\text{x!=A})$ and $(q_1,\text{x==A})$. Furthermore, since $q_1$ is final state, we can split the stream, and aggregate the sum. Thus, we add a new guarded case $((1,q_0),\text{x==A})$. Figure 3.7(b) shows the maintained guarded cases after reading each packet. The arrows between two guarded cases stand for the fact that the latter guarded case is emitted from the former one.
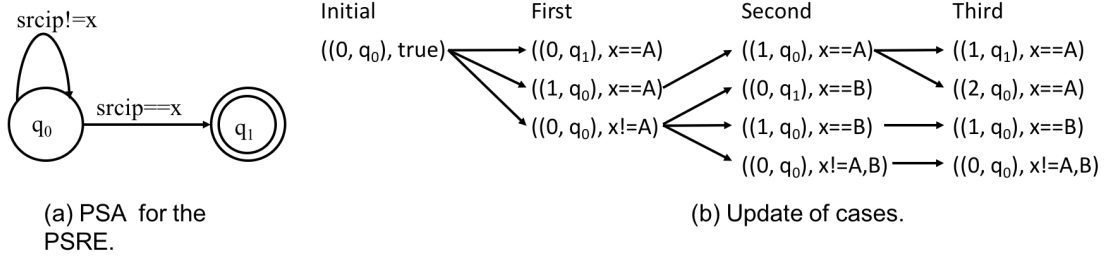
Figure 3.7: Example run of `iter`.

### 3.4.4 Compilation of Aggregation

Now we describe the aggregation function `aggop{f | T x}`. Followed by the definition of the aggregation function, an aggregation function maintains guarded states $(s_f,\phi)$. To update the aggregation expression, we just need to update each guarded state using `f`'s updating procedure. To illustrate the evaluation procedure, suppose the aggregation operator is `sum`. Given values for the parameters, we need to iterate through all guarded states $(s_f,\phi)$. If $\phi$ is satisfied, we evaluate `f` on $s_f$, say the return value is $v$; and count the number of values of `x` which satisfies $\phi$, say it is $k$; we sum up $v * k$ into the aggregated sum. For other aggregation operators, the evaluation process works similarly.

### 3.4.5 Compilation of Stream Composition

Lastly, we discuss stream composition. Recall that stream composition allows to process the input stream using a function `f`, and then apply the second function `g` on the processed stream. Following its definition, each time a new packet arrives, we need to first process it with `f`, and the returned value (e.g. a filtered packet) is then fed into `g`. A guarded state it maintains is $((s_f, s_g), \phi)$, where $s_f$ and $s_g$ is a state of `f` and `g` respectively. The initial guarded state is $((s_f^0, s_g^0), \texttt{true})$, which stands for the fact that for all instantiations, `f` and `g` are in their initial state. Algorithm 11 shows the algorithm to update a guarded state. To evaluate the expression, we return the value of evaluating `g` on $s_g$, where $s_g$ is from the guarded state $((s_f,s_g),\phi)$, and the guard $\phi$ is satisfied.

**Algorithm 11** update_comp$((S, \phi)$, packet)

---

1: suppose $S = (s_f, s_g)$
2: **for all** emitted $(s'_f, \phi')$ from update_f$((s_f, \phi)$,packet) **do**
3:    **if** f is defined on $s'_f$ **then**
4:       let ret be the returned value of f on $s'_f$
5:       emit $((s'_f, s'_g), \phi'')$, for all emitted $(s'_g, \phi'')$ from update_g$((s_g, \phi')$, ret)
6:    **end if**
7: **end for**

---

## 3.5   Implementation

We have implemented a prototype of the NetQRE system in C++, which consists of two main components, namely the compiler for NetQRE, and the NetQRE runtime.

**NetQRE Compiler.**     The NetQRE compiler implements the compilation algorithm described in Section 3.4. The compiler first generates a C++ program from an input NetQRE program, which is then compiled by the gcc compiler into executable. Our compiled program uses a tree structure to represent predicates over parameters. The choice of trees is driven by its simplicity, ease of encoding, and lookup performance.

In addition to the basic compilation algorithm, we include additional optimizations to the compiler. First, we use the standard minimization algorithm to minimize the state machine for a regular expression. Second, for aggregation expressions with `sum` and `avg`, we use an incremental updating algorithm to update the expression: we maintain the running sum as the state of the aggregation expression, and we update the sum incrementally when the state of inner expression is updated.

**NetQRE Runtime.** The NetQRE runtime includes a packet capture agent implemented using the pcap library [63]. Each packet that arrives at the runtime is parsed, and then processed by the compiled NetQRE program as shown in Figure 3.1. Currently, our runtime supports actions that include sending alert events to a controller, and directly installing a rule on a switch. The NetQRE runtime is not specifically optimized for fast packet capture and processing, and as future work, we plan to explore the use of DPDK [50].

## 3.6 Evaluation

We evaluate our NetQRE prototype centered around answering three key questions. First, can the NetQRE language express a wide range of quantitative network policies in a concise and intuitive manner? Second, is the generated code efficient in terms of throughput? Third, can NetQRE be used in a real-time monitoring setting that provides rapid mitigation based on quantitative network policies? Unless otherwise specified, all our experiments are carried out on a cluster of commodity servers, each of which has 16-core 2.1GHz Xeon E5-2450. Each core has a 2MB L2 cache, and each processor includes a 20 MB L3 cache. The total memory size is 24 GB. The OS is Ubuntu 12.04, and the kernel version is 3.13.0.

### 3.6.1 Expressiveness

We have implemented a set of quantitative network policies using NetQRE. The examples are drawn from a literature survey on quantitative network policies [28, 36, 77, 91, 92]. Table 3.1 summarizes the example policies and the lines of code of each NetQRE program.

|  | LoC |
|---|---|
| Heavy Hitter (Section 4.3.1) | 6 |
| Super Spreader (Section 4.3.1) | 2 |
| Entropy Estimation (Section 4.3.1) | 6 |
| Flow size dist. [36] | 8 |
| Traffic change detection [77] | 10 |
| Count traffic [91] | 2 |
| Completed flows (Section 4.3.2) | 6 |
| SYN flood detection (Section 4.3.2) | 9 |
| Lifetime of connection | 8 |
| Newly opened connection recently | 11 |
| # duplicated ACKs | 5 |
| # VoIP call | 7 |
| VoIP usage (Section 4.3.3) | 18 |
| Key word counting per email | 23 |

Table 3.1: Example policies NetQRE supports.

We make the following observations. First, NetQRE is able to express a large variety of policies, ranging from flow-level traffic measurement to application-level quantitative policies. Second, programming in NetQRE is concise. All example policies can be specified within 23 lines of code in NetQRE. This count includes commonly used filter functions as

well as regular expressions, which can be built into a library for reference. As an interesting comparison, we encoded the VoIP call use case in Bro [73], a well-known intrusion detection system. Bro required 51 lines of code, as compared to only 7 in NetQRE. However, Bro is unable to easily support the VoIP usage case written in NetQRE. In particular, it seems hard for Bro to count bandwidth consumption per VoIP session. This is not surprising as Bro's primary use is that of an intrusion detection system. We validated the correctness of our implementation on actual SIP traffic by comparing Bro's output against NetQRE's.

### 3.6.2 Throughput Performance

Our next set of experiments evaluate the performance of NetQRE's compiled implementations. NetQRE runs on a single machine based on the setup described earlier. We use a single core to measure NetQRE's throughput. Throughput can be increased via orthogonal parallelization techniques (e.g. [43]), which is not the main focus of the paper. Hence, all reported numbers are based on single-threaded execution. As a point of comparison, we compare each NetQRE program against an equivalent carefully hand-crafted C++ implementation. We note that all our hand-crafted implementations require at least 100 lines of code, and often times, require users to explicit manage internal state across packets – a programming task that NetQRE abstracts away.
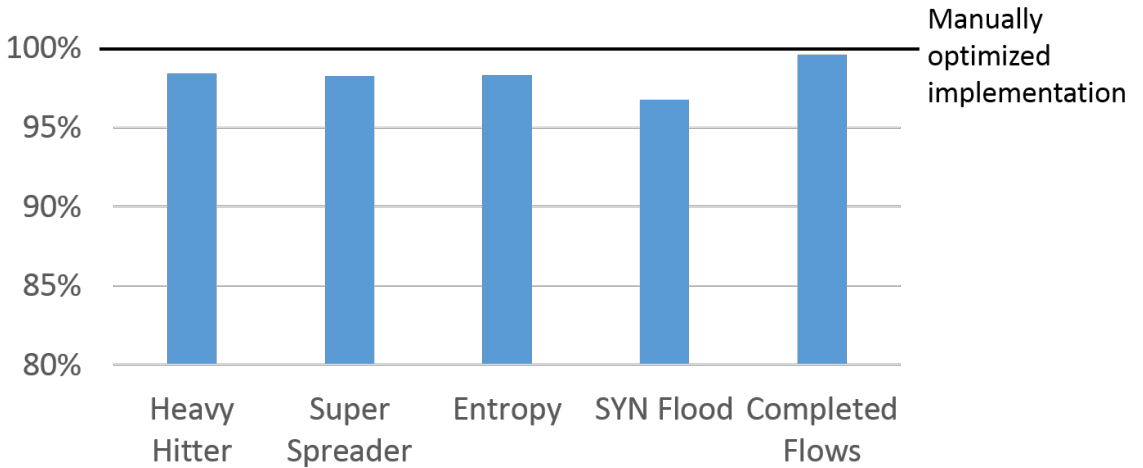
Figure 3.8: Normalized throughput comparison.

Figure 3.8 shows the performance of NetQRE implementations compared with manu-

ally optimized implementations, where NetQRE throughput performance is presented as a percentage of manually optimized implementations. We use as example policies, the heavy hitter, super spreader, entropy, SYN flood detection, and completed flows count use cases presented in Section 3.3. As workload, we use a CAIDA traffic trace [1] which contains 37 million packets, and replay the traffic as input to our implementations. The compiled NetQRE implementation incurs negligible overhead compared with the manually optimized low-level implementation. The difference between the throughput of the compiled NetQRE implementation and that of the manual implementation is within 4% across all use cases we studied.

To understand the performance compared with alternative tools, we further compare with OpenSketch and Bro.

**Comparison with OpenSketch.** We use the heavy hitter example (with a single field, as in the OpenSketch's reference code) to compare the performance of NetQRE compiled code with that of OpenSketch (in software). We follow the default setting of the reference code of OpenSketch [9]. The throughput of NetQRE compiled code is 72% higher than that of OpenSketch.

**Comparison with Bro.** We further compare with Bro. Given the limitations of Bro in handling the complete VoIP use cases, we simplify the use case to simply counting the number of VoIP calls (as opposed to bandwidth consumption) per user. NetQRE compiled implementation can finish counting within 1 second, while Bro takes about 23 seconds. Both programs output the same results for this use case. We believe there are at least two reasons for Bro's slower performance. First, Bro is designed for intrusion detection, and not aimed at and optimized for monitoring tasks. Second, unlike NetQRE which compiles high-level code to efficient low-level code, Bro uses an interpreter to execute the script written in Bro's language, which could result in considerable overhead.

### 3.6.3 Real-time Response

In the final set of experiments, we validate the use of NetQRE for enforcing quantitative network policies, through a combination of switch-level monitoring, followed by updates from the controller. We set up a network of two clients $C_1$ and $C_2$, one server, and one SDN

switch that mirrors traffic to a NetQRE runtime running the SYN flood detector presented in Section 3.3.2. In addition, a NetQRE controller based on POX [64] controls the switch. The network is emulated using Mininet [59] with link bandwidth set to 100Mbps.

In the experiment, $C_1$ sends normal traffic to $S$ using iperf at rate 1Mbps, and at the 7th second, $C_2$ starts SYN flood attack generated using our generator to the same server. The attack is detected by NetQRE, which generates an alert event to the controller, which subsequently installs a forwarding rule on the switch to block traffic from $C_2$. To illustrate the attack detection and blocking, Figure 3.9 shows the bandwidth utilization (Mbps) on the server. We observe that NetQRE successfully blocks $C_2$'s attacking traffic in real-time.
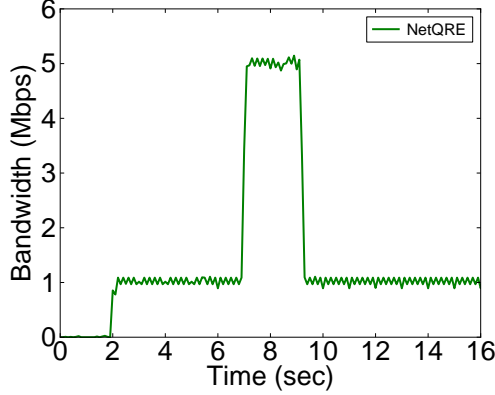


Figure 3.9: SYN flood: Bandwidth utilization (Mbps) at the server.

As a second experiment, using a similar setup, our NetQRE heavy hitter program (Section 3.3.1) running at a switch-side NetQRE runtime monitors the traffic, and issues an alert to the controller when detecting a heavy hitter, which further blocks the traffic. As a point of comparison, we compare our approach against two alternatives: (1) sending all packets to the controller which runs an equivalent heavy hitter detection program, (2) monitoring the flow counters on the switch to detect heavy hitters, as considered in other languages [67] and systems [69]. In the second alternative, the controller reads the counter every 1 second, and compute the bandwidth usage for each flow over a 5 seconds window.

Figure 3.10 plots the bandwidth utilization of the server. The above alternative solutions are labeled as *forward* and *stats* respectively. Compared with these two approaches, our proposed approach can detect the heavy hitter and further respond to it in a more timely

fashion. Moreover, compared with the *forward* approach, we send significantly fewer traffic to the controller, and is a more scalable solution.
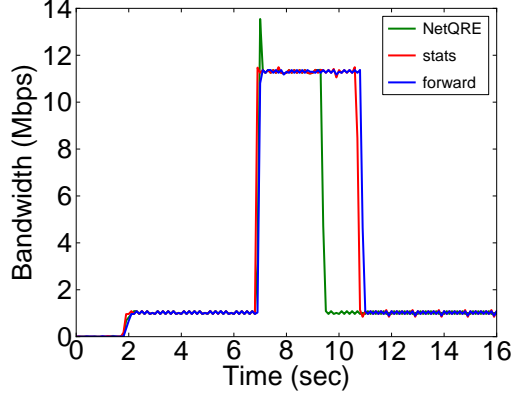


Figure 3.10: Heavy hitter detection: Bandwidth utilization (Mbps) at the server.
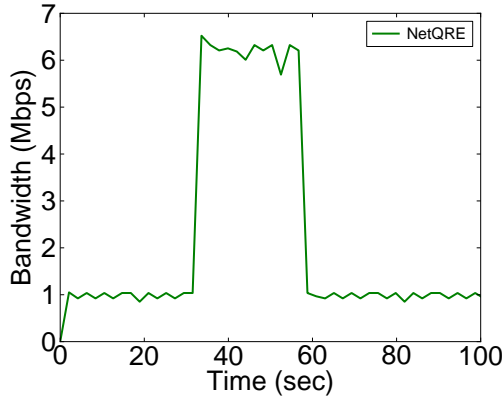


Figure 3.11: VoIP: Bandwidth utilization (Mbps) at the server.

In our final experiment, we validate the VoIP use case first presented in our introduction. We use a synthetic VoIP traffic trace generated using SIPp [12] replay a VoIP call (with accompanying H.264 MPEG video) made from a single user via client $C_2$. We replay the traffic at 5Mbps, and run iperf on $C_1$ as the background traffic. Our NetQRE program enforces a policy that blocks a call after the user making the call exceeds a SIP bandwidth usage of 18.75MB (around 30 seconds). We configure the NetQRE program to send an alert to the NetQRE controller which then blocks the traffic. Figure 3.11 plots the bandwidth utilization of the server. Again, this verifies that NetQRE can be used to intercept a SIP

session, monitor usage on a per-user basis, and react to high usage in a timely fashion.

### 3.6.4 Summary of Evaluation

Revisiting the three questions at the beginning of our evaluation, we observe that (1) NetQRE can express a wide range of quantitative network policies with a few lines of code, (2) achieves performance that is similar to that of carefully hand-crafted optimized low-level code while significantly outperforms other measurement and IDS tools, and (3) can be used in an SDN setting to monitor network traffic and update switches in real-time in response to specified NetQRE policies.

# Chapter 4

# Related Work

We highlight several work in a variety of topics that are related to the dissertation.

## 4.1   Domain Specific Languages in Networking

Network Datalog (NDlog) [61] is a declarative language to program network protocols, which is based on the well-known recursive query language Datalog. In NDlog, network state, such as network links and paths between a source and a destination, is modeled as relations. A user uses a set of rules to specify the derivation of tuples in the relations. NDlog extends Datalog by allowing a user to specify storage locations of tuples explicitly. By specifying the storage locations, a NDlog program models the distributed computation over network state without specifying imperative details such as sending and receiving a message.

With the emergence of SDN, there are several recent proposals on domain specific languages in SDN, which include Frenetic [40], Pyretic [67], NetCore [66] NetKAT [19], FlowLog [72], Merlin [84], FatTire [75], and Maple [88].

Frenetic [40], NetCore [66], and Pyretic [67] propose high-level abstractions to specify packet forwarding and network queries based on OpenFlow. These languages use high-level predicates to capture the semantics of flow-table rules in OpenFlow, and allow logical combinations such as conjunction, disjunction, and negation on the predicates to simplify the specification of flow tables. The semantics of these languages are compositional, which allows users to compose two modules in a parallel and sequential fashion. In addition to the high-level abstractions for static flow-table policies, these languages support dynamic

policies using general-purpose programs in order to respond to network events and change the flow tables.

NetKAT [19] is proposed with a solid theoretical foundation based on Kleene algebra with tests [57] to specify static packet forwarding policies. As a result, checking whether two forwarding policies specified in NetKAT are equivalent can be decided in PSPACE complexity, which allows network verification of properties such as reachability and traffic isolation.

The SDN languages described above focus on raising the level of abstractions for data plane policies. Below we highlight some work involving abstractions for control plane.

Flowlog [72] proposes a unified programming framework to capture data plane, control plane and controller state. Flowlog uses relations to store network state, and also offers a SQL-like language to specify the reaction to network events. Programs in Flowlog can be proactively compiled and installed onto flow tables. Furthermore, Flowlog offers sound and complete verification on the programs using the verifier Alloy [51].

Kinetic [55] proposes to use the state machine abstraction to program dynamic policies, where a state corresponds to a static policy which can be specified in Pyretic, and a transition corresponds to network events, user-defined events, or unexpected failures. Kinetic utilizes the NuSMV model checker [31] to automatically verify the correctness properties specified in computation tree logic (CTL) [33].

Similar state machine abstractions have also been studied in OpenState [22] and FAST [68]. These works aim at designing state machine models that can be embedded into switches, and thus executing stateful policies on the data plane efficiently.

There are also language proposals aimed at providing abstractions for packet-routing paths in the network [70, 75] using regular expressions.

FatTire [75] is a language for fault-tolerant network routing policies. FatTire uses regular expressions to specify the set of paths for packets in a specified flow space, and allows a user to specify the degree of faults to tolerate. A program in FatTire can be compiled into OpenFlow-based flow tables using the group tables [13], which is able to tolerate the given degree of faults.

The path query language proposed in [70] is a declarative language aimed at traffic monitoring. A query in this language uses regular expressions over packet locations and

header values to specify the packets (together with its routing paths) to be monitored. The packets satisfying the regular expression are counted. In addition, the language further leverages SQL-like "groupby" operation for aggregating the monitoring results.

The work in this dissertation shares similar goals with the work described above. However, there are several technical differences between our work and the work described above. For example, while the abstraction of policy tables in NetEgg is similar in spirit to these state machine abstractions, but however, NetEgg focuses on providing an intuitive programming framework which can generate policies directly from examples. Although some languages above have features to support traffic monitoring that NetQRE supports, however, to the best of our knowledge, none of these systems support monitoring queries beyond basic flow-level counters provided by OpenFlow.

## 4.2 Network Verification and Testing

In recent years there have been a number of proposals in network verification and testing. We highlight several work in this section, including techniques for the data plane ([38, 52–54, 62, 95]) as well as the techniques for the control plane ([20, 27]).

First, we highlight some work aimed at stateless data plane verification.

Header Space Analysis (HSA) [53] models each device in the network as a network transfer function, which maps a located packet (i.e. a packet together with its location) to a set of located packets. Using a customized symbolic execution technique, HSA is able to verify properties such as reachability, forwarding loops and traffic isolation.

Veriflow [54] and NetPlumber [52] introduce real-time data plane verification. Unlike HSA which needs to re-check data plane properties every time a rule is changed, these techniques only consider the incremental change of the data plane introduced by each forwarding rule's update, and check properties only on the delta part.

On the other hand, Anteater [62] uses a different approach. Anteater encodes the data plane configuration and the property as a boolean satisfiability formula (SAT). Using SAT solvers for the SAT problem, Anteater is able to verify the properties or generate a counterexample for violations.

There are works aimed at runtime testing for data plane. ATPG [95] generates test

packets that are injected to the network to test functional properties as well as performance properties. More recently, Buzz [38] proposes a technique based on symbolic execution to generate test packets in order to test stateful data plane.

Second, we highlight some work for control plane verification and testing. Nice [27] employs both model checking and symbolic execution to test SDN controller programs. Vericon [20] proposes a verification techniques that can verify the correctness of a controller application for all admissible network topologies and all network events.

## 4.3 Program Synthesis

This dissertation work is also related to the topic on program synthesis.

The Sketch system [81–83] has been developed for synthesizing bit-stream programs [83], stencil computations [81] and concurrent data structures [82]. The Sketch system allows users to specify a partial implementation (which is called a sketch), and then synthesizes a complete implementation that is equivalent to a given program from the sketch.

The NetEgg work is motivated by related work on programming by examples. In particular, the proposals in [47–49] implement reactive controllers from specification of behaviors in live sequence charts. Recently, programming by examples has been applied for a variety of domains [17, 21, 44, 45, 56, 60, 79]. FlashFill [44] generates string transformation macros in Excel from input/output string examples. Transit [87] uses both symbolic and concrete example to synthesize distributed protocols. Examples are also used for synthesizing recursive programs [17]. The NetEgg work is similar in spirit to above works, but technically different. Our input examples and target program are designed specific to the SDN domain, and have different characteristics, which require different synthesis algorithms.

## 4.4 Other Topics

In addition to the topics addressed above, there are several other topics that are related to the work presented in this dissertation. In this section, we describe the related work in the following topics.

### 4.4.1 Streaming Database Languages

Database-style query languages [29, 34] provide SQL-like language support for running continuous queries over data streams. While there are constructs to do aggregation over sliding windows, they are designed with simple relational queries over packet headers or packet counts, and cannot handle the complex queries based on traffic patterns that NetQRE supports.

### 4.4.2 Intrusion Detection Systems and Protocol Analyzers

There are a number of intrusion detection systems (IDS) and protocol analyzers, which can support some part of policies that NetQRE supports. However, there are several key differences between NetQRE and these systems. First of all, systems such as Snort [76] allow regular expression matches over packet payloads, but is not perform regular expression matches that span multiple packets, let alone track sessions across packets. While Bro [73] supports aspects of NetQRE, the scripting language requires operators to think imperatively in terms of states and events, as opposed to NetQRE's approach of using high-level declarative constructs. Moreover, IDS such as Bro is ill-suited to handle some of the use cases involving quantitative monitoring, e.g. the VoIP usage example.

### 4.4.3 Deep Packet Inspection

In NetQRE, we studied its support for deep packet inspection. There are a variety of commercial products that perform Deep Packet Inspection (DPI). In recent literature, Sc-analytics [43] provides a parallel Datalog-based DPI engine, but the focus is on parallelism and high performance, and hence the language does not lend itself naturally to customization of regular expressions and aggregation. DPI poses a challenge on encrypted data, an orthogonal issue that is addressed by other work (e.g. BlindBox [78]).

# Chapter 5

# Conclusion

In this chapter, we conclude our contributions in this dissertation and propose future work related to this topic.

## 5.1    Contributions

This dissertation studied the feasibility of using high-level abstractions for programming network policies. In particular, we studied two approaches aimed at stateful SDN policies and quantitative network policies, respectively.

First, we proposed NetEgg, a scenario-based programming framework for SDN policies. To program a network policy using NetEgg, network operators simply describe example behaviors of the desire network policy in representative scenarios. NetEgg then automatically synthesizes a policy implementation that is consistent with all given scenarios, including inferring the state that needs to be maintained and also the rules to update the state. NetEgg can run the policy on top of a centralized controller as well as automatically infer the flow-table rules to install onto switches. Our experiments showed that NetEgg was able to synthesize a wide range of network policies from scenarios in seconds. Moreover, the synthesized policy implementations achieved close performance with manually crafted implementations. We conducted a user study including 12 students. The results showed that NetEgg was reasonably intuitive to learn and use, and was able to reduce the programming time for 2 out of 3 network policies we studied.

Second, we proposed NetQRE, a specification language and toolkit for quantitative net-

work policies. NetQRE offers a declarative specification language that integrates regular-expression-like patter matching and aggregation operations. Given a NetQRE specification program, the NetQRE compiler automatically infers the state that needs to be maintained, and generates an efficient policy implementation. We showed that NetQRE is expressive to specify a wide range of quantitative network policies using a small number of lines of code. Our experiments demonstrated that the performance of the policy implementations generated by NetQRE was comparable to the implementations manually crafted and optimized in low-level code, and was significantly more efficient than alternative solutions.

## 5.2  Future Work

We discuss some potential research directions to extend the work described in the dissertation.

### 5.2.1  Interaction between Network Programmer and Programming Framework

The emergence of programmable networks triggers a new wave of innovations in network programming frameworks. Oftentimes, these programming frameworks are driven by specific types of applications, and seldom address their interaction with network programmers. Therefore, it remains a key challenge to understand how network programmers interact with these programming frameworks. In particular, it is interesting to address the following problems: how to make a programming framework accessible to network programmers; how to improve the productivity of network programmers; and how to enhance the reliability of networks with the aid of network programming frameworks.

### 5.2.2  Network Verification and Synthesis

The programmability of networks stimulates increasing interest in formal verification of network programs and configurations, in order to ensure the correctness of network services. However, given the fact that verification techniques are often expensive, network verification faces the following challenges: 1) how to enable verification for interesting properties such as those involving quantitative objectives; 2) how to scale the verification techniques up

to large-size networks. Additionally, it is appealing to automatically synthesize network topologies, configurations, and control programs directly from network operators' high-level requirements, with the correctness guaranteed by the process of synthesis. Recent progress on data plane programing frameworks such as P4 [23] also offers new opportunities in verifying and synthesizing data plane implementations.

In response to these challenges and opportunities, a potential extend to this dissertation work is to develop verification technique for the proposed domain-specific languages. Moreover, it is appealing to develop domain-specific specification languages that are expressive to specify a variety of interesting properties, and their corresponding verification and synthesis techniques that are scalable up to real-world networks.

# Bibliography

[1] Anonymized 2015 internet traces. `https://data.caida.org/datasets/passive-2015/`.

[2] Application layer packet classifier for linux. `http://www.mcafee.com/us/products/network-security-platform.aspx`.

[3] Kinetic source code. `https://github.com/frenetic-lang/pyretic/tree/kinetic/pyretic/kinetic`.

[4] Mcafee network security platform. `http://l7-filter.sourceforge.net/`.

[5] Mininet sample workflow. `http://mininet.org/sample-workflow/`.

[6] Mininet walkthrough. `http://mininet.org/walkthrough/`.

[7] Nsf i-corps program. `http://www.nsf.gov/awardsearch/showAward?AWD_ID=1564730`.

[8] Openflow tutorial. `http://archive.openflow.org/wk/index.php/OpenFlow_Tutorial`.

[9] Opensketch reference code. `https://github.com/USC-NSL/opensketch`.

[10] Philadelphia network programmability user group meetup. `http://www.meetup.com/Philadelphia-Network-Programmability-User-Group-Meetup/`.

[11] Pox wiki. `https://openflow.stanford.edu/display/ONL/POX+Wiki`.

[12] Sipp. `http://sipp.sourceforge.net/`.

[13] Openflow switch specification 1.3.1. `http://bit.ly/of-131`, March 2013.

[14] Sugam Agarwal, Murali Kodialam, and TV Lakshman. Traffic engineering in software defined networks. In *INFOCOM, 2013 Proceedings IEEE*, pages 2211–2219. IEEE, 2013.

[15] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM, 2008.

[16] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, volume 10, pages 19–19, 2010.

[17] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *Proceedings of the 25th International Conference on Computer Aided Verification*, CAV'13, pages 934–950, Berlin, Heidelberg, 2013. Springer-Verlag.

[18] Rajeev Alur, Dana Fisman, and Mukund Raghothaman. Regular programming for quantitative properties of data streams. In *25th European Symposium on Programming*. ESOP, 2016.

[19] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: Semantic foundations for networks. In *Proceedings of the 41st annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 113–126. ACM, 2014.

[20] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. Vericon: towards verifying controller programs in software-defined networks. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 31. ACM, 2014.

[21] Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. Flashrelate: Extracting relational data from semi-structured spreadsheets using examples. In *Proceed-*

*ings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 218–228, New York, NY, USA, 2015. ACM.

[22] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. Openstate: programming platform-independent stateful openflow applications inside the switch. *ACM SIGCOMM Computer Communication Review*, 44(2):44–51, 2014.

[23] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[24] Teren Bryson. `http://searchsdn.techtarget.com/opinion/Your-next-job-may-be-network-programmer-What-youll-need-to-know`.

[25] Kirk Byers. `http://www.networkcomputing.com/data-centers/programming-essential-skill-network-engineers/1722440870`.

[26] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[27] Marco Canini, Daniele Venzano, Peter Peresini, Dejan Kostic, Jennifer Rexford, et al. A nice way to test openflow applications. In *NSDI*, pages 127–140, 2012.

[28] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009.

[29] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 668–668, New York, NY, USA, 2003. ACM.

[30] Margaret Chiosi, Don Clarke, Peter Willis, Andy Reid, James Feger, Michael Bugenhagen, Waqar Khan, Michael Fargano, Chunfeng Cui, Hui Deng, Javier Benitez,

Uwe Michel, Herbert Damker, Kenichi Ogaki, Tetsuro Matsuzaki, Masaki Fukui, Katsuhiro Shimano, Dominique Delisle, Quentin Loudier, Christos Kolias, Ivano Guardini, Elena Demaria, Roberto Minerva, Antonio Manzalini, Diego Lpez, Francisco Javier Ramn Salguero, Frank Ruhl, and Prodip Sen. Network functions virtualisation: An introduction, benefits, enablers, challenges & call for action. `https://portal.etsi.org/nfv/nfv_white_paper.pdf`.

[31] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*, pages 359–364. Springer, 2002.

[32] Benoit Claise. Cisco systems netflow services export version 9. 2004.

[33] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[34] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: A stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 647–651, New York, NY, USA, 2003. ACM.

[35] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[36] Nick Duffield, Carsten Lund, and Mikkel Thorup. Estimating flow distributions from sampled flow statistics. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 325–336. ACM, 2003.

[37] Cristian Estan and George Varghese. *New directions in traffic measurement and accounting*, volume 32. ACM, 2002.

[38] Seyed K. Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. Buzz: Testing context-dependent policies in stateful networks. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 275–289, Santa Clara, CA, March 2016. USENIX Association.

[39] Seyed Kaveh Fayazbakhsh, Vyas Sekar, Minlan Yu, and Jeffrey C Mogul. Flowtags: enforcing network-wide policies in the presence of dynamic middlebox actions. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 19–24. ACM, 2013.

[40] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *ACM SIGPLAN Notices*, volume 46, pages 279–291. ACM, 2011.

[41] Pedro Garcia-Teodoro, J Diaz-Verdejo, Gabriel Maciá-Fernández, and Enrique Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *computers & security*, 28(1):18–28, 2009.

[42] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. Opennf: Enabling innovation in network function control. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 163–174. ACM, 2014.

[43] Harjot Gill, Dong Lin, Xianglong Han, Cam Nguyen, Tanveer Gill, and Boon Thau Loo. Scalanytics: a declarative multi-core platform for scalable composable traffic analytics. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 61–72. ACM, 2013.

[44] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46, pages 317–330. ACM, 2011.

[45] Sumit Gulwani, William R Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

[46] Arpit Gupta, Laurent Vanbever, Muhammad Shahbaz, Sean P Donovan, Brandon Schlinker, Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, and Ethan

Katz-Bassett. Sdx: A software defined internet exchange. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 551–562. ACM, 2014.

[47] David Harel. Can programming be liberated, period? *Computer*, 41(1):28–37, 2008.

[48] David Harel and Rami Marelly. *Come, lets play: Scenario-based programming using LSCs and the Play-Engine*, volume 1. Springer, 2003.

[49] David Harel, Assaf Marron, and Gera Weiss. Behavioral programming. *Communications of the ACM*, 55(7):90–100, 2012.

[50] DPDK Intel. Data plane development kit. `URLhttp://dpdk.org`.

[51] Daniel Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[52] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 99–111, Lombard, IL, 2013. USENIX.

[53] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[54] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 15–27, Lombard, IL, 2013. USENIX.

[55] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. Kinetic: Verifiable dynamic network control. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 59–72, Oakland, CA, May 2015. USENIX Association.

[56] Dileep Kini and Sumit Gulwani. Flashnormalize: Programming by examples for text normalization. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 776–783. AAAI Press, 2015.

[57] Dexter Kozen. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(3):427–443, 1997.

[58] Ashwin Lall, Vyas Sekar, Mitsunori Ogihara, Jun Xu, and Hui Zhang. Data streaming algorithms for estimating entropy of network traffic. In *ACM SIGMETRICS Performance Evaluation Review*, volume 34, pages 145–156. ACM, 2006.

[59] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *HotNets*. ACM, 2010.

[60] Vu Le and Sumit Gulwani. Flashextract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 55. ACM, 2014.

[61] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative Networking. *CACM*, 2009.

[62] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 290–301, New York, NY, USA, 2011. ACM.

[63] Steve McCanne, Craig Leres, and Van Jacobson. Libpcap. `http://www.tcpdump.org`, 1989.

[64] J Mccauley. Pox: A python-based openflow controller, 2014.

[65] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[66] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. *ACM SIGPLAN Notices*, 47(1):217–230, 2012.

[67] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, David Walker, et al. Composing software defined networks. In *NSDI*, pages 1–13, 2013.

[68] Masoud Moshref, Apoorv Bhargava, Adhip Gupta, Minlan Yu, and Ramesh Govindan. Flow-level state transition as a new switch primitive for sdn. In *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'14)*, August 2014.

[69] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Dream: dynamic resource allocation for software-defined measurement. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 419–430. ACM, 2014.

[70] Srinivas Narayana, Mina Tahmasbi, Jennifer Rexford, and David Walker. Compiling path queries. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 207–222, 2016.

[71] Ankur Kumar Nayak, Alex Reimers, Nick Feamster, and Russ Clark. Resonance: dynamic access control for enterprise networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 11–18. ACM, 2009.

[72] Tim Nelson, Andrew D Ferguson, Michael JG Scheer, and Shriram Krishnamurthi. Tierless programming and reasoning for software-defined networks. *NSDI, Apr*, 2014.

[73] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23):2435–2463, 1999.

[74] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. Simple-fying middlebox policy enforcement using sdn. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 27–38. ACM, 2013.

[75] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. Fattire: Declarative fault tolerance for software-defined networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 109–114. ACM, 2013.

[76] Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *LISA*, volume 99, pages 229–238, 1999.

[77] Vyas Sekar, Michael K Reiter, and Hui Zhang. Revisiting the case for a minimalist approach for network flow monitoring. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 328–341. ACM, 2010.

[78] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 213–226, New York, NY, USA, 2015. ACM.

[79] Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *Computer Aided Verification*, pages 634–651. Springer, 2012.

[80] Armando Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[81] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching stencils. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 167–178, New York, NY, USA, 2007. ACM.

[82] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching concurrent data structures. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 136–148, New York, NY, USA, 2008. ACM.

[83] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. Programming by sketching for bit-streaming programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 281–294, New York, NY, USA, 2005. ACM.

[84] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. Merlin: A language for provisioning network resources. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 213–226. ACM, 2014.

[85] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, Jan 1997.

[86] Amin Tootoonchian, Sergey Gorbunov, Yashar Ganjali, Martin Casado, and Rob Sherwood. On controller performance in software-defined networks. In *Presented as part of the 2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Berkeley, CA, 2012. USENIX.

[87] Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. Transit: specifying protocols with concolic snippets. In *ACM SIGPLAN Notices*, volume 48, pages 287–296. ACM, 2013.

[88] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying sdn programming using algorithmic policies. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 87–98. ACM, 2013.

[89] Mea Wang, Baochun Li, and Zongpeng Li. sflow: Towards resource-efficient and agile service federation in service overlay networks. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 628–635. IEEE, 2004.

[90] Richard Wang, Dana Butnariu, Jennifer Rexford, et al. Openflow-based server load balancing gone wild, 2011.

[91] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *NSDI*, volume 13, pages 29–42, 2013.

[92] Lihua Yuan, Chen-Nee Chuah, and Prasant Mohapatra. Progme: towards programmable network measurement. *IEEE/ACM Transactions on Networking (TON)*, 19(1):115–128, 2011.

[93] Yifei Yuan, Rajeev Alur, and Boon Thau Loo. Netegg: Programming network policies by examples. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, page 20. ACM, 2014.

[94] Yifei Yuan, Dong Lin, Rajeev Alur, and Boon Thau Loo. Scenario-based programming for sdn policies. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15. ACM, 2015.

[95] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 241–252. ACM, 2012.