NetEgg: A Scenario-Based Programming Toolkit for SDN Policies

Yifei Yuan, Dong Lin, Siri Anil, Harsh Verma, Anirudh Chelluri, Rajeev Alur, and Boon Thau Loo

***Abstract*— Recent emergence of software-defined networks offers an opportunity to design domain-specific programming abstractions aimed at network operators. In this paper, we propose scenario-based programming, a framework that allows network operators to program network policies by describing example behaviors in representative scenarios. Given these scenarios, our synthesis algorithm automatically infers the controller state that needs to be maintained along with the rules to process network events and update state. We have developed the NetEgg scenario-based programming tool, which 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 evaluate the performance of NetEgg based on the computational requirements of our synthesis algorithm as well as the overhead introduced by the generated policy implementation, and we study the usability of NetEgg based on a user study. Our results show that our synthesis algorithm can generate policy implementations in less than a second for all policies we studied, and the automatically generated policy implementations have performance comparable to their hand-crafted implementations. Our user study shows that the proposed scenario-based programming approach can reduce the programming time by 50% and the error rate by 32% compared with an alternative programming approach.**

***Index Terms*— NetEgg, SDN, programming-by-examples.**

I. Introduction

SOFTWARE-DEFINED networking (SDN) holds the promise of extensible routers that can be customized directly by network operators. Major router vendors now provide APIs (OpenFlow or vendor specific) that provide various forms of extensibility for traffic steering, on-demand network virtualization, security policies, and dynamic service chaining. The enhanced programming interface of SDN offers an opportunity to design domain-specific programming abstractions for network operators to take advantage of the flexibility to program network policies.

Manuscript received July 25, 2017; revised June 6, 2018; accepted July 10, 2018; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor S. Uhlig. Date of publication August 27, 2018; date of current version October 15, 2018. This work was supported by the NSF under Grant CNS-1513679, Grant CCF 1763514, Grant ITR-1138996, and Grant IIP-1564730. *(Corresponding author: Yifei Yuan.)*

Y. Yuan, R. Alur, and B. Thau Loo are with the Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104 USA (e-mail: yifeiy@cis.upenn.edu; alur@cis.upenn.edu; boonloo@cis.upenn.edu).

D. Lin is with LinkedIn, Sunnyvale, CA 94085 USA (e-mail: dolin@linkedin.com).

S. Anil is with Bloomberg LP, New York, NY 10022 USA (e-mail: siri.rao13@gmail.com).

H. Verma is with Intentionet, Seattle, WA 98104 USA (e-mail: harsh@intentionet.com).

A. Chelluri is with Facebook, Menlo Park, CA 94025 USA (e-mail: ani.chelluri@gmail.com).

Digital Object Identifier 10.1109/TNET.2018.2861919

To take advantage of this new wave of innovation, recently proposed domain-specific languages or DSLs (e.g. declarative networking [1], Frenetic [2], Pyretic [3], NetKAT [4], NetCore [5], FlowLog [6], Merlin [7], FatTire [8]) make it easier to program controllers with orders of magnitude reduction in code sizes by raising the level of abstraction.

A key challenge that has yet to be addressed is providing an intuitive programming abstraction that allows network operators even with little programming experiences to program their own protocols and policies, hence taking advantage of the new programming interface.

Motivated by recent work on programming by examples [9]–[11], we investigate an alternative approach aiming at providing network operators intuitive programming interfaces. Our approach is based on *synthesizing* an implementation automatically from *example scenarios* and providing a platform whereby operators can observe the synthesized implementation at runtime, and then tweak their input scenarios to refine the synthesized program.

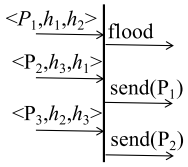
Our proposed approach is based on the observation that network operators typically like to use examples such as timing diagrams to design new network configurations and policies. In most cases, these examples would be generalized into design documents, followed by pseudocode and then finally implementation. We aim to facilitate the entire process by generating implementations directly from the examples themselves, hence giving the power of network programmability to all network operators. While the focus of this paper is on SDN settings, the approach is general and can be applied to any network protocol design and implementation.

Specifically, this paper makes the following contributions:

Scenario-Based Programming Framework: We propose the framework of scenario-based programming (Section IV), which allows operators to specify network policies using example behaviors. Instead of implementing a network policy by programming, the operator simply specifies the desired network policy using *scenarios*, which consist of examples of packet traces, and corresponding actions to each packet.

Design and Implementation: We have developed the NetEgg tool, including a synthesis algorithm (Section V), an interpreter for executing policies, and a web-based graphical user interface. 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, relevant fields associated with the state and rules for processing packets and updating states. 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 (Section VI).

Validation: We validate NetEgg by synthesizing SDN programs that use the POX controller directly from examples. Our tool is agnostic to the choice of SDN controllers, and



Scenario 1

Fig. 1. A scenario describing the learning switch. In the scenario, a packet is denoted by a 3-tuple: ⟨incoming port, source MAC, destination MAC⟩.

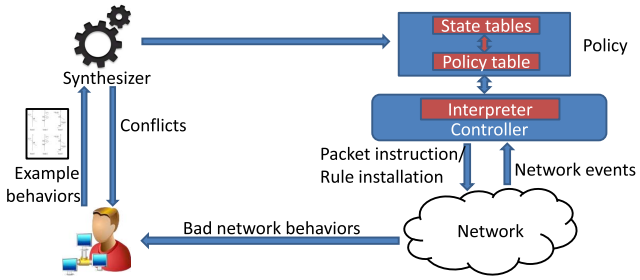


Fig. 2. NetEgg architecture.

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 (Section VII). The synthesized controller program achieves comparable performance to equivalent imperative programs implemented manually in POX (Section VIII). Moreover, our user study (Section IX) shows that NetEgg can reduce the programming time and the error rate compared with programming in POX.

We vision NetEgg as a rapid prototyping platform and an educational tool, where users can iterate through example scenarios, observe runtime behavior to determine correctness, and tweak their scenarios otherwise. NetEgg is correct and consistent with respect to the input scenarios. Synthesizing the input scenarios themselves based on high-level correctness properties is an avenue of future work.

II. Illustrative Example

To illustrate the use of NetEgg, we 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 program the policy, the network operator simply describes example behaviors of the policy in representative scenarios, in the form of network timing diagrams. Fig. 1 shows a scenario described by network operators.

The Scenario: In this scenario, the vertical line denotes the time line and 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 implementing the learning switch policy. 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 VII.

III. NetEgg Overview

Fig. 2 provides a high-level overview of NetEgg. The network operator describes example behaviors about the desired network policy in timing diagrams, where each timing diagram shows the behavior of the policy in a specific scenario (thus we also refer to the timing diagrams as scenarios). The timing diagrams can be drawn using our GUI or written in a simple configuration language, which we will describe in Section IV.

Given all scenarios, NetEgg first checks whether there exist conflicting behaviors 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 consists of a set of *state tables* and a *policy table*. State tables are used to remember the history of policy execution (e.g., the learned port for a MAC address in the learning switch example or the connection state for TCP connections). The policy table dictates actions for incoming network events and updates of state tables in 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), which determines 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.

Revisiting the learning switch example from the previous section, we first describe the state tables that are generated by our tool, before describing the policy.

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 learning switch needs to maintain a state table ST, which stores a state to indicate whether the port is learned and a value (in this case the associated port number) for each MAC address. See Fig. 4 for examples.

We design state tables in such a form in order to achieve a reasonable balance between the expressiveness of the policy model and also the efficiency of our synthesizer. Particularly, compared with a standard key-value map, the state table uses the additional state to explicitly indicate the number of cases the policy should care about (e.g., there are 2 cases for the

TABLE I
THE POLICY TABLE FOR THE LEARNING SWITCH

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

learning switch while other policies may have multiple cases), in order to support efficient policy synthesis (see Section V for more detail).

Given input scenarios, our tool automatically derives how many states need to be maintained in a state table, and what values need to be stored as the value for a key. For the learning switch example, NetEgg automatically infers the fact that for a given MAC address $macaddr$, ST needs to keep two states: state 0 indicates that the port associated with $macaddr$ is not learned, and state 1 indicates that the port is learned.

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 I for the learning switch example. We delay the discussion of its generation to Section V.

The policy table contains two *rules*, represented as the two rows in the table. We extend traditional match-action rules with additional state test and update in order to support stateful policies. Thus, 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 state value. For example, the test in the second rule has one check ST(dstmac).state=1, which checks whether the state associated with the dstmac address of the packet is 1 in ST. Note that we do not allow the check of values in a state table in order to achieve efficient policy synthesis. However, with this restriction, NetEgg can still support a wide range of policies as shown in Section VII. 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(dstmac).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(srcmac):=(1,port) changes the state associated with the srcmac address of the packet to 1 in ST, and stores the value associated with the srcmac address of the packet to the port of it. These two rules correspond to two cases of the learning switch: 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 learned.

Interpreter: The interpreter processes incoming packets at the controller using the policy table. The pseudocode of the interpreter is shown in Fig. 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 VI.

```
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
apply default actions to p
```

Fig. 3. The interpreter.

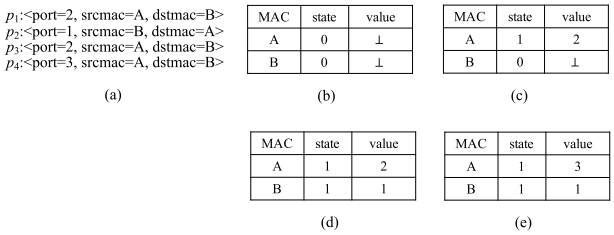


Fig. 4. An illustrative execution. (a) An example packet trace. (b) The initial state table. (c) The state table after p_1. (d) The state table after p_2. (e) The state table after p_4.

Example: Fig. 4 shows an illustrative execution of the policy in Table I for the incoming packet trace in subfigure (a). The purpose of this example is to illustrate how a policy table is executed, and thus we assume that every packet is processed on the controller. In Section VI we will describe how to automatically infer rules to be installed onto switches. 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 I 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 A has migrated to port 3, and it matches the second rule in the policy table and gets sent to port 1. Subfigure (e) shows the state table after applying the update. In Section VII, we will demonstrate other use cases including the use of timeout events to keep soft state.

IV. NETEGG MODEL

In this section, we first describe the scenario-based programming model of NetEgg, and explain how it 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.

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

Fig. 5. Scenario-based programming model.

A. Programming Language

NetEgg provides a configuration language for expressing network timing diagrams (Fig. 5). 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 field names along with their types. In our example, 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. In our current implementation, we assume that all packets appearing in a scenario-based program have the same packet-type. However, one can easily allow different packet-types in a scenario-based program.

Thus, the scenario of Fig. 1 can also be written as:

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

A scenario is *concrete* if all the symbolic packets and actions appearing in the scenario have only concrete values. A concrete scenario can be viewed as a test case that describes a specific sequence of packets coming to the network (in order) as input and also its corresponding applied actions as output of the desired policy. 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 Fig. 1 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. Thus, the use of symbolic values allows one to concisely describe a large set of concrete scenarios showing example behaviors of the desired policy in different cases.

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 generates the configuration.

B. 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 $\mathtt{T}_{i_1} \times .. \times \mathtt{T}_{i_d}$. Though we only allow one value per key in our model, one may use multiple state tables if multiple values need to be maintained.

The operations we allow on a state table are reads, checks and writes. Let ST be a state table of type $\mathtt{T}_1 \times .. \times \mathtt{T}_d \rightarrow S \times \mathtt{T}_{d+1}$, $\mathrm{f}_1, .. \mathrm{f}_d$ 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 $ST(\mathrm{f}_1, .., \mathrm{f}_d)$.state=s, where $\mathtt{s} \in S$ is a state. In our example, ST(dstmac).state=0 is a check with the field dstmac. The use of the state in state tables together with the restriction that we can only check the state allows us to develop efficient synthesis algorithm as we will see in the next section. 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(srcmac):=(1,port) is a write of ST with the field srcmac.

We use the term configurations for the snapshots of state tables. For example, the initial configuration of the state table in our example maps every MAC address to $(0, \perp)$ as shown in figure 4(b). Here, we use $\perp$ to represent the fact that no value is stored. A read $ST(\mathrm{f}_1, .., \mathrm{f}_d)$ for a packet p at a configuration c returns the state-value pair stored in ST for the key $(p.\mathrm{f}_1, .., p.\mathrm{f}_d)$ at c. We use $ST(\mathrm{f}_1, .., \mathrm{f}_d)$.state and $ST(\mathrm{f}_1, .., \mathrm{f}_d)$.value to denote the state and value in the returned pair. A check $ST(\mathrm{f}_1, .., \mathrm{f}_d)$.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 Fig. 4, ST(dstmac).state=0 is true for p_1 at the initial configuration (subfigure (b)) of ST. A write $ST(\mathrm{f}_1, .., \mathrm{f}_d) := (\mathrm{sv}, \mathrm{fv})$ 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 $\langle \mathrm{f}_1 = \mathrm{v}_1, .., \mathrm{f}_k = \mathrm{v}_k \rangle$, where f_i is a name of a packet field, and v_i is a concrete value or a wildcard. A packet p matches $\langle \mathrm{f}_1 = \mathrm{v}_1, .., \mathrm{f}_k = \mathrm{v}_k \rangle$ iff v_i is a wildcard, or $p.\mathrm{f}_i = \mathrm{v}_i$ for all $i = 1$ to k. The actions is a list of actions

TABLE II
AN INCONSISTENT POLICY TABLE

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

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 I 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.

V. POLICY GENERATION

Given a set of scenarios describing a policy, our synthesizer first checks if there are conflicts among scenarios. This process is described elsewhere [23] and thus omitted due to the space constraint. In this section, we focus on how to generate a policy, consisting of a set of state tables and a policy table, given a set of scenarios without conflicts. We start this section by discussing the objective policies NetEgg aims to generate. Then we present the synthesis algorithm in details.

A. 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 I is consistent with the scenario in Fig. 1. However, the policy in Table II is not consistent with the scenario, since it floods the third packet in the scenario instead of sending it to P_2.

In addition to consistency, NetEgg also aims to generate a generalized policy from input scenarios. Therefore, we also try to minimize the number of rules in a synthesized policy. To see how this heuristic can help to generate a general policy, let us consider the policy table in Table III with 3 rules. It can be verified that the policy is consistent with the scenario in Fig. 1. 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 I only uses two rules, and can handle the fourth packet mentioned above correctly.

TABLE III
A CONSISTENT YET RESTRICTIVE POLICY TABLE

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 IV
AN EXAMPLE POLICY TABLE SKETCH

match	test	actions	update
*	ST(dstmac) .state=0	ax_1	ST(srcmac) :=(sx_1, ux_1) ST(dstmac) :=(sx_2, ux_2)
*	ST(dstmac) .state=1	ax_2	ST(srcmac) :=(sx_3, ux_3) ST(dstmac) :=(sx_4, ux_4)

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.

B. Synthesis Algorithm Overview

In order to generate a consist policy table, a naïve algorithm is to enumerate all possible policy tables by increasing the number of rules, and then check the consistency with each enumerated policy table. The first consistent policy table is the desired one. While this simple algorithm meets the two requirements of the policy learning problem, however, the exhaustive enumeration is prohibitively inefficient in practice given the large number of policy tables.

To improve the efficiency of the algorithm, our key idea is to summarize a large number of policy tables using a symbolic representation called a *policy table sketch* (or *sketch* in short). As an example, Table IV shows a sketch. In a sketch, actions and parameters in update are represented using variables, while match and tests remain concrete. By substituting concrete values for the variables in a sketch, we can obtain a concrete policy table from the sketch. Therefore, a sketch naturally represents a large number of concrete policy tables. In this example, if each variable takes 3 different values, this sketch can represent 3^{10} policy tables. Here, we keep match and tests concrete in order to enable efficient search for variables in actions and update as shown later.

Based on policy table sketches, we improve the naïve enumeration algorithm by enumerating all possible sketches instead of concrete policy tables. Since a sketch may represent potentially a large number of policy tables, our enumeration can be significantly efficient. Now, given an enumerated sketch, our algorithm further searches concrete values for variables in the sketch in order to generate a consistent policy table. In the worst case, all values for each variable need to be searched, and we essentially enumerate concrete policy tables. However, in practice we oftentimes only need to search values for a small subset of variables by using efficient back tracking heuristics, and thus skipping the checking for a large number of potential policy tables. For example, when ax_1 is not flood, we can conclude that no matter what values are set for other variables, we can not obtain a consist policy table from the sketch. Therefore, we simply skip the search for other variables when ax_1 is not flood. Putting in other words, we can skip the enumeration and consistency checking for a large number of concrete policy tables.

Thus, our overall synthesis algorithm consists of two phases as shown in in Algorithm 1. First, the synthesis algorithm enumerates sketches by increasing the number of rules. For each sketch $sketch$, it invokes the procedure search_sketch($[SC_i]$, $sketch$) to search concrete values for variables in the sketch in order to generate a consistent policy table using the sketch.

Algorithm 1 synthesize($\{SC_i\}$)

```
1: for all sketch generated from generate_all_
   sketches({SC_i}) do
2:   pt = search_sketch([SC_i], sketch)
3:   if pt is not NONE return pt
```

In the rest of this section, we first show how to enumerate sketches given all input scenarios (i.e., generate_all_sketches($\{SC_i\}$). Next, we show how to use back tracking heuristics to efficiently search the values for variables in each sketch (i.e., search_sketch($[SC_i]$, $sketch$).

C. Sketch Enumeration

Our algorithm generates a sketch by composing a list L of match**es** together with a list of **tests**. Here, the list of L can be naturally learned from the symbolic packets in the input scenarios. However, we may not infer the list of **tests** from the input scenarios. Therefore, our algorithm enumerates all possible **tests**, and further infers all necessary **update** to complete the sketch. The reason that we can efficiently enumerate all possible **tests** is that we only allow the check of states. To ease the presentation of our algorithm, in this section we will assume that a rule in a policy table contains at most one **check** in its **test**. However, our algorithm and implementation handles the case of multiple **check**s.

Algorithm 2 shows how to enumerate sketches. First, the algorithm infers the list L from the input scenarios (line 2), and then enumerates all possible **tests**. The enumeration consists of two parts. First, the algorithm enumerates the size s of the state set used in the sketch (line 3) and then it enumerates all possible **read**s constructed from all combinations of packet fields (line 4). For the learning switch example, since all fields used in the input scenario are port, srcmac, and dstmac, there are 8 possible **read**s including ST_1(port), ST_2(port, srcmac), ... ST_8(port,srcmac,dstmac). Note that we enumerate s first to ensure that all generated sketches are ordered by their number of rules (the larger s is, the more rules a sketch has). With the match list L, enumerated **read** $read$ and also the state number s, the algorithm generates a new sketch by constructing all components in each rule (line 5).

Algorithm 2 generate_all_sketch($[SC_i]$)

```
1: sketch_list = []
2: L = generate_match_list([SC_i])
3: for all s = 1, ..., MAX do
4:   for all read do
5:     sketch = generate_sketch(L, read, s)
6:     add sketch to sketch_list
7: return sketch_list
```

In the following, we first describe how to infer the list L of match**es**, and then describe our algorithm to generate a sketch from L, $read$ and s.

Algorithm 3 generate_match_list($[SC_i]$)

```
1: L = []
2: for all packet sp = ⟨f_i=v_i⟩ in every scenario SC_i do
3:   m ← ⟨f_i=m_i⟩, s.t. m_i = v_i if v_i is a concrete value else
     *
4:   insert m to L
5: sort L
6: return L
```

Generate Ordered Match *List:* Given the input scenarios, we can naturally learn a list of all match**es** from the packets, 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 with concrete field values. Therefore, Algorithm 3 generates a **match** for each symbolic packet in the scenarios by replacing symbolic values using $*$ (line 3). After all match**es** are learned from the input packets, the algorithm needs to sort the list L such that no **match** on the top completely covers some **match** below it (line 5). This ensures that each symbolic packet can match its corresponding learned **match**. Note that for two generated match**es** which are partially overlapping with each other, we can order either one above the other.

Generate a Sketch: Given the match list L, the **read** $read$ used in **test** in the sketch, and the state set size s, we generate a sketch by composing L with all possible **check**s generated from $read$ and a state ranging from 0 to $s-1$. The algorithm is shown Algorithm 4. For each **match** $match$ in L and each state number l, the algorithm constructs a rule using $match$ and the **check** $read$.state $= l$ (line 8). In addition, the algorithm also needs to fill in all other sketch variables. For the **action**, the algorithm simply introduces a fresh variable. However for the **update**, the algorithm needs to consider all possible ways to update the state table by constructing all necessary **write**s (line 6 and 7). For the example in Table IV, since the $read$ is ST(dstmac), we need to consider ST(srcmac) and ST(dstmac) as shown in the table (Here we intentionally ignored the effect of the order of **write**s for the simplicity of presentation. However, we handle the order in our implementation.).

Algorithm 4 generate_sketch(L, $read$, s)

1: $sketch = []$
2: let $read$ be $ST(\mathrm{f}_1,..,\mathrm{f}_k)$
3: **for all** match $match$ in L **do**
4: **for all** $l = 0, ..., s-1$ **do**
5: $update = []$
6: **for all** read $ST(\mathrm{f}_1',..,\mathrm{f}_k')$ s.t. f_i' and f_i have the same type **do**
7: add $ST(\mathrm{f}_1',..,\mathrm{f}_k')$:=$(sx, ux)$ to $update$, with sx, ux being fresh variables
8: $r \leftarrow (match, (read.\text{state} = l), ax, update)$, with ax a fresh variable
9: add rule r to $sketch$
10: **return** $sketch$

D. Sketch-Based Search

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 simulate the run of the policy table and perform a backtracking search algorithm over necessary sketch variables. The algorithm is shown in Algorithm 5.

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

1: $stack = []$; $V = \{\}$; $ST =$ new state table
2: **for all** scenario SC_i **do**
3: clear ST
4: **for all** events e_j in SC_i **do**
5: let $r = (match, test, ax, update)$ be the first matched rule
6: **if** ax not in V **then**
7: $V[ax] \leftarrow$ drop; $stack$.push(ax)
8: **if** $V[ax]$ is consistent with e_j **then**
9: **for all** x in $update$ s.t. x not in V **do**
10: $x \leftarrow$-; $stack$.push(x)
11: apply $update$
12: **else**
13: **while** $stack$ is not empty **do**
14: $x = stack$.pop(); $v = V(x)$; V.remove(x)
15: **if** getNextValue(x, v) **then**
16: $V(x) \leftarrow$ getNextValue(x, v)
17: $stack$.push(x)
18: restart from line 2
19: **return** NONE
20: **return** pt obtained by substituting values in V for variables

The algorithm maintains the needed state table, a stack of sketch variables together with a map V storing the values assigned to the variables. Whenever a sketch variable is assigned a value, it ensures that the sketch variable is pushed to the stack. For each symbolic event in every scenario, the algorithm checks consistency of the first matched rule's actions (line 8) (if the action is not assigned a value, then initialize it first as in line 7). If the action is consistent with the event, the algorithm continues to apply the update of the rule, which updates the maintained state table (We may first initialize all sketch variables in update before the update can be applied, as in line 9, 10). However, whenever inconsistency encountered (line 12), the algorithm needs to backtrack. This procedure involves identifying the top most sketch variable that has new values to be searched, assigning the next possible value to it, and then restart consistency checking (line 13-18). Note that the range of values each sketch variable can take is determined by the sketch and scenarios. For example, if (sx, ux) appears in some write, the values of sx can only range from $\{0, .., s-1, \text{-}\}$, where s is the size of state set used to generate the sketch; and ux can only range over -, field names and concrete values appearing in the scenarios. It is similar for variables ax's appearing as actions. If the algorithm has searched all values for all sketch variables on the stack, we can conclude that no consistent policy table exist for this sketch (line 19); otherwise the algorithm returns the first found consistent policy table (line 20).

E. 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 IV are not used when checking consistency for any symbolic packet in Fig. 1, 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.

VI. 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 6 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: $match \leftarrow \langle \mathrm{f}_{i_1}{=}p.\mathrm{f}_{i_1},..,\mathrm{f}_{i_k}{=}p.\mathrm{f}_{i_k}\rangle$, 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: **for all** installed rule $match' \rightarrow [a'_1, .., a'_l]$ in the flow table **do**
6: let p' be a packet matches $match'$
7: let rule r be the matched rule in the policy table for p'
8: **if** r does not update state tables or the updated state tables remain unchanged **then**
9: update the installed rule to $match' \rightarrow r$.actions, if the actions a_j applied to p by r is supported by the switch
10: **else**
11: remove the installed rule from the flow table

Algorithm 6 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 in Fig. 4. By the interpreter's algorithm shown in Fig. 3, the first packet is processed on the controller, and the state table is updated to the one shown in subfigure (c). Applying Algorithm 6, the matching rule r for p_1 would be the first rule in the policy table shown in Table I. Since port 2 is already remembered for the srcmac A, r would not change the state table. Therefore, a flow table rule fr_1 = ⟨port=2,srcmac=A,dstmac=B⟩ →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 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).

scenario 1:
s_1, P_1, h_1, h_2 ⇒ flood
s_1, P_2, h_3, h_1 ⇒ send(P_1)
s_1, P_3, h_2, h_3 ⇒ send(P_2)

Fig. 6. Scenario-based program for the learning switch.

TABLE V
THE POLICY TABLE 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)

VII. 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 [12] and Nice [13]. We plan to explore light-weight verification tools for the custom policy abstraction in the future.

A. 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 Fig. 1. 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 ⟨switch : SWITCH, port : PORT, srcmac : MAC_ADDR, dstmac : MAC_ADDR ⟩. For the scenario, we simply add the switch field to each symbolic packet in the scenario in Fig. 1. This modified scenario suffices for NetEgg to synthesize the network-wide learning switch policy. The scenario and synthesized policy table is shown in Fig. 6 and Table V.

B. 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 Fig. 7. In the first scenario, the switch blocks the traffic from port 2,

scenario 1:
2, h_2, h_1 $\Rightarrow$ drop

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

Fig. 7. Scenario-based program for the 1st stateful firewall.

TABLE VI
THE POLICY TABLE FOR THE 1ST 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)	-

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

Fig. 8. The modified scenario for the 2nd stateful firewall.

TABLE VII
THE POLICY TABLE FOR THE 2ND STATEFUL FIREWALL

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)	-

and the second scenario demonstrates the case where the firewall allows the traffic from port 2. Notice that while the first packet in scenario 1 and the second packet in scenario 2 are identical, the applied actions are different. This demonstrates again the key difference between scenarios and traditional rules. It turns out that these two scenarios are sufficient to generate the desired policy, shown in Table VI.

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 Fig. 7 and change the dstip of the second packet in scenario 2, as shown in Fig. 8.

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

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 that 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 Fig. 9. 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

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

Fig. 9. The added scenario for the 3rd stateful firewall.

TABLE VIII
THE POLICY TABLE FOR THE 3RD STATEFUL FIREWALL

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,-)

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

Fig. 10. Scenario-based program for the TCP firewall.

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 Fig. 7 and Fig. 8 respectively.

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

C. 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 Fig. 10. 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

TABLE IX
THE POLICY TABLE 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 X
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

	#EV	#SC	Time
maclearner1	3	1	11 ms
maclearner2	3	1	15 ms
auth	3	2	13 ms
gardenwall	5	3	52 ms
ids	3	2	15 ms
monitor	3	2	13 ms
ratelimiter	10	5	147 ms
serverlb	7	3	143 ms
stateful firewall1	3	2	12 ms
stateful firewall2	3	2	16 ms
stateful firewall3	6	3	107 ms
trafficlb	7	3	402 ms
ucap	3	2	13 ms
vmprov	3	2	24 ms
TCP firewall	9	5	64 ms
ARP proxy	5	2	49 ms

the synthesizer generates the desired policy. The generated policy table is shown in Table IX, and the state table maintains states for each tuple of srcip, dstip, srcport and dstport.

VIII. PERFORMANCE EVALUATION

We have developed the NetEgg tool (including all components in Fig. 2) in Python, as well as a web-based graphical user interface. We evaluate the performance of NetEgg along three dimensions: (1) the efficiency of NetEgg in policy generation for a range of SDN policies, (2) the performance and overhead of the synthesized policies, and finally, (3) correctness of the flow table rule installation strategy.

A. Policy Generation

We survey recent literature on SDN policies, collect a range of SDN policies, and then use NetEgg to implement these policies [12], [14], [15]. Table X summarizes our results.

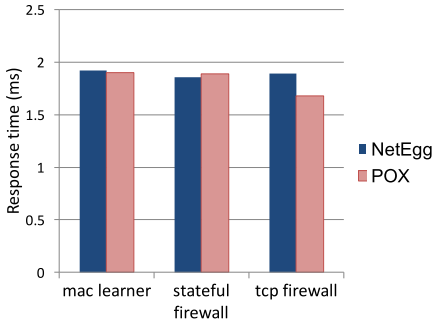


Fig. 11. Response time.

We report the total number of events in the scenarios used to program each policy, the number of scenarios, and the computation time of the synthesizer to generate the policy from scenarios.

We make the observation that NetEgg can generate a wide range of SDN policies considered in literature efficiently. In all examples, it requires no more than 402 ms for the synthesizer to generate a policy, which allows real-time interaction with the users. We also notice that programming in NetEgg using scenarios can be concise. All of the policies are expressed in less than 5 scenarios, with up to 10 events in total. Our user study (described in Section IX) also confirms our findings.

B. Policy Execution

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.

1) Cbench Evaluation: We first use Cbench [16], an SDN controller performance testing tool, to evaluate the performance of policy implementations on NetEgg.

Experiments: We evaluate the response time and throughput of NetEgg. For response time, 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. We evaluate the throughput using a number of switches. For comparison, we also evaluate the policies' implementations in POX.

Results: Fig. 11 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 ($<1\%$). For throughput, we measure the number of requests that the MAC learning policy can handle. When saturated using 20 switches, the NetEgg implementation handles 1098 requests per second, while POX handles 2484. We leave as future work to explore how to further improve the throughput via parallelization techniques. However, as we will see later, the end-to-end performance is not significantly degraded.

2) 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.

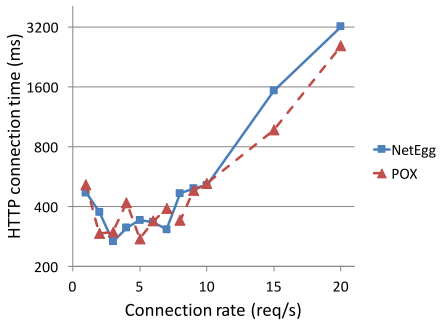


Fig. 12. HTTP connection time.

Fig. 13. The network topology for the Firewall++ assignment. The network has a single switch with interface 1 connected with the CS department and interface 2 connected with the Internet. The controller (top-right box in the figure) needs to be programmed to implement the Firewall++ policy.

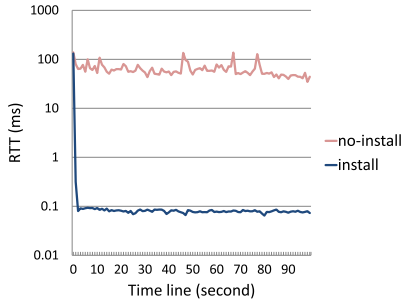


Fig. 14. Effects of flow table rule installation.

Experiments: We emulate a fattree topology [17] 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 POX and NetEgg.

Results: Fig. 12 reports the average connection time over all 15 clients. The x-axis is the rate of HTTP requests. 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.

C. 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 [18]. 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 Fig. 14. 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.

IX. User Study

To evaluate the usability of our approach, we conducted user studies centered around the following questions:

- Q1: Can NetEgg reduce the error rate of programming SDN policies?
- Q2: Can NetEgg reduce the programming time?
- Q3: How well does the user interact with NetEgg?

In the following subsections, we first describe our user study design, followed by our results.

A. User Study Design

We conduct the user study among masters/PhD students in the School of Engineering within our university, given that they are readily available to us. Given that NetEgg is also geared towards an educational tool, these students represent the primary users of the tool. Since most students were not proficient in recently proposed high-level SDN programming languages such as Pyretic and Kinetic, we compare NetEgg with POX, a popular SDN controller based on Python.

We divide students into a NetEgg group and a POX group. Since POX is based on Python, we require all users in the POX group to be proficient in Python to ensure the quality of the programming task. However, since NetEgg does not require any programming experience, we have not imposed any programming knowledge constraints on the NetEgg group. At the end, 15 users are in the NetEgg group and 9 users in the POX group. All users reported no SDN programming experience before, and all users except for one in the NetEgg group have taken classes in networking or distributed systems – they are a good proxy of actual network operators, given their knowledge of networking but not the SDN programming background.

POX is a mature software while NetEgg is still in development stage. Prior to this study, we carry out three additional user studies on NetEgg (on 21 additional students), mostly to iron out any bugs of NetEgg and provide feedback on the UI design before our actual study. We exclude these 21 students from the study results since they worked on earlier version of NetEgg with some bugs, though the general observations are similar even if they are included.

Programming Assignment: We asked the users to program a stateful firewall application (called Firewall++) to protect an

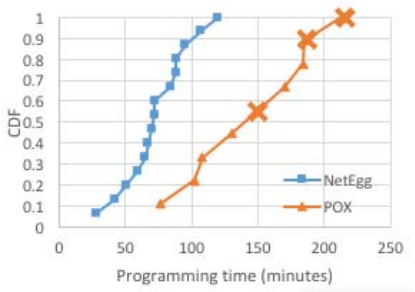


Fig. 15. The CDF of the programming time for NetEgg and POX.

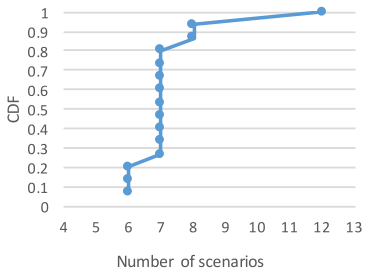


Fig. 16. The CDF of the number of scenarios for NetEgg.

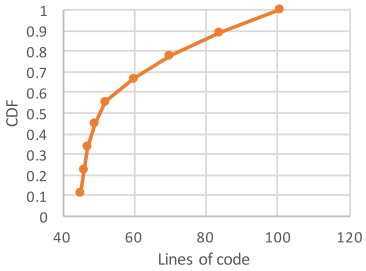


Fig. 17. The CDF of the lines of code for POX.

internal network (i.e. the CS department) from the Internet in the network topology shown in Fig. 13. We chose the firewall application because of its importance and wide deployment in today's networks, and also its simplicity in the functionality.

The requirement of Firewall++ was described as the following rules to the users:

0. The firewall should allow any non-IP traffic.
1. All UDP traffic from the Internet are dropped.
2. All UDP traffic from the CS department network to the Internet are allowed to go through.
3. All TCP traffic from the CS are allowed to go through.
4. SSH packets (i.e. TCP packets with dstport 22) coming from the Internet are blocked.
5. TCP traffic from any host A in the Internet at TCP port P to any host B in the CS at TCP port Q is allowed if B sends TCP traffic using TCP port Q to A at TCP port P before; otherwise the TCP traffic from the Internet should be dropped.

To test the users' programs, we provided the users with a script that automatically tested their programs in 5 different test cases, corresponding to rule 1-5 described above. The rule 0 is tested along with these test cases. This test script runs the controller with the policy program synthesized from user's scenarios or coded in POX, and emulates the network in Fig. 13 using the network emulator Mininet [18]. In each test case, the script instructs hosts in the emulated network to send traffic to the network. When testing each case, the script displayed a short description of the test case, and only showed to the user PASS or FAIL after testing finished.

Setup: Before the user started programming, we first gave a 20-minute tutorial to both groups on the basics of SDN programming. This tutorial covered a brief introduction to SDN, step-by-step illustration on how to use NetEgg/POX to program the example learning switch policy, and finally the description of the Firewall++ programming assignment. We also prepared an instruction on the usage of NetEgg, highlighting the usage of how to create new scenarios and how to add events into a scenario. Since POX is widely used and well documented, we referred the users to the detailed official POX instructions [19], [20].

To help users focus on the programming of Firewall++ in POX, we prepared a program template where the user only needs to implement the body of the function of handling the events triggered by the packets sent to the controller.

B. Results

We present the results based on both the logs from the UI instrumentation and the surveys with the users.

Q1: Error Rate: We observe that NetEgg reduces the error rate of programming the SDN policy. We collect all users' final policy programs, and consider a policy program to be correct if it can pass all 5 test cases. For the NetEgg group, all users' final policy programs are correct. However, for the POX group, only 6 (67%) users' programs are correct, while the incorrect programs only pass 4 test cases.

Q2: Programming Time: We observe that NetEgg reduces the programming time by 50% on average. Fig. 15 shows the CDF of the programming time for both groups. In particular, every marker in the figure corresponds to a user's program and a cross marker in the POX line denotes the fact that the corresponding user's policy program is incorrect. On average, the NetEgg group took 73 minutes for the Firewall++ assignment. The minimal and maximal were 28 and 120 minutes. In contrast, the POX group took 146 minutes on average. The minimal and maximal were 76 and 213 minutes. This includes the users who did not pass all test cases. When excluding these users, the average programming time for the POX group is 128 minutes, NetEgg still reducing 43% programming time. In short, the results are encouraging: the NetEgg group used significantly less time while achieved much higher correct rate compared with the POX group.

Program Size: In addition to the programming time, one typically used metric to measure the programming effort is the program size (i.e. lines of code, or LoC). However, it is hard to compare directly the program sizes between NetEgg and POX, given that the scenario-based programming approach does not have a notion equivalent to LoC. Thus, we use the number of scenarios to approximate the size of a NetEgg project. Fig. 16 and Fig. 17 show the CDF of the number of scenarios and LoC for NetEgg and POX. On average, the NetEgg group used 7.2 scenarios (with 1.32 events per scenario), and the average LoC of POX programs is 61.6. As an interesting comparison, we manually programmed this policy in Pyretic, and the LoC is 30. Though this comparison should not be viewed as a quantitative one given the reasons explained above, we do believe that qualitatively speaking, NetEgg allows a smaller program size compared with POX.

Q3: User's Interaction With NetEgg: We logged a user's scenarios each time she built the policy program by calling the NetEgg synthesizer, in order to understand the users' inter-

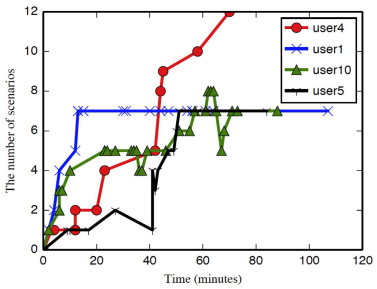


Fig. 18. The number of scenarios for representative users along with time.

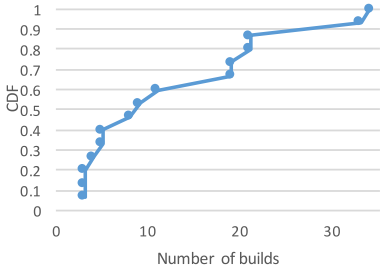


Fig. 19. The CDF of the number of unique builds for NetEgg.

TABLE XI
SUMMARY OF INTERACTION PATTERNS WITH NETEGG

	SI	BFE	SE
# of users	9	2	4
Percentage	60%	13%	27%

action with NetEgg. Fig. 18 exhibits the number of scenarios along with the time line for representative users in the NetEgg group. To ensure that this figure reflects the user's progress in the user study, each marker in the figure corresponds to a unique build of a user (i.e. at least one scenario is different from previous build).

We observe three patterns of the users' interaction with NetEgg: smooth interaction (SI), back-and-forth edits (BFE), and stuck edits (SE). Table XI summarizes the number and percentage of users in each category.

The SI pattern consists of fairly smooth interaction with NetEgg. For example, see user4 (red line) in Fig. 18. At the beginning, user4 quickly added 2 scenarios in the first 20 minutes, and then incrementally added more scenarios until he got a correct program (recall all NetEgg users succeeded in passing all test cases). This pattern demonstrates the advantage of NetEgg's approach: It allows the user to adaptively demonstrate the desired SDN policy's behaviors and the synthesized program gradually converges to the desired one. We find that most of the NetEgg users' interactions (9 users) have similar patterns to this one. Their average programming time is 58 minutes, 40% of the average POX programming time.

The BFE pattern (user10, green line; user5, black line) has the following feature. The users were able to use scenarios to demonstrate the policy's behavior, however, they were not sure whether their scenarios demonstrated the correct behavior of the policy, and ended up removing some scenarios from the project. For example, at the 63rd minute, user10 deleted 3 scenarios and then added some more new scenarios. The similar behavior is also observed at the 40th minute and 41st minute for user5, as well as the 36th minute for user10. After talking with the users, we find that they were confused of the use of symbolic values, and sometimes misunderstood a scenario as a list of rules. We believe NetEgg would achieve much smoother interaction with these users if the concept of scenario-based programming was well conveyed. As user10 told us, he could have finished this assignment in 10 minutes if he had resolved the confusion sooner. For this category, the average programming time is 86 minutes.

The SE pattern (e.g. user1, blue line in Fig. 18) shows that some users were able to use multiple scenarios to demonstrate the example behaviors of the policy. However, instead of describing new behaviors using more scenarios, they tried to modify existing scenarios to tweak the synthesized policy program (e.g. from 13th minutes on for user1). They told us that they thought "scenarios" as "rules", and thus attempted to figure out how the synthesized program was generated from the "rules". This phenomenon is not unique to NetEgg, but shows a fundamental challenge of the programming-by-examples approach: How to improve the user's visibility of the synthesizing process, and guide the user to make high-quality examples to increase the user's confidence on the synthesized program [21]. We plan to address this challenge in the future work. We find 4 (27%) users in this category, and they finished the assignment in 102 minutes on average.

Our surveys also confirm some of our findings. A user thought NetEgg intuitive and easy to use:"the tool has a small learning curve and is quite intuitive …it is probably easier to use than a programming language.". Another user also gave a positive review:"I am very clear about what I am doing when I use the interface, and know exactly how the packets are handled under what situations." On the other hand, the user also said he "may get confused about how to define variables (symbolic values)".

Though NetEgg performed differently for users in the three categories, we find that NetEgg allows rapid iterations for all users across the three categories. Fig. 19 shows the CDF for the number of unique builds (i.e. if the user built a project without any changes, we do not count it) across all users. On average, a user performed 13 builds in the NetEgg user study. That is 1.7 builds every 10 minutes. Since each build was different for a user, this metric approximates the iteration rate for each user. Unfortunately, we could not find a similar metric for POX, since Python does not require build/compilation.

X. RELATED WORK

This paper extends our early version [22] with a user study to evaluate the usability of the proposed approach.

SDN Programming Languages: SDN domain-specific languages [2]–[8], [24] make programming policies easier using high-level abstractions. Our approach is different – we target at designing intuitive abstractions for network operators who can take advantages of their domain expertise and generate examples for our tool.

Programming by Examples: Our work is motivated by related work in the formal methods community in programming by examples. References [9]–[11] implement finite-state reactive controllers from specification of behaviors. Reference [25] generates string transformation macros in Excel from input/output string examples. Reference [26] uses both symbolic and concrete example to synthesize distributed protocols.

Our 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.

Policy Abstractions: Recent work proposes new abstractions of policies based on state machines [14], [15], [27]. These work shows the state machine abstraction benefits from fast execution on data plane [14], [27], and conciseness of programming [15]. Our abstraction of policy tables is similar in spirit to these state machine abstractions and thus can benefit from the advantages of previous work. But however, our work focuses on providing an intuitive programming framework which can generate policies directly from examples.

XI. Conclusion

In this paper, we explore the design and implementation of scenario-based programming that automatically generates network policy implementations from example scenarios.

We observe that NetEgg is expressive and can support a wide range of policies at reasonable overhead compared to imperative implementations. Our user study shows that programming in NetEgg is intuitive and concise, and can reduce both the programming time and error rate compared to alternative approaches. This approach lends itself naturally to rapid prototyping and shortening the design/implementation iteration cycle, as well as SDN programming education.

NetEgg is designed for state-oriented policies, and does not suit well for objective-oriented policies, such as shortest-path routing and traffic engineering. We leave the exploration of programming such policies using scenarios to future work.

Moving forward, we plan to carry out a user study to gather feedback from a larger pool of users. We also observe that NetEgg is slightly cumbersome for supporting policies that depend on stateful aggregate values, for example, take a particular action if a threshold is met. We plan to explore the combination of imperative languages with NetEgg, or using NetEgg with a database query language for enabling such complex policies. We plan to explore the use of formal verification techniques to check scenarios against high-level properties. Finally, while the paper focuses on SDN polices, the programming model is not restricted to SDN, and we plan to apply this approach to other settings, for example Internet and wireless routing policies.

References

[1] B. T. Loo *et al.*, "Declarative networking," in *Proc. CACM*, 2009, pp. 87–95.

[2] N. Foster *et al.*, "Frenetic: A network programming language," in *Proc. ICFP*, 2011, pp. 279–291.

[3] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software-defined networks," in *Proc. NSDI*, 2013, pp. 1–14.

[4] C. J. Anderson *et al.*, "NetKAT: Semantic foundations for networks," in *Proc. POPL*, 2014, pp. 113–126.

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

[6] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi, "Tierless programming and reasoning for software-defined networks," in *Proc. NSDI*, 2014, pp. 519–531.

[7] R. Soulé *et al.*, "Merlin: A language for provisioning network resources," in *Proc. CoNEXT*, 2014, pp. 213–226.

[8] M. Reitblatt, M. Canini, A. Guha, and N. Foster, "FatTire: Declarative fault tolerance for software-defined networks," in *Proc. HotSDN*, 2013, pp. 109–114.

[9] D. Harel and R. Marelly, *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Berlin, Germany: Springer-Verlag, 2003.

[10] D. Harel, "Can programming be liberated, period?" *Computer*, vol. 41, no. 1, pp. 28–37, Jan. 2008.

[11] D. Harel, A. Marron, and G. Weiss, "Behavioral programming," in *Proc. CACM*, vol. 55, no. 7, pp. 90–100, 2012.

[12] T. Ball *et al.*, "VeriCon: Towards verifying controller programs in software-defined networks," in *Proc. PLDI*, 2014, pp. 282–293.

[13] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE way to test openflow applications," in *Proc. NSDI*, 2012, p. 10.

[14] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan, "Flow-level state transition as a new switch primitive for SDN," in *Proc. HotSDN*, 2014, pp. 377–378.

[15] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark, "Kinetic: Verifiable dynamic network control," in *Proc. NSDI*, 2015, pp. 59–72.

[16] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks," in *Proc. HotICE*, 2012, p. 10.

[17] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. ACM SIGCOMM*, 2008, pp. 63–74.

[18] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proc. HotNets*, 2010, Art. no. 19.

[19] *POX Wiki*. Accessed: Mar. 24, 2017. [Online]. Available: https://openflow.stanford.edu/display/ONL/POX+Wiki

[20] *Openflow Tutorial*. Accessed: Mar. 24, 2017. [Online]. Available: http://archive.openflow.org/wk/index.php/OpenFlow_Tutorial

[21] M. Mayer *et al.*, "User interaction models for disambiguation in programming by example," in *Proc. UIST*, 2015, pp. 291–301.

[22] Y. Yuan, D. Lin, R. Alur, and B. T. Loo, "Scenario-based programming for SDN policies," in *Proc. CoNEXT*, 2015, Art. no. 34.

[23] Y. Yuan, "High-level programming abstractions for network policies," Ph.D. dissertation, Dept. Comput. Inf. Sci., Univ. Pennsylvania, Philadelphia, PA, USA, 2016.

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

[25] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *Proc. POPL*, 2011, pp. 317–330.

[26] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. K. Martin, and R. Alur, "TRANSIT: Specifying protocols with concolic snippets," in *Proc. PLDI*, 2013, pp. 287–296.

[27] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "OpenState: Programming platform-independent stateful openflow applications inside the switch," in *Proc. ACM SIGCOMM*, 2014, pp. 44–51.

Yifei Yuan received the Ph.D. degree in computer and information science from the University of Pennsylvania in 2016. He is currently a Post-Doctoral Researcher of computer and information science with the University of Pennsylvania.

Dong Lin received the Ph.D. degree in computer and information science from the University of Pennsylvania. He is currently a Staff Software Engineer with LinkedIn.

Siri Anil received the M.S. degree in embedded systems from the University of Pennsylvania in 2017. She is currently a Software Developer with Bloomberg LP.

Harsh Verma received the M.S. degree in computer and information science from the University of Pennsylvania. He is currently a Software Engineer with Intentionet.

Anirudh Chelluri received the M.S. degree in embedded systems from the University of Pennsylvania in 2018. He is currently a Production Engineer with Facebook.

Rajeev Alur received the Ph.D. degree in computer science from Stanford University in 1991. He is currently a Zisman Family Professor of computer and information science with the University of Pennsylvania.

Boon Thau Loo received the Ph.D. degree in computer science from the University of California at Berkeley in 2006. He is currently a Professor of computer and information science with the University of Pennsylvania.