SNAP: Stateful Network-Wide Abstractions for Packet Processing

Mina Tahmasbi Arashloo[1], Yaron Koral[1], Michael Greenberg[2], Jennifer Rexford[1], and David Walker[1]

[1]Princeton University , [2]Pomona College

Abstract

Early programming languages for Software-Defined Networking were built on top of the simple match-action paradigm offered by OpenFlow 1.0. However, emerging hardware and software switches offer much more sophisticated support for persistent state in the data plane, without involving a central controller. Nevertheless, managing stateful, distributed systems efficiently and correctly is known to be one of the most challenging programming problems. To simplify this new SDN programming problem, we introduce SNAP.

SNAP offers a simpler "centralized" stateful programming model, by allowing programmers to develop programs on top of *one* big switch rather than *many*. These programs may contain reads and writes to global, persistent arrays, and as a result, SNAP programmers can implement a broad range of applications, from stateful firewalls to fine-grained traffic monitoring. The SNAP compiler relieves programmers of having to worry about how to distribute, place, and optimize access to these stateful arrays by doing it all for them. More specifically, the compiler translates one-big-switch programs into an efficient internal representation based on a novel variant of binary decision diagrams. Using this internal representation, it discovers read-write dependencies, and constructs a mixed-integer linear program, which jointly optimizes the placement of state and the routing of traffic across the underlying physical topology. We have implemented a prototype compiler and applied it to about 20 SNAP programs over various topologies to demonstrate our techniques' scalability.

1 Introduction

The first generation of programming languages for software-defined networks (SDNs) [10,14,18,29,42] was built on top of OpenFlow 1.0, which offered simple match-action processing of packets. As a result, these systems were partitioned into (1) a stateless packet-processing part that could be analyzed statically, compiled, and installed on OpenFlow switches, and (2) a general stateful component that ran on the controller.

This "two-tiered" programming model can support any network functionality by running the stateful portions of the program on the controller and modifying the stateless packet-processing rules accordingly. However, many simple stateful programs—such as detecting SYN floods, DNS amplification attacks, and DNS tunneling—cannot be implemented *efficiently* because packets need to travel back-and-forth between the switches and the controller frequently, incurring significant overhead and delay. Thus, in practice, stateful programs on the controller are limited to those that do not require *per-packet* stateful processing.

Today, however, SDN technology has advanced considerably: there is a raft of new proposals for switch interfaces that *expose persistent state on the data plane*, including those in P4 [6], OpenState [4], POF [36], and Open vSwitch [24]. Stateful programmable data planes enable us to *offload* programs that require *per-packet* stateful processing onto programmable switches, subsuming a variety of functionality normally relegated to middleboxes. However, the mere existence of these low-level stateful mechanisms does not make networks of these devices easy to program. In fact, programming distributed collections of stateful devices is typically one of the most difficult kinds of programming problems. We need new languages and abstractions to help us manage the complexity and optimize resource utilization effectively.

For these reasons, we have developed SNAP, a new language that allows programmers to mix primitive stateful operations with pure packet processing. However, rather than ask programmers to program a large, distributed collection of independent, stateful devices manually, we provide the abstraction that the network is *One* Big Switch (OBS). Programmers can allocate persistent arrays on that one big switch, and as a result, they do not have to worry about where or how such arrays are stored in the physical network. Such arrays can be indexed by fields in incoming packets and modified over time as conditions in the network change. Moreover, if multiple arrays must be updated simultaneously, we provide programmers with a form of *network transaction* to ensure such updates occur atomically. As a result, it is easy to write SNAP programs that learn about the network environment and record its state, store per-flow information or statistics, or implement a variety of stateful mechanisms.

While it simplifies programming for the user, the OBS programming model, together with the stateful primitives, generate a number of implementation challenges. In particular, multiple different flows may depend upon the same stateful components. To process these flows correctly and efficiently,

the compiler must simultaneously determine which flows depend upon which stateful components, how to route those flows, and where to place the stateful components. Hence, to map programs written in terms of the OBS to concrete topologies, the SNAP compiler automatically discovers read-write dependencies between statements. It then translates the program into an xFDD, a variant of forwarding decision diagrams (FDDs) [33] extended to incorporate stateful operations. Next, the compiler generates a system of integer-linear equations that jointly optimizes the placement of arrays and the routing of traffic. Finally, the compiler generates the proper underlying switch-level configurations from the xFDD and the optimization results. We assume that the switches chosen for array placement are capable of atomic read-write operations on arrays indexed by one or more packet fields; switches that do not support persistent state can still play a role in routing flows efficiently through the appropriate state variables. To summarize, our main contributions are:

- A stateful and compositional SDN programming language with persistent global arrays, a one-big-switch programming model, and network transactions. (See §2 for an overview and §3 for more technical details).
- A set of algorithms for compiling SNAP programs into low-level switch mechanisms (§4):
  - An algorithm for compiling SNAP programs into an intermediate representation that detects program errors, such as race conditions introduced by parallel access to stateful components. This algorithm uses our extended forwarding decision diagrams (xFDD).
  - An algorithm to generate a mixed integer-linear program, based on the xFDD, which jointly decides array placement and routing while minimizing network congestion and satisfying the constraints necessary to support network transactions.
- An implementation and evaluation of our language and compiler using about 20 applications. (§5 and §6).

We discuss various data-plane implementations for SNAP's stateful operations, how it relates to middleboxes, and possible extensions in §7, and explore related work in §8.

2 System Overview

This section overviews the key concepts in SNAP's language and compilation process with a set of example programs.

2.1 Writing SNAP Programs

DNS tunnel detection. The DNS protocol is designed to resolve information about domain names. However, it is possible to use DNS messages as a tunnel to leak information since they are not typically intended for general data transfer. Detecting DNS tunnels is one of many real-world scenarios that require state to track the properties of network flows [5]. The following steps can be used to detect DNS tunneling:

1. For each client, keep track of the IP addresses resolved by DNS responses.

DNS-tunnel-detect

```
1  if dstip = 10.0.6.0/24 & srcport = 53 then
2    orphan[dstip][dns.rdata] <- True ;
3    susp-client[dstip]++;
4    if susp-client[dstip] = threshold then
5      blacklist[dstip] <- True
6    else id
7  else
8    if srcip = 10.0.6.0/24 & orphan[srcip][dstip]
9    then orphan[srcip][dstip] <- False;
10        susp-client[srcip]--
11   else id
```

Figure 1: SNAP implementation of `DNS-tunnel-detect`.

2. For each DNS response, increment a counter. This counter tracks the number of resolved IP addresses that a client does not use.
3. When a client sends a packet to a resolved IP address, decrement the counter for the client.
4. Report tunneling for clients that exceed a threshold for resolved, but unused IP addresses.

Figure 1 shows a SNAP implementation of the above steps that detects DNS tunnels in the CS department subnet 10.0.6.0/24 (see Figure 2). Intuitively, a SNAP programs can be thought of as a function that takes in a packet plus the current state of the network and produces a set of transformed packets as well as updated state. The incoming packet is read and written by referring to its fields (such as `dstip` and `dns.rdata`). The "state" of the network is read and written by referring to user-defined, array-based, global variables (such as `orphan` or `susp-client`). Before explaining the program in detail, note that it does not refer to specific network device(s) on which it is implemented. SNAP programs are expressed as if the network was *one-big-switch* (OBS) connecting edge ports directly to each other. Our compiler automatically distributes the program across network devices, freeing programmers from such details and making SNAP programs portable across topologies.

The `DNS-tunnel-detect` program examines two kinds of packets: incoming DNS responses (which may lead to possible DNS tunnels) and outgoing packets to resolved IP addresses. Line 1 checks whether the input packet is a DNS response to the CS department. The condition in the `if` statement is an example of a simple *test*. Such tests can involve any boolean combination of packet fields.[1] If the test succeeds, the packet could potentially belong to a DNS tunnel, and will go through the detection steps (Lines 2–6). Lines 2–6 use three global variables to keep track of DNS queries. Each variable is a mapping between keys and values, persistent across multiple packets. The `orphan` variable, for example, maps each pair of IP addresses to a boolean value. If `orphan[c][s]` is `True` then `c` has received a DNS response for IP address `s`. The variable `susp-client` maps the client's IP to the number of DNS responses it has received but not accessed yet. If the

[1]The design of the language is unaffected by the chosen set of fields. For the purposes of this paper, we assume a rich set of fields, e.g. DNS response data. New architectures such as P4 [6] have programmable parsers that allow users to customize their applications to the set of fields required.

packet is not a DNS response, a different test is performed, which includes a stateful test over `orphan` (Lines 8). If the test succeeds, the program updates `orphan[srcip][dstip]` to `False` and decrements `susp-client[srcip]` (Lines 10–11). This step changes the global state and thus, affects the processing of future packets. Otherwise, the packet is left unmodified — `id` (Line 12) is a no-op.

Routing. `DNS-tunnel-detect` cannot stand on its own—it does not explain where to forward packets. In SNAP, we can easily *compose* it with a forwarding policy. Suppose our target network is the simplified campus topology depicted in Figure 2. Here, I_1 and I_2 are connections to the Internet, and D_1–D_4 represent edge switches in the departments, with D_4 connected to the CS building. C_1–C_6 are core routers connecting the edges. External ports (marked in red) are numbered 1–6 and IP subnet `10.0.i.0/24` is attached to port `i`. The `assign-egress` program assigns outports to packets based on their destination IP address:

```
assign-egress = if dstip = 10.0.1.0/24
then outport <- 1
else if dstip = 10.0.2.0/24 then outport <- 2
else ...
else if dstip = 10.0.6.0/24 then outport <- 6
else drop
```

Note that the policy is independent of the internal network structure, and merely needs to be recompiled if the topology changes. By combining `DNS-tunnel-detect` with `assign-egress`, we have implemented a useful end-to-end network program: `DNS-tunnel-detect;assign-egress`

Monitoring. Suppose the operator wants to monitor packets entering the network at each ingress port (ports 1-6). She might use an array indexed by `inport` and increment the corresponding element on packet arrival: `count[inport]++`. Monitoring should take place *alongside* the rest of the program; thus, she might combine it using parallel composition (`+`): `(DNS-tunnel-detect + count[inport]++); assign-egress`. Intuitively, `p + q` makes a copy of the incoming packet and executes both `p` and `q` on it simultaneously.

Note that it is not always legal to compose two programs in parallel. For instance, if one writes to the same global variable that the other reads, there is a race condition, which leads to ambiguous state in the final program. Our compiler detects such race conditions and rejects ambiguous programs.

Network Transactions. Suppose that an operator sets up a honeypot at port 3 with IP subnet 10.0.3.0/25. The following program records, per inport, the IP and dstport of the last packet destined to the honeypot:

```
if dstip = 10.0.3.0/25
then hon-ip[inport] <- srcip;
     hon-dstport[inport] <- dstport
else id
```

Since this program processes many packets simultaneously, it has an implicit race condition: if packets p_1 and p_2, both destined to the honeypot, enter the network from port 1 and get reordered, each may visit `hon-ip` and `hon-dstport` in a different order (if the variables reside in different locations).

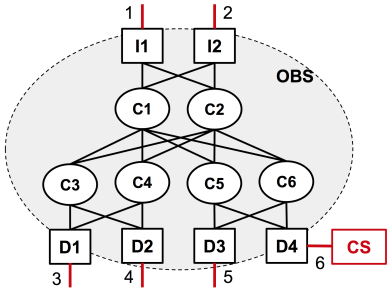


Figure 2: Topology for the running example.

Therefore, it is possible that `hon-ip[1]` contains the source IP of p_1 and `hon-dstport[1]` the destination IP of p_2 while the operator's intention was that both variables refer to the same packet. To establish such properties for a collection of state variables, programmers can use *network transactions* by simply enclosing a series of statements in an `atomic` block. Atomic blocks co-locate their enclosed state variables so that a series of updates can be made to appear atomic.

2.2 Realizing Programs on the Data Plane

Consider the program `DNS-tunnel-detect; assign-egress`. To distribute this program, defined on top of OBS, across network devices, the SNAP compiler should decide (i) where to place state variables (`orphan`, `susp-client`, and `blacklist`), and (ii) how packets should be routed across the physical network. These decisions should be made in such a way that each packet passes through devices storing *every* state variable it *needs*, in the correct *order*. Therefore, the compiler needs information about which packets need which state variables. In our example program, for instance, packets with `dstip = 10.0.6.0/24` and `srcport = 53` need to pass all three state variables, with `blacklist` accessed after the other two.

Program analysis. To extract the above information, we transform the program to an intermediate representation called *extended forwarding decision diagram (xFDD)* (see Figure 3). FDDs were originally introduced in an earlier work [33]. We extended FDDs in SNAP to support stateful packet processing. An xFDD is like a binary decision diagram: each intermediate node is a test on either packet fields or state variables. The leaf nodes are sets of action sequences, rather than merely 'true' and 'false' as in a BDD [1]. Each interior node has two successors: *true* (solid line), which determines the rest of the forwarding decision process for inputs passing the test, and *false* (dashed line) for failed cases. xFDDs are constructed compositionally; the xFDDs for different parts of the program are combined to construct the final xFDD. Composition is particularly more involved with stateful operations: the same state variable may be referenced in two xFDDs with different header fields, e.g., once as `s[srcip]` and then as `s[dstip]`. How can we know whether or not those fields are equal in the packet? We add a new kind of test, over pairs of packet fields (`srcip = dstip`), and new ordering requirements on the xFDD structure.

Once the program is transformed to an xFDD, we analyze the xFDD to extract information about which groups of pack-

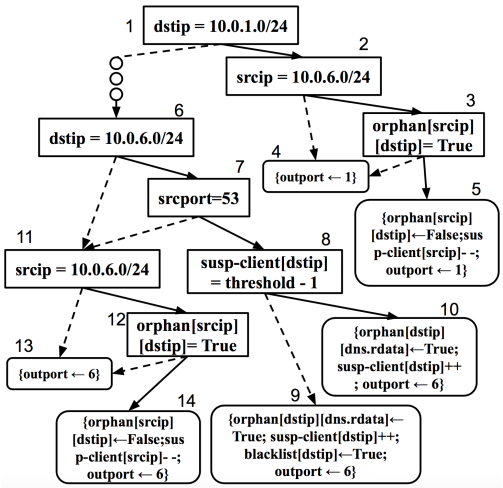


Figure 3: The equivalent xFDD for `DNS-tunnel-detect; assign-egress`

ets need which state variables. In Figure 3, for example, leaf number 10 is on the true branch of `dstip=10.0.6.0/24` and `srcport=53`, which indicates that all packets with this property may end up there. These packets need `orphan`, because it is modified, and `susp-client`, because it is both tested and modified on the path. We can also deduce these packets can enter the network from any port and the ones that are not dropped will exit port 6. Thus, we can aggregate state requirement information across OBS ports, and choose paths for traffic between these ports accordingly.

Joint placement and routing. At this stage, the compiler has the information it needs to distribute the program. It uses a mixed-integer linear program (MILP) that solves an extension of the multi-commodity flow problem to *jointly* decide state placement and routing while minimizing network congestion. The constraints in the MILP guarantee that the selected paths for each pair of OBS ports take corresponding packets through devices storing every state variable that they need, in the correct order. Note that the xFDD analysis can identify cases in which both directions of a connection need the same state variable s. Thus, the MILP constraints ensure that they both traverse the device holding s.

In our example program, the MILP places all state variables on D_4, which is the optimal location as all packets to and from the protected subnet must flow through D_4.[2] Note that this is not obvious from the `DNS-tunnel-detect` code alone, but rather from its *combination* with `assign-egress`. This highlights the fact that in SNAP, program components can be written in a modular way, while the compiler makes globally optimal decisions using information from all parts. The optimizer also decides forwarding paths between external ports. For instance, traffic from I_1 and D_1 will go through C_1 and C_5 to reach D_4. The path from I_2 and D_2 to D_4 goes through C_2 and C_6, and D_3 uses C_5 to reach D_4. The paths between the rest of the ports are also determined by the MILP in a way that minimizes link utilization. The compiler takes state placement and routing results from the MILP, partitions the program's intermediate representation (xFDD) among switches, and generates rules for all stateless and stateful switches in the network. The rules are then pushed to the data plane by the controller.

Reacting to network events. The above phases only run if the operator wishes to change the OBS program that is running on the network. Once the program compiles, and to respond to network events such as failures or traffic matrix shifts, we use a simpler and much faster version of the MILP that given the current state placement, only re-optimizes for routing. Moreover, with state on the data plane, policy changes become considerably *less frequent*. This is because the policy, and consequently switch configurations, *do not* change upon changes to state. In `DNS-tunnel-detect`, for instance, attack detection and mitigation are both captured in the program itself, happen *on the data plane*, and therefore react rapidly to malicious activities in the network. This is in contrast to the case where all the state is on the controller. There, the policy needs to change and recompile multiple times both during detection and on mitigation, to reflect the state changes on the controller in the rules on the data plane.

3 SNAP

SNAP is a high-level language with two key features: programs are *stateful* and are written in terms of an abstract network topology comprising a *one-big-switch* (OBS). It has an algebraic structure patterned on the NetCore/NetKAT family of languages [2, 20], with each program comprising one or more *predicates* and *policies*. SNAP's syntax is in Figure 4. Its semantics is defined through an (elided) evaluation function "eval." eval determines, in mathematical notation, how an input packet should be processed by a SNAP program. Note that this is part of the *specification* of the language, *not* the implementation. Any implementation of SNAP, including ours, should make sure that packets are processed as defined by the eval function: when we talk about "running" a program on a packet, we mean calling eval on that program and packet. We discuss eval's most interesting cases here; see the extended version for a full definition [40].

eval takes the SNAP term of interest, a packet, and a starting state and yields a set of packets and an output state. To properly define the semantics of multiple updates to state when programs are composed, we need to know the reads and writes to state variables performed by each program while evaluating the packet. Thus, eval also returns a *log* that contains this information. More specifically, eval adds "$R\,s$" to the log whenever a read from state variable s occurs, and "$W\,s$" on writes. Note that these logs are part of our formalism, but not our implementation. We express the program state as a dictionary that maps state variables to their contents. The contents of each state variable is itself a mapping from values to values. Values are defined as packet-related fields (IP address, TCP ports, MAC addresses, DNS domains) along with integers, booleans and vectors of such values.

[2] All the state need not go in the same place, it can be spread out across the network. It just so happens that in this case, one location turns out to be optimal.

$e \in \text{Expr}$	::=	$v \mid f \mid \vec{e}$	
$x, y \in \text{Pred}$	::=	id	Identity
	|	$drop$	Drop
	|	$f = v$	Test
	|	$\neg x$	Negation
	|	$x \mid y$	Disjunction
	|	$y \& x$	Conjunction
	|	$s[e] = e$	**State Test**
$p, q \in \text{Pol}$	::=	x	Filter
	|	$f \leftarrow v$	Modification
	|	$p + q$	Parallel comp.
	|	$p; q$	Sequential comp.
	|	$s[e] \leftarrow e$	**State Modification**
	|	$s[e]{+}{+}$	**Increment value**
	|	$s[e]{-}{-}$	**Decrement value**
	|	if a then p else q	**Conditional**
	|	$\text{atomic}(p)$	**Atomic blocks**

Figure 4: SNAP's syntax. Highlighted items are not in NetCore.

Predicates. Predicates have a constrained semantics: they will never update the state (but may read from it), and they either return the empty set or the singleton set containing the input packet. That is, predicates either pass or drop their input packets. id, passes the packet and $drop$, drops it. Given a packet pkt, the test $f = v$ passes the packet if the field f of pkt is v. Both predicates yield empty logs.

The novel predicate in SNAP is the *state test*, written $s[e_1] = e_2$ and read "state variable (array) s at index e_1 equals e_2". Here e_1 and e_2 are *expressions*, where an expression is either a value v (like an IP address or TCP port), a field f, or a vector of expressions $\vec{e}$. For $s[e_1] = e_2$, function eval evaluates the expressions e_1 and e_2 on the current packet to yield two values v_1 and v_2. It then checks whether the store says state variable s indexed at v_1 is equal to v_2. The packet can pass if they are equal, and is dropped otherwise. The returned log will include $R\,s$, to record that the predicate read from the state variable s.

We evaluate negation $\neg x$ by running eval on x and then complementing the result, propagating whatever log x produces. SNAP uses $\mid$ for disjunction and & for conjunction. $x \mid y$ unions the results of running x and y individually while doing the reads of both x and y. $x \& y$ intersects the results of running x and y while doing the reads of x and then y.

Policies. Policies can modify packets and the store. Every predicate is a policy—they simply make no modifications. Field modification $f \leftarrow v$ takes an input packet pkt and yields a new packet, pkt', such that $pkt'.f = v$ but otherwise pkt' is the same as pkt. State update $s[e_1] \leftarrow e_2$ passes the input packet through while (i) updating the store so that s at $\text{eval}(e_1)$ is set to $\text{eval}(e_2)$, and (ii) adding $W\,s$ to the log. The $s[e]{+}{+}$ (resp. $-{-}$) operators increment (decrement) the value of $s[e]$ and add $W\,s$ to the log.

Parallel composition $p + q$ runs p and q in parallel and tries to merge the results. If the logs indicate a state read/write or write/write conflict for p and q then there is no consistent semantics we can provide, and we leave the semantics undefined. Take for example $(s[0] \leftarrow 1) + (s[0] \leftarrow 2)$. The left branch updates $s[0]$ to 1, while the right branch updates it to 2. There is no good choice here, so we leave the semantics undefined and raise *compile error* in the implementation. On the other hand, $(s[0] \leftarrow 1) + (s'[0] \leftarrow 2)$ is fine if $s \neq s'$.

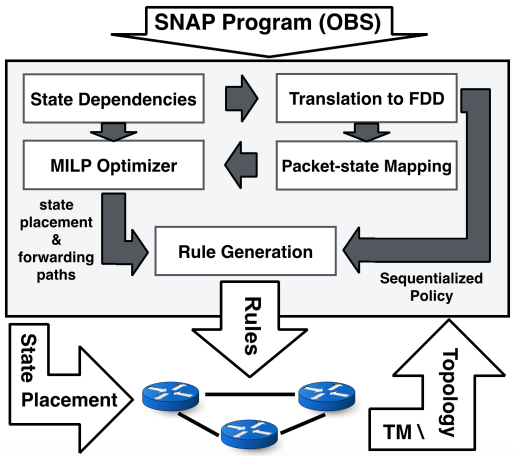


Figure 5: Overview of the compiler phases.

Sequential composition $p; q$ runs p and then runs q on each packet that p returned, merging the final results. We must ensure the runs of q are pairwise consistent, or else we will have a read/write or write/write conflict. For example, let p be $(f \leftarrow 1 + f \leftarrow 2)$. Assume that $pkt[f \mapsto v]$ means "update pkt's f field with v". Given a packet pkt, the policy p produces two packets: $pkt_1 = pkt[f \mapsto 1]$ and $pkt_2 = pkt[f \mapsto 2]$. Let q be $g \leftarrow 3$. If we run $p; q$ on pkt, then q must process each packet output by p, and merge the outputs. Since no state reads or writes occurred, this policy runs without exception, producing two packets: $pkt[f \mapsto 1, g \mapsto 3]$ and $pkt[f \mapsto 2, g \mapsto 3]$. However, if q' is $s[0] \leftarrow f$, running $p; q'$ fails because we cannot merge properly: running q' on pkt_1 updates $s[0]$ to be 1; running q on pkt_2 updates $s[0]$ to be 2.

We have an explicit conditional "if a then p else q," which indicates *either* p or q are executed. Hence, both p and q can perform reads and writes to the same state. Such conditionals can sometimes be *encoded* as $a; p + \neg a; q$. However, such encoding is not possible in cases like "if $s[0] = 0$ then $s[0] \leftarrow 1$ else $s[0] \leftarrow 0$," and that is why such conditionals are built into the language. Finally, we have a notation for *atomic blocks*, written $\text{atomic}(p)$. As described in §2, there is a risk of inconsistency between state variables residing on different switches on a real network, when many packets are in flight concurrently. When compiling $\text{atomic}(p)$, our compiler ensures that all the state in p is updated atomically (§4).

4 Compilation

To implement a SNAP program specified on the *one-big-switch* (OBS) abstraction, we must fill in two critical details: *traffic routing* and *state placement*. The physical topology may offer many paths between edge ports, and many possible locations for placing state.[3] The routing and placement problems interact: if two flows (with different input and output

[3] In this work, we assume that each state variable must reside in a single place, though it is conceivable that it could be distributed (see §4.4).

OBS ports) both need some state variable s, we would like to select routes for the two flows that pass through a common location where we place s. Further complicating the situation, the OBS program may specify that certain flows read or write multiple state variables in a particular order. The routing and placement on the physical topology must respect that order. In `DNS-tunnel-detect`, for instance, routing must ensure that packets reach wherever `orphan` is placed before `susp-client`. In some cases, two different flows may depend on the same state variables, but in different orders.

We have designed a *compiler* that translates OBS programs into forwarding rules and state placements for a given topology. As shown in Figure 5, the two key phases are (i) translation to *extended forwarding decision diagrams* (xFDDs)—used as the intermediate representation of the program and to calculate which flows need which state variables—and (ii) optimization via *mixed integer linear program* (MILP)—used to decide routing and state placement. In the rest of this section, we present the compilation process in phases, first discussing the analysis of state dependencies, followed by the translation to xFDDs and the packet-state mapping, then the optimization problems, and finally the generation of rules sent to the switches.

4.1 State Dependency Analysis

Given a program, the compiler first performs *state dependency analysis* to determine the ordering constraints on its state variables. A state variable t *depends* on a state variable s if the program writes to t after reading from s. Any realization of the program on a concrete network must ensure that t does not come before s. Parallel composition $p+q$, introduces no dependencies: if p reads or writes state, then q can run independently of that. Sequential composition $p;q$, on the other hand, introduces dependencies: whatever reads are in p must happen before writes in q. In explicit conditionals "if a then p else q", the writes in p and q depend on the condition a. Finally, atomic sections $\mathsf{atomic}(p)$ say that all state in p is inter-dependent. In `DNS-tunnel-detect`, for instance, `blacklist` is dependent on `susp-client`, itself dependent on `orphan`. This information is encoded as a dependency graph on state variables and is used to order the xFDD structure (§4.2), and in the MILP (§4.4) to drive state placement.

4.2 Extended Forwarding Decision Diagrams

We use an extended forwarding decision diagrams (xFDDs) as our internal representation for the OBS program. Formally (see Figure 6), an xFDD is either a *branch* $(t\ ?\ d_1 : d_2)$, where t is a test and d_1 and d_2 are xFDDs, or a *set* of action sequences $\{as_1,\ldots,as_n\}$. Each branch can be thought of as a conditional: if the test t holds on a given packet pkt, then the xFDD continues processing pkt using d_1; if not, processes pkt using d_2. There are three kinds of tests. The *field-value test* $f = v$ holds when $pkt.f$ is equal to v. The *field-field test* $f_1 = f_2$ holds when the values in $pkt.f_1$ and $pkt.f_2$ are equal. Finally, the *state test* $s[e_1] = e_2$ holds when the state variable s at index e_1 is equal to e_2. The last two tests are our extensions

$$
\begin{array}{rcll}
d & ::= & t\ ?\ d_1 : d_2 \mid \{as_1,\ldots,as_n\} & \text{xFDDs} \\
t & ::= & f = v \mid f_1 = f_2 \mid s[e_1] = e_2 & \text{tests} \\
as & ::= & a \mid a;a & \text{action sequences} \\
a & ::= & id \mid drop \mid f \leftarrow v \mid s[e_1] \leftarrow e_2 & \text{actions} \\
 & & \mid s[e_1]{+}{+} \mid s[e_1]{-}{-} &
\end{array}
$$

$$
\begin{array}{rcl}
\text{TO-XFDD}(a) & = & \{a\} \\
\text{TO-XFDD}(f = v) & = & f = v\ ?\ \{id\}\ :\ \{drop\} \\
\text{TO-XFDD}(\neg x) & = & \ominus\text{TO-XFDD}(x) \\
\text{TO-XFDD}(s[e_1] = e_2) & = & s[e_1] = e_2\ ?\ \{id\}\ :\ \{drop\} \\
\text{TO-XFDD}(\mathsf{atomic}(p)) & = & \text{TO-XFDD}(p) \\
\text{TO-XFDD}(p+q) & = & \text{TO-XFDD}(p) \oplus \text{TO-XFDD}(q) \\
\text{TO-XFDD}(p;q) & = & \text{TO-XFDD}(p) \odot \text{TO-XFDD}(q) \\
\text{TO-XFDD}(\text{if } x \text{ then } p \text{ else } q) & = & (\text{TO-XFDD}(x) \odot \text{TO-XFDD}(p)) \\
 & & \oplus\ (\ominus\text{TO-XFDD}(x) \odot \text{TO-XFDD}(q))
\end{array}
$$

Figure 6: xFDD syntax and translation.

to FDDs. The state tests support our stateful primitives, and as we will see later in this section, the field-field tests are required for their correct compilation. Each leaf in an xFDD is a set of action sequences, with each action being either the identity; drop; field-update $f \leftarrow v$; or state update $s[e_1] \leftarrow e_2$, which is another extension to the original FDD.

A key property of xFDDs is that the order of their tests ($\sqsubset$) must be defined in advance. This ordering ensures that each test is present at most once on any path in the final tree when merging xFDDs. In our xFDDs, we ensure that all field-value tests precede all field-field tests, themselves preceding all state tests. Field-value tests themselves are ordered by fixing an arbitrary order on fields and values. Field-field tests are ordered similarly. For state tests, we first define a total order on state variables by looking at the dependency graph from §4.1. We break it into strongly connected components (SCCs) and fix an arbitrary order on state variables within each SCC. Then for every edge from one SCC to another—i.e., where some state variable in the first SCC depends on some state variable in the second—s_1 precedes s_2 in the order, where s_2 is the minimal element in the second SCC and s_1 is the maximal element in the first SCC. The state tests are then ordered based on the order of state variables.

We translate a program to an xFDD using the TO-XFDD function (Figure 6), which translates small parts of program directly to xFDDs. Composite programs get recursively translated and then composed using a corresponding normalization operator: we use $\oplus$ for $p+q$, $\odot$ for $p\ ;\ q$, and $\ominus$ for $\neg p$. Figure 7 gives a high-level definition of the semantics of these operators. For example, $d_1 \oplus d_2$ tries to merge similar test nodes recursively by merging their true branches together and false ones together. If the two tests are not the same and d_1's test comes first in the total order, both of its subtrees are merged recursively with d_2. The other case is similar. $d_1 \oplus d_2$ for leaf nodes is the union of their action sets.

The hardest case is surely for $\odot$, where we try to add in an action sequence as to an xFDD $(t\ ?\ d_1 : d_2)$. Suppose we want to compose $f \leftarrow v_1$ with $(f = v_2\ ?\ d_1 : d_2)$. The result of this xFDD composition should behave as if we first do the update and then the condition on f. If $v_1 = v_2$, the composition should continue only on d_1, and if not, only on

$$\{as_{11},\cdots,as_{1n}\}\oplus\{as_{21},\cdots,as_{2m}\}=\{as_{11},\cdots,as_{1n}\}\cup\{as_{21},\cdots,as_{2m}\}$$
$$(t\ ?\ d_1:d_2)\oplus\{as_1,\cdots,as_n\}=(t\ ?\ d_1\oplus\{as_1,\cdots,as_n\}:d_2\oplus\{as_1,\cdots,as_n\})$$

$$(t_1\ ?\ d_{11}:d_{12})\oplus(t_2\ ?\ d_{21}:d_{22})=\begin{cases}(t_1\ ?\ d_{11}\oplus d_{21}:d_{12}\oplus d_{22}) & t_1=t_2\\ (t_1\ ?\ d_{11}\oplus(t_2\ ?\ d_{21}:d_{22}):d_{12}\oplus(t_2\ ?\ d_{21}:d_{22}) & t_1\sqsubset t_2\\ (t_2\ ?\ d_{21}\oplus(t_1\ ?\ d_{11}:d_{12}):d_{22}\oplus(t_1\ ?\ d_{11}:d_{12}) & t_2\sqsubset t_1\end{cases}$$

$$\ominus\{id\}=\{drop\}$$
$$\ominus\{drop\}=\{id\}$$
$$\ominus(t\ ?\ d_1\ :\ d_2)=(t\ ?\ \ominus d_1\ :\ \ominus d_2)$$

$$\begin{aligned}as\odot\{as_1,\cdots,as_n\}&=\{as\odot as_1,\cdots,as\odot as_n\}\\ as\odot(t\ ?\ d_1:d_2)&=\text{(see explanations in §4.2)}\\ \{as_1,\cdots,as_n\}\odot d&=(as_1\odot d)\oplus\cdots\oplus(as_n\odot d)\\ (t\ ?\ d_1:d_2)\odot d&=(d_1\odot d)|_t\oplus(d_2\odot d)|_{\sim t}\end{aligned}$$

$$\{as_1,\cdots,as_n\}|_t=(t\ ?\ \{as_1,\cdots,as_n\}:\{drop\})$$

$$(t_1\ ?\ d_1:d_2)|_{t_2}=\begin{cases}(t_1\ ?\ d_1:\{drop\}) & t_1=t_2\\ (t_2\ ?\ (t_1\ ?\ d_1:d_2):\{drop\}) & t_2\sqsubset t_1\\ (t_1\ ?\ d_1|_{t_2}:d_2|_{t_2}) & t_1\sqsubset t_2\end{cases}$$

Figure 7: Definitions of xFDD composition operators.

d_2. Now let's look at a similar example including state, composing $s[srcip] \leftarrow e_1$ with $(s[dstip] = e_2\ ?\ d_1 : d_2)$. If *srcip* and *dstip* are equal (rare but not impossible) and e_1 and e_2 always evaluate to the same value, then the whole composition reduces to just d_1. The field-field tests are introduced to let us answer these equality questions, and that is why they always precede state tests in the tree. The trickiness in the algorithm comes from generating proper field-field tests, by keeping track of the information in the xFDD, to clear out these cases. The full algorithm is given the extended version [40].

Recall from §3 that inconsistent use of state variables is prohibited by the language semantics when composing programs. We enforce the semantics by looking for these violations while merging the xFDDs of composed programs and raising a compile error if the final xFDD contains a leaf with parallel updates to the same state variable.

4.3 Packet-State Mapping

For a given program p, the corresponding xFDD d offers an explicit and complete specification of the way p handles packets. We analyze d, using an algorithm called *packet-state mapping*, to determine which *flows* use which states. This information is further used in the optimization problem (§4.4) to decide the correct routing for each flow. Our default definition of a flow is those packets that travel between any given pair of ingress/egress ports in the OBS, though we can use other notions of flow (see §4.4). Traversing from d's root down to the action sets at d's leaves, we can gather information associating each flow with the set of state variables read or written. We omit the full algorithm due to space constraints.

Furthermore, the operators can give hints to the compiler by specifying their network *assumptions* in a separate policy:

```
assumption = (srcip = 10.0.1.0/24 & inport = 1)
+ (srcip = 10.0.2.0/24 & inport = 2) + ...
+ (srcip = 10.0.6.0/24 & inport = 6)
```

We require the assumption policy to be a predicate over packet header fields, only passing the packets that match the operator's assumptions. `assumption` is then sequentially composed with the rest of the program, enforcing the assumption by dropping packets that do not match the assumption. Such assumptions benefit the packet-state mapping. Consider our example xFDD in Figure 3. Following the xFDD's tree structure, we can infer that all the packets going to port 6 need all the three state variables in `DNS-tunnel-detect`. We can also infer that all the packets coming from the 10.0.6.0/24 subnet need `orphan` and `susp-client`. However, there is nothing in the program to tell the compiler that these packets can only enter the network from port 6. Thus, the above assumption policy can help the compiler to identify this relation and place state more efficiently.

Variable	Description
u, v	edge nodes (ports in OBS)
n	physical switches in the network
i, j	all nodes in the network
d_{uv}	traffic demand between u and v
c_{ij}	link capacity between i and j
dep	state dependencies
$tied$	co-location dependencies
S_{uv}	state needed for flow uv
R_{uvij}	fraction of d_{uv} on link (i, j)
P_{sn}	1 if state s is placed on n, 0 otherwise
P_{suvij}	d_{uv} fraction on link i, j that has passed s

Table 1: Inputs and outputs of the optimization problem.

4.4 State Placement and Routing

To decide state placement and routing, we generate an optimization problem that takes the form of a *mixed-integer linear program* (MILP), and is an extension of the multi-commodity flow problem. The MILP has three key inputs: the concrete network topology, the state dependency graph G, and the packet-state mapping, and two key outputs: routing and state placement (Table 1). The objective is to minimize the sum of link utilization (or a convex function of it) in the network.

Inputs. The topology is defined in terms of the following inputs to the MILP: (i) the nodes, some distinguished as edges (ports in OBS), (ii) expected traffic d_{uv} for every pair of edge nodes u and v, and (iii) link capacities c_{ij} for every pair of nodes i and j. State dependencies in G are translated into input sets dep and $tied$. $tied$ contains pairs of state variables which are in the same SCC in G, and must be co-located. dep identifies state variables with dependencies that do not need to be co-located; in particular, $(s,t) \in dep$ when s precedes t in variable ordering, and they are not in the same SCC in G. The packet-state mapping is used as the input variables S_{uv}, identifying the set of state variables needed on flows between nodes u and v.

Outputs and Constraints. The routing outputs are variables R_{uvij}, indicating which fraction of the flow from edge node u to v should traverse the link between nodes i and j. The constraints on R_{uvij} (left side of Table 2) follow the multi-commodity flow problem closely, with standard link capacity and flow conservation constraints, and edge nodes distinguished as sources and sinks of traffic.

State placement is determined by the variables P_{sn}, which indicate whether the state variable s should be placed on the physical switch n. Our constraints here are more unique to our setting. First, every state variable s can be placed on exactly one switch, a choice we discuss at the end of this section. Second, we must ensure that flows that need a given state variable s traverse that switch. Third, we must ensure that each flow traverses states in the order specified by the dep relation; this is what the variables P_{suvij} are for. We require that $P_{suvij} = R_{uvij}$ when the traffic from u to v that goes over the link (i, j) has already passed the switch with the state variable s, and zero otherwise. If dep requires that s should come before some other state variable t—and if the (u, v) flow needs both s and t—we can use P_{suvij} to make sure that the (u, v) flow traverses the switch with t only after it has traversed the switch with s (the last state constraint in Table 2). Finally, we must make sure that state variables $(s,t) \in tied$ are located on the same switch.

The MILP can be configured to decide paths for more fine-grained notions of flows. Suppose packet-state mapping finds that only packets with $srcip = x$ need state variable s. We refine the MILP input to have two edge nodes per port, one for traffic with $srcip = x$ that needs s, and one for the rest, to enable the MILP choosing paths for them separately. Moreover, the MILP forces each state variable s to be placed on *one* physical switch. This way, we can avoid the overhead of synchronizing multiple instances of the same variable in different locations. Besides, most network topologies are hierarchical and have core switches that can serve as good candidates for placing state variables common across multiple flows. Still, distributing a state variable remains a valid option. For instance, the compiler can partition $s[inport]$ into k *disjoint* state variables, each storing s for one port. The MILP can decide placement and routing as before, this time with the option of distributing partitions of s with no concern for synchronization.

4.5 Generating Data-Plane Rules

Rule generation happens in two phases, combining information from the xFDD and MILP to configure the switches in the network. In generating the configurations, we assume each packet is augmented with a SNAP-header upon entering the network, which contains its original OBS inport and future outport, and the id of the last processed xFDD node, the purpose of which will be explained shortly. This header is stripped off when the packet exits the network. We use `DNS-tunnel-detect` from §2 as a running example, with its xFDD in Figure 3. For exemplary clarity, we assume that all the state variables are stored on C_6 instead of D_4.

Routing Constraints	State Constraints
	$\sum_n P_{sn} = 1$
$\sum_j R_{uvuj} = 1$	$\forall u, v.\ \forall s \in S_{uv}.\ \sum_i R_{uvin} \geq P_{sn}$
$\sum_i R_{uviv} = 1$	$\forall (s,t) \in tied.\ P_{sn} = P_{tn}$
$\sum_{u,v} R_{uvij} d_{uv} \leq c_{ij}$	$P_{suvij} \leq R_{uvij}$
$\sum_i R_{uvin} = \sum_j R_{uvnj}$	$P_{sn} + \Sigma_i P_{suvin} = \Sigma_j P_{suvnj}$
$\sum_i R_{uvin} \leq 1$	$\forall s \in S_{uv}.P_{sv} + \sum_i P_{suviv} = 1$
	$P_{sn} + \Sigma_i P_{suvin} \geq P_{tn}$

Table 2: Constraints of the optimization problem.

In the first phase, we break the xFDD down into 'per switch' xFDDs, since not every switch needs the entire xFDD to process packets. Splitting the xFDD is straightforward, given placement information: stateless tests and actions can happen anywhere, but reads and writes of states have to happen on switches storing them. For example, edge switches (I_1 and I_2, and D_1 to D_4) only need to process packets up to the state tests, e.g., tests 3 and 8, and write the test number in the packet's SNAP-header showing how far into the xFDD they made progress. Then, they send the packets to C_6, which has the corresponding state variables, `orphan` and `susp-client`. C_6, on the other hand, does not need the top part of the xFDD. It just needs the subtrees containing its state variables to continue processing the packets sent to it from the edges. The per-switch xFDDs are then translated to switch-level configurations, by a straightforward traversal of the xFDD.[4]

In the second phase, we generate a set of match-action rules that take packets through the paths decided by the MILP. These paths comply with the state ordering also used in the xFDD, thus they will get packets to switches with the right states in the right order. Note that packets contain the path identifier (the OBS inport and outport, (u, v) pair in this case) and the "routing" match-action rules are generated in terms of this identifier to forward them on the correct path. Additionally, note that it may not always be possible to decide the egress port v for a packet upon entry if its outport depends on state. We make the observation that in that case, all the paths for possible outports of the packet pass the state variables it needs. We pick one of these paths in proportion to their capacity and show, in the extended version [40], that traffic on these paths remains in their capacity limit.

As an example of packets being handled by generated rules, consider a DNS response with source IP 10.0.1.1 and destination IP 10.0.6.6, entering the network from port 1. The rules on I_1 process the packet up to test 8 in the xFDD, tag the packet with the path identifier (1, 6) and number 8. The packet is then sent to C_6. There, C_6 will process the packet from test 8, update state variables accordingly, and send the packet to D_4 to exit the network from port 6.

5 Implementation

The compiler is mostly implemented in Python, except for the state placement and routing phase (§4.4) in which we use the Gurobi Optimizer [15] to solve the MILP. The compiler's output for each switch is a set of switch-level instructions

[4] See §5 for the details of implementation of stateful operations.

in a low-level language called NetASM [31], which comes with a software switch capable of executing those instructions. NetASM is an assembly language for programmable data planes designed to serve as the "narrow waist" between high-level languages such as SNAP, and NetCore [20], and programmable switching architectures such as RMT [7], FPGAs, network processors and Open vSwitch.

As described in §4.5, each switch processes the packet by its customized per-switch xFDD, and then forwards it based on the fields of SNAP-header using a match-action table. To translate the switch's xFDD to NetASM instructions, we traverse the xFDD and generate a *branch* instruction for each test node, which jumps to the instruction of either the true or false branch based on the test's result. Moreover, we generate instructions to create two tables for each state variable, one for the indices and one for the values. In the case of a state test in the xFDD, we first retrieve the value corresponding to the index that matches the packet, and then perform the branch. For xFDD leaf nodes, we generate *store* instructions that modify the packet fields and state tables accordingly. Finally, we use NetASM support for atomic execution of multiple instructions to guarantee that operations on state tables happen atomically.

While NetASM was a proper tool to test our compiler, any programmable device that supports match-action tables, the branch instruction, and stateful operations can be used as a target for SNAP. The prioritized rules in match-action tables, for instance, are effectively branch instructions. Thus, one can use multiple match-action tables to implement xFDD in the data plane, generating a separate rule for each path in the xFDD. Several emerging switch interfaces support stateful operations [4, 6, 24, 36]. We discuss possible software and hardware implementations for SNAP stateful operations in §7.

6 Evaluation

This section evaluates SNAP in terms of language expressiveness and compiler performance.

6.1 Language Expressiveness

We used SNAP to create several stateful network functions that are typically delegated to middleboxes. Table 3 lists applications we implemented, drawing examples from the Chimera [5], FAST [21], and Bohatei [8] systems. The code can be found in the extended version [40]. Most examples use protocol-related fields in fixed packet-offset locations, which are parsable by emerging programmable parsers. Some fields used in Sidejack detection and Snort policies require session reassembly. However, this is orthogonal to the language expressiveness. As long as these fields are available to the tables in the forwarding plane, they can be used in SNAP programs and our compiler can generate suitable switch configurations. To make them available to the switch, one could extract these fields in the data plane by placing a "preprocessor" before the switch pipeline, similar to middleboxes. For instance, Snort [34] uses a sequence of preprocessors prior to the detection engine, which can use the fields extracted by the preprocessors.

	Application
Chimera [5]	# domains sharing the same IP address
	# distinct IP addresses under the same domain
	DNS TTL change tracking
	DNS tunnel detection
	Sidejack detection
	Phishing/spam detection
FAST [21]	Stateful firewall
	FTP monitoring
	Heavy-hitter detection
	Super-spreader detection
	Sampling based on flow size
	Selective packet dropping (MPEG frames)
	Connection affinity in LB
Bohatei [8]	SYN flood detection
	DNS amplification mitigation
	UDP flood mitigation
	Elephant flows detection
Others	Bump-on-the-wire TCP state machine
	Snort flowbits [34]

Table 3: Applications written in SNAP.

6.2 Compiler Performance

The compiler has several phases upon the system's cold start, yet most events require only part of them. Table 4 summarizes these phases and their sensitivity to network and policy changes. Our report of results is organized accordingly:

Cold Start. When the very first program is compiled, the compiler goes through all phases, including MILP model creation, which happens *only once*. Once created, the model supports incremental additions and modifications of variables and constraints in a few milliseconds.

Policy Changes. Compiling a *new* program goes through the three program analysis phases and rule generation as well as *both* state placement and routing, which are decided using the MILP in §4.4, denoted by "ST". Note that policy changes become considerably *less frequent* as compared to cases where state lives on the controller (§2.2) since most dynamic changes are captured by the state variables that reside on the data plane. The policy, and consequently switch configurations, *do not* change upon state changes. Thus, we expect policy changes to happen infrequently, and be planned in advance. The Snort rule set, for instance, gets updated every few days [35].

Topology/TM Changes. Once the policy is compiled, we fix the decided state placement, and only re-optimize routing in response to network events such as failures or shifts in the traffic matrix. For that, we formulated a variant of ST, denoted as "TE" (traffic engineering), that receives state placement as input, and decides forwarding paths while satisfying state requirement constraints. We expect TE to run every few minutes since in a typical network, the traffic matrix is fairly stable and traffic engineering happens on the timescale of *minutes* or more [16, 22, 39, 41].

ID	Phase		Topology or TM Change	Policy Change	Cold Start
P1	State dependency		-	✓	✓
P2	xFDD generation		-	✓	✓
P3	Packet-state map		-	✓	✓
P4	MILP creation		-	-	✓
P5	MILP solving	State placement and routing (ST)	-	✓	✓
		Routing (TE)	✓	-	-
P6	Rule generation		✓	✓	✓

Table 4: Compiler phases and their sensitivity to different events. For each scenario, phases that get executed are checkmarked.

Topology	# Switches	# Edges	# Demands
Stanford	26	92	20736
Berkeley	25	96	34225
Purdue	98	232	24336
AS 1755	87	322	3600
AS 1221	104	302	5184
AS 6461	138	744	9216
AS 3257	161	656	12544

Table 5: Statistics of evaluated enterprise/ISP topologies.

	P1-P2-P3 (s)	P5 (s)		P6(s)	P4 (s)
		ST	TE		
Stanford	1.1	29	10	0.1	75
Berkeley	1.5	47	18	0.1	150
Purdue	1.2	67	27	0.1	169
AS 1755	0.6	19	6	0.04	22
AS 1221	0.7	21	7	0.04	32
AS 6461	0.8	116	47	0.1	120
AS 3257	0.9	142	74	0.2	163

Table 6: Runtime of compiler phases when compiling `DNS-tunnel-detect` with routing on enterprise/ISP topologies.

6.2.1 Experiments

We evaluated the performance using the applications listed in Table 3. Traffic matrices are synthesized using a gravity model [30]. We used an Intel Xeon E3 server with 3.4 GHz CPU and 32GB memory, and PyPy compiler [26]. Moreover, we report the maximal rule-generation time per switch.

Enterprise/ISP topologies. We used a set of three campus networks and four inferred ISP topologies from RocketFuel [38] (see Table 5).[5] For ISP networks, we considered 70% of the switches with the lowest degrees as edge switches to form OBS external ports. The "# Demands" column shows the number of distinct OBS ingress/egress pairs. We assume directed links. Table 6 shows the compilation time for the DNS tunneling example (§2) on each network, broken down by compiler phase. Figure 8a compares the compiler runtime for different scenarios, combining the runtimes of phases relevant for each.

Scaling with topology size. We synthesize networks with 10–180 switches using IGen [27]. In each network, 70% of the switches with the lowest degrees are chosen as edges and the DNS tunnel policy is compiled with that network as a target. Figure 8b shows the compilation time for different scenarios, combining the runtimes of phases relevant for each. Note that by increasing the topology size, the policy size also increases in the `assign-egress` and `assumption` parts.

Scaling with number of policies. We use the network from the previous experiment with 50 switches and gradually increase the number of policies by parallel composing those from Table 3, each destined to a separate egress port. The policies are mostly similar in complexity to the DNS tunnel example (§2). Figure 8c depicts the compilation time as a function of the number of policies. The 10-second jump from 18 to 19 takes place when the TCP state machine policy is added, which is considerably more complex than others.

6.2.2 Analysis of Experimental Results

The experiments allow us to make several conclusions:

Creating the MILP takes longer than solving it, in most cases, and much longer than other phases. Fortunately, this is a *one-time* cost. After creating the MILP instance, incrementally adding or removing variables and constraints (as the topology or state requirement changes) takes just a few milliseconds, even for large topologies.

Solving the ST MILP unsurprisingly takes longer as compared to the rest of the phases when topology grows. It takes around 2.5 minutes for the biggest synthesized topology and around 2.3 minutes for the biggest RocketFuel topology, AS 3257. The curve is close to exponential as the problem is inherently computationally hard. However, this phase takes place only in cold start or upon a *policy* change, which as mentioned earlier, are infrequent and planned in advance.

Re-optimizing routing with fixed state placement is much faster. In response to network events (e.g., link failures or traffic-matrix shifts), the TE MILP can recompute paths in around a minute across all our experiments, which is the timescale we expected for this phase. Moreover, TE MILP can be used even on *policy* changes, if the user settles for a sub-optimal state placement using heuristics rather than ST MILP. We plan to explore such heuristics for state placement.

Given the kinds of events that require complete (policy change) or partial (network events) recompilation, we believe that our compilation techniques meet the requirements of enterprise networks and medium-size ISPs, such as the ones in Table 5. Moreover, if needed, our compilation procedure could be combined with common traffic-engineering techniques once the state placement is decided, to avoid resolving the original or even TE MILP on small timescales.

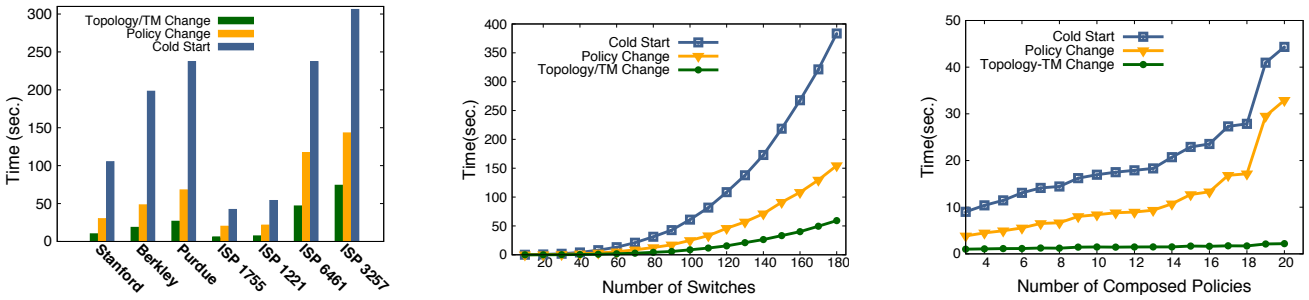
7 Discussion

This section discusses data-plane implementation strategies for SNAP's stateful operations, how it relates to middleboxes, and possible extensions to our techniques that enable support for a broader range of applications.

7.1 Stateful Operations in the Data Plane.

A state variable (array) in SNAP is a key-value mapping, or a *dictionary*, on header fields, which is persistent across multiple packets. When the key (index) range is small, it is feasible to pre-allocate all the memory the dictionary needs and im-

[5] The publicly available Mininet instance of Stanford campus topology has 10 extra dummy switches to implement multiple links between two routers.



(a) Compilation time of `DNS-tunnel-detect` with routing on enterprise/ISP networks.

(b) Compilation time of `DNS-tunnel-detect` with routing on IGen topologies.

(c) Compilation time for policies from Table 3 incrementally composed on a 50-switch network.

Figure 8: Compiler runtimes for scenarios in Table 4 on various policies and topologies. Once a policy is compiled for the first time (cold start , policy change), it reacts to traffic using its state variables on the data plane. Topology/TM changes result in reoptimizing forwarding paths.

plement it using an array. A large but *sparse* dictionary can be implemented using a *reactively*-populated table, similar to a MAC learner table. It contains a single default entry in the beginning, and as packets fly by and change the state variable, it *reactively* adds/updates the corresponding entries.

In software, there are efficient techniques to implement a dictionary in either approach, and some software switches already support similar reactive "learning" operations, either atomically [31] or with small periods of inconsistency [24]. The options for current hardware are: (i) arrays of registers, which are already supported in emerging switch interfaces [6]. They can be used to implement small dictionaries, as well as hash tables and Bloom Filters as sparse dictionaries. In the latter case, it is possible for two different keys to hash to the same dictionary entry. However, there are applications such as load balancing and flow-size-based sampling that can tolerate such collisions [21]. (ii) Content Addressable Memories (CAMs) are typically present in today's hardware switches and can be modified by a software agent running on the switch. Since CAM updates triggered by a packet are not immediately available to the following packets, it may be used for applications that tolerate small periods of state inconsistency, such as a MAC learner, DNS tunnel detection, and others from Table 3. Our NetASM implementation (§5) takes the CAM-based approach. NetASM's software switch supports atomic update to the tables in the data plane and therefore can perform *consistent* stateful operations.

In summery, any software/hardware switch with similar capabilities as described above, can implement all or a useful subset of SNAP's stateful operations and therefore can fit as SNAP's data-plane target for switches that carry state.

7.2 SNAP and Middleboxes

Networks traditionally rely on middleboxes for advanced packet processing, including stateful functionalities. However, current advances in switch technology enables stateful packet processing in the data plane, which naturally makes the switches capable of subsuming a subset of middlebox functionality. SNAP provides a *high-level programming framework* to exploit this ability, hence, it is able to express a wide range of stateful programs that are typically relegated to middleboxes (see Table 3 for examples). This helps the programmer to think about a single, explicit network policy, as opposed to a disaggregated, implicit network policy using middleboxes, and therefore, get more control and customization over a variety of simpler stateful functionalities.

This also makes SNAP subject to similar challenges as managing stateful middleboxes. For example, many network functions must observe all traffic pertaining to a connection *in both directions*. In SNAP, bi-directional analyses are straightforward: If traffic in both directions causes a shared stateful variable to be updated, the MILP optimizer will force all traffic in both directions through the same node If the two directions do not interact through shared state, they may be routed along different paths. State migration is another middlebox challenge. Here, previous work such as Split/Merge [28] and OpenNF [13] show how to migrate *internal* state from one network function to another, and Gember-Jacobson et al. [12] manage to migrate state without buffering packets at the controller. SNAP currently focuses on static state placement. However, because SNAP's state variables are explicitly declared as part of the policy, rather than hidden inside blackbox software, SNAP is well situated to adopt these algorithms to support smooth transitions of state variables in dynamic state placement. Additionally, the SNAP compiler can easily analyze a program to determine whether a switch modifies packet fields to ensure correct traffic steering—something that is challenging today with blackbox middleboxes [9, 17].

While SNAP goes a step beyond previous high-level languages to incorporate stateful programming into SDN, we neither claim it is as expressive as all stateful middleboxes, nor that it can replace them. To interact with middleboxes, SNAP may adopt techniques such as FlowTags [17] or SIMPLE [9] to direct traffic through middlebox chains by tagging the packets to mark their progress. This paper focuses on *programming* networks rather than *verifying* their properties, but if verification is of interest in future work, one might adopt techniques such as RONO [23] to verify isolation properties in presence of stateful middleboxes. In summary, interacting with existing middleboxes is no harder or easier in SNAP than it is in other global SDN languages such as NetKAT [2].

7.3 Extending SNAP

Modifying fields with state variables. An interesting extension to SNAP is the following action, `f <- s[e]`, allowing a field to be directly modified with the value of a state variable at a specific index. This action can be used to express applications such as NATs and proxies, which can store connection mappings in state variables and modify packets accordingly as they fly by. Moreover, it enables SNAP programs to modify a field by the output of an arbitrary function on a set of packet fields, such as a hash function. Such a function is nothing but a fixed mapping between input header fields and output values. Thus, when analyzing the program, the compiler can treat these functions as fixed state variables with the function's input fields as index for the state variable and place them on switches with proper capabilities when distributing the program across the network. However, adding this action results in complicated statement dependencies, which we will explore as future work.

Deep packet inspection (DPI). Several applications such as intrusion detection (IDS) require searching the packet's payload for specific patterns. From a linguistic point of view, SNAP can be extended rather easily with an extra field called *content*, containing the packet's payload. Moreover, the semantics of tests on the content field can be extended to match on regular expressions. The compiler can also be modified to assign content tests to switches that have DPI capabilities.

Dynamic dependencies. SNAP identifies dependencies between state variables and flows statically, at compile time, and chooses forwarding paths accordingly. Consequently, if a flow may depend upon a particular variable x in the future (because a particular state change *may* happen), it will be routed through the node containing x. This observation generates interesting opportunities for optimizations, which dynamically shift routes as state changes. We plan to explore such optimization in the context of *dynamic* service chaining.

Cross-packet fields. There are several protocols for which packet fields may be scattered among multiple packets. Middleboxes such as IDS perform session reconstruction to extract these fields. Although SNAP's language is agnostic to the chosen set of fields, the compiler currently supports fields stored *in the packet itself* and the state associated with them. However, it is interesting to explore abstractions for expressing how multiple packets (e.g. in a session) can form "one big packet" and use its fields. The compiler can further place sub-programs that use cross-packet fields on devices that are capable of reconstructing the "one big packet".

Queue-based policies. SNAP currently has no notion of queues and therefore, cannot be used to express queue-based performance-oriented policies such as active queue management, queue-based load balancing, and packet scheduling. There is ongoing research on finding the right set of primitives for expressing such policies [32], which is largely orthogonal and complementary to SNAP's current goals.

8 Related Work

Stateful languages. Stateful NetKAT [19], developed concurrently with SNAP, is a stateful language for "event-driven" network programming, which guarantees consistent update when transitioning between configurations in response to events. SNAP source language is richer and exponentially more compact than stateful NetKAT as it contains multiple *arrays* (as opposed to one) that can be indexed and updated by contents of *packet headers* (as opposed to constant integers only). Moreover, they place multiple copies of state at the edge, proactively generate rules for all configurations, and optimize for rule space, while we distribute state and optimize for congestion. Kinetic [18] provides a per-flow state machine abstraction, and NetEgg [43] synthesizes stateful program from user's examples. However, they both keep the state at the controller.

Compositional languages. NetCore [20], and the family of languages evolved around it [2, 10, 29], have primitives for tests and modifications on packet fields as well as composition operators to combine programs. SNAP builds on these languages by adding primitives for stateful programming (§3). To capture the joint intent of two policies, sometimes the programmer needs to decompose them into their constituent pieces, and then reassemble them using `;` and `+`. PGA [25] allows programmers to specify access control and service chain policies using graphs as the basic building block, and tackles this challenge by defining a new type of composition. However, PGA does not have linguistic primitives for stateful programming, such as those that read and write the contents of global arrays. Thus, we view SNAP and PGA as complementary research projects, with each treating different aspects of language design space.

Stateful switch-level mechanisms. FAST [21] and OpenState [4] propose flow-level state machines as a primitive for a *single* switch. SNAP offers a network-wide OBS programming model, with a compiler to distribute the programs across the network. Thus, although SNAP is exponentially more compact than a state machine in cases where the state stores contents of packet header, both FAST and OpenState can be used as a target for a subset of SNAP's programs.

Optimizing placement and routing. Several projects have explored optimizing placement of middleboxes and/or routing traffic through them. These projects and SNAP share the mathematical problem of placement and routing on a graph. Merlin programs specify service chains as well as optimization objectives [37], and the compiler uses an MILP to choose paths for traffic with respect to specification. However, it does not decide the placement of service boxes itself. Rather, it chooses the paths to pass the existing instances of the services in the physical network. Stratos [11] explores middlebox placement and distributing flows amongst them to minimize inter-rack traffic, and Slick [3] breaks middleboxes into fine-grained elements and distributes them across the network while minimizing congestion. However, they both have a separate algorithm for placement. In Stratos, placement results

is used in an ILP to decide distribution of flows. Slick uses a virtual topology on the placed elements with heuristic link weights, and finds shortest paths between traffic endpoints.

9 Conclusions

In this paper, we introduced a stateful SDN programming model with a one-big-switch abstraction, persistent global arrays, and network transactions. We developed a suite of algorithms to analyze such programs, compile them and distribute their state across the network. Based on these ideas, we implemented and evaluated the SNAP language and compiler for programming with stateful SDN.

References

[1] S. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978. (Cited on page 3.)

[2] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 113–126, New York, NY, USA, January 2014. ACM. (Cited on pages 4, 11 and 12.)

[3] B. Anwer, T. Benson, N. Feamster, and D. Levin. Programming slick network functions. In *ACM SIGCOMM Symposium on SDN Research*, pages 14:1–14:13, New York, NY, USA, 2015. ACM. (Cited on page 12.)

[4] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. OpenState: Programming platform-independent stateful OpenFlow applications inside the switch. *ACM SIGCOMM Computer Communication Review*, 44(2):44–51, 2014. (Cited on pages 1, 9 and 12.)

[5] K. Borders, J. Springer, and M. Burnside. Chimera: A declarative language for streaming network traffic analysis. In *USENIX Security Symposium*, pages 365–379, 2012. (Cited on pages 2 and 9.)

[6] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014. (Cited on pages 1, 2, 9 and 11.)

[7] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. ACM. (Cited on page 9.)

[8] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey. Bohatei: Flexible and elastic ddos defense. In *USENIX Security Symposium*, pages 817–832, Washington, D.C., August 2015. USENIX Association. (Cited on page 9.)

[9] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul. Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions. In *Hot Topics in Software-Defined Networks*, pages 19–24, 2013. (Cited on page 11.)

[10] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *ACM SIGPLAN International Conference on Functional Programming*, September 2011. (Cited on pages 1 and 12.)

[11] A. Gember, R. Grandl, A. Anand, T. Benson, and A. Akella. Stratos: Virtual middleboxes as first-class entities. *UW-Madison TR1771*, 2012. (Cited on page 12.)

[12] A. Gember-Jacobson and A. Akella. Improving the safety, scalability, and efficiency of network function state transfers. In *ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, pages 43–48, New York, NY, USA, 2015. ACM. (Cited on page 11.)

[13] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling innovation in network function control. In *ACM SIGCOMM*, pages 163–174, New York, NY, USA, 2014. ACM. (Cited on page 11.)

[14] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an operating system for networks. *ACM SIGCOMM Computer Communications Review*, 38(3), 2008. (Cited on page 1.)

[15] Gurobi optimizer. `http://www.gurobi.com`. Accessed: September 2015. (Cited on page 8.)

[16] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 3–14. ACM, 2013. (Cited on page 9.)

[17] D. A. Joseph, A. Tavakoli, and I. Stoica. A policy-aware switching layer for data centers. In *ACM SIGCOMM*, pages 51–62, New York, NY, USA, 2008. ACM. (Cited on page 11.)

[18] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. Kinetic: Verifiable dynamic network control. In *Networked Systems Design and Implementation*, 2015. (Cited on pages 1 and 12.)

[19] J. McClurg, H. Hojjat, N. Foster, and P. Cerný. Specification and compilation of event-driven SDN programs. *CoRR*, abs/1507.07049, July 2015. (Cited on page 12.)

[20] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A compiler and run-time system for network programming languages. *ACM SIGPLAN Notices*, 47(1):217–230, 2012. (Cited on pages 4, 9 and 12.)

[21] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan. Flow-level state transition as a new switch primitive for SDN. In *Hot Topics in Software-Defined Networks*, pages 61–66, New York, NY, USA, 2014. ACM. (Cited on pages 9, 11 and 12.)

[22] A. Nucci, A. Sridharan, and N. Taft. The problem of synthetically generating IP traffic matrices: Initial recommendations. *ACM SIGCOMM Computer Communication Review*, 35(3):19–32, 2005. (Cited on page 9.)

[23] A. Panda, O. Lahav, K. J. Argyraki, M. Sagiv, and S. Shenker. Verifying isolation properties in the presence of middleboxes. *CoRR*, abs/1409.7687, 2014. (Cited on page 11.)

[24] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of Open vSwitch. In *Networked Systems Design and Implementation*, May 2015. (Cited on pages 1, 9 and 11.)

[25] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang. Pga: Using graphs to express and automatically reconcile network policies. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 29–42. ACM, 2015. (Cited on page 12.)

[26] Pypy. `http://pypy.org`. Accessed: September 2015. (Cited on page 10.)

[27] B. Quoitin, V. Van den Schrieck, P. François, and O. Bonaventure. IGen: Generation of router-level Internet topologies through network design heuristics. In *International Teletraffic Congress*, pages 1–8. IEEE, 2009. (Cited on page 10.)

[28] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/Merge: System support for elastic execution in virtual middleboxes. In *NSDI*, pages 227–240, 2013. (Cited on page 11.)

[29] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker. Modular SDN Programming with Pyretic. *USENIX; login*, 38(5):128–134, 2013. (Cited on pages 1 and 12.)

[30] M. Roughan. Simplifying the synthesis of Internet traffic matrices. *ACM SIGCOMM Computer Communication Review*, 35(5):93–96, 2005. (Cited on page 10.)

[31] M. Shahbaz and N. Feamster. The case for an intermediate representation for programmable data planes. In *ACM SIGCOMM Symposium on SDN Research*. ACM, 2015. (Cited on pages 9 and 11.)

[32] A. Sivaraman, S. Subramanian, A. Agrawal, S. Chole, S.-T. Chuang, T. Edsall, M. Alizadeh, S. Katti, N. McKeown, and H. Balakrishnan. Towards programmable packet scheduling. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, page 23. ACM, 2015. (Cited on page 12.)

[33] S. Smolka, S. A. Eliopoulos, N. Foster, and A. Guha. A fast compiler for NetKAT. In *ACM SIGPLAN International Conference on Functional Programming*, 2015. (Cited on pages 2 and 3.)

[34] Snort. `http://www.snort.org`. (Cited on page 9.)

[35] Snort blog. `http://blog.snort.org`. Accessed: September 2015. (Cited on page 9.)

[36] H. Song. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In *Hot Topics in Software-Defined Networks*, August 2013. (Cited on pages 1 and 9.)

[37] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A language for provisioning network resources. In *ACM SIGCOMM CoNEXT Conference*, pages 213–226, New York, NY, USA, 2014. ACM. (Cited on page 12.)

[38] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson. Measuring ISP topologies with Rocketfuel. *IEEE/ACM Transactions on Networking*, 12(1):2–16, 2004. (Cited on page 10.)

[39] M. Suchara, D. Xu, R. Doverspike, D. Johnson, and J. Rexford. Network architecture for joint failure recovery and traffic engineering. In *ACM SIGMETRICS*, pages 97–108. ACM, 2011. (Cited on page 9.)

[40] Snap : Stateful network-wide abstractions for packet processing, anonymous technical report. `https://sites.google.com/site/snaptr16`. (Cited on pages 4, 7, 8 and 9.)

[41] R. Teixeira, N. Duffield, J. Rexford, and M. Roughan. Traffic matrix reloaded: Impact of routing changes. In *Passive and Active Network Measurement*, pages 251–264. Springer, 2005. (Cited on page 9.)

[42] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: Simplifying SDN programming using algorithmic policies. In *ACM SIGCOMM*, 2013. (Cited on page 1.)

[43] Y. Yuan, R. Alur, and B. T. Loo. NetEgg: Programming network policies by examples. In *ACM SIGCOMM Hot Topics in Networking*, pages 20:1–20:7, New York, NY, USA, 2014. ACM. (Cited on page 12.)