Genesis : End-to-End Policy Enforcement by Switch Table Synthesis

Kausik Subramanian
University of Wisconsin-Madison

ABSTRACT

Operators in enterprise and datacenter networks deal with complex end-to-end policies for a large number of classes like reachabilities, middlebox traversals and isolation policies, and dealing with these policies separately and enforcing them at switch-level rules is cumbersome. To tackle this, we create a tool Genesis which supports a rich suite of end-to-end policies which the network operator can specify using a high-level API. Genesis converts the policy enforcement problem to a SMT instance and synthesizes switch rules using a SMT solver (Z3) which can be used by a controller to deploy to the topology.

1. INTRODUCTION

Network operators of enterprise and cloud datacenter networks deal with hundreds of switches and thousands of traffic classes desiring diverse end-to-end policies. However, in real-life, the process of policy enforcement by network operators is manual and ad-hoc, leading to mis-configurations which have severe performance and security impacts. With the boom in cloud services, datacenter networks deal with thousands of packet classes which are not constant, but in flux, thus, making it difficult to enforce them manually.

The bare minimum need is *reachability* for hosts from ingress to egress switches in these networks. With the recent advent of complex packet processing middleboxes like like intrusion detection systems, firewalls, load balancers etc, a feature of use is to specify the middleboxes a particular flow would like to traverse.

With the advent of multi-tenant cloud services, a desired feature cloud providers need to support is various Quality of Service guarantees. The current Service Level Agreements (SLA) provided are centered around compute, storage, or Internet traffic. The lack of a good network management system, there exists no provisions for providing SLAs in terms of network performance guarantees among the VMs. Lack of network guarantees leads to unpredictability of performance for distributed applications. Tenants may have to run VMs for a longer time, because the network turns out to be a bottleneck. This leads to increased expense for tenants. Also, multi-tenant clouds are susceptible to network attacks. For example, if two tenants share a link in the datacenter, one tenant could hog the bandwidth on the link, and the other tenant would suffer due to unequal distribution of bandwidth. To combat this, cloud operators rate-limit the VMs, but this can reduce link utilization as tenants may not be using the complete share of their bandwidth. An approach to remove interference between tenant flows is to ensure the tenant flows do not share links in the datacenter, and recent datacenter topologies like fat-trees provides multiple paths from one layer to another and thus, we can route different tenants through different paths. Thus, support for link isolation policies is a very important feature for cloud operators.

There has been a lot of work in the field of policy enforcement in networks, like bandwidth provisioning in Merlin [**?**], and middlebox policy enforcement in SIMPLE [2]. However, the approach followed in these works are catered to the specific policies, and thus, difficult to extend it to, for example, provide isolation of paths. An ideal policy enforcement system for network operators would support complex and diverse policies, and thus an important feature is *generality* of the approach, so that it can be extended to enforce custom policies required by the operator. To this end, we propose applying *synthesis* to the problem of policy enforcement by use of SMT-solvers.

In recent years, the space of program synthesis has seen great progress, and in the context of SDNs, controller synthesis [4]. The work on controller synthesis is towards enforcing switch-level behaviour, for example, learning switches or firewalls. However, when dealing with enterprise and datacenter networks, network operators need support for specifying end-to-end proactive policies without the need to program individual switch behaviour. Also, policies useful to operators are proactive (not dependent on the actual packets flowing), and this enables to enforce policies by synthesis of switch-table rules, and using a skeleton controller to deploy the forwarding rules to the switches. In contrast, trying to synthesize reactive policies (like a firewall), the

controller needs to store the state of flows it has received and have a control module following the specifications, which is an interesting synthesis problem, but orthogonal to our problem.

Enforcement of isolation, waypoint policies are NP-complete(see appendix A for proofs), so to provide support for policies which are fundamentally hard to compute, we leverage recent advances in creating fast SMT solvers (like Z3) to perform synthesis by encoding the policy enforcement problem to a SMT instance, and use the SMT solver to find a solution which we translate to switch rules. To this end, we design the Genesis tool with a rich set of policy support where the network operators express the end-to-end policies using a high-level API and Genesis will synthesize the lower-level switch forwarding rules for realising these policies, abstracting out the need for operators to work on switch-level behaviours. <Write about main contributions>

2. PROBLEM STATEMENT

3. SYNTHESIS

Given a set of policies, we perform switch table synthesis by modeling the network forwarding model, and for each policy, adding a set of constraints to the SMT Solver Z3 [**?**], such that the solution model satisfying the constraints can be used to extract the forwarding rules for the switches. One of the key points of the network forwarding model is that the reachability, waypoint and isolation constraints are completely encoded using SAT i.e using boolean variables.

3.1 Network Forwarding Model

We define the physical switch topology as an undirected graph $T = \{S, L\}$, where S is the set of switches and L is the set of links. We use the neighbour function $N(s) = \{s' \mid (s, s') \in L\}$ to denote the set of neighbour switches of L. The set of reachability policies is denoted as R and each reachability policy $r \in R$ is of the form : $\{predicate, src, dst, W\}$.

The field *predicate* is defined over the space of network headers and is used to identify the class of the packet. Assuming that the intersection of predicates is empty for policies in R, we create a mapping $\gamma : R \rightarrow PC$ to associate each reachability policy with a unique integer called packet class in the set PC. Switches $src, dst \in S$ denote the ingress and egress switches respectively for the packet class $pc = \gamma(r)$ and Genesis finds a path from src to dst for pc. If a waypoint policy is specified, W is the set of switches the path from src to dst must traverse through in no particular order.

We define a static integer μ to be the maximum path length for any packet class, and define the set $K = [0, \mu]$ to be the set of all permissible path lengths.

The key functions of the network forwarding model are *two-fold*, abstracting the actual forwarding rules at each node, and encoding the reachability of each flow.

DEFINITION 1. *$Fwd : S \times S \times PC \rightarrow \{True, False\}$ is the forwarding rule model. $Fwd(sw_1, sw_2, pc)$ is set to $True$ means that sw_1 will forward packets of class pc to switch sw_2.*

DEFINITION 2. *$Reach : N \times PC \times K \rightarrow \{True, False\}$ is the reachability model. $Reach(sw, pc, k)$ function is set to True if switch sw is reachable in the path from source switch of packet class pc in exactly k steps.*

The Fwd function can be $True$ only for switches which are neighbours in the topology. For sw_1, sw_2 who are not connected by a link, $\forall pc, Fwd(sw_1, sw_2, pc) = False$.

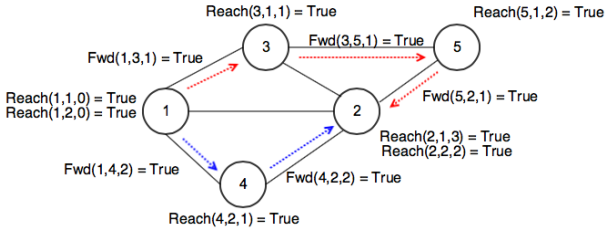

Figure 1: Values of the network forwarding model. The red and blue arrows denote the paths taken by packet classes 1 and 2 respectively from source switch 1 to destination switch 2

The network forwarding model is demonstrated by an example in fig. 1. There are two reachability policies, $r1 : 1 >> [5] >> 2$ with $pc = 1$ and $r2 : 1 >> 2$ with $pc = 2$ and $r1$ is isolated to $r2$. Using the value of Fwd function, we can find out paths for each packet class, and the set of forwarding rules.

3.2 Reachability Constraints

For a reachability policy $s >> d$ and packet class pc, the constraints added must ensure that the solution model contains a path from source from destination. One of the constraints required for this is that there must exist a forwarding rule at source to one of its neighbours, and these constraints act as the base case relating the Fwd and $Reach$ functions.

$$\exists n \in N(s).Fwd(s, n, pc) \wedge Reach(n, pc, 1) \quad (1)$$

$Reach(s.pc, 0)$ is taken to be $True$. We need a path to destination, thus destination switch must be reachable for some path length :

$$\exists k.Reach(dst, pc, k) \quad (2)$$

Also, since the destination is supposed to be the last switch in the path, we need to ensure that the destination does not have forwarding rules further :

$$\forall n \in N(dst).Fwd(dst, n, pc) = False \quad (3)$$

For the solver to find a path, we need inductive constraints propagating reachability backward from destination to source. If a node n_1 is reachable in k, there must a node n_2 which is reachable in $k-1$ steps and there exists a forwarding rule $n_2 \rightarrow n_1$.

$$\forall n_1, k.Reach(n_1, pc, k) \implies \exists n_2.n_2 \in N(n_1) \wedge Reach(n_2, pc, k-1) \wedge Fwd(n_2, n_1, pc) \quad (4)$$

The backward propagation is responsible for ensuring there is a valid path from source to destination by using the unit clauses in eq. (2), and finding a path from destination back to a switch sw which is a neighbour of src. thus $Reach(sw, pc, 1)$ would be true from eq. (1) and the reachability policy would be satisfied.

The above mentioned constraints are sufficient to ensure there is a path from src to dst for packet class pc in terms of the forwarding function Fwd. However, since there is no restriction on number of Fwd values that can be true at a switch, we can get multiple forwarding rules at switches, and also multiple paths to the destination. Also, there can exist forwarding loops as well. Howvever, for a reachability policy, we merely require to find a path from src to dst, so we can just extract a path from the model by performing a breadth-first search on the model-graph from source to destination. A directed edge exists in the model-graph : $n_1 \rightarrow n_2$ if there is forwarding rule indicated by the model $Fwd(n_1, n_2, pc) = True$. Thus, we extract the relevant rules of the shortest path from source to destination from the model, and the additional rules obtained in the solution (extra paths, forwarding loops) are ignored.

3.3 Waypoint Constraints

For a reachability policy with waypoints $s >> W >> d$ and packet class pc, we add all the constraints specified in § 3.2 to ensure the existence of a path from source to destination. To satisfy waypoint policies, we need to add constraints so that all $w \in W$ is reachable.

$$\forall w \in W. \exists k.Reach(w, pc, k) \quad (5)$$

However, just ensuring reachability of waypoints is not sufficient. Since, we do not have any restrictions on the count of forwarding rules for a packet class at a switch, it is possible that waypoints are reachable from the source through separate paths, and do not lie in the path from source to destination. Thus, to ensure that all waypoints are reachable in the path from source to destination, we need to add constraints on the count of forwarding rules at each switch.

Forwarding rule constraints are to ensure that the forwarding function Fwd at a switch forwards to a *single* node which is a *neighbour* or to no node at all (Switches which are not reached in the path and the destination will not have any forwarding rules). We define the forwarding set $FwdSet$ as follows:

$$FwdSet(sw, pc) = \{n \mid Fwd(sw, n, pc) = True\} \quad (6)$$

The $|A|$ function is used to the denote the size of set A. The constraint on the forwarding set are that the size of each set must not exceed 1 (0 or 1). The forwarding set constraint is as follows :

$$\forall sw, pc.\ |FwdSet(sw, pc)| \leq 1 \quad (7)$$

The forwarding set constraints ensure that the forwarding rules exist only on the path from source to destination, and no other rules exist in the solution. If a switch has a forwarding rule elsewhere, then it would not have a rule for the path, and the destination will not be reachable. These restrictions will also ensure there are no forwarding loops in the path. Since, there is only one path in the model from source and destination, and eq. (5) ensures that waypoints are reachable, therefore, the path from source to destination must traverse through all the waypoints. Since eq. (5) imposes no order on waypoint traversal, the solution will traverse the waypoints in no particular order.

3.4 Isolation Constraints

For a traffic isolation policy $pc_1 || pc_2$ means that the paths do not share an link in the same direction in their paths, therefore, at every switch, the forwarding rules for pc_1 and pc_2 must not forward it to the same switch.

$$\forall n_1. \neg(\exists n_2.Fwd(n_1, n_2, pc_1) \wedge Fwd(n_1, n_2, pc_2)) \quad (8)$$

These constraints are sufficient to ensure that the packet classes pc_1 and pc_2 would be isolated.

The isolation constraints is intuitive when coupled with the forwarding set constraints (eq. (7)) as the model only has forwarding rules for the path from source to destination. However, for a reachability policy, we argue that the forwarding set constraints are not required when coupled with the isolation constraints. The reasoning behind this is that the solver would simply remove the extra forwarding rules of a packet class in the model which conflict with the other packet class, as there are no constraints which require the need of these extra forwarding rules for correctness, but are just one solution model in the space of solutions.

For a security isolation policy, pc_1 and pc_2 must not share links at all. eq. (8) constraint can be modified to enforce bidirectional isolation :

$$\forall n_1. \neg(\exists n_2.Fwd(n_1, n_2, pc_1) \wedge (Fwd(n_1, n_2, pc_2) \vee Fwd(n_2, n_1, pc_2)) \quad (9)$$

3.5 Maintenance Constraints

<Write about link and switch maintanences>

3.6 Capacity Constraints

For a link capacity policy on the link $sw_1 \rightarrow sw_2$ with capacity ω, we use the theory of integer linear arithmetic to add constraints on the link so that capacity used by the flows traversing the link does not exceed ω. In terms of our model, the link capacity policy translates to constraints to ensure that the the occurences of $Fwd(sw_1, sw_2, pc) = True$ for $pc \in PC$ conforms to the capacity specified in the policy, as the forwarding rule $sw1 \rightarrow sw2$ means that the link is being used by the particular packet class.

Let $C(sw_1, sw_2, pc)$ be the cumulative capacity function of the link used by all packet classes less than equal to pc. Since we use integers for denoting the packet class, and thus, we have total order of the set of packet classes. We use this to create inductive constraints to sum over the set of Boolean variables $Fwd(sw_1, sw_2, pc)$. Let $PC : [0, \lambda]$ be the set of packet classes and the $W(pc)$ is the capacity of packet class pc (W can be uniform or non-uniform for all packet classes). The base case constraint for the capacity function is for $pc = 0$ (the minimum element of PC) :

$$Fwd(sw_1, sw_2, 0) \implies C(sw_1, sw_2, 0) == W(0) \quad (10)$$

$$\neg Fwd(sw_1, sw_2, 0) \implies C(sw_1, sw_2, 0) == 0 \quad (11)$$

The inductive constraints for the capacity function are as follows if link is used :

$$\forall pc > 0. Fwd(sw_1, sw_2, pc) \implies$$
$$C(sw_1, sw_2, pc) == C(sw_1.sw_2, pc - 1) + W(pc) \quad (12)$$

If the link is not used, the constraints are as follows :

$$\forall pc > 0. \neg Fwd(sw_1, sw_2, pc) \implies$$
$$C(sw_1, sw_2, pc) == C(sw_1.sw_2, pc - 1) \quad (13)$$

If pc uses the link $sw_1 \rightarrow sw_2$, we add $W(pc)$ to $C(sw_1, sw_2, pc - 1)$, or add 0 to it if link is not being used. Thus, we define the cumulative capacity constraints using inductive constraints. To satisfy the capacity policy, the total capacity used should not exceed input ω. λ is the greatest element in the set PC.

$$C(sw_1, sw_2, \lambda) \leq \omega \quad (14)$$

4. CONSTRAINT EVALUATION

In the section, we evaluate the number of constraints required to add to the z3 Solver. Let $|S|$ be the number of switches in the topology, $|L|$ be the number of edges in the topology, $|PC|$ be the number of packet classes, $|I|$ be the number of traffic isolation policies and μ be the max path length. Let deg be the maximum number of neighbours for a switch (degree of the topology).

We encode the forwarding set constraint (eq. (7)) using a SAT encoding rather than an SMT encoding for better performance. To express the constraint that the count of forwarding rules is not more than one, we can do this by adding an *or* of clauses, where each clause is an *and* of one rule set to true and all others set to false. An example for switch s and neighbours {t,u} is

```
(or
  (and Fwd(s, t, pc) (not Fwd(s, u, pc)))
  (and Fwd(s, u, pc) (not Fwd(s, t, pc)))
  (and (not Fwd(s, u, pc)) (not Fwd(s, t, pc)))
)
```

Thus, the number of terms in a forwarding rule constraint for a switch and packet class is $deg \times (deg + 1)$. We add a single `or` constraint for each switch and packet class in the network. Therefore, the count of constraints is $|S| \times |PC|$.

The major bulk of time is consumed by the creation of the constraints for backward propagation of reachability (eq. (4)). The number of constraints added to the solver is $|S| \times |PC| \times \mu$. We are using a quantifier-free encoding for better performance, so expressing the *exists* is done by a *or* clause of the set of neighbours for a switch. Therefore, the size of each constraint is deg.

For each traffic isolation policy, we need to add constraints for each edge of the network, so that the two packet classes dont share the edge. Therefore, the number of constraints added is $|L| \times |I|$ and the size of each constraint is constant size of two terms.

Constraints	Number	Size
Forwarding Rules	$\|S\| \times \|PC\|$	$deg \times (deg + 1)$
Reachability Propagations	$\|S\| \times \|PC\| \times \mu$	deg
Isolation	$\|L\| \times \|I\|$	2

Table 1: Number and Size of Constraints

5. TACTICS

If we consider reachability between two end-switches, the synthesis problem translates to choosing a path from the solution space of all paths from the source to destination such that the chosen path satisfies all policies like waypoints and isolation. Datacenter topologies have multiple paths between endpoints to provide full bisection bandwidth. This feature of datacenters ensues that the solution space of paths for a pair of endpoints is large.

For example, consider the fat-tree topology in fig. 2. The number of paths under length 10 between two edge switches in the same pod is 242 and two edge switches in different pods is 272. If we consider the synthesis of n reachability policies, the problem roughly translates to finding a solution in the solution space of size $n \times 242$ which satisfies all policies.

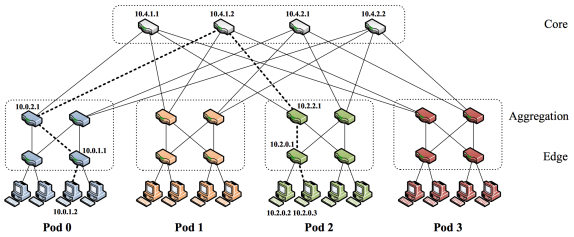

Figure 2: Fat Tree Topology

We can leverage the network structure of specific topologies to reduce the solution space. In the case of fig. 2, the 272 paths between two edge switches would contain many paths which can be avoided, like paths which traverse through edge switches other than the source/destination, or paths which would be going from the aggregate to core and back multiple times.

Tactics are abstractions to impose a high-level restriction on the paths by leveraging the network structure. An example tactic for the fat-tree topology would be that a path connecting two edge switches must not traverse through another edge switch (unless we have a edge switch which is a waypoint). The advantage of tactics is that we would like to provide an expressive framework for operators to provide some insights on the properties of the path without drastically reducing the solution space of paths.

Let us define Lb be the set of labels and S be the set of switches in the topology. Let $\phi : S \rightarrow Lb$ be the labelling function and maps switches to a label in Lb. One example for ϕ in the fat-tree topology in fig. 2 can be that we map all switches in the same level(core, aggregate or edge) to the same level, leveraging the hierarchical structure.

DEFINITION 3. *A tactic T is a regular expression over the alphabel Lb. We construct an associated automaton A^T which is the automaton of the regular expression with every state accepting except the sink state (non-final state with a self-loop over all alphabets).*

A path P is a word over the alphabet S. Let $\Phi : S^* \rightarrow Lb^*$ be the path-labelling function which maps each switch in the word to its corresponding label.

$$P \models T \iff \Phi(P) \in L(A^T) \tag{15}$$

where $L(A^T)$ is the language of automaton A^T. The rationale behind the associated automaton A^T is that the automaton accepts prefixes of the label-path, and rejects prefixes which cannot lead to a path satisfying the regular expression of the tactic (by reaching a sink state).

We demonstrate the example of a tactic with the fat-tree topology in fig. 2. Let $Lb = \{c, a, e\}$. The labelling function ϕ maps each core switch to c, each aggregate switch to a and every edge switch to e. For a path connecting two edge switches, let us define T_e as $(e.^*e) \wedge \neg(e.^*e.^*e)$. This tactic says that the path connecting the two edge switches must not traverse through a edge switch.

5.1 Synthesis with Tactics

The tactic defined in the earlier section is a general framework of specify path properties based on network structure. However, adding constraints to ensure the path satisfies the tactic is *slower* than without using tactics. In our synthesis algorithm, the backward reachability propagation constraints are the most significant in terms of time taken and complexity.

$$\forall n_1, k. Reach(n_1, pc, k) \implies \exists n_2. n_2 \in N(n_1) \wedge Reach(n_2, pc, k-1) \wedge Fwd(n_2, n_1, pc) \tag{16}$$

To prune these type of constraints, the question we need to answer is this: If switch n_1 with label $\phi(n_1)$ exists in a word at position k, what labels can exist in the word at position $k-1$ such that the word does not reach a sink state in the tactic automaton. By eliminating words which lead the automaton to the sink state, we eliminate the paths which will not satisfy the tactic. For example, the tactic $T_e : (e.^*e) \wedge \neg(e.^*e.^*e)$ which specifies that the path must start and end at a edge switch, and traverse through other edge switches. For $k > 2$ and label a, we can observe that the label preceeding it cannot be e if the automaton is not in a sink state (since label e does not come up in the word except the first and the last character). Thus, while adding constraints for a switch of label a and $k > 2$, we can remove neighbours which have label e.

Let us formalise this notion of finding local label patterns based on a tactic T and associated automaton A^T over the alphabet Lb. Let $w[k+1]$ denotes the $(k+1)^{th}$ character of the word. The automaton A^T will accept all words except those which lead it to the sink state. We use the automaton to construct a set of reachable paths for switch label lb and path length k.

DEFINITION 4. *$\Gamma(lb, k)$ is the set of reachable paths such that for each $w \in \Gamma(lb, k)$:*

1. $w \in L(A^T)$
2. $w \in Lb^{k+1}$
3. $w[k+1] = lb$
4. *w is a valid label-path in the topology*

The first and second requirement is to reason about the backward reachability propagation constraints $Reach(sw, pc, k)$ for switches $sw \in S$ for which $\phi(sw) = lb$. The input to the automaton A^T is restricted based on the topology and label-mapping, and the third requirement is to prune the words to valid label-paths

of the topology. In our running example of the fat-tree topology, we can see that neighbours of core switches(c) are only aggregate switches(a). Therefore, in a valid label-path, after reading a c, we can only read the character a. We create a label adjacency matrix of the topology and prune the words in Γ.

DEFINITION 5. *$\Delta(lb, k)$ is the set of labels such that for each $l \in \Delta(lb, k) \iff \exists w \in \Gamma(lb, k).w[k] = l$*

where $w[k]$ denotes the k^{th} character in the word.

After computing $\Delta(lb, k)$, we can modify the backward reachability propagation constraints to include neighbours whose labels are in $\Delta(lb, k)$.

$$\forall n_1, k.Reach(n_1, pc, k) \implies \exists n_2.n_2 \in N(n_1) \wedge \phi(n_2) \in \Delta(\phi(n_1), k) \wedge Reach(n_2, pc, k-1) \wedge Fwd(n_2, n_1, pc) \quad (17)$$

If $\Delta(lb, k) = \emptyset$, then $Reach(sw, pc, k) = False$ for switches $sw \in S$ for which $\phi(sw) = lb$.

To find the Δs for a tactic, we use a dynamic programming approach of finding all valid label words of length k accepted by the associated automaton ($\forall lb.\ \Gamma(lb, k)$) by using the valid label words of length $k-1$ (we also store the last state the automaton is in), and make one valid transition on labels (validity is based on two factors : automaton doesn't reach the sink state, and the current label and previous label are connected in the topology). After computing the Γs, we can compute $\Delta(lb, k)$ by examining $\Gamma(lb, k)$ to find labels at the k^{th} position.

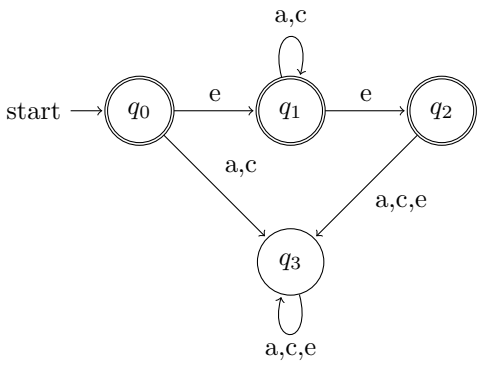

Figure 3: Associated automaton A^T for tactic T_e

We demonstrate the advantages of tactics with an example. Consider the tactic $T_e : (e.^*e) \wedge \neg(e.^*e.^*e)$ and its associated automaton(fig. 3).

$$\Gamma(a, 2) = \emptyset,\ \Gamma(e, 2) = \{eae\},\ \Gamma(c, 2) = \{eac\}$$

$$\Delta(a, 2) = \emptyset,\ \Delta(e, 2) = \{a\},\ \Delta(c, 2) = \{a\}$$

Thus, $\forall sw, pc.\ \phi(sw) = a \ \wedge\ \Delta(a, 2) = \emptyset \implies Reach(sw, pc, 2) = False$ and we do not add backward reachability propagation constraints for the term $Reach(sw, pc, 2)$.

Incorporating tactics in the synthesis is sound. If the tactic imposes an restriction to the path other than the source and destination, then the procedure is incomplete. However, tactics can be used to specify restrictions which would be reasonably complete, thus speeding up the synthesis. Even though we have defined a general framework to specify tactics using regular expressions, since the synthesis constraints depend on local behaviours, not every tactic would provide a speedup. For example the tactic $\neg(.^*cacac.^*)$ which specifies that a path must not traverse through the core switch layer more than twice, will not provide speedup, bacause we cannot deduce any local patterns from the associated automaton as the regular expression is trying to count the number of times the path traverses a core switch, and thus it cannot remove any label from its preceding set as count is a global pattern and our approach can only use local patterns to speedup synthesis.

6. NETWORK SURGERY

Network surgery is the technique of performing equivalent network transformations to eliminate redudant constraints required for synthesis of switch rules. One of the properties of the path found by the synthesis solver is that it is simple (i.e no loops). Using this property, we can create slices of the topology where a packet class' path will reside completely, thus not requiring to add constraints for switches in other slices of the topology.

To create the topology slices, we use Schmidt's linear-time algorithm[**?**] to find bridges in the graph. A bridge is an edge in the topology, which when removed, partitions the graph into two disconnected components. <write-about-slices>.

Consider a reachability policy where the source and destination switches belong to the same topology slice. Since, the bridge edge is the only edge connected vertices of this slice with the rest of the graph, the path for the reachability policy will be contained in the topology slice and not cross the bridge (otherwise the path would have to traverse through the bridge twice back and forth which is not permitted).

Formally, let us define the slices of the topology as $S_1, S_2, ..., S_n \subset S$. We define the slice neighbour function for the slices as $N_{S_i}(s) = \{v | v \in S_i \wedge (s, v) \in L\}$. If there is a reachability policy (r, pc) with $src, dst \in S_i$, we can replace the switch domain S by S_i and the neighbour function N by N_{S_i} in the constraint formation for reachability. For example, the backward reachability propagation constraints for a policy in slice S_i can be modified to :

$$\forall n_1, k.n_1 \in S_i \wedge Reach(n_1, pc, k) \implies \exists n_2.n_2 \in N_{S_i}(n_1) \wedge Reach(n_2, pc, k-1) \wedge Fwd(n_2, n_1, pc) \quad (18)$$

For packet classes isolated with packet class pc, only

links in S_i are needed to be isolated, as the path will be confined to topology slice S_i.

7. OPTIMISTIC SYNTHESIS

8. IMPLEMENTATION

9. EVALUATION

In fig. 4, we evaluate the performance of Genesis with respect to increasing topology sizes. We use fat-tree topology, and the policy inputted is two reachability policies where the source and destination are edge switches (one policy has a single aggregate switch as a waypoint). These two policies are isolated to each other. The number of nodes is the X-axis and the Y-axis is the time taken to synthesise the paths for the two input policies. As expected, we can infer a expotential increase with number of switches in the topology.
In fig. 5, we evaluate the performance of Genesis with respect to increasing number of reachability policies in a 45-node fat-tree topology. The reachability policies have both source and destination as edge switches (without a waypoint) and no isolation between them. The number of policies is the X-axis and the Y-axis is the time taken to synthesise the paths for the two input policies. Though, this problem can be solved simply by DFS, this experiment is more to rationalise the fact the z3 will not have linear complexity for such a case.

10. RELATED WORK

[**?**] tries to synthesize local forwarding rules for a single switch based on a reactive forwarding policy. One of the important considerations for correctness of forwarding rules is that the controller sees all relevant events and rules are not added prematurely. The abstraction for expressing forwarding policies are for individual switches, on the other hand, we try to enforce network-wide policies, so each switch will have different forwarding policy such that the entire network enforces the flow policies. Also, the policies we deal with are proactive, so we need not worry about the controller seeing the relevant events (which is a requirement in synthesizing rules for reactive policies). For all practical purposes, we needn't worry about switch-controller interactions or premature rule installations.
NetEgg [4] synthesizes the forwarding policy of a switch using examples of how the switch should function when it receives packets. This deals with the forwarding policy of individual elements, can cannot be used directly to synthesize network-wide policies.
[1] tries to tackle a similar problem to ours of flow policy enforcement. However their end-point policies are only concerned with reachability (two hosts can talk through specific ingress and egress points). Their rule placement algorithm takes the path of the flow in the network as an input (the routing policy) and place rules on this path to enforce the endpoint policy and taking in consideration switch table constraints. We are trying to tackle the problem without the routing policy as input, as we enforce flow policies which would require different routing policies (like traffic isolation), so we cannot determine the path of the flow beforehand. Our solution can support enforcement of policies which require different routing policies. [**?**] builts on the [1] abstraction to optimize the specific case of distributed firewall policy enforcement using ILP.
[3] NetGen solves the problem of network updates using synthesis. Given a specification which mentions the packet classes, the old path and the new path, NetGen solves the network change problem using a SMT solver. One salient aspect is the use of uninterpreted functions to reduce the number of constraints.

11. CONCLUSION

12. REFERENCES

[1] Kang, N., Liu, Z., Rexford, J., and Walker, D. Optimizing the "one big switch" abstraction in software-defined networks. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies* (New York, NY, USA, 2013), CoNEXT '13, ACM, pp. 13–24.

[2] Qazi, Z. A., Tu, C.-C., Chiang, L., Miao, R., Sekar, V., and Yu, M. Simple-fying middlebox policy enforcement using sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 27–38.

[3] Saha, S., Prabhu, S., and Madhusudan, P. Netgen: Synthesizing data-plane configurations for network policies. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research* (New York, NY, USA, 2015), SOSR '15, ACM, pp. 17:1–17:6.

[4] Yuan, Y., Alur, R., and Loo, B. T. Netegg: Programming network policies by examples. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2014), HotNets-XIII, ACM, pp. 20:1–20:7.

APPENDIX

A. PROOFS OF NP-HARDNESS

A.1 Enforcement of Isolation Policies

Given a undirected graph $G = \{V, E\}$ which represents the switch topology denoted in fig. 6 and undirected graph $P = \{R, I\}$ which represents the policy graph. Every vertex $p \in P$ is a reachability policy :

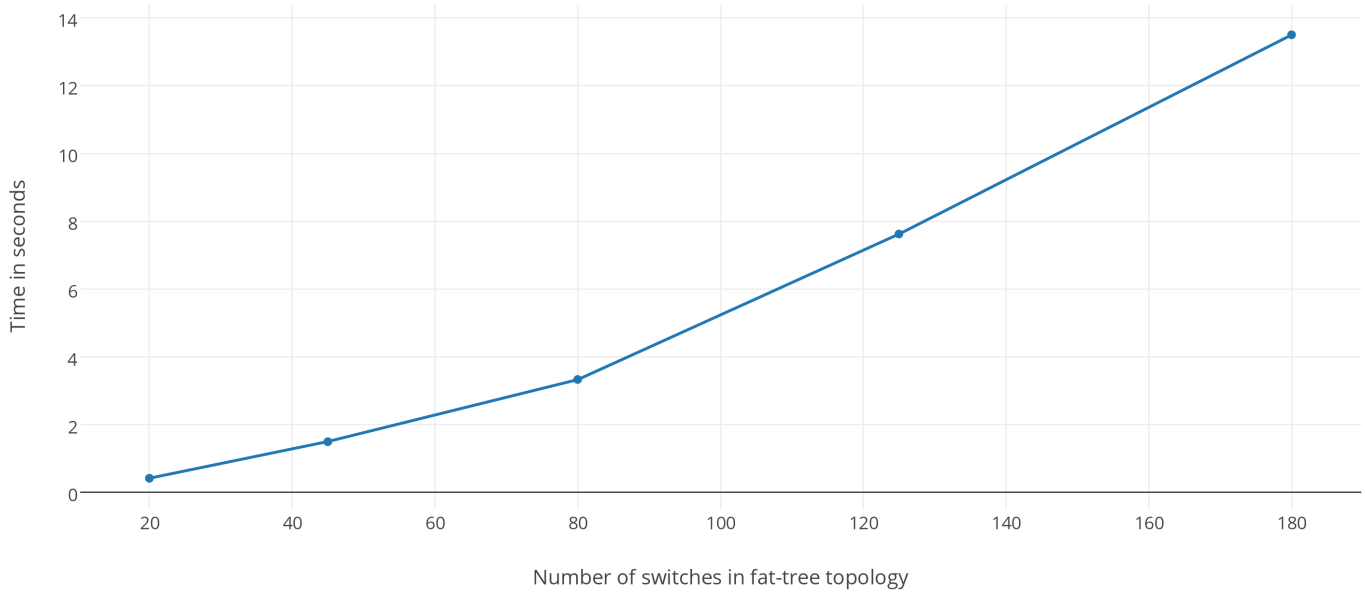

Figure 4: Time taken to synthesize two isolated flows with increasing fat-tree topology sizes

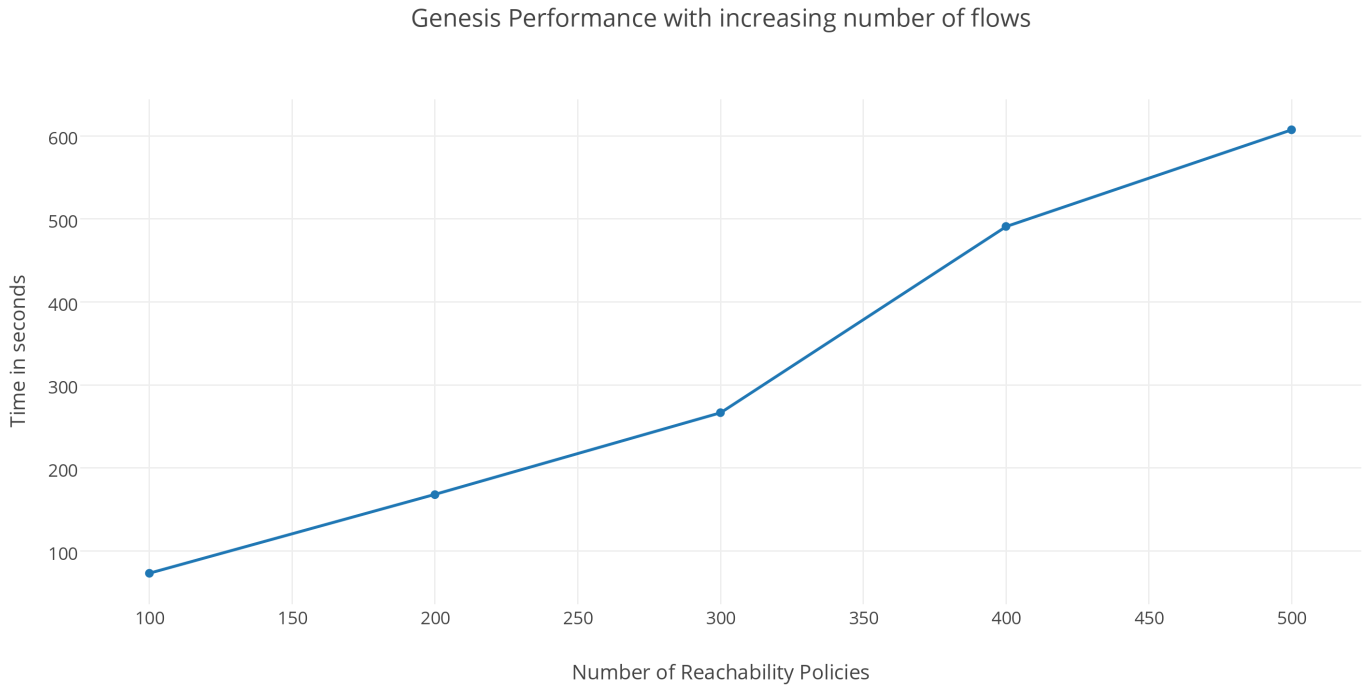

Figure 5: Performance plot to synthesis with increasing number of reachability policies in a 45-node fat-tree topology

$S >> D$ and each edge $i \in I$ which connects vertices $p1$ and $p2$ mean that the paths of $p1$ and $p2$ are isolated from each other.

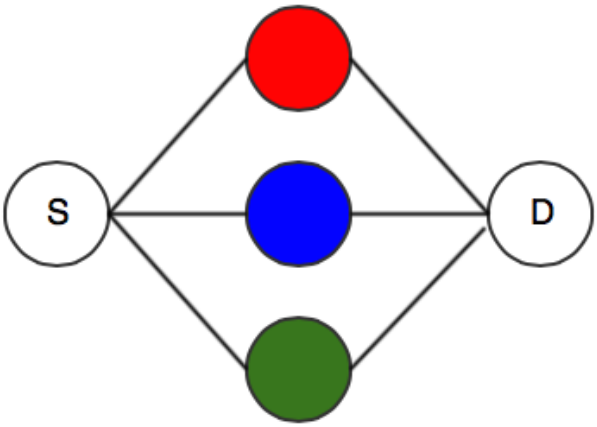

Figure 6: The switch topology G. All circles represent switches and all reachability policies are S to D

The solution to policy enforcement will be such that each reachability policy $p \in P$ from $S >> D$ will traverse through one of the colored switches $\{red, blue, green\}$. Color the vertices of P with the switch the path traverses through. If two vertices are connected by an edge in P, those flows would be isolated, and thus, will not have the same color. Thus, the problem reduces to finding a 3-graph coloring for P, which is NP-complete. Thus, the enforcement of isolation policies is NP-complete.

In genral, k-coloring (for k > 2) is NP-complete, the policy enforcement problem in a switch topology with k paths from source to destination reduce to a k-coloring problem, and thus is NP-complete.

A.2 Enforcement of Waypoint Policies

Given a undirected graph $G = V, E$. Let us assume there exists an polynomial-time algorithm to compute the reachability paths satisfying the policies of the following types on the graph :

- **P1** : $v_1 >> v_2 \Rightarrow$ There exists a path from v_1 to v_2 satisfying all input policies. A property of the path is that it does not have repeat a vertex (no forwarding loops).
- **P2** : $v_1 >> W >> v_2 \Rightarrow$ The path from v_1 to v_2 should pass through the vertices in the set W in any order, without repeating a vertex.

Reduction of Hamiltonian Cycle Problem : Given a undirected graph $G = V, E$, find $v \in V$ such that the degree of v is the minimum in the graph (Will work for any vertex actually). If a Hamiltonian cycle is present in the graph, it will have the vertex v in the cycle, and one of the edges from v.
Lets take a $n \in Neighbours(v)$. Let the input policies to our algorithm be :

- **P4** : $v >> W >> n$ where $W = V - \{v, n\}$

P4 cimputes a simple path from v to n which passes through all the other vertices in the graph which is the Hamiltonian path problem. Since computing the Hamiltonian path is NP-hard, the problem of path computation for the waypoint policies as specified is NP-hard.