Kuai: A Model Checker for Software-defined Networks

Rupak Majumdar
MPI-SWS
Germany

Sai Deep Tetali
UC Los Angeles
USA

Zilong Wang
MPI-SWS
Germany

Abstract—In software-defined networking (SDN), a software controller manages a distributed collection of switches by installing and uninstalling packet-forwarding rules in the switches. SDNs allow flexible implementations for expressive and sophisticated network management policies. We consider the problem of verifying that an SDN satisfies a given safety property. We describe Kuai, a distributed enumerative model checker for SDNs. Kuai takes as input a controller implementation written in Murphi, a description of the network topology (switches and connections), and a safety property, and performs a distributed enumerative reachability analysis on a cluster of machines. Kuai uses a set of partial order reduction techniques specific to the SDN domain that help reduce the state space dramatically. In addition, Kuai performs an automatic abstraction to handle unboundedly many packets traversing the network at a given time and unboundedly many control messages between the controller and the switches.

We demonstrate the scalability and coverage of Kuai on standard SDN benchmarks. We show that our set of partial order reduction techniques significantly reduces the state spaces of these benchmarks by many orders of magnitude. In addition, Kuai exploits large-scale distribution to quickly search the reduced state space.

I. Introduction

Software-defined networking (SDN) is a novel networking architecture in which a centralized software controller dynamically updates the packet processing policies in network switches based on observing the flow of packets in the network. SDNs have been used to implement sophisticated packet processing policies in networks, and there is increasing industrial adoption. By specifying the rules to be added or deleted, a controller can dynamically change how packets flow in the network, a bug in the controller code can lead to hard-to-analyze network errors at run time. We describe the design of Kuai, a distributed enumerative model checker for SDNs. The input to Kuai is a model of an SDN consisting of two parts. The first part is the controller, written in a simplified guarded-command language similar to Murphi. The second part is the description of a network, consisting of a fixed finite set of switches, a fixed set of client nodes, and the topology of the network (i.e., the connections between the ports of the clients and the switches). Given a safety property of the network, Kuai explores the state space of the SDN to check if the property holds on all executions.

Figure 1 shows a simple SDN. It consists of two switches $sw_1$ and $sw_2$ connected to two clients $c_1$ and $c_2$. Each client has a port and each switch has two ports to send and receive packets, and the figure shows how the ports are connected to each other. Each connection between ports represents a bidirectional communication channel that may reorder packets.

Moreover, the switches are connected to a controller through dedicated links. Packets are routed in the network using flow tables in switches. A flow table is a collection of prioritized forwarding rules. A rule consists of a priority, a pattern on packet headers, and a list of ports. A switch processes an incoming packet based on its flow table. It looks at the highest priority rule whose pattern matches the packet and forwards the packet to the list of ports specified in the rule, and drops the packet if the list of ports in the rule is empty. In case no rule matches a packet, the switch forwards the packet to the controller using a request queue and waits for a reply from the controller on a forward queue. The controller replies with a list of ports to which the packet should be forwarded, and optionally sends control messages to the control queue of one or more switches to update their flow tables. A control message can add or delete a rule in a switch.

By specifying the rules to be added or deleted, a controller can dynamically control the behaviors of all switches in an SDN network. For example, suppose we want to implement the policy that all SSH packets are dropped. The controller can update the switches with a rule that states that no SSH packets are forwarded, and another that states all non-SSH packets are forwarded. List 1 shows a possible controller that implements this policy. Essentially, the controller drops SSH packets, and adds three rules on the switches: $r_1$ to drop SSH packets, $r_2$ to forward packets from port 1 to port 2, and $r_3$ to forward packets from port 2 to port 1. Since dropping

---

List 1: Controller for SSH

```python
def pktIn(pkt):
    (sw, pt) = pkt.loc
    if pkt.prot = SSH:
        drop(pkt)
        else:
            dest = 2 if pt = 1 else 1
            fwd(pkt, [[dest]], sw)
    rule r1 = (5, {prot=SSH}, [])
    rule r2 = (1, {port=1}, [2])
    rule r3 = (1, {port=2}, [1])
    message cm1 = add(r1)
    message cm2 = add(r2)
    message cm3 = add(r3)
    for sw in [sw1, sw2]:
        send_message(cm1, sw)
        send_message(cm2, sw)
        send_message(cm3, sw)
```
SSH packets (rule r1) has higher priority, it will match SSH packets, and rules r2 and r3 will only match (and forward) non-SSH packets. The controller has a subtle bug. It turns out that a switch can implement rules in arbitrary order. Thus, the switches may end up adding rules r2 and r3 before adding r1, thus violating the policy. Our model checker confirms the bug. A possible fix in this case is to implement a barrier after line 15, to ensure that rule r1 is added before the other rules. Our model checker confirms the policy holds in the fixed version.

The verification of SDNs is challenging due to several reasons. First, even when the topology is fixed with a finite set of clients and switches, the state space is still unbounded, as clients may generate unboundedly many packets and these packets could be simultaneously progressing through the network. For example, client c1 may send a packet to sw1 at any point, and an unbounded number of packets can be in the network before sw1 processes them. Similarly, there may be an unbounded number of control messages (i.e., messages sent from the controller to a switch) between the controller and the switches. While there may be a physical limit on the number of packets and control messages imposed by packet buffers in the switches, the sizes of these buffers can be large (of the order of megabytes) and precise modeling of buffers will blow up the state space.

Second, the packets may be processed in arbitrary interleaved orders, and the processing of one packet may influence the processing of subsequent ones because the controller may update flow tables based on the first packet. Similarly, control messages between the controller and the switches may be processed in arbitrary order and this may lead to potential behaviors by processing all packets at a time, we do not lose behaviors by processing all packets at the packet queue of a switch atomically. Similarly, using the semantics of the barrier message [12], we show that a switch can atomically execute all control messages up to the last barrier in its control queue. Specifically, this optimization enables the model checker to bound the size of control queues. Additionally, we show that whenever there is a packet in a client’s packet queue, the client can receive and process it immediately, so that sends from switches can be atomically processed with receives at clients. Finally, we show that we can eagerly serve requests to the controller, that is, we do not lose behaviors if we restrict the controller’s request queue to size one and service these requests as soon as they appear.

We empirically demonstrate that our set of partial order reduction techniques significantly reduces the state spaces of SDN benchmarks, often by many orders of magnitude. For the simple SSH example, the number of explored states is approximately 2 million without partial order reductions, but only 13 with reductions!

To handle large state spaces, our model checker Kuai distributes the model checking over a number of nodes in a cluster, using the PReach distributed model checker [2] (based on Murphi [4]) as its back end. The large-scale distribution enables Kuai to model check large state spaces quickly.

Related Work. There is a lot of systems and networking interest in SDNs [9], [5] and standards such as Openflow [12]. From the formal methods perspective, research has focused on verified programming language frameworks for writing SDN controllers [6], [8]. Here, verification refers to correct compilation from Frenetic to executable code, or to checking composability of programs, not the correctness of invariants.

Previous model checking attempts for SDNs mostly focused either on proving a static snapshot of the network [10] or on model checking or symbolic simulation techniques for a fixed number of packets [3], [14]. Recent work extended to controller updates and arbitrary number of packets [17], but used a manual process to add non-interference lemmas. In contrast, our technique automatically deals with unboundedly many packets and, thanks to the partial-order techniques, scales to much larger configurations than reported in [17].

Program verification for SDN controllers using loop invariants and SMT solving has been proposed recently [1]. While the invariants can quantify over the network (and therefore not limited to finite topologies), the model of the network ignores asynchronous interleavings of packet and control message processing that we handle here.

Our work builds on the work of distributed enumerative model checking and the PReach tool [2]. Our contribution is identifying domain specific state space reduction heuristics that enable us to explore large configurations.

II. Software-defined Networks

Preliminaries. A multiset m over a set Σ is a function Σ → N with finite support (i.e., m(σ) ≠ 0 for finitely many σ ∈ Σ). By M[Σ] we denote the set of all multisets over Σ. We shall write m = [σ1/2, σ2/3] for the multiset m ∈ M[[σ1, σ2, σ3]] with m(σ1) = 2, m(σ2) = 0, and m(σ3) = 1. We write Ø for an empty multiset, mapping each σ ∈ Σ to 0. We write {σ} for an empty set. Two multisets are ordered by m1 ≤ m2 if for all σ ∈ Σ, we have m1(σ) ≤ m2(σ). Let m1 ⊕ m2 (resp. m1 ⊖ m2) be the multiset that maps every element σ ∈ Σ to m1(σ) + m2(σ) (resp. max{0, m1(σ) − m2(σ)}).

Given a set of states, a (guarded) action α is a pair (g, c) where g is a guard that evaluates the states to a boolean and c is a command. A action α is enabled in a state s if the guard of α evaluates s to true. If α is enabled in s, the command of α can execute and lead to a new state s’, denoted by s α ⇝ s’. We write α(s) = s’ if s α ⇝ s’. A transition system TS is a tuple (S, A, →, s0, AP, L) where S is a set of states, A is a set of actions, → ⊆ S × A × S is a transition relation, s0 ∈ S is the initial state, AP is a set of atomic propositions, and L : S → 2AP is a labeling function. We write α−* for the reflexive transitive closure of →. A state s’ is reachable from s if s →* s’. We write s →+ s’ if there is a state t such that s → t →* s’. For a state s, let A(s) be the set of actions enabled in s; we assume A(s) ≠ Ø for each s ∈ S. The trace of an infinite execution ρ = s α→ s1 α2→ ... is defined as trace(ρ) = L(s)L(s1)... The trace of a finite execution ρ = s α1→ s1 α2→ ... αn→ s0 is defined as trace(ρ) = L(s)L(s1)...L(sn). An execution is initial if it starts in s0. Let Traces(TS) be the set of traces of initial executions in TS. We define invariants and invariant satisfaction in the usual way.

Syntax of Software-defined Networks We model an SDN as a network consisting of nodes, connections, and a controller
program. Nodes come from a finite set Clients of clients and a (disjoint) finite set Switches of switches. Each node $n$ has a finite set of ports $\text{Port}(n) \subseteq \mathbb{N}$ which are connected to ports of other nodes. A location $(n, pt)$ is a pair of a node and a port $pt \in \text{Port}(n)$. Let $\text{Loc}$ be the set of locations. A connection is a pair of locations. A network is well-formed if there is a bijection function $\lambda : \text{Loc} \rightarrow \text{Loc}$, called the topology function, such that $\{(n, pt), \lambda(n, pt)\} \mid (n, pt) \in \text{Loc}$ is the set of connections and no two clients are connected directly.

We model a packet $pkt$ in the network as a tuple $(a_1, \ldots, a_k, loc)$, where $(a_1, \ldots, a_k) \in \{0, 1\}^k$ models an abstraction of the packet data and $loc \in \text{Loc}$ indicates the location of $pkt$. Let $\text{Packet}$ be the set of all packets.

Each switch contains a set of rules that determine how packets are forwarded. A rule is a tuple $(\texttt{priority}, \texttt{pattern}, \texttt{ports})$, where $\texttt{priority} \in \mathbb{N}$ determines the priority of the rule. $\texttt{pattern}$ is a proposition over $\text{Packet}$, and $\texttt{ports}$ is a multiset of ports. We write $\texttt{Rule}$ to denote the set of all rules. Intuitively, a packet matches a rule if it satisfies pattern. A switch forwards a packet along $\texttt{ports}$ for the highest priority rule that matches.

Rules are added or deleted by a switch through a set of control messages $\text{CM} = \{\text{add}(r), \text{del}(r) \mid r \in \text{Rule}\}$. Additionally, the controller uses a barrier message $b$ to synchronize.

type client {
  $\texttt{Port} : \text{set of nat}$
  $\texttt{pq} : \text{multiset of packets}$
}

rule "$\texttt{send}(c, pkt)$"
  true $\Rightarrow$ send(c, pkt)
end

rule "$\texttt{recv}(c, pkt, pts)$"
  exist(pkt:c.pq, true) $\Rightarrow$ recv(c, pkt, pts)
end

Listing 2: Client

A client $c \in \text{Clients}$ is modeled as in List 2. It consists of a finite set $\text{Port}$ of ports and a packet queue $\text{pq} \in \mathbb{M}[\text{Packet}]$ containing a multiset of packets which have arrived at the client. We use (guarded) actions to model behaviors of clients. An action is written as "$\texttt{name guard} \Rightarrow \texttt{command end}$." Predicate $\texttt{exist}(i : X, \varphi)$ asserts that there is an element $i$ in the set (or multiset) $X$ such that the predicate $\varphi$ holds. Additionally, if $\texttt{exist}(i : X, \varphi)$ holds, then the variable $i$ is bound to an element of $X$ that satisfies $\varphi$ and can be used later in the command part. In each step, a client $c$ can (1) send a non-deterministically chosen packet $pkt$ along some ports (rule $\texttt{send}$), or (2) receive a packet $pkt$ from its packet queue and (optionally) send a multiset of packets $\text{pkt}$ on some ports (rule $\texttt{recv}$).

A switch $sw$ is modeled as in List 3. It consists of a set of ports, a flow table $ft \subseteq \text{Rule}$, a packet queue $\text{pq}$ containing packets arriving from neighboring nodes, a control queue $\text{cq}$ containing control messages or barriers from the controller, a forward queue $\text{fq}$ consisting of at most one pair $(pkt, \text{ports})$ through which the controller tells the switch to forward packet $pkt$ along the ports $\text{ports}$, and a boolean variable $\texttt{wait}$. Predicate $\texttt{noBarrier}(sw)$ asserts $\text{sw}.\text{cq}$ does not contain a barrier. Predicate $\texttt{bestmatch}(sw, r, pkt)$ asserts that $r$ is the highest priority rule whose pattern matches the packet $pkt$ in switch $sw$'s flow table.

Intuitively, a switch has a normal mode and a waiting mode determined by the $\texttt{wait}$ variable. When the switch is in the normal mode, as long as there is no barrier in its control queue, it can either attempt to forward a packet from its packet queue based on its flow table, or update its flow table according to a control message in its control queue. When the switch cannot find a matching rule in its flow table for a packet, it can initiate a request to the controller, change to the waiting mode, and wait for a forward message from the controller telling it how to forward the packet. Once it receives a forward message $(\text{pkt}, \text{pts})$ and there is no barrier in the control queue, it forwards the pending packet $pkt$ to the ports in $\text{pts}$, and changes back to the normal mode. If the control queue contains one or more barriers, the switch dequeues all control messages up to the first barrier from its control queue and updates its flow table.

type switch {
  $\texttt{Port} : \text{set of nat}$
  $ft : \text{set of rules}$
  $\texttt{pq} : \text{multiset of packets}$
  $\texttt{cq} : \text{list of barriers and}$
  $\text{multisets of control messages}$
  $\texttt{fq} : \text{set of forward messages}$
  $\texttt{wait} : \text{boolean}$
}

rule "$\texttt{match}(sw, pkt, r)$"
  $\text{sw}.\texttt{wait} \& \texttt{noBarrier}(sw) \&$
  exist(pkt:sw.pq, 
  exist(r:sw.ft, bestmatch(sw,r,pkt)) $\Rightarrow$
  match(sw,pkt,r)
end

Listing 3: Switch

A controller $\text{controller}$ is modeled as in List 4. It is a tuple $(CS, cs_0, cs, csN, \kappa, pktIn)$ where $CS$ is a finite set of control states, $cs_0 \in CS$ is the initial control state, $cs$ is the current control state, $\kappa$ is a finite request queue of size $\kappa \geq 1$ consisting of packets forwarded to the controller from switches, and $\text{pktIn}$ is a function that takes a packet $pkt$ and a control state $cs_1$, and returns a tuple $(\eta, (pkt, pts), cs_2)$ where $\eta$ is a function from $\text{Switches}$ to $(\mathbb{M}[\text{CM}] \cup \{b\})^*$.
is a forward message, and $c_{eq}$ is a control state. Intuitively, in each step, the controller removes a packet $pkt$ from $rq$ and executes $pktIn(pt, controller, cs)$). Based on the result $(\eta, (pkt, pts), cs')$, it sends back to the source of the packet the forward message $(pkt, pts)$ that specifies $pkt$ should be forwarded along $pts$, and goes to a new control state $cs'$. Further, for each switch $sw$ in the network it appends $\eta(sw)$ to $sw$'s control queue.

**Semantics of Software-defined Networks** The semantics of an SDN is given as a transition system. Let $\mathcal{N} = (Clients, Switches, \lambda, Packet, Rule, controller)$ be an SDN, where each component is as defined above.

A state $s$ of the SDN $\mathcal{N}$ is a quadruple $(\pi, \delta, cs, rq)$, where $\pi$ is a function mapping each client $c \in Clients$ to its packet queue $pq$ and $\delta$ is a function mapping each switch $sw \in Switches$ to a tuple $(pq, cs, ft, wait)$ consisting of its packet queue, control queue, forward queue, flow table, and the wait variable.

For a non-empty list $l = [x_1,x_2,\ldots,x_n]$, define $l.hd=x_1$, $l.lt=[x_2,\ldots,x_n]$, and $l[i]$ as the $t$-th element in $l$. Given two lists $l_1$ and $l_2$, let $l_1+l_2$ be the concatenation of $l_1$ and $l_2$. For two non-empty lists $l_1 = [x_1,\ldots,x_n]$ and $l_2 = [y_1,\ldots,y_m] \in \{\mathcal{M}[CM] \cup \{b\}\}^*$, define $l_1+l_2$ to be the list $[x_1,\ldots,x_{n-m},x_{n-m+1}\oplus y_1,\ldots,y_m]$, if $x_i \neq y_j$ and $y_i \neq x_j$; $l_1+l_2$ otherwise.

Given a flow table $ft$ and a list $l \in \{\mathcal{M}[CM] \cup \{b\}\}^*$, let $update(ft, l)$ be a procedure that updates $ft$ based on $l$ as follows. It dequeues the head of $l$ and sets $l \leftarrow l.lt$. If the head is a barrier $b$, then ignore it. If the head is a multiset $m$, then nondeterministically chooses a fetching order $o$ and based on $o$, removes a control message $m' \in m.cm() > 0$ from $m$. If $cm$ is add($r$), then add the rule $r$ to $ft$. If $cm$ is del($r$), then remove $r$ from $ft$. Keeps updating $ft$ based on $l$ until $m$ becomes empty. Repeats the above instructions on $l$ until $l$ becomes empty. Then it returns the resulting flow table $ft$.

For a function $f : X \rightarrow Y$, $x \in X$, and $y \in Y$, if $f[x/y]$ denotes the function that maps $x$ to $y$ and all $x' \neq x$ to $f(x')$.

Let $f[x_1/y_1 \rightarrow x_2/y_2 \rightarrow \ldots \rightarrow x_n/y_n]$ denote the function $f[x_1/y_1][x_2/y_2][\ldots][x_n/y_n]$. Given a subset $X' = \{x_1,\ldots,x_n\} \subseteq X$, let $f[fo\text{rce } x_i \in X' : x_i \rightarrow y_i]$ be the function $f[x_1/y_1][\ldots][x_n/y_n]$ where $1 \leq i \leq n$. Given a tuple $t = (f_1,\ldots,f_k)$, let $t.f_i$ be the field $f_i$, for $1 \leq i \leq k$. By abuse of notation, we write $t[f_i \rightarrow v]$ to be the tuple such that $t[f_i \rightarrow v].f_i = v$ and for any $j \neq i$, $t[f_i \rightarrow v].f_j = t.f_j$.

We define the following basic operations over $\delta$ and $\pi$:

1. $\delta'$ and $\pi'$ are the result of applying $\delta$ and $\pi$ to the following basic operations over $\delta$ and $\pi$:

2. $\delta'$ is the aggregate of $\delta$.

3. $\delta'$ is the result of applying $\delta$ and $\pi$ to the following basic operations over $\delta$ and $\pi$:

4. $\delta'$ is the result of applying $\delta$ and $\pi$ to the following basic operations over $\delta$ and $\pi$:

5. $\delta'$ is the result of applying $\delta$ and $\pi$ to the following basic operations over $\delta$ and $\pi$:

6. $\delta'$ is the result of applying $\delta$ and $\pi$ to the following basic operations over $\delta$ and $\pi$:

7. $\delta'$ is the result of applying $\delta$ and $\pi$ to the following basic operations over $\delta$ and $\pi$:

8. $\delta'$ is the result of applying $\delta$ and $\pi$ to the following basic operations over $\delta$ and $\pi$:
9) \( \alpha = \text{ctrl}(\text{pkt}, \text{cs}) \). Let \( \text{pktIn}(\text{pkt}, \text{cs}) = (\eta, \text{msg}, \text{cs}') \) and \( \text{sw} = \text{pkt.loc.n} \). Let \( \delta'' = \text{addFwdMsg}(\delta, \text{sw}, \text{msg}) \), and \( \delta' = \text{addCtrlCmd}(\delta'', \eta) \).

An atomic proposition \( p \in \text{AP} \) is an assertion over packet fields or over control states. Define an SDN specification as a safety property \( \Box \phi \) where \( \phi \) is a formula over \( \text{AP} \) and \( \Box \) is the "globally" operator of linear-temporal logic. The model checking problem for an SDN asks, given an SDN \( \mathcal{N} \) and an SDN specification \( \Box \phi \), if \( \mathcal{TS}(\mathcal{N}) \) satisfies \( \Box \phi \). For example, blocking SSH packets can be specified as \( \Box \neg \text{pktIn}(\text{pkt.loc.n} \in \text{Clients} \land \text{pkt.src} \in \text{Clients} \land \text{pkt.loc.n} \neq \text{pkt.prot} \Rightarrow \text{pkt.prot} \neq \text{SSH}) \).

### III. Optimizations

We now describe partial-order reduction and abstraction techniques that reduce the state space. These techniques use the structure of SDNs and, as we demonstrate empirically, are crucial in making the model checking scale to non-trivial examples. We state the correctness theorems; the proofs are in the technical report [11].

#### Partial Order Reduction

Let \( \mathcal{TS} = (\mathcal{S}, A, \rightarrow, s_0, \mathcal{AP}, L) \) be an action-deterministic transition system, i.e., \( s \xrightarrow{\alpha} s' \) and \( s \xrightarrow{\beta} s'' \) implies \( s' = s'' \). Given two actions \( \alpha, \beta \in A \) with \( \alpha \neq \beta \), and \( \beta \) are independent if for any \( s \in \mathcal{S} \) with \( \alpha, \beta \in A(s) \), \( \beta \in A(\alpha(s)) \), and \( \alpha(\beta(s)) = \beta(\alpha(s)) \). The actions \( \alpha \) and \( \beta \) are dependent if \( \alpha \) and \( \beta \) are not independent. An action \( \alpha \in A \) is a stutter action if for each transition \( s \xrightarrow{\alpha} s' \) in \( \mathcal{TS} \), we have \( L(s) = L(s') \).

For \( i \in \{1, 2\} \), let \( \mathcal{TS}_i = (\mathcal{S}_i, A_i, \rightarrow, s_{i0}, \mathcal{AP}, L_i) \) be transition systems. Infinite executions \( p_1 \) of \( \mathcal{TS}_1 \) and \( p_2 \) of \( \mathcal{TS}_2 \) are stutter-equivalent, denoted \( p_1 \equiv p_2 \), if there is an infinite sequence \( A_0 A_1 A_2 \ldots \) with \( A_i \subseteq \mathcal{AP} \), and natural numbers \( n_0, n_1, n_2, \ldots, m_0, m_1, m_2, \ldots \geq 1 \) such that \( \text{trace}(p_1) = \sum_{n_0} A_0 \sum_{n_1} A_1 \sum_{n_2} A_2 \ldots \) and \( \text{trace}(p_2) = \sum_{m_0} A_0 \sum_{m_1} A_1 \sum_{m_2} A_2 \ldots \).

\( \mathcal{TS}_1 \) and \( \mathcal{TS}_2 \) are stutter equivalent, denoted \( \mathcal{TS}_1 \equiv \mathcal{TS}_2 \), if \( \mathcal{TS}_1 \subseteq \mathcal{TS}_2 \) and \( \mathcal{TS}_2 \subseteq \mathcal{TS}_1 \), where \( \subseteq \) is defined by: \( \mathcal{TS}_1 \subseteq \mathcal{TS}_2 \) iff for all \( p_1 \in \text{Traces}(\mathcal{TS}_1) \) and \( p_2 \in \text{Traces}(\mathcal{TS}_2) \), \( p_1 \equiv p_2 \).

#### A. Barrier Optimization

Intuitively, barrier optimization uses the observation that for any state, we can always flush out control queues of switches until there are no barriers in them. This implies that after a control action is executed, one can immediately update flow tables of switches whose control queue has barriers added by the controller. Hence a control action and successive barrier actions can be merged. We prove its correctness by viewing it as an instance of partial order reduction.

For an SDN \( \mathcal{N} \), note that \( \mathcal{TS}(\mathcal{N}) \) is action-deterministic due to barrier actions. With different filtering orders, \( \text{barrier}(\text{sw}) \) may lead to multiple states. Define \( b(s, \text{sw}) \) as the number of transitions of the form \( s \xrightarrow{\text{barrier}(\text{sw})} s' \). Note that a barrier action from any \( s \) leads to at most \( 2^{|\text{Rule}|} \) states. Hence for each transition \( s \xrightarrow{\text{barrier}(\text{sw})} s_i \) where \( 1 \leq i \leq b(s, \text{sw}) \), we can append the action with the index

i.e., \( s \xrightarrow{\text{barrier}(\text{sw})} s_i \). In the following, we redefine the set \( \text{Barrier} = \{ \text{barrier}(\text{sw}) \mid s \in \text{Switches} \land 1 \leq i \leq 2^{|\text{Rule}|} \} \), and assume that \( \mathcal{TS}(\mathcal{N}) \) is action-deterministic by renaming barrier actions.

A switch \( s \) has a barrier iff there is a barrier in \( s \)'s control queue. A state \( s \) has a barrier, denoted \( \text{hash}(s) \), if some switch \( s \in \text{Switches} \) has a barrier in \( s \). Define the ample set for every state \( s \in \mathcal{TS}(\mathcal{N}) \) as follows: if \( s \) has a barrier, then \( \text{ample}(s) = \{ \text{barrier}(\text{sw}) \mid 1 \leq i \leq b(s, \text{sw}) \land \text{sw} \text{ has a barrier in } s \} \), that is, all barrier actions enabled in \( s \). If \( s \) does not have a barrier, then \( \text{ample}(s) = \{ s \} \).

Given \( \mathcal{TS}(\mathcal{N}) \), we now define a transition system \( \bar{\mathcal{TS}} = (\bar{\mathcal{S}}, \bar{\mathcal{A}}, \rightarrow, \bar{s}_0, \bar{\mathcal{AP}}, \bar{L}) \) where \( \bar{\mathcal{S}} = \mathcal{S} \) is the set of states, and the transition relation \( \rightarrow \) is defined as: if \( s \xrightarrow{\alpha} s' \) and \( \alpha \in \text{ample}(s) \), then \( s \xrightarrow{\alpha} s' \).

**Theorem 1:** Let \( \mathcal{TS}(\mathcal{N}) \) be an action-deterministic transition system. \( \mathcal{TS}(\mathcal{N}) \equiv \bar{\mathcal{TS}} \).

Intuitively, Theorem 1 holds because any barrier action is independent of other actions and is a stutter action. Hence for an infinite execution \( s_{\alpha_1} s_{\alpha_2} \ldots s_{\alpha_n} \) in \( \mathcal{TS}(\mathcal{N}) \) where \( s \) has a barrier and \( \alpha_i \) is not a barrier action for all \( 1 \leq i \leq n \), we can permute \( \text{barrier}(\text{sw}) \) forward until \( s \) and obtain a stutter-equivalent execution in \( \bar{\mathcal{TS}} \).

Since Theorem 1 holds, we can merge a control action and successive barrier actions into a single transition \( s \xrightarrow{\text{ctrl}(\text{pkt}, \text{cs})} s' \) where we define the new semantics of \( \text{ctrl}(\text{pkt}, \text{cs}) \) under the transition relation \( \rightarrow \). Formally, Let \( \eta \), \( \text{pkt}, \text{pts} \), \( \text{cs}' \) = \( \text{pktIn}(\text{pkt}, \text{cs}) \) and \( \text{sw} = \text{pkt.loc.n} \).

A. Barrier Optimization

**Theorem 2:** Given an SDN \( \mathcal{N} \) and a safety property \( \Box \phi \), \( \mathcal{TS}(\mathcal{N}) \) satisfies \( \Box \phi \) iff \( \mathcal{TS}_2 \) satisfies \( \Box \phi \).

**B. Client Optimization**

Given transition system \( \mathcal{TS}_2 = (\mathcal{S}_2, A_2, \rightarrow, s_0, \mathcal{AP}, L_2) \), we further reduce the state space by observing that any receive action of a client is a stutter action and is independent of other actions. Formally, we define \( \text{ample}(s) \) for each state \( s \in \mathcal{S}_2 \) as follows: if there is a client in \( s \) such that its packet queue is not empty, then \( \text{ample}(s) = \{ \text{recv}(c, \text{pkt}, \text{pts}) \mid \text{pkt is in } c.pq \text{ at } s \} \), that is, all receive actions enabled in \( s \). Otherwise, \( \text{ample}(s) = \{ s \} \).

We now define a transition system \( \mathcal{TS}_3 = (\mathcal{S}_3, A_3, \rightarrow, s_0, \mathcal{AP}, L_3) \) where \( \mathcal{S}_3 = \mathcal{S}_2 \).
A_3 = A_2, AP_3 = AP_2, L_3 = L_2, and where the transition relation \( \rightarrow_3 \) is defined as: if \( s \xrightarrow{\alpha} s' \) and \( \alpha \in \text{ample}(s) \), then \( s \xrightarrow{\alpha} s' \).

**Theorem 3:** (1) \( TS_2 \cong TS_3 \). (2) Given a safety property \( \Box \phi, TS_2 \) satisfies \( \Box \phi \) iff \( TS_3 \) satisfies \( \Box \phi \).

C. \((0, \infty)\) Abstraction

The \((0, \infty)\) abstraction bounds the size of packet queues and the multiset in each control queue. The idea is as follows. One can regard a multiset as a counter that counts the number of elements in it exactly. Instead, \((0, \infty)\) abstraction abstracts a multiset so that for each element \( e \), it either does not contain \( e \) (i.e. 0) or contains unboundedly many copies of \( e \) (i.e. \( \infty \)). Then the size of an abstracted multiset is bounded. Note that for any state \( s \) in \( TS_3 \), any switch’s control queue contains exactly one multiset. Hence, the abstraction bounds the length of control queues.

Let \( \mathbb{N}_\infty = \mathbb{N} \cup \{\infty\} \). The following lemma claims that \( TS_4 \) simulates \( TS_3 \), where the size of the request \( pq \) is added \( k \geq 1 \) times into a packet queue \( pq \), we set \( pq \) to \( \overline{pq} \oplus \overline{pkt_k} \), and \( s_5 \) whenever \( \eta(sw) \) is added into switch \( s_6 \)’s control queue \( cq \), we set \( cq \) to \( \overline{cq + \eta(sw)} \).

The following lemma claims that \( TS_4 \) simulates \( TS_3 \), which leads to Theorem 4.

**Lemma 4:** For any infinite initial execution \( s_0 \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \ldots \) in \( TS_3 \), there is an infinite initial execution \( t_0 \xrightarrow{\gamma_1} t_1 \xrightarrow{\gamma_2} \ldots \) in \( TS_4 \) such that for all \( i \geq 0 \), \( s_i = (\pi_i, \delta_i, cs_i, rq_i) \) and \( t_i = (\pi'_i, \delta'_i, cs'_i, rq'_i) \) satisfy the following condition: for all \( c \in \text{Clients}, \pi_i(c) \leq c \pi'_i(c) \) and for all \( sw \in \text{Switches}, \delta_i(sw).pq \leq \delta'_i(sw).pq, \delta_i(sw).cq \leq c \delta'_i(sw).cq, \delta_i(sw).fq = \delta'_i(sw).fq, \delta_i(sw).ft = \delta'_i(sw).ft, \) and \( \delta_i(sw).wait = \delta'_i(sw).wait, \) and \( cs_i = cs'_i, \) and \( rq_i = rq'_i. \)

**Theorem 4:** Given a safety property \( \Box \phi, TS_2 \) satisfies \( \Box \phi \) then \( TS_3 \) satisfies \( \Box \phi \) and \( TS_4 \) satisfies \( \Box \phi \).

D. All Packets in One Shot Abstraction

So far, a switch processes a single packet at a time. We can further reduce the reachable state space by forcing a switch to process all packets matched by some rule at a time. The intermediate states produced by successive match actions in a switch are removed. Let \( TS_5 = (S_4, A_4, \rightarrow_5, s_0, AP_4, L_4) \).

Define a transition system \( TS_5 = (S_5, A_5, \rightarrow_5, s_0, AP_5, L_5) \) where \( S_5 = S_4, AP_5 = AP_4, L_5 = L_4, A_5 \) is the union of the new “multiple” match actions and \( A_4 \) excluding the old “single” match actions, and \( \rightarrow_5 \) is defined as:

\[ s \xrightarrow{\alpha}_5 s' \]

and if \( pkt_{\text{lst}} = [pkt_1, \ldots, pkt_n] \) and \( r_{\text{lst}} = [r_1, \ldots, r_n] \)

\[ (s_{\text{match}}(sw.pkt_1, r_1), \ldots, s_{\text{match}}(sw.pkt_n, r_n)) \xrightarrow{\alpha}_5 s' \]

We prove \( TS_5 \) simulates \( TS_3 \). We define a relation \( R \subseteq S_4 \times S_5 \) such that \( ((\pi, \delta, cs, rq), (\pi', \delta', cs', rq')) \in R \) iff for all \( pkt \in \text{Packet}, \) for all \( c \in \text{Clients}, \pi(c)(pkt) = \infty \rightarrow \pi'(c)(pkt) = \infty \) and for all \( sw \in \text{Switches}, \delta(sw).pq(pkt) = \infty \rightarrow \delta'(sw).pq(pkt) = \infty, \delta(sw).cq = \delta'(sw).cq, \delta(sw).fq = \delta'(sw).fq, \delta(sw).ft = \delta'(sw).ft, \) and \( \delta(sw).wait = \delta'(sw).wait, \) and \( cs = cs', \) and \( rq = rq'. \)

**Theorem 5:** (1) The relation \( R \) is a simulation relation. (2) For a safety property \( \Box \phi, TS_5 \) satisfies \( \Box \phi \), then \( TS_4 \) satisfies \( \Box \phi \).

E. Controller Optimization

We consider a restricted class of SDNs in which the size \( \kappa \) of the controller’s request queue is one. Under this restriction, we can define a new transition system \( TS_6 \) that is stutter equivalent to \( TS_5 \) and has fewer reachable states. The idea is to observe that a no-match action is a stutter action and is independent of any actions before a corresponding control action is executed. Formally, given \( TS_5 = (S_5, A_5, \rightarrow_5, s_0, AP_5, L_5) \), we can define a new transition relation \( \rightarrow_6 \) inductively:

\[ s_0 \xrightarrow{\alpha}_6 s_1 \]

where a new action \( \text{nomatch}_\text{ctrl}(sw, pkt, cs) \) merges \( \text{nomatch}(sw, pkt) \) and \( \text{ctrl}(pkt, cs) \) actions. We define a transition system \( TS_6 = (S_6, A_6, \rightarrow_6, s_0, AP_6, L_6) \), where \( S_6 = S_5 \) is the set of states, \( A_6 \) is the union of all \( \text{nomatch}_\text{ctrl}(sw, pkt, cs) \) actions and \( A_5 \setminus \{\text{NoMatch} \cup \text{Ctrl}\}, AP_6 = AP_5, \) and \( L_6 = L_5. \)

**Theorem 6:** Given an SDN \( N \) where the size of the request queue of the controller is one, and a safety property \( \Box \phi \), (1) \( TS_6 \cong TS_5 \). (2) \( TS_5 \) satisfies \( \Box \phi \) iff \( TS_6 \) satisfies \( \Box \phi \).

IV. Implementation and Evaluation

Kuai\(^1\) is implemented on top of PReach [2], a distributed enumerative model checker built on Murphi. We model switches, clients, and the controller as concurrent Murphi processes which communicate using message passing, with the queues modeled as multisets. We manually abstract IP packets using predicates used in the controller. We implement \((0, \infty)\)-counter abstraction as a library on top of Murphi multisets.

Kuai takes as input topology information such as the number of switches, clients, and their connections, (manually) abstracted packets, and the controller code written as a Murphi process, and invariants written in Murphi syntax. We found it fairly straightforward to port POX [15] controllers due to the imperative features of Murphi. Murphi allows arbitrary first order logic formulas as invariants and it is easy to specify

\(^1\)The tool is can be downloaded at https://github.com/t-saideep/kuai
We now describe the benchmarks in detail.

**SSH** We run Kuai on the SSH controller from Listing 1. It finds the control message reordering bug in 0.1 seconds. By adding a barrier after line 15, Kuai proves the correctness in 6.4 seconds by exploring 13 states. In contrast, the unoptimized version explores over 2 million states.

<table>
<thead>
<tr>
<th>Program</th>
<th>Bytes/ state</th>
<th>w/o optimizations</th>
<th>w/ optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSH 2×2</td>
<td>394</td>
<td>2,283,527</td>
<td>23.52s</td>
</tr>
<tr>
<td>ML 3×3</td>
<td>320</td>
<td>9,109,456</td>
<td>89.99s</td>
</tr>
<tr>
<td>ML 6×3</td>
<td>748</td>
<td>23,926,202</td>
<td>604.07s</td>
</tr>
<tr>
<td>ML 9×2</td>
<td>1276</td>
<td>18,615,767</td>
<td>793.84s</td>
</tr>
<tr>
<td>FW(S) 1×2</td>
<td>332</td>
<td>2,110,986</td>
<td>26.89s</td>
</tr>
<tr>
<td>FW(M) 2×4</td>
<td>448</td>
<td>4,507</td>
<td>8.03s</td>
</tr>
<tr>
<td>FW(M) 3×4</td>
<td>560</td>
<td>512,439</td>
<td>55.06s</td>
</tr>
<tr>
<td>FW(M) 4×4</td>
<td>676</td>
<td>5,360,871</td>
<td>475.54s</td>
</tr>
<tr>
<td>RS 4×4</td>
<td>764</td>
<td>4998</td>
<td>6.60s</td>
</tr>
<tr>
<td>RS 4×5</td>
<td>764</td>
<td>590,570</td>
<td>82.82s</td>
</tr>
<tr>
<td>RS 4×6</td>
<td>764</td>
<td>5,112,013</td>
<td>327.39s</td>
</tr>
<tr>
<td>SIM 5×6</td>
<td>632</td>
<td>167</td>
<td>6.23s</td>
</tr>
<tr>
<td>SIM 5×8</td>
<td>632</td>
<td>167</td>
<td>6.34s</td>
</tr>
<tr>
<td>SIM 5×12</td>
<td>1108</td>
<td>167</td>
<td>6.85s</td>
</tr>
</tbody>
</table>

**Table I:** Experimental results. Omitted entries indicate that model checking did not terminate. The number X×Y in the Program column means that there are X switches and Y clients in the example.

We have evaluated Kuai on a number of real world OpenFlow benchmarks. The experiments were performed on a cluster of 5 Dell R910 rack servers each with 4 Intel Xeon X7550 2GHz processors, 64 × 16GB Quad Rank RDIMMs memory and 174GB storage. Our experiments had access to a total of 150 cores and had access to 4TB of RAM.

Table I shows a summary of experimental results and compares the scalability of model checking with increasing distribution. We noticed that the performance of the distributed model checker plateaued around 70 Erlang processes on these and other large examples. Thus, times (in table I) are provided for configurations that use 70 Erlang processes. As introduced abstractions, it is possible that we get false positives. We verified the existence of all bugs reported by Kuai manually and there were no false positives.

Besides the table, we plot the MAC learning example in Figure 3, which shows how significantly our optimization techniques reduce the state space. Though we still suffer from the state-space explosion problem, our optimizations delay it and enable us to verify SDNs with much larger configurations.

We now describe the benchmarks in detail.

**MAC Learning Controller (ML)** This is based on the POX [15] implementation of the standard ethernet discovery protocol. We checked there are no forwarding loops (similar to [17]), i.e., a packet should not reach a switch more than once. Packets are augmented with a bit for each switch which gets set when the switch processes that packet. The invariant is specified using these visit-bits (called reached): □ ∀sw ∈ Switches. ∀pkt ∈ sw.pq. (¬pkt.reached(sw)).

A cycle in the topology will lead to forwarding loops as the controller does not compute the minimum spanning tree. We discover the bug in a cyclic topology of 3 switches 3 clients in 0.47 seconds. We re-ran the example on a topology containing the minimum spanning tree of the original cyclic topology and the tool is able to prove that there were no forwarding loops in 6.39 seconds. We scale the example by adding more switches. We notice that while the verification on topology with 9 switches and 2 clients has fewer states than the one with 6 switches and 3 clients, each state in the latter case is bigger than the former and hence the memory and communication overheads are higher.

**Single Switch Firewall (FW(S))** This is based on an advanced GENI assignment [7] on building an OpenFlow based firewall. The controller takes as input a simple configuration file which is a list of tuples of the form (client1, port1, client2, port2). This specifies that packets originating from client1 on port1 can be forwarded to client2 on port2. We abbreviate the tuples as (client1: port1 → client2: port2). Any flow not explicitly allowed is forbidden. The flows are uni-directional and the above flow will reject traffic initiated by client2 on port2 towards client1 on port1. However, once client1 initiates a flow, the firewall should allow client2 to reply back, making the flow bi-directional until client1 closes the connection.

The naive implementation of the controller is as follows: on receiving a packet (c1: p1 → c2: p2), check if there is a tuple matching the flow in the policy. If it does, add rules (c1: p1 → c2: p2) and (c2: p2 → c1: p1) and forward the packet to c2. Otherwise add a rule to drop packets of the form (c1: p1 → c2: p2). The invariant to verify here is to ensure the policy of the firewall, i.e., a packet from c1: p1 should be forwarded to c2: p2 if and only if (c1: p1 → c2: p2) exists in the firewall policy or if (c2: p2 → c1: p1) exists in the policy and c2 has already initiated the corresponding flow. The following formula specifies that allowed packets should not be dropped: □∀p ∈ Packet. on_dropped(p) ⇒ ¬flows[p.src][packet.src_port][packet.dest][p.dest_port], where on_dropped(p) is set if a packet-drop transition is fired on packet p (and reset at the beginning of every transition). flows is an auxiliary variable in the controller which keeps

Fig. 2: Verification time vs processes ◦ ML 9×2 □ ML 6×3 □ FW(M) 4×4 safety properties. Kuai compiles them into a single Murphi file and the model checking effort is then distributed across several machines using PReach. Finally the output of the tool is an error trace if the program invariant fails, or success otherwise.

Fig. 3: State space of MAC learning controller: Δ: optimized, ◦ unoptimized
track of allowed flows based on the firewall policy and initiating client.

We ran the experiment on a topology with 2 clients and a firewall. We found an interesting bug in our implementation which is caused by not assigning proper priorities to rules. For example, when \((c1: p1 \rightarrow c2: p2)\) is present in the policy but not \((c2: p2 \rightarrow c1: p1)\), the rule to drop flows should have a lower priority than the rules to allow flows. Otherwise, the following bug would occur. If \(c2\) initiates the flow \((c2: p2 \rightarrow c1: p1)\) then the controller adds a rule to drop packets matching that flow. Later on, if \(c1\) initiates \((c1: p1 \rightarrow c2: p2)\) and the controller adds the corresponding rules to allow the flow on both directions, the switch now has two conflicting rules of the same priority. One to allow and the other to drop \((c2: p2 \rightarrow c1: p1)\). The switch may non-deterministically choose to drop the packet. Once we fixed the bug, the tool could prove the invariant in 5.45 seconds.

**Multiple Switch Firewalls (FW(M))** We extend the above example to include multiple replicated firewalls for load balancing. We now allow the clients to send packets to all of these firewalls. We augment the implementation of the single switch controller to add the same rules on all firewalls. However, this implementation no longer ensures the invariant in the multi-switch setting.

Consider the case with two firewalls, \(f1\) and \(f2\). The tool reports the following bug: \(c1\) initiates \((c1: p1 \rightarrow c2: p2)\) on firewall \(f1\). The controller adds the corresponding rules to allow flows in both directions to \(f1\) and \(f2\) but only sends a barrier to \(f1\). Now \(f2\) delays the installation of \((c2: p2 \rightarrow c1: p1)\) and \(c2\) replies back to \(c1\) through \(f2\) which forwards the packet to the controller. The controller then drops the packet.

The fix here is to add the rules along with barriers on all switches and not just the switch from which the packet originates. With this fix the tool is able to prove the property in 8 seconds. In order to test the scalability, we tested the tool on increasing number of firewalls in the topology.

**Resonance (RS)** Resonance [13] is a system for ensuring security in large networks using OpenFlow switches. When a new client enters the network, it is assigned a registration state and is only allowed to communicate with a web portal. The portal either authenticates a client by sending a signal to the controller (and the controller assigns the client an authenticated state), or sets the client to quarantined state. In the authenticated state, the client is only allowed to communicate with a scanner. The scanner ensures that the client is not infected and sends a signal to the controller and lets the controller assign it an operational state. If an infection is detected, it is assigned a quarantined state. The clients in operational state are periodically scanned and moved to the quarantined state if they are infected. Quarantined clients cannot communicate with other clients.

In our model, the web portal non-deterministically chooses to authenticate or quarantine a client and the scanner non-deterministically marks a client operational or quarantined. We check the invariant that packets from quarantined clients should not be forwarded: \(\forall p \in \text{Packet. on\_forward}(p) \Rightarrow (\text{state}(p,\text{src}) \neq \text{Quarantined})\). Similar to on_dropped, on_forward is set when packet-forward transition is fired and reset before the beginning of every transition. The controller follows the Resonance algorithm [13].

We ran the experiment on a topology of two clients, one portal, one scanner and four switches. The topology is the same as in Figure 2 of [13] without DHCP and DNS clients. Kuai proves the invariant in 6.6 seconds. We scale up the example by increasing the number of clients.

**Simple (SIM)** Simple [16] is a policy enforcement layer built on top of OpenFlow to ensure efficient middlebox traffic steering. In many network settings, traffic is routed through several middleboxes, such as firewalls, loggers, proxies, etc., before reaching the final destination. Simple takes a middlebox policy as input and translates this to forwarding rules to ensure the policy holds. The invariant ensures that all source packets to a client will be received and forwarded by the middleboxes specified in a given policy before the packet reaches its destination.

We ran the experiment on a topology of two clients, two firewalls, one IDS, one proxy and five switches (see Figure 1 of [16]). Kuai can prove the invariant in 6.48 seconds.

We scale up the example by fixing the destination client and increasing the number of source clients that can send packets to it. Because of our “all packets in one shot” optimization (section III-D), no matter how many packets get queued initially, they are all forwarded in lock-step as the controller forwarding rule applies to all incoming packets.

### References


