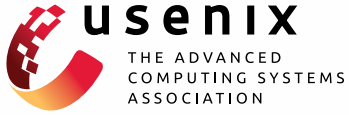


Kinetic: Verifiable Dynamic Network Control

Hyojoon Kim, *Georgia Institute of Technology;* **Joshua Reich,** *AT&T Labs–Research;*
Arpit Gupta, Muhammad Shahbaz, and Nick Feamster, *Princeton University;*
Russ Clark, *Georgia Institute of Technology*

https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/kim

This paper is included in the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15).

May 4–6, 2015 • Oakland, CA, USA

ISBN 978-1-931971-218

Open Access to the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15) is sponsored by USENIX

Kinetic: Verifiable Dynamic Network Control

Hyojoon Kim[‡], Joshua Reich[*], Arpit Gupta[†]
Muhammad Shahbaz[†], Nick Feamster[†], Russ Clark[‡]
[‡]*Georgia Tech* [*]*AT&T Labs – Research* [†]*Princeton University*
http://kinetic.noise.gatech.edu/

Abstract

Network conditions are dynamic; unfortunately, current approaches to configuring networks. Network operators need tools to express how a network's data-plane behavior should respond to a wide range of events and changing conditions, ranging from unexpected failures to shifting traffic patterns to planned maintenance. Yet, to update the network configuration today, operators typically rely on a combination of manual intervention and ad hoc scripts. In this paper, we present Kinetic, a domain specific language and network control system that enables operators to control their networks dynamically in a concise, intuitive way. Kinetic also automatically verifies the correctness of these control programs with respect to user-specified temporal properties. Our user study of Kinetic with several hundred network operators demonstrates that Kinetic is intuitive and usable, and our performance evaluation shows that realistic Kinetic programs scale well with the number of policies and the size of the network.

1 Introduction

Network conditions are always changing. Traffic patterns change, hosts arrive and depart, topologies change, intrusions occur, and so forth. Despite the fact that many of these changes are predictable—and, in some cases, even planned—an operator's *control* over the network remains relatively static. In response to changing conditions, network operators typically manually change low-level network configurations. Our previous study of network configuration changes found that a campus network may experience anywhere from 1,000 to 18,000 changes per month [20]. Although tools like Puppet [27] and Chef [3] can automate some network device configuration tasks, this level of automation is still relatively hands-on and error-prone.

To underscore the importance of this problem, we analyzed acceptable use policies from more than 20 campus networks (many of which are publicly available [22]) and also surveyed network operators about their experience with existing tools for implementing these kinds of policies. These policies are written in English and typically express how the network's forwarding behavior should change in response to changing network conditions. For example, the University of Illinois's network use policy has an unrestricted class, and four restricted classes of traffic shaping; a user's traffic is downgraded into different classes based on their past usage over a 24-hour sliding window. Such policies sound simple enough when expressed in prose, but in fact they require complex instrumentation and "wrappers" that dynamically change low-level network configuration. Network operators currently have no concise way to express these functions, nor do they have any way of checking whether their changes will result in the intended behavior. In a recent survey we conducted that included several hundred network operators, 89% of respondents said that they could not be certain that the changes they made to network configuration would not introduce new bugs.

Software-defined networking (SDN) is a powerful approach to managing computer networks [11] because it provides network-wide visibility of and control over a network's behavior; the Frenetic [13] family of languages provides higher-level abstractions for expressing network control. These languages are embedded in general-purpose programming languages (specifically, OCaml and Python), which makes it possible to write control programs that can respond to arbitrary events. Yet these languages do not provide intuitive abstractions for *automating* changes to network policy in response to dynamic conditions, nor do they make it possible to *verify* that these changes will match the operator's requirements for how network behavior should react to changing network conditions.

To address these problems, we present Kinetic, a domain specific language (DSL) and SDN controller that enables writing network control programs that capture responses to changing network conditions in a concise, intuitive, and verifiable language. Kinetic provides a structured language for expressing a network policy in terms of finite state machines (FSMs), which both concisely capture dynamics and are amenable to verification. States correspond to distinct forwarding behavior, and events trigger transitions between states. Kinetic's event handler listens to events and triggers transitions in policy, which in turn update the data plane.

Kinetic makes it possible to *verify* that changes to network behavior conform to a higher-level specification of correctness. For example, a network operator might want to prove that a control program would never allow a host access to certain parts of the network once an intrusion has been detected. Ongoing work has devoted much attention to verification of the network's data plane; tools such as VeriFlow [19] and HSA [18] can determine, for example, whether the forwarding table entries in a network's switches and routers would result in persistent loops or reachability problems. However, *these tools only operate on a snapshot of the data plane*; they do not allow operators to reason about network *control programs*, or how network control would change in response to various events or changes in network conditions. They do not provide any way for a network operator to find errors in the control programs that install erroneous data-plane state in the first place. Kinetic's focus on automating and verifying the control plane is complementary to this previous work. Kinetic's use of computation tree logic (CTL) [6]—and its ability to automatically verify policies with the NuSMV model checker [4]— can allow network operators to verify the dynamic behavior of the controller before the control programs are ever run.

One significant challenge we faced when designing Kinetic is the potential for state explosion in Kinetic programs, due to the large number of hosts, flows, network events, and policies. A naïve encoding of dynamic policies in an FSM would result in an exponential number of states, even for simple programs because every flow, with all possible combination of fields (*e.g.*, src/dst IP, src/dst MAC, etc), can have its own state. To control this state explosion, Kinetic introduces an abstraction called a *Located Packet Equivalence Class (LPEC)*, through which a programmer can specify a division of the flow space and map an independent copy of an FSM (FSM instance) to each class of flow space. Using LPECs, a programmer can define groups of flows that should always map to same FSM instances (*e.g.*, all flows from the same source MAC address). Thus, each defined group of flows will be in the same state. Additionally, because Kinetic is itself based on Pyretic (a Python-based SDN control language in the Frenetic family) [25], Kinetic inherits Pyretic's language and runtime features. Specifically, Kinetic uses Pyretic's composition operators to express larger FSMs as multiple smaller ones that correspond to distinct network tasks (*e.g.*, authentication, intrusion detection, rate-limiting). Applying Pyretic's composition operators to independent Kinetic FSMs and classic product construction of automata [10] (combining multiple FSMs with union or product) greatly simplifies the construction of Kinetic's FSM expressions and allows the FSM-based policies to scale.

We evaluated two aspects of Kinetic: (1) its usability, in terms of both conciseness and operators' facility with expressing realistic network policies; and (2) its performance, in terms of its ability to efficiently compile network policies into flow-table entries, particularly as the number of policies, the size of the network, and the rate of events grow. We conducted a user study with Kinetic of more than 650 participants, many of whom were network operators with no prior programming experience; most found Kinetic quite accessible: 79% thought that configuring the network with Kinetic was easier than current approaches, and 84% thought that Kinetic makes it easier to verify network configuration than existing alternatives.

Kinetic is open-source and publicly available; the project webpage provides access to the source code, a tutorial on Kinetic, and all of the code for the experimental evaluation [21]. The system has been used by SDN practitioners [14] and has served as the basis for projects and assignments in several university courses, as well as in a Coursera [8] course, where it has been used by thousands of students over the past two years.

Profession		Experience (years)		# Users in Network	
Operator	216	1	32	1–10	156
Developer	251	1–5	310	10–100	137
Student	123	5–10	187	100–1,000	136
Vendor	80	10–15	150	1,000–10,000	118
Manager	69	15–20	122	> 10,000	322
Other	138	> 20	73		
Total	877		874		869

Table 1: *Demographics of participants in the Kinetic user study. We asked these participants about their experiences configuring existing networks, as well as their experiences using Kinetic. Section 5 discusses the participants' experience with Kinetic. Not all participants answered every question.*

2 Motivation and Background

To motivate the need for Kinetic, we present the results of a survey of network operators about problems automating and verifying network configuration. We then present background on Pyretic, the language on which Kinetic is based; and on model checking and computation tree logic, which we use to design Kinetic's verification engine.

2.1 Motivation: Network Operator Survey

To gain a better understanding of the extent to which network operators have to change their network configurations, as well as their level of confidence in their changes, we conducted an institutional review board (IRB)-approved survey of more than 800 participants, concerning their experience with configuring existing networks, as part of a Coursera course on software-defined networking that we offer [8]. Table 1 summarizes the demographics of the participants: about 870 students completed the survey, 216 of whom were full-time network

operators. The majority of the students who completed the assignment and survey had more than 5 years of experience in networking, and many had more than 15 years of experience. More than 200 of the students had experience with networks of more than 1,000 devices, and more than 300 of the students had experience with networks with more than 10,000 users. Most of the participants had the most experience with campus or enterprise networks.

The responses we received demonstrate a clear need for better tools for automating and verifying network control. Nearly 20% of participants said that they must change their network configurations more than once a day. The most common causes of changes were provisioning, planned maintenance, and updates to security policies—exactly the types of configuration changes that we aim to automate with Kinetic. More strikingly, 89% of respondents indicated that they were never completely certain that their changes to the configuration would not introduce a new problem or bug, and 82% were concerned that the changes would introduce problems with *existing* functionality that was unrelated to the change. The two most common aspects of configuration that operators wanted to see *automated* were correctness testing (37%) and quality of service and performance assurances (24%). The two most common aspects of configuration that participants wanted to see *verified* were general correctness problems (37%) and security properties (26%). We asked these same participants to write programs in Kinetic and other SDN controllers; we discuss the results of that part of our user study in Section 5.1.

2.2 Background: Pyretic and CTL

Pyretic. To develop a language for expressing control dynamics that is both concise and easy to use, we based the Kinetic language on Pyretic [25], a Python-embedded domain-specific programming language for writing SDN control programs. It encodes network data-plane behavior in terms of policy functions that map an incoming "located packet" (*i.e.*, a packet and its location) to an outgoing set of located packets. Pyretic has a `policy` variable that determines the actions that the control program applies to incoming packets (*e.g.*, filtering, modification, forwarding). Pyretic ultimately compiles policies to OpenFlow-based switches. Pyretic's composition operators provide straightforward mechanisms for composing multiple distinct policies into a single coherent control program. Pyretic's parallel composition operator (+) makes a copy of the original packet and applies the corresponding policies to each copy in parallel. Sequential composition (>>) applies policies to a packet in sequence, so that the second policy is applied to the packet that is the output of the first policy. Pyretic is extensible, and its support for composing distinct policies and dynamically recompiling flow-table entries whenever the `policy` variable is updated are useful features for Kinetic. Still, the language itself does not provide a framework for writing concise, intuitive policies that respond to changing conditions, which is Kinetic's goal.

Operator	Meaning
	(Quantifiers over Groups of Paths)
A ϕ	ϕ holds for all possible paths from the current state.
E ϕ	There exists a paths from the current state where ϕ holds.
	(Quantifiers over a Specific Path)
X ϕ	ϕ holds for ne**X**t state.
F ϕ	ϕ eventually holds sometime in the **F**uture.
G ϕ	ϕ holds for all current and following states, **G**lobally.
ϕ U ψ	ϕ holds at least **U**ntil ψ.

Table 2: *Computation tree logic (CTL) operators.*

Model checking. We wanted to design Kinetic so that policies were not only easy to automate, but also easy to verify. To do so, we applied a model checking framework developed by Clarke and Emerson [5, 6] and subsequently refined by McMillan [24]. Model checking can guarantee that a finite state machine (FSM) satisfies certain properties that are expressed in different types of logics; this feature makes FSMs a logical choice for expressing Kinetic policies. One such logic is computation tree logic (CTL), a branching-time logic that represents time as a tree structure. The initial state of an FSM is the root, and each node represents a different future state. A path through the tree represents an execution path of the FSM. CTL allows the expression of various types of temporal logic statements, such as those expressed in Table 2. NuSMV is a widely used symbolic model checker for FSMs [4]. The Kinetic compiler automatically translates Kinetic programs into an SMV model, which can be tested against various CTL-based assertions.

3 Kinetic by Example

We illustrate various features of Kinetic by way of example programs. All of the examples that we present in this section are verifiable; we defer a discussion of verification, as well as the details of the Kinetic language and runtime, to Section 4. We have selected examples that demonstrate the design features of Kinetic; the Kinetic Github repository has more examples [21].

Kinetic programs capture control dynamics with a finite state machine (FSM) abstraction. To illustrate this abstraction, we start with a simple example involving intrusion detection. Although FSMs are intuitive, representing all possible network states in a monolithic FSM would result in state explosion; the second and third examples illustrate two abstractions that address this challenge: Located Packet Equivalence Classes (LPECs) and FSM composi-

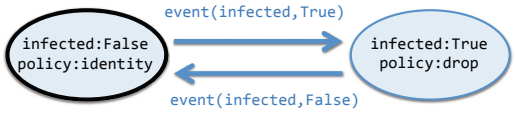


Figure 1: *Intrusion detection FSM.*

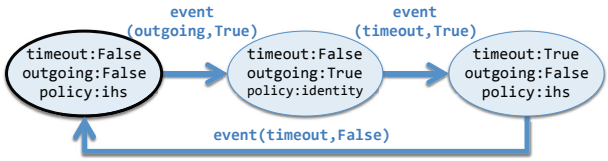


Figure 2: *Stateful firewall FSM.*

tion. Finally, to show Kinetic's generality, we present a MAC learning switch implementation.

3.1 Capturing Dynamics

We begin with a simple dynamic policy involving intrusion detection. Suppose that a network operator wants the network to drop all packets to and from a host once it receives an event indicating that the host is infected (*e.g.*, from an intrusion detection system). Kinetic allows operators to concisely express these dynamics with finite state machines that determine how a policy should evolve in response to events such as intrusions. We chose FSMs as the basic abstraction for expressing Kinetic programs because (1) they intuitively and concisely capture control dynamics in response to network events; and (2) their structure makes them amenable to verification.

In this example, each host would have a single state variable, `infected`. When `infected` is false, the controller applies Pyretic's `identity` (allow) policy for traffic from that host; when it is true, the controller applies Pyretic's `drop` policy for the host's traffic. Figure 1 shows this logical FSM. To support verification, the actual specification of the FSM for this policy is slightly more complicated; we expand on this example in Section 4.2.

3.2 Capturing State for Groups of Packets

Defining FSMs in Kinetic has the potential to create state explosion, since dynamic policies must be defined over a state space that is exponential in the number of hosts and flows (and possibly other aspects of the network). For example, consider the previous example, a two-state FSM indicating whether a host is infected. If the network has N hosts, then representing the state of the network requires an FSM with 2^N states, which is intractable, particularly as the size of the network and the complexity of policies grow. Instead of directly encoding an FSM that explicitly encodes all variable values, Kinetic encodes a single generic FSM that can be applied to any given group of

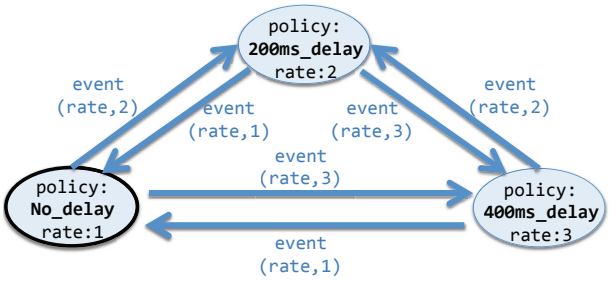


Figure 3: *Data usage-based rate limiter FSM.*

packets (*e.g.*, all packets from the same host, in the case of the previous example). Each group of packets has a separate FSM instance; packets in the same group will always be in the same state. We call such a group of packets a *located packet equivalence class (LPEC)*.

To illustrate the use of LPECs, we describe the implementation of a stateful firewall that implements a common security policy. Figure 2 shows the Kinetic representation of the policy. This program always allows outbound traffic, but blocks inbound traffic unless the traffic flow is in response to corresponding outbound traffic for that flow. For example, if internal host ih_1 pings external host eh_2 then packets sent from eh_2 should be allowed back through the firewall until a certain timeout occurs, but only if ih_1 is the destination.

The firewall's initial state, in the left of the figure, shows the policy, `ihs`, which is a filter policy matching all traffic whose source address in the set of internal hosts. A Pyretic-encoded query collects outbound packets from hosts in `ihs` and produces (`outgoing, True`) event. This triggers the update of the `policy` variable to `identity` (indicating that traffic is now allowed), and `outgoing` is reset. The `timeout` event is provided by Kinetic event driver. After certain amount of time (*e.g.*, five seconds), a (`timeout, True`) event is invoked unless another outgoing packet is seen within the timeout. The program should regard inbound and outbound flows between the same pairs of endpoints with the same state, and the programmer should not have to explicitly encode state for every pair of endpoints. To implement such a policy, the programmer can define an LPEC to correspond to a distinct source-destination IP address pair:

```
def lpec(pkt):
  h1 = pkt['srcip']
  h2 = pkt['dstip']
  return (match(srcip=h1,dstip=h2) |
          match(srcip=h2,dstip=h1) )
```

3.3 Composing Independent Policies

Many aspects of network state are logically independent. For example, whether a host has authenticated is independent of whether it is infected or whether it has exceeded

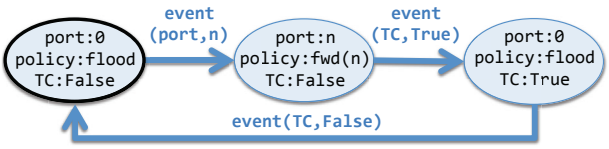


Figure 4: *MAC learner FSM.*

a usage cap. This independence allows a programmer to represent the overall network state as a product automaton that can be decomposed in terms of simpler *tasks*, where each task has simpler (and smaller) FSMs. This example shows the composition of four independent network tasks.

In our survey of campus network policies, we found nearly 20 university campuses [22] that implemented some form of usage-based rate-limiting (*e.g.*, [7]). Network operators currently implement these policies using low-level scripts that interact with monitoring devices. Kinetic provides intuitive mechanisms for implementing such a policy. Figure 3 illustrates the FSM for a usage-based rate limiter, which forward traffic with different delays depending on the user's historical data usage patterns. By default, traffic is forwarded with no delay; depending on the events that the controller receives concerning usage, the controller may institute a policy that introduces additional delay on user traffic. (OpenFlow 1.0 does not support traffic shaping, so we use variable delay as an illustrative example; Kinetic could be coupled with controllers that support later versions of OpenFlow that can do traffic shaping.)

Naturally, a real network would not only have policies involving quality-of-service, but also other policies, such as those relating to authentication and security. For example, a control program might first check whether a host is authenticated, either through a Web login or via 802.1X mechanism. Subsequently, the host's traffic might be subject to an intrusion detection policy that allows traffic by default but blocks the traffic if an infection event occurs. Finally, it might be sequentially composed with the rate-limiting policy above, yielding the resulting policy:

```
(web_auth + 802.1X_auth) >> ids >>
    rate_limiter
```

To verify this program, Kinetic generates a single FSM model for input to a model checker. Thus, programmers can write CTL specifications for the resulting composed policy, not only for individual policies. For example, a logic statement involving the combination of policies such as "If a host is authenticated either by the web authentication system or with 802.1X and is not infected, the resulting policy should never drop packets" can be verified with a single CTL assertion, as shown in Table 3. (Section 4.4 discusses verification in more detail.)

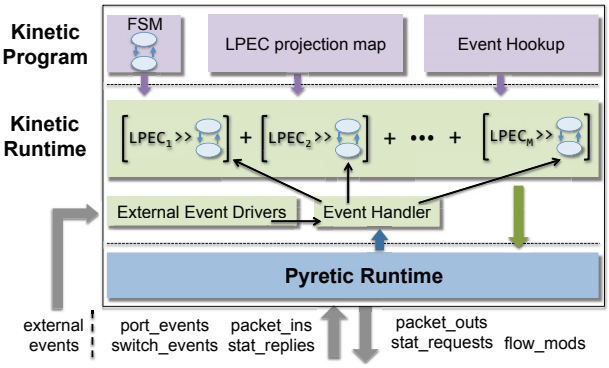


Figure 5: *Kinetic architecture.*

3.4 Handling General Event Types

Figure 4 shows a Kinetic FSM for a MAC learner that responds to both packets from hosts and topology changes. Although the implementation of a MAC learning switch is just as simple in other languages (indeed, it is the "canonical" reference program for SDN controllers), we present this example to illustrate that Kinetic programs can handle a variety of event types, including packet arrivals.

This program responds to two different types of events: `TC` (topo_change) and `port` events. The `TC` event is a built-in event that is invoked automatically whenever a topology change occurs. In Kinetic, programs can register and react to this built-in event. The `port` events are generated by a Pyretic query that collects the first packet for each (switch,srcmac) pair. The values of `policy` are defined by that of `port`: the value is `flood` when `port` is 0, and `fwd(n)` when `port=n`. Initially `port` is 0 (indicating the port has not yet been learned), and `TC` is `False`. When a `(port,n)` event arrives, which is invoked by the Pyretic runtime when it sees a packet from an unseen host, a transition occurs, setting the port to the value learned and the policy to unicast out that port. The MAC learner then unicasts packets to the appropriate hosts until a topology change occurs, triggering the transition to the right-most state in which `TC` is `True`, resulting in flooding for packets corresponding to that LPEC (*i.e.*, switch-source MAC address pair).

4 Kinetic Design & Implementation

We describe the details of Kinetic's architecture, language, runtime, and verification engine.

4.1 Architecture

We now describe the Kinetic system architecture, including the design of the Kinetic programming language. Figure 5 shows the Kinetic architecture, which is built on the Pyretic runtime. At the highest level, a Kinetic program

Kinetic	K ::= P | FSMPolicy(L,M) | K + K | K >> K L ::= *f : packet -> F* M ::= FSMDef([*var_name*=W]) W ::= VarDef(*type*,*init_val*,T) T ::= [case(S,D)] S ::= D==D | S & S | (S | S) | !S D ::= C(*value*) | V(*var_name*) | event
Pyretic	P ::= Dynamic() | N | P + P | P >> P
Static Pyretic	N ::= B | F | modify(h=v) | N + N | N >> N F ::= A | F & F | (F | F) | ~F A ::= identity | drop | match(h=v) B ::= FwdBucket() | CountBucket()

Figure 6: *The Kinetic language grammar.*

has three parts: (1) a finite state machine (FSM) specification; (2) a specification of portions of flow space that are always in the same state in any given FSM (an LPEC); and (3) mechanisms for incorporating external events that could change the state of any given LPEC's FSM.

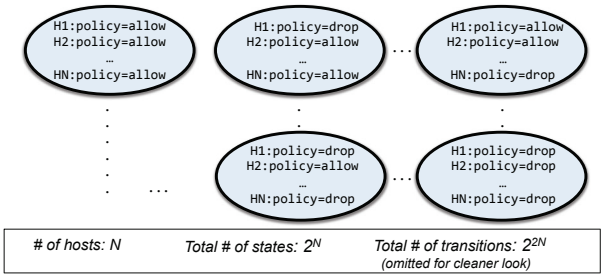
Kinetic instantiates copies of programmer-specified FSMs (one per LPEC); the Kinetic event handler sends incoming events, which can arrive either from external event hookups or from the Pyretic runtime (*e.g.*, in the case of certain types of events such as incoming packets), to the appropriate FSMs. Kinetic FSMs register with one or more event drivers and update their states when new events arrive, responding to incoming events that may be processed by those drivers. Kinetic supports both native events and generic JSON events. Because Kinetic is embedded in Pyretic, these functions can be executed using Pyretic's runtime. We use the Pyretic runtime to exchange OpenFlow messages with the network switches; we also use the Pyretic runtime to handle certain types of events, such as those related to either network topology or traffic.

4.2 Language and Abstractions

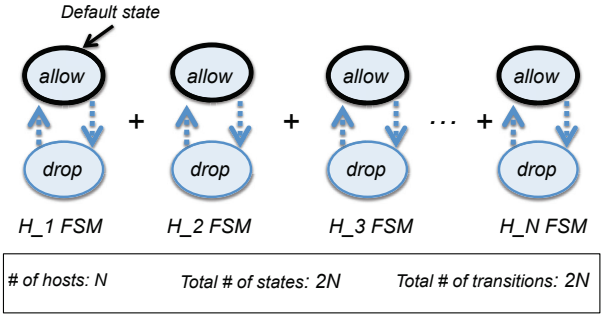
We offer a complete description of the language and then discuss LPECs and FSM composition in more detail.

4.2.1 Language Overview

Figure 6 defines the Kinetic language, which extends Pyretic (**P**). Pyretic has *bucket* policies (notated by **B**) which collect packets and count packet statistics, respectively; *primitive filters* (**A**) and *derived filters* (**F**) that allow only matching packets through; and *static policies* (**N**). Static policies include buckets, filters, the `modify` policy, and the combination of these via parallel and sequential composition. `Dynamic` generates a stream of static policies and can be combined with other policies in parallel or sequence.



(a) *Explicit encoding is exponential in N.*



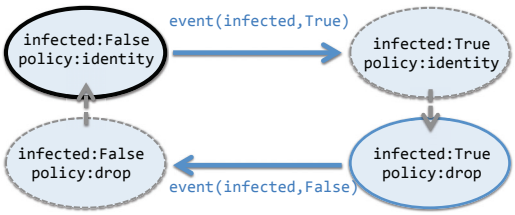
(b) *Decomposing to N LPEC FSMs.*

Figure 7: *Reducing state explosion using an LPEC FSM.*

Kinetic extends the Pyretic DSL with a subclass of `Dynamic`—`FSMPolicy`—which takes two arguments: an LPEC projection map (**L**) and an FSM description (**M**). The LPEC projection map takes a packet and returns a filter policy. The FSM description is set of assignments from a variable name to a variable definition (**W**). Each variable is defined by its type, initial value, and associated transition function (**T**). Each transition function is a list of cases, each of which contains a test (**S**) and an associated basic value (**D**) to which this corresponding state variable will be set, should this case be the first one in which the test is true. Tests are the logical combination of other tests (using *and*, *or*, *not*) or equality comparison between basic values. Finally, basic values are constants (**C(value)**), state variables (**V(variable_name)**), and events (**event**).

4.2.2 Located Packet Equivalence Classes

Recall from Section 3 that an LPEC allows an operator to encode a generic FSM for groups of packets (*e.g.*, all packets with the same source MAC address). Each distinct LPEC will have its own FSM instance, and the group of packets in each LPEC will be in the same state. Because each LPEC refers disjoint sets of packets, their FSMs (and corresponding policies) can be maintained independently, thus allowing their policies to be encoded in parallel. This mechanism allows the programmer to avoid explicit encoding of all combinations of network states (as shown in Figure 7a) and instead express each LPEC's FSM in-



(a) *Actual implementation of the Kinetic FSM.*

```
1  @transition
2  def infected(self):
3    self.case(occurred(self.event),self.event)
4
5  @transition
6  def policy(self):
7    self.case(is_true(V('infected')),C(drop))
8    self.default(C(identity))
9
10 self.fsm_def = FSMDef(
11   infected=FSMVar(type=BoolType(),
12                   init=False,
13                   trans=infected),
14   policy=FSMVar(type=Type(Policy,{drop,identity}),
15                 init=identity,
16                 trans=policy))
17
18 def lpec(pkt):
19   return match(srcip=pkt['srcip'])
20
21 fsm_pol = FSMPolicy(lpec,self.fsm_def)
```

(b) *Kinetic code that implements the Kinetic FSM.*

Figure 8: *Logical FSM for an IDS in Kinetic, and the Kinetic code that implements the policy.*

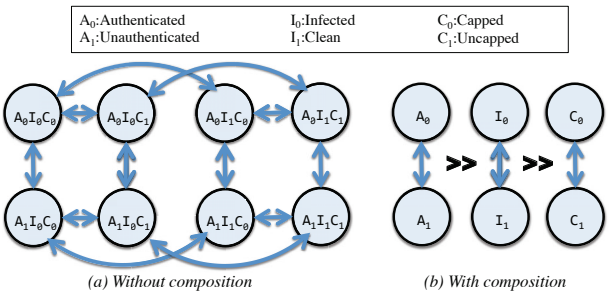


Figure 9: *Composing independent tasks in sequence.*

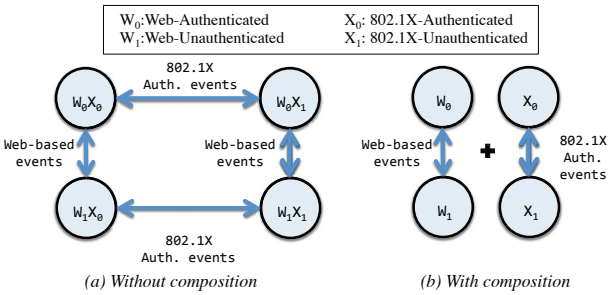


Figure 10: *Composing multiple authentication tasks in parallel. Any successful authentication would result in allowing the host's traffic.*

dependently and compose them in parallel, as shown in Figure 7b.

Each LPEC has an FSM, which has a set of states, where each state has a Pyretic policy; and a set of transitions between those states, where transitions occur in response to events that the operators defines. When events arrive, the respective LPEC FSMs may transition between states, ultimately inducing the Pyretic runtime to recompile the policy and push updated rules to the switches. In Kinetic, a programmer can specify an LPEC in terms of a Pyretic filter policy. For example, `match(srcip=pkt['srcip'])` defines an LPEC FSM for each unique source IP address.

Returning to our IDS example from Section 3 (Figure 1), Figure 8b shows the code for the Kinetic program that implements the simple intrusion detection example from Section 3. Each host (*i.e.*, source IP address) can have a distinct state, so we need an LPEC FSM per source IP address; lines 18–19 define the LPEC. To define an FSM that is amenable to model checking, we must separate the `infected` variable and the corresponding `policy` variable into two separate states, as shown in Figure 8a. *Exogenous* events trigger transitions between the `infected` variable states; a change in this variable's value in turn triggers an *endogenous* transition of the `policy` variable, which ultimately causes the Pyretic runtime to recompile flow-table entries for the network switches. Lines 1–3 in Figure 8b define the exogenous transition for `infected`; lines 5–8 defined the endogenous transition for `policy` (note that the value of `policy` is defined in terms of the value of `infected`). Finally, lines 10–16 define the FSM itself, in terms of the two variables; the FSM definition is simply a set of FSM variables, each of which has a type, an initial value, and a transition function.

4.2.3 FSM Composition

In Section 3, we showed an example of a campus network policy that composed FSMs for independent network tasks to control state explosion. Without FSM composition, a programmer would need to define FSMs for $\Pi_{i=1}^{N} a_i$ possible states, where a_i is the number of possible states for task i and N is the total number of tasks. Decomposing the product automaton reduces state complexity from exponential to linear in the number of independent tasks. For example, given ten tasks, each with two states, a monolithic program would require 1,024 states, as opposed to just 20.

Pyretic allows policies to be composed either in parallel (*i.e.*, on independent copies of the same packet) or in sequence (*i.e.*, where the second policy is applied to the output from the first). It turns out that these operators are *also* useful for reducing state explosion. Figure 9 illustrates how sequential composition can reduce state

complexity by decomposing a larger product automaton. Consider a simple control program that puts the host into a walled-garden until it has authenticated, quarantines the host if an infection has been detected, and rate limits a host if it has exceeded a usage cap. Each of these tasks has two possible states: authenticated or not, quarantined or not, capped or not, resulting in 2^3 possible states. By applying `auth >> IDS >> cap`, the same network control program requires only $2 \cdot 3$ states.

Figure 10 shows how parallel composition reduces state complexity. A Kinetic program might specify that either a network flow should be authenticated by a Web authentication mechanism or 802.1X. If *either* of these tasks places the host in an authenticated state, the host should be allowed to send traffic. Without composition, the network state machine would need a second set of states, requiring 2^N states, where N is the number of authentication tasks (in this case, $N = 2$). (Clearly, even more states would be needed if any independent task could assume more than two states.) As before, decomposition reduces this to $\Sigma_{i=1}^{N} a_i$ states, where a_i is the number of possible states for task i, and N is the number of tasks.

4.3 Runtime

We now explain optimizations to the Kinetic runtime to support the efficient compilation of the large finite state machines that might result from networks with many hosts and policies and high event rates. The Kinetic runtime's main challenge is storing and processing the joint state of all LPEC FSMs to produce a single set of forwarding table entries in the network switch. To accomplish this goal, the runtime first decomposes the FSMs with combinators to achieve a representation of the network state that is linear in the number of hosts and policies. Second, Kinetic optimizes the compilation process itself by recognizing that the LPEC FSMs typically operate on disjoint flow space, which allows for optimizations that dramatically speed up parallel composition. Finally, Kinetic only expands the LPEC FSMs for which a transition has actually occurred. We describe each of these optimizations below.

Decomposing the product automata. A Kinetic `FSMPolicy` encodes the complete FSM as the *product automaton* [10] of the individual LPEC FSMs. We can represent the Pyretic policy for the entire network, given a global network state s as the following product automata:

$$\text{policy} \quad = \quad \sum_{i=1}^{N} (\text{lpec}_\text{i} \;>>\; \text{lfsm}_i(s))$$

where the summation operator represents parallel composition of the corresponding policies, and each LPEC

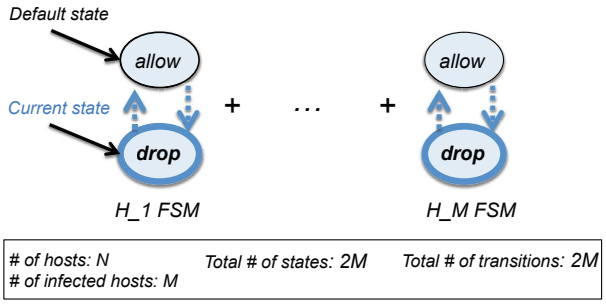


Figure 11: *Expanding only* ***M*** *LPEC FSMs that have changed.*

generator produces the appropriate packets that are processed by the corresponding LPEC FSM in state s.

Fast compilation of disjoint LPECs. Compilation of policies that are composed in parallel is computationally expensive, as it requires producing the cross-product of all match and action rules: it involves computing the intersection of match statements and the union of actions, for every pair of match and action pairs between the two policies. If LPECs are disjoint, however, the resulting policies can simply be combined without explicitly computing the intersection of the match statements: the rules from each LPEC FSM can simply be inserted into the flow tables.

Default policies and on-demand LPEC FSM expansion. Even a linear-sized representation may not scale. For example, the LPEC generator shown in Section 4.2.2 would generate 2^{32} LPECs if it were fully expanded, while a generator for each pair of hosts (based on hardware address) would produce 2^{96} LPECs. Fortunately, because all LPEC FSMs are generated from the same FSM specification, they start with the same initial state and, hence, the same default policy. Thus, Kinetic does not need to expand the FSM for an LPEC *unless and until it experiences a state transition*; until that point, the Kinetic runtime can simply apply whatever default policy is defined for that FSM. Figure 11 highlights this on-demand expansion.

Kinetic's runtime optimizations reduce the computational complexity of compilation from exponential in the number of LPECs (Figure 7a), to linear in the number of LPECs (Figure 7b), and finally to linear in the (much smaller) number of LPECs that have actually experienced a transition (Figure 11). Kinetic additionally employs additional optimizations, such as memoizing previously compiled policies, as other applications have used [15].

4.4 Verification

When the programmer executes a Kinetic program, Kinetic automatically creates an FSM model for the NuSMV model checker. Kinetic obtains information about each state variable (*e.g.*, type, initial value, and transition relationship) by parsing the `fsm_def` data structure; Kinetic

```
1  MODULE main
2    VAR
3      policy    : {identity,drop};
4      infected  : boolean;
5    ASSIGN
6      init(policy)   := identity;
7      init(infected) := FALSE;
8      next(infected) :=
9        case
10         TRUE     : {FALSE,TRUE};
11       esac;
12     next(policy) :=
13       case
14         infected : drop;
15         TRUE     : identity;
16       esac;
```

Figure 12: *NuSMV FSM model for IDS policy from Figure 8b.*

parses the transition function for additional information about transitions, which often depend on other variables.

Kinetic then uses NuSMV to test CTL specifications that the programmer writes against the FSM model. Kinetic outputs the CTL specifications that passed; for any failed specifications, Kinetic produces a counterexample, showing the sequence of events and variable changes that violated the specification. In addition to single `FSMPolicy` objects, Kinetic can convert composed policies into a single model that can be verified. For example, although the programmer specifies a composed policy as in Figure 9b and Figure 10b, verification will execute on a combined FSM model as in Figure 9a and Figure 10a.

Figure 12 shows the NuSMV FSM model corresponding to the IDS policy from Figure 8b. The model definition has two parts. The first is `VAR`, which declares the names and types of each variable (lines 2–4). The second is `ASSIGN`, where current and future variable values are assigned, using two functions for each variable: an `init` function that determines the variable's initial value (line 5–7), and a `next` function that specifies what value or values the variable may take, as a function of the current values of other variables in the model (line 8–16).

Within the `case` clause of each `next` function, the left-hand side shows the condition, while the right-hand side shows the variable's next value if the condition holds. `TRUE` on the left-hand side refers to a default transition. Lines 8–11 indicate that the `infected` variable can change between `FALSE` and `TRUE`, independent of any other state variable (in reality, the value changes based on external event of the same name). The `policy` variable in lines 12–16 shows that the value transitions to `drop` if `infected` is `True`, while the default is `identity`. Thus, it shows that `policy`'s next value depends on the `infected` variable. Table 3 shows examples of the types of temporal properties that Kinetic can verify.

5 Evaluation

In this section, we evaluate two aspects of Kinetic: (1) Does Kinetic make it easier for network operators to configure realistic network policies? (Section 5.1); and (2) How does Kinetic's performance scale with the number of flows, users, and policies? (Section 5.2).

5.1 Programming in Kinetic: User Study

Evaluating whether a new network configuration paradigm such as Kinetic makes it easier for network operators to write network policies is challenging. Network operators already know how to use existing tools and infrastructure, and deploying a new control framework requires overcoming both the inertia of network infrastructure that is already deployed *and* the knowledge base of network operators, many of whom are not programmers by training. We needed to find a way to ask many network operators to evaluate Kinetic in light of these obstacles. Fortunately, the Coursera course on software-defined networking that we teach [8] offers precisely this captive audience, as the course's demographic includes many network operators who are both educated about SDN and willing to experiment with cutting-edge tools. (Section 2 and Table 1 explained the initial survey and described the demographics of the participants.) We obtained approval from our institutional review board (IRB) to ask students to use Kinetic and other SDN controllers to complete a simple network management task and subsequently survey them.

We asked the students in the course to write a "walled garden" controller program that is inspired from real enterprise network management task that we have learned about in our discussions with network operators [9]. In summary, the students were asked to write a program that permitted all traffic to and from the Internet unless a host was deemed to be infected (*e.g.*, as determined from an intrusion detection system alert) *and* not a host that was exempt from the policy (one might imagine that certain classes of users, such as high-ranking administrators or executives would get different treatment than strict interruption of service). In the assignment, we asked the students to: (1) Write a Kinetic program that implements the policy; (2) Choose either Pyretic or POX to implement the same policy; (3) Optionally implement the policy in the remaining controller; (4) Answer survey questions about their experiences with each controller.

The course devoted one week each to each of the three controller platforms, and students had already completed assignments in both POX and Pyretic, so if anything, students should have found those platforms at least as familiar as Kinetic. In fact, there were three programming assignments in Pyretic while there was only one for Kinetic. Kinetic was discussed in only one lecture out of eight

Program	CTL	Description
Mac learner	AG (topo_change → AX policy=flood)	Always resets to flooding when topology changes.
	AG (policy=flood → AG EF (port>0))	Can always go from flooding to unicasting a learned port.
	! AG (port=1 → EX port=2)	It is impossible to update the learned port without first flooding.
	AG (port>0 → A [port>0 U topo_change])	Port will stay learned until there is a topology change.
Stateful firewall	AG (outgoing & !timeout → AX policy=identity)	If first packet originated from internal host and timeout did not occur, the system should allow traffic.
	AG (outgoing & timeout → AX policy=matchFilter)	If first packet originated from internal host, but timeout occurred, the system should shut down traffic (apply match filter).
	AG (!outgoing → AX policy=matchFilter)	If first packet is not from internal host, the system should not allow traffic (apply match filter).
Composed policy	AG (infected → AX policy=drop)	If host is infected, drop packets.
	AG ((authenticated_web | authenticated_1x) & !infected → AX policy!=drop)	If host is authenticated either by Web or 802.1X, and is not infected, packets should never be dropped.
	AG (authenticated_web & !infected & rate=2 → AX policy=delay200)	If host is authenticated by Web, not infected, and the rate is 2, delay packets by 200ms.

Table 3: *NuSMV CTL rules for different Kinetic programs.*

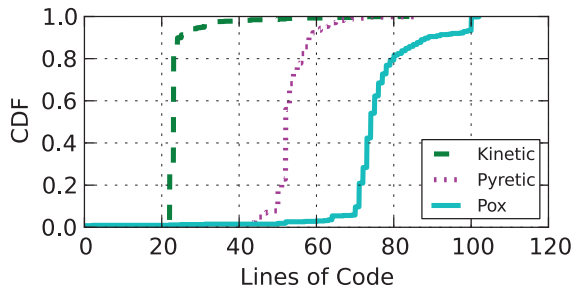


Figure 13: *The lines of code required to implement the walled-garden program in different controller languages.*

Programs	FL	Pox	Pyretic	Kinetic
ids/firewall	416	22	46	17
mac_learner	314	73	17	33
server load balancer	951	145	34	37
stateful firewall	–	–	25	41
usage-based rate limiter	–	–	–	30

Table 4: *Lines of code to implement programs in each controller.*

lectures in the course, and was not treated specially. To further minimize the bias in favor of Kinetic, students were instructed to complete the assignment in Kinetic first, as the first attempt is usually the hardest. With better understanding of the assignment, it is likely that programming in Pyretic or POX would have been easier.

Of the students who completed the survey from Section 2, 667 attempted the assignment, and 631 successfully completed it (a 95% completion rate), and 70% of those students completed the programming assignment in less than three hours. We asked students who did not complete the assignment why they did not complete it; most referenced external factors such as time constraints, as opposed to anything pertinent to Kinetic.

To compare the complexity of the different control programs, we compare the lines of code in programs implemented with different controllers; we then conduct qualitative measurements by surveying the students of the course. Although the lines of code for a program depends on the language, programmer, and implementation style, a high-level comparison can nevertheless yield a rough but meaningful sense for the relative simplicity of a Kinetic program. Figure 13 shows the distribution of the lines of code that students needed to implement the walled-garden program in different controller languages. About 80% of the implementations using Kinetic required about 22 lines of code; in contrast, more than half of the Pyretic implementations required more than 50 lines, and half of the assignments written in POX required more than 75 lines of code. The fact that Kinetic requires fewer lines of code to implement this program highlights the utility of the abstractions that Pyretic provides. In addition to the experiment from the Coursera course, where we could find publicly available implementations on the Web, we compared the number of lines of code for several different programs to our own implementations of the same programs in Kinetic. (Blank entries in the table indicate that no implementation was available.) Table 4 shows these results, for four different controllers: Floodlight, POX, Pyretic, and Kinetic. The public Pyretic programs are occasionally slightly shorter than the corresponding Kinetic programs because they only handle built-in events such as packet arrivals and topology changes. Programs that need to handle arbitrary events would likely always be shorter in Kinetic.

In addition to analyzing quantitative measures such as lines of code, we asked students more qualitative questions about their experiences using Kinetic to implement the walled-garden assignment, relative to their experiences with Pyretic and POX. We asked students to rank the controllers based on ease of use, as well as which platform they preferred. Figure 14 shows some highlights from this part of the survey. Of the three controllers, more than half of the students preferred writing the assignment in Kinetic

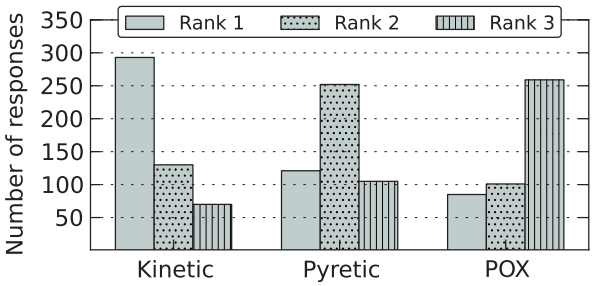


Figure 14: *Number of students who preferred each controller.*

versus either Pyretic or POX. We asked students whether Kinetic could make it easier to configure and verify policies in their networks. About 79% of students thought that configuring the network with Kinetic would be easier than current approaches, and about 84% agreed that Kinetic would make it easier to verify network policies.

Students who chose Kinetic as the language that they preferred best cited its abstractions, FSM-based structure, and support for intuition (*e.g.*, "Kinetic is more intuitive: the only thing I need to do is to define the FSM variable.", "intuitive and easy to understand", "reduces the number of lines of code", "programming state transitions in FSMs makes much more sense", "the logic is more concise"). Some students still preferred Kinetic, despite the fact that the syntax had a steeper learning curve: "Kinetic took less time and was actually more understandable using the templates even though the structure was very 'cryptic'... I thought the Pyretic would be the easiest...[but] I spent a lot more time chasing down weird bugs I had because of things I left out or perhaps didn't understand." Interestingly, many of the students actually preferred the lower-level trappings of POX to Pyretic (*e.g.*, "Pyretic was friendly, but the logic more intricate"). The results of this experiment and survey highlight both the advantages and disadvantages of Kinetic's design, as well as the difficulty of designing "northbound" languages for SDN controllers: without intuitive abstractions, operators may even *prefer* the lower-level APIs to higher-level abstractions.

5.2 Performance and Scalability

We evaluate Kinetic's performance and scalability when handling incoming events, as well as the performance and scalability of verification, as those are the two main contributions of our work. We evaluated the Kinetic controller on a machine with an Intel Xeon CPU E5-1620 3.60 GHz processor and 32 GB of memory. We measured raw packet forwarding performance but do not focus those numbers, as the forwarding performance is not the focus of our work and is equivalent to what can be achieved with POX and Pyretic, in any case. Similarly, the rate at which updated

Statistic	# per day
Total Unique Authenticated Users	22,586
Total Unique Devices Authenticated	41,937
Number of WPA authentication Events	1,330,220
Number of WEB authentication Events	1,850

Table 5: *The frequency of network events on a primary campus network, which we use for a trace-driven evaluation of Kinetic.*

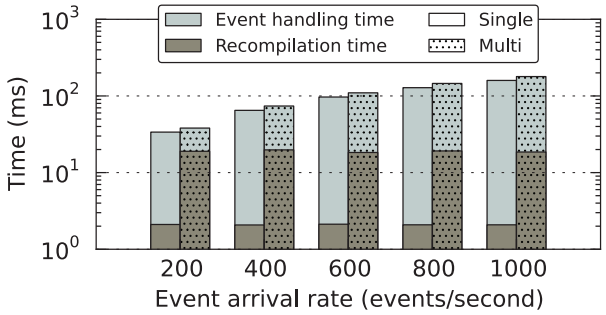


Figure 15: *Time to handle a batch of incoming events and recompile policies in Kinetic, for different event arrival rates and policies.*

rules can be installed depends on lower layers (*e.g.*, POX). Optimizing the number of rule updates [32] and applying them consistently [28] have been studied in previous work, so we do not focus on those aspects here.

Event handling and policy recompilation. Because Kinetic recompiles the policy whenever an event causes a state transition, we must evaluate how fast Kinetic can react to events and recompile the policy for realistic network scenarios. We used the wireless network from a large university campus with more than 4,300 access points deployed across 200 buildings; the network authenticates nearly 42,000 unique devices for more than 22,000 users every day. Table 5 summarizes these statistics. On such a network, Kinetic would have to keep track of an equivalent number of devices, and each authentication event (about 1.3 million per day) would require Kinetic to recompile policy, resulting in an average of 15 events per second (though certainly higher during peak periods). We evaluate Kinetic for event arrival rates for up to 1,000 events per second, for both a single-FSM policy and a policy involving the composition of multiple FSMs, based on the example in Section 3.3. We create a Kinetic program that results in 42,000 LPEC FSMs and randomly distribute authentication events across these FSMs (*i.e.*, devices).

Figure 15 presents the results of this experiment. Recompilation time is longer for the program with multiple FSMs composed together as it embeds a more complex policy than the program with a single FSM. For both programs, event handling time increases as event arrival rate increases. Even for event arrival rates that are several orders of magnitude more frequent than an actual campus

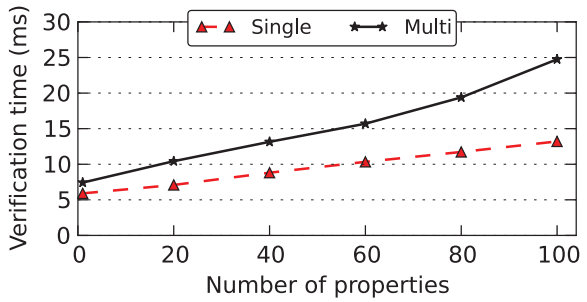


Figure 16: *Verification time as a function of the number of CTL properties that Kinetic checks, for a policy with a single FSM and a policy with a composition of multiple FSMs.*

network, Kinetic's event handling and recompilation times are around (or less than) a few hundred milliseconds.

Verification. The speed of verification depends on the performance of NuSMV, which in turn depends on three factors: (1) the size of the given FSM (*i.e.*, number of states and transitions), (2) the number of properties to verify, and (3) the kinds of properties to verify. To observe whether Kinetic's verification time is reasonable, we evaluate Kinetic's verification performance with the same programs that we used to evaluate Kinetic's event handling and recompilation performance. The single-FSM authentication program produces an FSM with four states, and the program with multiple FSMs produces a combined FSM with 384 states. To test Kinetic with more than the handful of CTL specifications we manually created, we generate over one hundred specifications using random combinations of CTL operators. They are all syntactically correct (*i.e.*, NuSMV will not complain about the syntax), but are generated regardless of whether they will be true or false when each goes through the model checker, as our goal is merely to measure verification time.

Figure 16 shows the time to complete verification for different Kinetic programs with different numbers of properties. Each experiment had 1,000 independent trials; the variance across experiments was small, so we do not show the error bars. As expected, a Kinetic program with a larger FSM model takes longer to finish. The figure also shows that the number of properties affects verification time, but all verification finishes within 35 milliseconds. Kinetic performs verification before the Kinetic program ever runs, so this process has no effect on performance.

6 Related Work

We discuss SDN controllers with verifiable properties, approaches for formally verifying data-plane behavior, and other high-level SDN control languages.

Formal verification of SDN control programs. FlowLog [26] provides a database-like programming model that unifies the control-plane logic with data-plane state and controller state. Aspects of FlowLog programs can be verified, but because the language does not naturally capture state transitions and temporal relationships, it cannot verify arbitrary temporal relationships, such as those that can be verified with CTL in Kinetic. FlowLog uses Alloy to perform bounded verification, so its analysis is not complete, and certain aspects of verification are manual. FlowLog has not been evaluated for realistic network policies or for large networks. It requires storing multiple database entries for each network state variable and handles certain aspects of control logic by sending data packets to the controller, so it is unlikely to scale. VeriCon [1] verifies that a program written in its language (CSDN) is correct for all topology and packet events (*e.g.*, packet arrivals, switch joins). It does not handle arbitrary network events, and there is no OpenFlow-based implementation, so its practicality is unclear.

Formal verification of data-plane behavior. Recent work in network verification has focused on verifying static properties of the data-plane state [19, 28, 29]. Anteater [23] and HSA [17, 18] can verify properties of a static snapshot of a network's data-plane state. These systems can determine whether a static snapshot of data-plane state violates some invariant, but they do not verify the logic of the control program that generated the state in the first place, making it difficult to identify which aspect of the network's control-plane logic caused the incorrect data-plane state. In contrast, Kinetic helps operators verify control logic, such as "if an intrusion detection system determines that a host is infected, the host's traffic should be dropped". This capability helps operators both reason about future data-plane states that a control program could install and troubleshoot incorrect behavior when it does arise. Because Kinetic's verifies the static programs themselves, it can detect logic errors before the control program is ever run on a live network. NICE [2] can test control-plane properties that might result from arbitrary sequences of standard OpenFlow events; it is not a controller, but rather a test harness for control programs written in existing low-level controllers (*e.g.*, NOX) and hence does not permit reasoning about arbitrary events.

Other SDN control languages. Many languages raise the level of abstraction for writing network control programs, yet these languages do not offer constructs for concisely encoding policies that capture network dynamics, nor do they incorporate formal verification of control-plane behavior. FML [16] allows network operators to write and maintain policies in a declarative style. Nettle [30] is a domain specific language in Haskell. Procera [31] applies functional reactive programming to help operators express policies. Frenetic [12] is a family of languages that share fundamental constructs and techniques for efficient compilation to OpenFlow switches.

7 Conclusion

One of the reasons that network configuration is so challenging is that network conditions are continually changing, and network operators must adapt the network configuration whenever these conditions change. Network operators need means not only to automate these configuration changes but also to verify that the changes will be correct. Existing general-purpose SDN controllers lack intuitive constructs for expressing dynamic policy and ways to efficiently verify that the control programs conform to expected behavior.

To address these problems, we designed and developed Kinetic, a domain specific language and SDN controller for implementing dynamic network policies in a concise, verifiable language. Kinetic exposes a language that allows operators to express network policy in an intuitive language that maps directly to a CTL-based model checker. We evaluated Kinetic's usability and performance through both a large-scale user study and trace-driven performance evaluation on realistic policies and found that network operators find Kinetic easy to use for expressing dynamic policies and that Kinetic can scale to a large number of policies, hosts, and network events.

Kinetic sits squarely in the realm of ongoing work on network verification and complements the growing body of work on data-plane verification, such as Veriflow [19] and NetPlumber [17]. As these tools can help network operators ask questions about snapshots of data-plane state, and Kinetic can help network operators reason about the dynamics of network policies (which ultimately compile to the corresponding data-plane state), the approaches are complementary. Similarly, Kinetic needs the path guarantees that consistent updates [28] provide to guarantee that the properties it verifies are preserved during state transitions; conversely, consistent updates could be extended to reason about temporal properties such as those that Kinetic can express. One natural next step would be to combine these approaches.

Acknowledgements

We thank the NSDI reviewers and our shepherd Dejan Kostic for providing valuable feedback for this paper. We also thank Jennifer Rexford, Nate Foster, David Walker, and Yoshio Turner for their helpful comments on earlier versions of this paper. This work was supported by NSF Awards CNS-1261357 and CNS-1409076.

References

[1] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. Vericon: towards verifying controller programs in software-defined networks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 31. ACM, 2014. (Cited on page 12.)

[2] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE Way to Test OpenFlow Applications. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2012. (Cited on page 12.)

[3] Chef. `http://www.opscode.com/chef/`. (Cited on page 1.)

[4] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An Open-source Tool for Symbolic Model Checking. In *Computer Aided Verification*, pages 359–364. Springer, 2002. (Cited on pages 2 and 3.)

[5] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986. (Cited on page 3.)

[6] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT press, 1999. (Cited on pages 2 and 3.)

[7] CMU Network Bandwidth Usage Guidelines. `http://www.cmu.edu/computing/network/connect/guidelines/bandwidth.html`. (Cited on page 5.)

[8] Coursera: SDN class. `https://www.coursera.org/course/sdn`, 2014. (Cited on pages 2 and 9.)

[9] Coursera: SDN Walled Garden Assignment. `http://goo.gl/JYMret`, 2014. (Cited on page 9.)

[10] S. Eilenberg. *Automata, languages, and machines*, volume 1. Access Online via Elsevier, 1974. (Cited on pages 2 and 8.)

[11] N. Feamster, J. Rexford, and E. Zegura. The Road to SDN. *Queue*, 11(12):20:20–20:40, Dec. 2013. (Cited on page 1.)

[12] N. Foster, M. J. Freedman, A. Guha, R. Harrison, N. P. Katta, C. Monsanto, J. Reich, M. Reitblatt, J. Rexford, C. Schlesinger, A. Story, and D. Walker. Languages for Software-Defined Networks. *IEEE Communications*, 51(2):128–134, Feb. 2013. (Cited on page 12.)

[13] N. Foster, R. Harrison, M. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *International Conference on Functional Programming (ICFP)*, Sept. 2011. (Cited on page 1.)

[14] Frenetic, Pyretic and Resonance. `http://blog.sflow.com/2013/08/frenetic-pyretic-and-resonance.html`. Note: Kinetic was previously known as Resonance. (Cited on page 2.)

[15] A. Gupta, L. Vanbever, M. Shahbaz, S. P. Donovan, B. Schlinker, N. Feamster, J. Rexford, S. Shenker, R. Clark, and E. Katz-Bassett. SDX: A Software Defined Internet Exchange. In *ACM SIGCOMM*, pages 579–580, Chicago, IL, 2014. (Cited on page 8.)

[16] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker. Practical declarative network management. In *ACM/USENIX Workshop on Research on Enterprise Networking (WREN)*, pages 1–10, Barcelona, Spain, 2009. (Cited on page 12.)

[17] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real-time Network Policy Checking using Header Space Analysis. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013. (Cited on pages 12 and 13.)

[18] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2012. (Cited on pages 2 and 12.)

[19] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Lombard, IL, Apr. 2013. (Cited on pages 2, 12 and 13.)

[20] H. Kim, T. Benson, A. Akella, and N. Feamster. The Evolution of Network Configuration: A Tale of Two Campuses. In *ACM SIGCOMM Internet Measurement Conference (IMC)*, pages 499–514, Berlin, Germany, 2011. (Cited on page 1.)

[21] Kinetic source code. `https://github.com/frenetic-lang/pyretic/tree/kinetic` Note: Evaluation specific data & scripts are in the kinetic_test branch. (Cited on pages 2 and 3.)

[22] List of Real Campus Network Acceptable Use Policies. `http://goo.gl/pdR6Sd`, 2014. (Cited on pages 1 and 5.)

[23] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteater. In *ACM SIGCOMM*, pages 290–301, Toronto, Ontario, Canada, 2011. (Cited on page 12.)

[24] K. L. McMillan. *Symbolic model checking*. Springer, 1993. (Cited on page 3.)

[25] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software-Defined Networks. In *USENIX NSDI*, 2013. (Cited on pages 2 and 3.)

[26] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 519–531, Seattle, WA, Apr. 2014. (Cited on page 12.)

[27] Puppet. `http://puppetlabs.com/solutions/juniper-networks`. (Cited on page 1.)

[28] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. In *ACM SIGCOMM*, pages 323–334, Helsinki, Finland, 2012. (Cited on pages 11, 12 and 13.)

[29] D. Sethi, S. Narayana, and S. Malik. Abstractions for model checking sdn controllers. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 145–148, 2013. (Cited on page 12.)

[30] A. Voellmy and P. Hudak. Nettle: Taking the sting out of programming network routers. In *Practical Aspects of Declarative Languages (PADL)*, pages 235–249. Springer, 2011. (Cited on page 12.)

[31] A. Voellmy, H. Kim, and N. Feamster. Procera: a language for high-level reactive network control. In *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networks (HotSDN)*, pages 43–48, Helsinki, Finland, 2012. (Cited on page 12.)

[32] X. Wen, L. Li, C. Diao, X. Zhao, and Y. Chen. Compiling Minimum Incremental Update for Modular SDN Languages. In *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, pages 193–198. ACM, 2014. (Cited on page 11.)