Switches are Monitors Too! Stateful Property Monitoring as a Switch Design Criterion

Tim Nelson Nicholas DeMarinis Timothy Adam Hoff
Rodrigo Fonseca Shriram Krishnamurthi
Brown University

Abstract

Testing and debugging networks *in situ* is notoriously difficult. Many vital correctness properties involve histories over multiple packets (e.g., prior established connections). Checking such properties requires *cross-packet state*, which cannot be fully captured on stateless switch hardware.

Recent SDN work is enabling limited switch operations on persistent state. We present runtime checking of cross-packet correctness properties as a unique and instructive use case for developing stateful switch primitives. In this paper, we examine a set of cross-packet properties and distill from them switch features needed to monitor their correctness. We then contrast these against features provided by current approaches to switch state in SDNs and identify semantic gaps with an eye toward informing future switch instruction sets.

Categories and Subject Descriptors

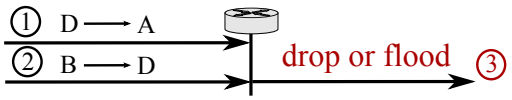
C.2.3 [**Network Operations**]: Network management

1. INTRODUCTION

Testing and debugging network behavior can be uniquely challenging. Naive tests, such as pinging, may capture connectivity, but not deeper aspects of the system. These may include ARP resolution, timing of responses, and—most challenging of all—how network state evolves in response to events. Software Defined Networking underscores this problem, as the network may now be controlled by third-party or home-grown software, rather than the limited configuration languages provided by switch manufacturers.

An alternative testing approach involves checking *correctness properties* of the network. In this paper, we propose runtime monitoring for *correctness* of *stateful* network behavior, which has received limited attention to date, rather than the more well-studied problems of monitoring for per-flow measurement (e.g., [21]) or to detect patterns in traffic.

In Sec. 2, we will present complex and realistic properties. For illustration, we begin with a simple one. Consider a standard learning switch, which remembers arrival ports for layer-2 source addresses to avoid broadcasting. One of its many correctness properties is: "*Once a destination* `D` *is learned, packets to* `D` *are unicast on the appropriate port*." To show this property is violated, a monitor must find an initial packet from which address `D` should be learned, followed by another packet, addressed to `D`, forwarded incorrectly:



In canonical SDN platforms, a learning switch implementation is typically part of a program running on the controller. Many tools are available to statically check properties in network programs [1, 8, 11, 13, 17], but these fail to capture the "full stack" of network behavior since they reason about a program or configuration in isolation. VeriFlow [10] and similar approaches, such as the property-based work of Beckett, et al. [5], examine the network at runtime, but are limited to analyzing flow-table rules installed by a known, white-box application, and thus are of limited use in a hybrid environment, when integrating with black-box code, or—as is becoming more common—when some stateful operations happen on-switch rather than via the controller.

Checking correctness properties at runtime intrinsically requires *cross-packet state*, *i.e.* retaining information about packet history, due to the dependency between initial packets (that trigger learning) and subsequent packets (that use the learned destination). As multiple addresses will be seen over time, a monitor must manage a growing set of learned destinations. Thus, maintaining packet *history* differs greatly from runtime methods that monitor packet *trajectory*, such as Path Queries [15] and NetSight [9], which are unable to capture the rich, stateful nature of many correctness properties.

Monitoring the necessary packets, rather than only controller messages, quickly becomes expensive to do externally:

HotNets-XV, November 09 - 10, 2016, Atlanta, GA, USA

 ISBN 978-1-4503-4661-0/16/11…$15.00

DOI: http://dx.doi.org/10.1145/3005745.3005755

in the learning switch example, *any* outgoing packet could potentially violate the property (e.g., if after D has been learned, packets sent to it are broadcast instead of unicast). Thus, an external monitor must either see all such packets, or else update the switch's behavior dynamically to filter out idempotent updates, which in this case amounts to keeping the full state table in its forwarding base.

For these reasons, we claim that the *ideal* location for monitoring cross-packet properties is directly on switches, a feat which is becoming possible due to emerging developments to increase switch programmability. Proposals such as OpenState [6], POF [20], and P4 [7] provide stateful operations on switches, enabling applications such as MAC learning and connection tracking without controller interaction. In light of this trend, we make a case for runtime, on-switch monitoring of cross-packet correctness properties, which has several advantages: • it obviates the need to redirect potentially large volumes of traffic to an external monitor; • switches may run stateful programs without controller interaction, making controller-based monitoring infeasible; and • by analyzing packets directly in the data plane, switches can monitor properties on a broader scope than static analysis tools or controller-based monitoring schemes support, even in the presence of non-SDN devices or third-party middleboxes.

Related Work in Stateful Property Checking.

Runtime property checking of software has a strong pedigree. MonPoly [4], MaC [12], Eagle [3], and many others analyze or enable analyzing event streams to verify stateful correctness properties. While one could build a middlebox that runs, say, MonPoly, that approach would abandon the benefits of switch-centric monitoring and it is unclear that these approaches would scale to line rate. DMaC [23] leverages custom on-switch Datalog support for stateful monitoring; our aim is necessarily closer to standard switches. Approaches like OFRewind [22] capture only control packets, which limits the scope of properties that can be checked.

Middlebox-centric monitoring approaches depend on routing traffic through computationally powerful nodes. These are challenging and expensive to deploy for checking behavior throughout the network core—as in the above learning switch property—rather than at its edge.

In SIMON [18] a central server sees every network event, allowing operators to write stateful monitoring scripts in a fully-expressive programming language. In spite—indeed, largely due to—this extensive power, SIMON runs only in a simulator, where it can be used for prototyping.

Contributions.

As SDN switches rapidly become more programmable, now is the time to consider vital use cases for switch features that point to unique requirements but have not been adequately explored. In this paper, we examine a set of cross-packet properties and identify key features needed to monitor their correctness. As we will show, even relatively simple correctness properties can raise subtle semantic questions and point to unique challenges not considered in current proposals for extending switches.

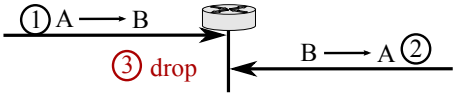
This paper stems from our experience with Varanus [16], an experiment in on-switch stateful property monitoring. Varanus provides a query language for properties, adapted to run on switches by using a restricted, limited-power set of SIMON's features. Though its language is promising, the Varanus system is not yet practical. Through it we have identified challenges in monitoring stateful properties, and call out both semantic requirements and scalability concerns. In doing so, we hope to *promote support for monitoring as a first-class consideration in future switch instruction set designs*.

2. SEMANTIC CHALLENGES

We begin by identifying key semantic features needed for on-switch stateful property monitoring. Since properties vary in the features they need, we examine several basic examples in succession (Tab. 1 lists more complex properties). To separate semantic requirements from on-switch implementation, we defer discussing implementation to Sec. 3.2.

2.1 Example: Stateful Firewall

A stateful firewall opens to external traffic as internal hosts initiate connections. One correctness property for such a device is: "*After seeing traffic from internal host A to external host B, packets from B to A are not dropped*". Monitoring for *violations* of this property requires identifying event traces, consisting of one packet arrival and one departure, for arbitrary values of A and B such that the B→A packet is dropped:



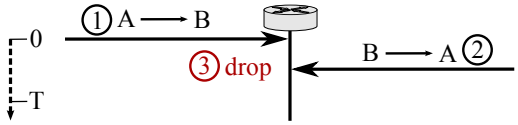
This requires a number of semantic features from the switch.
Feature 1: Access to Necessary Fields. The monitor must be able to obtain values for A and B. Extracting header fields is a primary function of most switches, yet standard switches only parse packet headers to a limited depth. Our stateful firewall monitor reads no deeper than can be accessed in standard hardware. Checking application-layer fields, however, requires richer parsing. The monitor must also be able to observe egress packets and distinguish them from new arrivals.
Feature 2: Access to Event History. Here, the switch must remember the set of outgoing (A,B) address pairs seen. It must then match against that stored state to detect violations.

We define a property as a sequence of *observations* that, when completed, witness a violation. For instance, violating this firewall property requires two observations: an A→B packet and a B→A packet whose egress action is drop.
Feature 3: Timeouts. To rein in space consumption and make the switch more resilient to network failure, stateful firewalls often erase "stale" connections after a reasonable timeout. The property as written so far does not soundly capture this new requirement: a drop that comes after a valid

Property		Fields	History	Timeouts	Obligation	Identity	Neg Match	T.Out. Acts	Inst. ID
ARP Cache Proxy	Requests for known addresses are not forwarded	L3	•						exact
	Requests for unknown addresses are forwarded	L3	•		•	•		•	exact
Port Knocking	Intervening guesses invalidate sequence	L4	•				•		exact
	Recognize valid sequence	L4	•		•		•		exact
Load Balancing	New flows go to hashed port	L4	•		•	•			symmetric
	New flows go to round-robin port	L4	•		•	•			symmetric
	No change in port until flow closed	L4	•			•	•		symmetric
FTP	Data L4 port matches L4 port given in control stream	L7	•				•		symmetric
DHCP	Reply to lease request within T seconds	L7	•	•				•	symmetric
	Leased addresses never re-used until expiration or release	L7	•	•					symmetric
	No lease overlap between DHCP servers	L7	•				•		symmetric
DHCP + ARP Proxy	Pre-load ARP cache with leased addresses	L7	•				•	•	wandering
	No direct reply if neither pre-loaded nor prior reply seen	L7	•		•				wandering

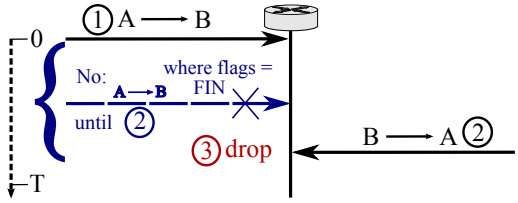
Table 1: Select additional properties not discussed in Sec. 2. Port knocking and the first ARP example are taken from Varanus [16]; FTP example is from FAST [14]; others are from our own experience. The **Fields** column gives the maximum layer of parsing required as a rough indicator of complexity. The **Inst. ID** field indicates the variety of instance matching required (Feature 8). Other columns contain • if the feature is required, and are blank otherwise. Side-effect control and provenance are intrinsic features of the monitoring implementation and independent of the property.

timeout will still trigger a violation. Instead, we must express: "*for T seconds after seeing traffic from* A *to* B*...*":



The monitor must maintain *separate* timers for each A, B pair, to be reset whenever a new A→B packet is seen.

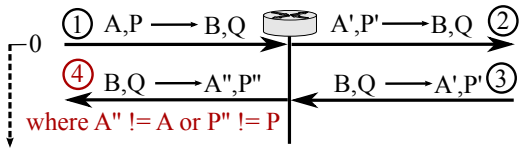
Feature 4: Persistent Obligation. An intelligent stateful firewall also deletes state when it detects a connection has been closed. As before, A→B packets allow return traffic, but as soon as either party closes the connection, B→A packets should be dropped until A re-establishes the connection. The property is now: "*for T seconds after seeing traffic from* A *to* B*, or until the connection is closed...*". The second observation cannot always occur if a B→A is dropped within the timeout; the monitor is obligated to watch for the connection to close:



This sort of temporary obligation is analogous to the well-studied "until" formulas in temporal logics, and can be challenging to implement—especially in the presence of timeouts. Furthermore, prior observations may partition the obligation space: a separate obligation is necessary *for each* A, B *pair*: one pair may close its connection, but not another.

2.2 Example: Network Address Translation

NAT alters packets as they cross the switch, which requires additional semantic features to monitor. Suppose we expect that reverse translation is working, i.e., "*Return packets are translated according to their corresponding initial outgoing translation.*" A violation of this property consists of *four* observations: (1) a packet arrival A,P→B,Q[1] from the internal network; (2) the same packet departing with new source A′,P′ (where A′ and P′ are a fresh address and port combination assigned by the switch); (3) a packet arrival B,Q→A′,P′ from the external network; and (4) the same packet departing with destination not equal to A,P.



Feature 5: Maintaining Packet Identity. Feature 1 allows extraction of packet headers: we can observe an arrival and then, some time later, an egress event that shares some of the arrival's fields. This is insufficient to capture "the same packet" in steps (2) and (4) above. Here the monitor needs to connect an arrival and its corresponding set of egress events—information that is most reliably captured on the switch itself.

Dropped-packet detection is a related issue. All the firewall examples above take action when a B→A packet is dropped. As Sec. 3.2 explains, this valuable feature is almost universally unsupported, even by recent versions of OpenFlow.

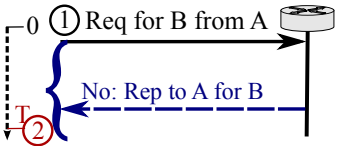
Feature 6: Negative Match. Step (4) detects departures with destinations *not* equal to some previous value. State must therefore be available in a way that can be negatively matched.

2.3 Example: ARP Cache Proxy

An ARP proxy should, either via the controller or through local state, learn address mappings and reply immediately upon seeing a request for a known hardware address. This reply is a different packet from the request, so packet identity

[1] A and B are source and destination layer-3 addresses and P and Q respective layer-4 ports.

(Feature 5) cannot be used here. While the property is certainly violated if the switch *never* sends a reply, we impose a maximum wait time to make checking practical: "*If the switch receives a request for a known MAC address, it will send a reply within T seconds*." Violating this property involves T seconds passing without a reply being sent:



Feature 7: Timeout Actions. Such negative observations (orthogonal to negative matching) trigger when a timeout fires, rather than just expiring state. This differs substantially from the usual meaning of timeout on a switch, and is another feature almost universally unsupported by current switches.

Refreshing timeouts corresponding to negative observations is also subtle. If—like ordinary timeouts (Feature 3)—they were reset whenever the preceding observation fired, a never-answered sequence of requests every (T-1) seconds would not be detected as a violation.

2.4 Additional Semantic Challenges

Finally, some challenges arise regardless of property.

Feature 8: Instance Identification. Monitor state at any given time comprises a set of partially completed attempts to show property violation, which we call *instances*. An instance consists of a set of header values matching previously seen observations, plus the next observation stage to be matched. For example, a monitor instance for the NAT property on the cusp of raising an alert would contain concrete values for the original source and destination `A,P`→`B,Q`, the translated source `A',P'`, and the fact it is now attempting stage (4). When an event occurs, the monitor must decide which instance(s), if any, it advances. This task is more or less challenging depending on data flow between observation stages.

• *Symmetric Match.* In the stateful-firewall properties above, data flow is straightforward. The first observation binds address values `A` and `B` which match (when inverted) return packets. `A,B` pairs thus fully describe instances at any stage.

• *Wandering Match.* The "DHCP + ARP" properties in Tab. 1 extend the ARP proxy example by populating the cache by observing addresses in DHCP traffic. Checking these properties requires *mapping observations with different protocol fields to the same instance*, rather than matching on a fixed set of fields for each stage (e.g., a 5-tuple match).

• *Multiple Match.* Switches can react to information that is out-of-band with respect to the data plane. For example, a learning switch (Sec. 1) may react to a downed link: "*link-down messages delete the set of learned destinations*." If this property fails, it is because a packet from address `D` arrived, followed by a link-down message, and then a packet with destination `D` was unicast without intervening `D`-sourced packets. To handle this, the link-down message must advance one partially-complete instance for each `D` seen so far.

Feature 9: Side-Effect Control. Even in a highly optimized ideal switch, contention for shared state will impact throughput. If a packet must cause a state change, a switch has two options: forward the packet *inline* with state update—blocking forwarding until the update is complete—or asynchronously *split* the forwarding and update actions.

If the switch splits processing, the monitor has minimal impact on throughput, but its state might lag behind any packets issued in response, leading to monitor errors. In contrast, if the switch inlines updates, its state will be up to date, but at the expense of increased forwarding latency, which could also cause incorrect monitor behavior. Some properties and networks may be more forgiving of delay than others. We therefore believe that this option should be explicitly exposed, which *none* the approaches we study in Sec. 3 do.

Feature 10: Provenance. Once a property violation is detected, the monitor is able to send a failure notification. So far that message conveys only the final, trigger event. Ignoring the events leading up to the violation is suboptimal for debugging, but recording each packet that advances an observation is not feasible. Thus, the implementation must provide a balance between *full* provenance and performance.

3. IMPLEMENTATION STRATEGIES

There is an ongoing trend toward more programmable switches, which often leads to adding stateful primitives. However, building a monitor that provides the semantic features discussed in Sec. 2 involves many aspects of a switch's design, of which state is only one component. In Sec. 3.1, we review recent approaches providing state on switches; Sec. 3.2 discusses the different ways in which they fall short of certain requirements for stateful property monitoring.

3.1 Existing approaches to on-switch state

Standard OpenFlow provides basic quantitative state, such as counters and meters, to track flow statistics. Any additional state requires OpenFlow extensions or controller involvement.

Different proposals have added limited on-switch state to offload controller interaction. OpenState [6] proposes a design, based on Mealy machines, for OpenFlow tables that maintain per-flow state. When packets arrive, each is mapped to a state based on pre-determined header fields, after which the state may be modified. This simple state-machine primitive allows functionality like MAC learning, tracking established connections, and port knocking on-switch. FAST [14] encodes similar state machines and provides a software implementation using the Open vSwitch [19] *learn* action, which allows a switch to modify its flow tables as packets are seen. FAST combines *extrinsic* state information and per-flow state by incorporating hash functions, enabling applications such as load balancers and heavy-hitter detection.

POF [20] and P4 [7] describe more generic packet processing architectures. Programs written in P4 define a processing pipeline for match-action switch hardware. POF has similar

objectives, but its implementation is geared toward network processors. Both architectures provide generic, per-flow persistent state that can be updated during packet processing. In addition, they support rich, protocol-independent matching, providing access to a wide range of protocol fields. P4 also describes an "egress pipeline", allowing further processing on switch metadata before packets are forwarded.

These proposals clearly support stateful operations. Indeed, they provide all of the features for monitoring sequences of positive observations that use exact or symmetric instance identification. POF and P4 also give additional support for matching and identifying related events. Despite this added functionality, they do not account for multiple match, timeout actions, or preserving history for provenance.

Looking beyond these generic architectures, SNAP [2] provides a high-level abstraction for stateful operations. SNAP programs operate on a set of persistent global arrays, and enable stateful tests along with advanced matching capabilities. While this provides a powerful language for writing programs, SNAP shares the limitations of P4 and POF (indeed, these are some of its potential target platforms) in terms of support for stateful monitoring. Moreover, SNAP uses the "one big switch" abstraction, relying on its compiler to determine forwarding paths in the network core. This hides details about the behavior of individual switches, which may be of interest to a monitor—highlighting the essential difference in design goals between monitoring and network-control programs.

Varanus [16] is unique among these approaches: it was designed with property monitoring as an explicit goal and so provides a different set of stateful extensions. Varanus's approach encodes each active monitor instance as its own OpenFlow table and uses an extended, recursive form of the Open vSwitch *learn* action to "unroll" instances into new tables as events arrive. This permits multiple instances to be matched at once, as well as allowing timeouts to advance state. Varanus provides these additional monitoring primitives, but at the expense of performance. Sec. 3.2 discusses these primitives in detail as well as their performance tradeoffs.

3.2 Identifying semantic gaps for monitoring

The above approaches add many features to enhance programmability in SDNs. However, they tend to focus on implementing programs that determine packet-processing behavior and thus do not consider the challenges raised when *monitoring the correctness* of that behavior. Tab. 2 observes some common semantic gaps in what these approaches support.

Timeout Actions. Simple rule timeouts have been provided since OpenFlow 1.0, as they are directly applicable to network-control applications. In contrast, Feature 7 describes extending timeouts to perform *explicit actions* that update state for an instance. Using a timeout in this manner has limited usage in forwarding programs, but in monitoring it enables powerful negative observations. Adding such support would entail a new type of event not triggered by a packet, as well as processing the additional state updates. To our knowledge, Varanus uniquely supports timeout actions, using custom extensions to Open vSwitch.

Parsing and match support. Access to a wide range of fields (Feature 1) is important, and not only for monitoring. Programmable parsing is becoming more prevalent in proposed SDN architectures like P4. A more critical gap for monitoring is the ability to parse and match on a switch's metadata information, such as its output port, as it traverses the processing pipeline. This is useful for checking properties based on the switch's behavior, like determining if the output port is correct and discerning multicast from unicast. This could be implemented by adding pipeline stages (*i.e.* tables) for monitoring after making output decisions. Matching on output port is supported in OpenFlow 1.5's egress tables, but dropped packets never enter the egress pipeline, which increases the difficulty of extracting this information. P4 is unique in considering this requirement, revealing an important design gap in most current proposals.

State updates. Monitoring applications may use persistent state quite differently from network-control applications. In a stateful firewall application that tracks connections, state updates may not be necessary after a connection is established. However, when *monitoring* the firewall, state updates may be required more frequently—potentially on every packet. Thus, monitoring poses new design challenges in how state updates occur. This is true regardless of whether the switch is implemented in hardware, software, or as a hybrid network processor. We discuss some of our experiences with state updates when discussing performance challenges in Sec. 3.3.

Provenance. Network-control programs have little need to preserve packet history information beyond existing requirements for persistent state (like tracking open connections). In contrast, an operator may wish to know *what led up to* a property violation. Supporting this is clearly challenging due to the extra state required, although rich approaches like POF/P4 could theoretically provide it. Moreover, *limited* provenance could be recovered without added cost: since some header information is retained for matching purposes, those values (where used in the final observation) could be conveyed along with the final event. A more complete provenance could be selectively constructed via an approach like NetSight [9], which sends postcards to a central monitoring server.

Instance identification. All of the approaches discussed provide some form of "per-flow" persistent state. Broadly, a "flow" defines a set of related packets that should map to the same state, such as all traffic in a TCP connection or local to a switch port. Methods for mapping a packet to a particular flow state differ by implementation, including: using specific fields to index into a state table (OpenState), arbitrary hash functions (FAST), or user-defined methods in programmable architectures (P4, POF, and SNAP).

Monitoring can require subtly different criteria for mapping packets to states, since a monitor tracks *instances* of partially completed observations (Feature 8). This differs from typical definitions of a "flow": instances are defined by

Semantic Challenge	OpenFlow 1.3	OpenState [6]	FAST [14]	POF [20] and P4 [7]	SNAP [2]	Varanus [16]	Static Varanus
State mechanism	Controller only	State machine	*Learn* action	Flow registers	Global arrays	Recursive *learn*	Recursive *learn*
Update datapath	—	Fast path	Slow path	Fast path	Fast path	Slow path	Slow path
Processing Mode	Inline	Inline	Inline			Split	Split
Event History		✓	✓	✓	✓	✓	✓
Identification of related events	✓ (1.5 only)			✓	✓	✓	✓
Field access	Fixed	Fixed	Fixed	Dynamic	Dynamic	Fixed	Fixed
Negative match	✓	✓	✓	✓	✓	✓	✓
Rule timeouts	✓	✓	✗	✓	✗	✓	✓
Timeout actions	✗	✗	✗	✗	✗	✓	✓
Symmetric match		✓	✓	✓	✓	✓	✓
Wandering match		✗	✗			✓	✓
Out-of-band events		✗	✗	✗	✗	✓	✗
Full provenance		✗	✗	✗	✗	✗	✗

Table 2: Comparison of existing approaches. A ✓ indicates that a work provides a given semantic feature. An ✗ means that the approach's architecture precludes implementation of a feature, or indicates that it is not included in its design. Where a feature does not apply or where support is unclear, we leave a blank space. We limit our analysis of OpenFlow to actions supported in version 1.3 (1.5 for egress matching) without controller interaction. Since the state processing mode and hash function support for wandering match in POF, P4, and SNAP is target dependent, we leave these spaces blank.

an observation's timestamp and previous history, in addition to packet header information. Supporting wandering match stretches the definition of a flow to span multiple protocols, which complicates parsing packets for a single state machine. Multiple match semantics requires advancing more than one instance per packet event to support out-of-band events. To our knowledge, this challenge is unique to monitoring and is prohibitively costly in current methods since it differs greatly from well-studied match-action semantics.

Varanus enables multiple match by separating monitor instances into different OpenFlow tables. This uniquely supports out-of-band events, but at a high cost. Since Varanus isolates each instance in its own table, the depth of the switch pipeline (*i.e.* the number of tables to check) is no smaller than the number of active instances, which is infeasible in practice.

3.3 Key Performance Challenges

Developing switch primitives for monitoring that function at line rate is a challenging task due to the increased requirements for matching and persistent state that go beyond even relatively new proposals for stateful forwarding. In this section, we discuss our experiences in building the Varanus prototype as an example of the inherent tradeoffs between monitoring features and performance in switch design.

As Tab. 2 shows, Varanus offers a great deal of expressive power for monitoring compared to other on-switch works. However, Varanus does not scale for two reasons: the number of active instances determines the pipeline depth, which can greatly affect packet processing time; and its OpenFlow extensions for maintaining state cannot achieve a high throughput. The former issue can be mitigated by bounding the number of monitoring tables in the switch pipeline, which provides, in principle, a constant packet processing time, at the expense of some expressivity. Limiting the processing pipeline to one table per observation stage preserves support for wandering match while sacrificing support for out-of-band events, which required an unbounded amount of tables. We view removing multiple-match support as a reasonable tradeoff since our experiences (and Tab. 1) indicate that properties involving out-of-band events are relatively rare.

However, even this "static" Varanus remains an intractable approach so long as it stores and updates its state using OpenFlow rules, which cannot be modified at line rate. A scalable implementation would need to involve more rapid state mechanisms, such as the register-based approach in P4. Monitoring on-switch unavoidably incurs a latency cost, however small, since it lengthens the switch's pipeline. Where the switch's regular stateful operations overlap with the monitor's, some redundancy in computation and storage is to be expected, and limiting overhead via appropriate tradeoffs will be vital.

4. CONCLUSION

New architectures are driving the future of SDN by increasing switch programmability. In light of this inexorable trend, it is vital to identify unconsidered use-cases for switch programmability, such as stateful property monitoring. We show that the requirements of stateful monitoring go beyond the typical concept of "per-flow" state: they interact deeply with other switch design concerns such as parsing, pipelining, and event capture. This reveals significant gaps between the requirements of monitoring and the state-of-the-art.

For simplicity, our scope has been limited to properties that can be monitored using a single switch and expressed with boolean conditions, rather than quantitative measurements. Even with these limits, we have identified ways that existing approaches fall short of the features that property monitoring requires. We hope that our experiences motivate switch designers to consider stateful monitoring as a use-case in their future work.

Acknowledgments.

We are grateful to Theophilus Benson and the anonymous reviewers for their insightful feedback. This work was partially supported by the NSF.

5. REFERENCES

[1] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In *Principles of Programming Languages (POPL)*, 2014.

[2] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker. SNAP: Stateful network-wide abstractions for packet processing. In *Conference on Communications Architectures, Protocols and Applications (SIGCOMM)*, 2016.

[3] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Program monitoring with LTL in EAGLE. In *International Parallel and Distributed Processing Symposium*, 2004.

[4] D. Basin, F. Klaedtke, S. Müller, and E. Zălinescu. Monitoring metric first-order temporal properties. *Journal of the ACM*, May 2015.

[5] R. Beckett, X. K. Zou, S. Zhang, S. Malik, J. Rexford, and D. Walker. An assertion language for debugging SDN applications. In *Workshop on Hot Topics in Software Defined Networking*, 2014.

[6] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. OpenState: Programming platform-independent stateful OpenFlow applications inside the switch. *ACM Computer Communication Review*, 2014.

[7] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *ACM Computer Communication Review*, 2014.

[8] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *Networked Systems Design and Implementation*, 2015.

[9] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *Networked Systems Design and Implementation*, 2014.

[10] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *Networked Systems Design and Implementation*, 2013.

[11] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. J. Clark. Kinetic: Verifiable dynamic network control. In *Networked Systems Design and Implementation*, 2015.

[12] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *Euromicro Conference on Real-Time Systems*, 1999.

[13] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *Conference on Communications Architectures, Protocols and Applications (SIGCOMM)*, 2011.

[14] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan. Flow-level state transition as a new switch primitive for SDN. In *Workshop on Hot Topics in Software Defined Networking*, 2014.

[15] S. Narayana, M. Tahmasbi, J. Rexford, and D. Walker. Compiling path queries. In *Networked Systems Design and Implementation*, 2016.

[16] T. Nelson, N. DeMarinis, T. A. Hoff, R. Fonseca, and S. Krishnamurthi. Compiling Stateful Network Properties for Runtime Verification. *ArXiv e-prints*, July 2016. http://arxiv.org/abs/1607.03385.

[17] T. Nelson, A. D. Ferguson, M. J. G. Scheer, and S. Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *Networked Systems Design and Implementation*, 2014.

[18] T. Nelson, D. Yu, Y. Li, R. Fonseca, and S. Krishnamurthi. Simon: Scriptable interactive monitoring for SDNs. In *Symposium on SDN Research (SOSR)*, 2015.

[19] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of Open vSwitch. In *Networked Systems Design and Implementation*, 2015.

[20] H. Song. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In *Workshop on Hot Topics in Software Defined Networking*, 2013.

[21] N. L. M. van Adrichem, C. Doerr, and F. A. Kuipers. OpenNetMon: Network monitoring in OpenFlow software-defined networks. In *Network Operations and Management Symposium*, 2014.

[22] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: Enabling record and replay troubleshooting for networks. In *USENIX Annual Technical Conference*, 2011.

[23] W. Zhou, O. Sokolsky, B. T. Loo, and I. Lee. *DMaC*: Distributed monitoring and checking. In *Runtime Verification*, 2009.