Runtime Verification for P4 Networks

Benny Rubin
Cornell University
bcr57@cornell.edu

Sundararajan Renganathan
Stanford University
rsundar@stanford.edu

Nate Foster
Cornell University
jnfoster@cs.cornell.

Abstract

Software defined networking (SDN) has changed the landscape for innovation and control in campus and enterprise networks. While it has simplified much when it comes to network management, SDN has added new complexities and surface area for misconfigurations, bugs, or even malicious behavior in a network. However, we can leverage the programmability of networks to provide end to end verification of packet behavior. Tiny Packet Checkers (TPC) is an approach that provides real time, runtime verification of every packet in the data plane. Network properties are compiled into monitors (or *tiny packet checkers*) that are run at each switch in a packet's path. The monitor collects data from packets and the checker verifies that no packet violates any properties. Errant packets are stopped and sent to the control plane for further analyses.

TPC incurs modest overhead, stemming from additional checker computations and packet header data, when compiled to switch hardware. It is possible to reduce this overhead by capturing network state in the form of packet telemetry and only checking for property violations at the end of a packet's journey. We introduce a class of properties that can be checked at the last hop, either on a leaf switch or a smart NIC on the end-host. This approach reduces strain on the network core switches and allows for more compute-heavy hardware to run the checker. We also provide a set of TPC programs written using last-hop semantics and an analyses of performance tradeoffs.

I. Introduction

In the past 2 decades, the way that networks are configured and operated has been changing significantly. Historically, these complex networks of switches, routers, NAT and firewall middleboxes, and other specialized hardware ran proprietary closed software that undergo years of testing and standardization. Additionally, they are configured using interfaces that vary across vendors and products. This approach as slowed innovation and led to unnecessary complexity [3].

These days, we are seeing a paradigm shift towards programmable networks, where a piece of hardware can act as a switch, NAT device, router, firewall, etc. depending on how they are configured. The defining characteristic of these software defined networks is the separation of the *control plane* from the *data plane.* The control plane is responsible for deciding *how* to handle network traffic, and the data plane is responsible for forwarding the traffic based on information it receives from the control plane [3].

SDN control protocols like OpenFlow don't provide the flexibility to deal with the increase of complexity and protocols that appear in modern networks. The development of P4, a domain specific language for programming packet parsers, was a major step forward in the design of programmable networks as it allows for three major goals. It allows for reconfiguring the way that switches process packets once they are deployed by simply compiling a new program to run on it. Network hardware is abstracted away from specific network protocols. Finally, packet processing functionality is completely separated from the underlying hardware. These goals are achieved through the P4 compiler and runtime environment [4]. The rest of this paper assumes mild familiarity with P4.

P4 paves the way for more innovation and complexity in software defined networks. This introduces a new set of challenges for verifying correct functionality of programmable networks. A network can experience incorrect behavior for any number of the following reasons: a bug in the P4 program, incorrect match-action rules installed into the data plane by a controller, a misconfigured or faulty piece of hardware, or malicious attacks. Often these bugs only manifest themselves once a packet has gone through a specific sequence of switches and tables.

Runtime verification is a technique that can verify the runtime behavior of a system in real time. The approach in this paper is called Tiny Packet Checkers (TPC), which performs runtime verification in the data plane. TPC has a domain specific language that describes a set of runtime properties that we expect packet in the network to satisfy and compiles them into tiny packet checkers that run in the data plane along with the P4 forwarding code. These properties are checked by switches at line rate. Packets that do not satisfy these properties can be alerted on and dropped. TPC programs are compiled to P4 and linked with the P4 code running on switches.

The work presented in this paper will reason about the feasibility and tradeoffs of checking properties on the last hop of a packet. This reduces the overhead of checking during each hop of a packet's journey yet requires more telemetry data to be attached to each packet. This approach has a number of benefits.

Typically, network hardware on the edge of the core has more resources and support more complex interfaces and functionality; by saving the checking phase until the last hop, we allow for fewer requirements and less overhead in the devices along the route. Additionally, this approach may serve better for certain networks, so by implementing a TPC compiler module that can be configured to translate programs between every hop checking and last hop checking, we add flexibility to the TPC runtime verification system.

It is worthwhile to mention several degradations that could be incurred. It is possible that last hop checking reduces the accuracy of pinpointing where failures occur along the path. This approach also loses the benefit of immediately dropping packets that violate the specified properties.

The contributions of this paper include:

1. We present a practical system and semantics for checking properties at the last hop instead of hop by hop
2. We introduce a class of properties that can safely be checked at the end
3. We wrote a set of TPC programs that check properties at the end, along with a discussion on the tradeoffs of hop-by-hop and last hop property checking

II. Related Work

There has been extensive work in the area of static analyses for programmable networks and P4 programs. P4Assert uses annotated assertions in P4 programs to verify a model using symbolic execution [1]. This approach is able to quickly evaluate various p4 applications to verify correctness and uncover bugs. P4v allows you to specify a control-plane interface that specifies the proper behavior of a P4 program [6].

These classes of static verification tools while helpful for catching bugs at compile time, cannot yet catch every possible set of bugs and furthermore are restricted to the level of the P4 program and cannot detect errors in the compiler, switch hardware, or tables filled in by the control plane at runtime. These static verification approaches are complementary to TPC and should be used alongside it for additional assurance.

Runtime verification, on the other hand, can catch bugs, configuration mistakes, or malicious behaviors that stem from a number of sources during the execution of the P4 program on the network. However, there has not been a lot of work in this area. The following section will explain why TPC is novel compared to the literature that exists on runtime verification for SDN.

P4Consist uses probe packets with special tags that collect telemetry data that are forwarded to dedicated servers that compare expected network behavior to the ground truth behavior from the probes. [5] However, the tags are only added to the special probe packets. Additionally, the verification happens offline, meaning inconsistencies cannot be detected at line rate as they occur. TPC checks for property violations on every single packet and the check is done on switch, rather than requiring a dedicated server. DBVal implements assertions in P4 that can verify runtime behavior in the dataplane, however it focuses network behavior on the execution of a single switch, while TPC can capture network-wide properties such as loops and slicing [1]. Thus, TPC introduces novel concepts in the area of runtime verification for programmable networks.

```
bit<32> slice;

for switch in path {
    init {
        if (%path_length == 1)
            slice = @switch_slice;
    }

    checker {
        if (slice != @switch_slice) {
            reject;
        }
    }
}
```

Fig. 1. TPC program for end-to-end slicing

III. The TPC Language

The following is a brief overview on the semantics and syntax of the TPC language. Note, this work was done by Sundararajan Renganathan, advised by Nick McKeown at Stanford, in collaboration with Nate Foster's research group at Cornell. TPC programs are compiled to P4 and linked with the P4 code that runs on the programmable switches in a network alongside the forwarding code. While TPC is still a specification language, it is written as a program in a scripting language. This style is more familiar to programmers than typical logical specifications frameworks such as Linear Temporal Logic. In addition to traditional types, expressions, and statements in imperative programming languages TPC has a few constructs that are specific to the language.

It is easiest to understand TPC through an example. A simple property to illustrate the expressibility of TPC is end-to-end slicing. In network slicing, each switch is assigned a slice and a packet may only traverse switch's allocated to the same slice. A common use case of this is VLAN isolation.

Figure 1 is a TPC program that enforces slicing at each hop. The first thing to note is that at the top level, the program is written as a for loop that models the packet's journey through each switch. At the first hop, the slice variable is set to the slice of the current switch. At each subsequent hop, this value is checked against the current slice of the switch. If any of them are not equal, the packet is rejected. The slice variable is called a checker variable, and has no annotation. This means the data is carried with the packet for verification. P4 has the ability to add in-band telemetry data to packets, which is one of the SDN features crucial for TPC and last hop checking.

Forwarding variables are prepended with a %, such as %path_length, and correspond to the headers of the packet and metadata of the P4 program. Finally, static variables correspond to configuration data and information that is managed by the control plane, such as @switch_slice. As evident, they are written with an @ symbol. There is another type of variable called sensor data that is collected and aggregated by the switches, but is not useful for last hop checking.

The init block is run at the start of the ingress pipeline, when the packet is first read in at the switch. The checker block is then run at the end of the egress pipeline, when the packet is about to be sent out. This allows them to capture different state, before the forwarding program is run on the switch. There is also an optional sensor{} block that can be used to keep track of state at a switch to be used over a flow of multiple packets, however it is not important for the discussion in this paper.

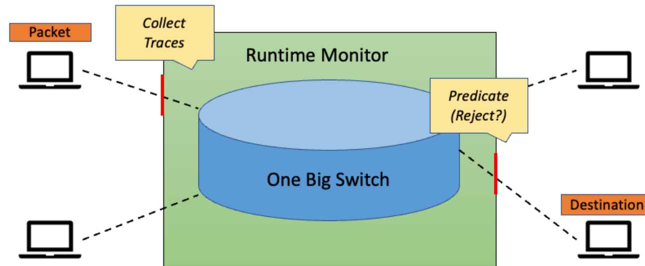

Fig. 2. Model of TPC as a Runtime Monitor

IV. TRACES AND PREDICATES

While TPC has a domain specific language for easily encoding properties, at the highest level of abstraction it is a runtime monitoring system that collects traces on packets, comprised of network state as it traverses the network, and a predicate that runs on a trace and either halts the packet or allows it to be forwarded to the end host.

In classical software runtime verification a runtime monitor is instrumented into a software system to observe the behavior and determine whether or not it violates a correctness specification [9]. More recent work on runtime monitoring for other application domains defines a useful abstraction for thinking about a runtime monitor. The monitor is instrumented into the system in such a way that it collects traces of the runtime execution. These traces can capture state at certain points, checkpoints, or often interactions with outside systems. In offline runtime verification, the entire trace is passed into a checker that verifies that certain predicates hold. In an online system, the predicates are run as the trace is collected. These predicates correspond to properties that the system administrator want to verify. This approach has been used in distributed systems, security applications, and even in hardware [8].

To view the network verification problem as a classical runtime verification scheme, it is helpful to think of a network as one big switch. Packets enter the switch from a source end host and then are routed to their destination host. The details of the network are not important in this model. As can be seen in the model in figure 3, there is a monitor around the network that captures the state as packets traverse the network. This is analogous to how a runtime verification system would be instrumented to see the trace of a program as it executes. When the packet exits the network, before it is forwarded to the end host, the monitor checks that the trace passes a predicate. At the highest level, this is all that TPC does.

The language was originally designed so that properties are checked at each hop of a packet's journey. However, this runtime monitor abstraction does not specify where a predicate is run on a trace, only that it is either halted or allowed to continue before it reaches the end host. It is possible to reason about properties that could be checked only at the end of a packet's journey, either in the leaf switch or a smart NIC aboard the endhost. Instead of checking the property at each switch, the necessary telemetry data could be forwarded along with the packet in the data plane and then it could be checked before it reaches the end-host either at the leaf switch or a smart NIC. By using the one big switch model, it is easy to reason about last hop checking in the same way as hop by hop checking – packets that violate a property are not delivered to the destination and instead are sent to the control plane. This is analogous to an offline check for runtime verification, where hop by hop can be seen as online.

The TPC abstraction only specifies the trace to be collected and the predicate on the trace. Checking the predicate hop by hop or at the edge does not affect the property being checked. If you model the predicate as a state machine, S that operates on a trace T. S either enters a rejecting state and halts the packet or allows the packet to continue to its destination. T contains a sequence of state $(t_0, t_1, t_2, \ldots, t_n)$ the packet picks up as it traverses the network. In the edge check, thinking of the network as one big switch, the state machine would take in the trace once it has been fully collected and either forward the packet to the end host or reject it. By in-lining the check into the network, the state machine, including the transition function, does not change. The trace itself is also independent of where the check occurs. The only difference is that the state machine is distributed to different pieces of hardware and the transitions occur as the trace state becomes available, rather than at the end. Thus, these two approaches are equivalent. Another way to think about this is that one could have an offline verification system or an online monitor for the same, equivalent correctness specification.

Where the check happens is flexible and up to the programmer. It is easy to visualize the check happening at the edge, as the packet is leaving the monitor, as in figure 3, however it can be in-lined into the network and distributed to switches inside, as seen in the program in figure 1. This has the advantage of immediately dropping packets as soon as TPC knows it will eventually be halted. There exists some total compiler that can translate the same equivalent program between edge checking and hop by hop checking by in-lining the program into the network. This is considered future work for this paper.

There is a class of properties that cannot be checked at the edge. The only case is when the packet does not reach the edge, as then the trace is lost, and the predicate cannot be checked. Such examples are if a packet is stuck in an infinite loop or if it is dropped in a black hole. In this case, other tools can be run alongside TPC (such as hop by hop checking for these violations), to ensure packets make it to the edge.

```
bit<32> slice1;
bit<32> slice2;
bit<32> slice3;
bit<32> slice4;

for switch in path {
  init {
      if (%path_length == 1){
        slice1 = @switch_slice;
      }
      if (%path_length == 2){
        slice2 = @switch_slice;
      }
      if (%path_length == 3){
        slice3 = @switch_slice;
      }
      if (%path_length == 4){
        slice4 = @switch_slice;
      }
  }
  checker {
      if (last_hop) {
        if (slice1 & slice2 & slice3 & slice4 != slice1) {
          reject;
        }
      }
  }
}
```

Fig. 3. TPC program for last hop checking of slicing

V. Checking Property Violations

A. Hop by Hop Checking

As seen in the example in fig. 1, the for loop specifies that the initialization and checker block is run at each switch. This allows for the switch to keep track of the state of the network in memory as packets are forwarded through. Additionally, it means that if the checker finds a packet that has violated a property, it can immediately be dropped and sent to the control plane. This is the main advantage of the hop-by-hop checking approach. There are, however, a number of limitations. Due to the fixed number of pipeline stages in popular programmable switches, the entire TPC program might not fit on all pieces of hardware. Additionally, the network core often has little room for overhead and more congestion. Compared to edge switches and smart NICs, there is much less available compute.

B. *Last Hop Checking*

It is possible to reason about properties that could be checked only at the end of a packet's journey, either in the leaf switch or a smart NIC aboard the end host. Instead of checking the property at each switch, the necessary telemetry data could be forwarded along with the packet in the data plane and then it could be checked before it reaches the end-host either at the leaf switch or a smart NIC. By viewing the network core as one big switch, it is easy to reason about last hop checking in the same way as hop by hop checking – packets that violate a property are not delivered to the destination and instead of sent to the control plane. Additionally, some core switches may not have certain functionality. There could be restrictions on accessing stateful registers, depending on the hardware, and limited resources for adding new logic to the pipeline.

This approach significantly reduces overhead on network core switches and still acts as a runtime verification system that can check every packet as it traverses the network. It is semantically straightforward to convert a TPC program for a property from hop-by-hop checking to last hop checking. There are 2 necessary pieces of information:

- Telemetry data to keep track of
- Predicate on the data to check for a violation

Each switch along the path will forward the telemetry data and the final hop will do the checking. P4 allows for a number of fields to be collected as packet telemetry. This includes packet headers, switch state such as table content, registers, queue lengths, and parser statistics. Any property that uses predicates on this data can be expressed as a last hop TPC program. Figure 2 is an example of a TPC program for last hop checking. For simplicity, it assumes each path is 4 hops. With compiler improvements, the trace would be modeled as a list and the predicate would be verifying that all the elements are the same (by the transitive property on the first element).

The main difference is that instead of doing the check at each hop, the telemetry data is simply forwarded with the packet and then the check happens all at once at the end.

Unfortunately, TPC does not currently support lists. We are working on compiler support for this feature. Thus, the code is not very elegant, but it will be greatly improved soon.

As shown in the example, right now the only way to check for properties at the end is with a conditional in the checker block. However, this still requires the program to be compiled to every switch in the path. We plan on optimizing this by only compiling the required blocks to each switch, so only the leaf switches will include the checker.

Below is a (non-exhaustive) list of properties that can be expressed in TPC and have been implemented as equivalent hop by hop and edge checking TPC programs. Each of the entries contains a property name, a description of the property, the trace to be collected, and the predicate on the trace. The form is as follows:

- Property Name
  - Description
  - Telemetry Data
  - Predicate on Data for Rejection

- Slicing
  - Each packet only traverses through the same slice
  - The slice of each switch in the path
  - Not all the slices are the same
- General loop
  - No packet traverses the same switch twice
  - Each switch in the path
  - A switch is visited twice
- VLAN Isolation
  - All packets traverse the same VLAN
  - The VLAN of each switch
  - Packet enters a different VLAN
- Leaf Spine Invariant
  - First and Last hop of a packet's path are leaf switches
  - The state of the first switch and last switch
  - First switch or last switch is not a leaf switch
- Reachability
  - host s is reachable from host t
  - Reachability matrix from control plane and the end host
  - If host t is not reachable from the last hop
- Isolation
  - Negation of Reachability
- Waypointing
  - Packets sent from s can reach t, going through switch w
  - A Boolean value, whether the packet has gone through switch w
  - w is false
- Egress port validity
  - Packets may only egress the network at allowed ports
  - Destination port of egress leaf switch and allowed ports from control plane
  - The destination port is not an allowed port
- Path length validation
  - The length of the path is as expected

- The number of switches in the path
- The number of switches in the path is not the same as the expected distance

- Path validation
  - The path taken by a packet is as expected
  - Each switch in the path
  - The path is not the same as the expected path

VI. ENGINEERING TRADEOFFS

Besides the set of properties that cannot be checked at the edge, the only differences between the two approaches are practical engineering tradeoffs. By giving network administrators the ability to choose how to implement their properties, TPC remains flexible to different network requirements.

One of the biggest advantages of edge checking is that it allows for incremental deployment of TPC to networks. P4 programmable switches are expensive and are not commonly used in production networks. However, we are seeing more and more switches adding the capability to add in-band telemetry to packets. Checking at the edge allows for switches that cannot be arbitrarily programmed, yet can add telemetry to packets, to be verified by TPC. When the packet reaches the edges, either a programmable switch or more likely a smartNIC or even the end host kernel, can perform the property violation check. This one property of edge checking allows for a significantly higher number of networks to support TPC. As networks migrate towards deep programmability, it is important to add incremental support to tool that leverage it.

Another point of tradeoff between the two approaches is telemetry overhead. The amount of telemetry overhead depends on the property. Certain properties, such as maximum hop length, only require a single integer to be carried along with the network. This integer can be thought of as mutable state that each switch updates as the packet traverses the network. However, other properties, such as ones that are dependent on each switch in the path (e.g. switch slice isolation), can require more telemetry. In this case, the amount of telemetry scales with the diameter of the network, as each switch adds 4 bytes to the list of switch slices. This list can be thought of as an immutable local variable that each switch appends data to. For this property, checking hop by hop only requires 4 bytes total, while checking at the edge requires 4 bytes per switch. Often, especially if the computation is simple, it is better to perform it on each switch rather than carrying significantly more telemetry. However, it is convenient, when a switch doesn't support deep programmability, to be able to append the data to the packet to be processed later.

The network core is highly congested, responsible for forwarding all packets to their destination. Especially in modern networks, a little bit of overhead can be the difference between line rate forwarding and significantly slower throughput. In fact, the P4 compiler will not even allow you to install a program if it uses more pipeline space than is available. Thus, to realistically support TPC, it is important to have flexibility in where the computation takes place. By moving the computation from the switches in the core that highly congested and sensitive to small overheads, to the switches or smartNICs at the edge that have more available compute and often more capabilities for programmability, edge checking is easier to support; in some networks it might be the only option.

Finally, checking hop by hop has the benefit that packets are halted *as soon* as it violates the property. If you know a packet will eventually be dropped instead of forwarded to its destination end host, it clearly reduces strain on the network to drop it immediately rather than allowing it to add to the congestion of the network and must be processed at every subsequent switch. Edge checking guarantees the same level of correctness, but the packet will necessarily travel to the edge before it is dropped.

Qualitatively, there are significant differences between writing programs for the edge and hop by hop. In our experience, it is easier to reason about the behavior of a TPC program that specifies the telemetry to add to a packet and a predicate on the telemetry. This is especially the case when designing the language to support list constructs and comprehensions on lists. Additionally, this style of programming is more conducive to the high level abstraction of viewing a TPC program as a trace and a predicate on a trace. Thus, the lists specify the trace (i.e. the state collected at each switch) and the list comprehensions specify a predicate on the trace. Meanwhile, it can be tricky to reason about the behavior of a program that is running the check at each hop, as the predicate is working on a partial trace and is thus a "partial" check.

When designing the compiler, the edge check has a full view of the trace, while the hop by hop checker only performs "partial" checks on partial traces. Thus, a program written in the edge check format (i.e. specifying a trace and a predicate on the trace) could be easily optimized to be compiled to different P4 programs to be run only on the necessary switches, introducing the least amount of overhead, while a hop by hop program might have to be manually written separately for each switch. This is because the hop by hop program semantically specifies the behavior of a single switch on the partial trace it has so far, rather than the entire trace available at the edge.

The language design itself only serves to make it easy for a programmer to share their intent with the system and to make it easy for the system to compile the program to the switches. It is important to note that

independently of the language design, the TPC program should be able to run as a hop by hop check or as an edge check. This qualitative experience only serves to show that it appears easier to specify programs as edge checks and to translate them to hop by hop programs as needed than to specify programs as hop by hop programs and translate them to edge checks.

VII. CONCLUSION

The emergence of software defined networking enables innovation in ways that hasn't been seen before, but also comes with new surface area for vulnerabilities and bugs in the P4 compiler, P4 programs, control plane, and hardware. Tiny Packet Checkers is an approach that leverages these new programmable networks to check the correctness of every packet at runtime as it traverses the network. In this paper, we have introduced a high level abstraction for TPC that is independent of where the check happens. We have also shown the equivalence of checking at each hop and at the edge, along with example properties and a discussion of the practical engineering tradeoffs of each approach. Checking at the edge enables incremental deployment and reduces strain on the network core, but leads to higher telemetry overheads and does not halt packets as soon as they violate the correctness specification. Ultimately, the ability to switch between different methods for checking properties allows for flexibility in how operators run TPC programs.

VIII. FUTURE WORK

There are numerous directions to go for future research. Verification for programmable networks, and even software define networking as a whole is an emerging field that requires a lot of work before it comes feasible or cost effective. The first direction for future work is to make improvements to the compiler and add language support for checking at the edge. The most important feature is to add list support so that traces can be modeled as lists, that can be arbitrarily long, depending on how many hops.

Another nice feature would be a total compiler that can translate between equivalent hop by hop and edge checking programs. This would give flexibility to operators on how they want to run TPC, depending on what their network topology looks like and how much support they have for P4 programmability. Together with this is a compiler that can compile a program to different switches, depending on the topology. This would reduce the overhead on all switches, except for the necessary ones. For example in waypointing, the waypoint switch is the only one in the network that really needs to add any state to the trace.

Finally, we are working on a testing framework for TPC. This either requires a test network with programmable switches, which is costly and difficult to setup, or a simulation framework which is a project in and of itself to deploy. For these reasons, there has not been a full scale experiment or testing of the different approaches to running TPC, however this is currently being worked on. These results would allow us to quantitatively compare the effectiveness of hop by hop checking and edge checking with different properties and network topologies.

REFERENCES

[1] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. Uncovering bugs in p4 pro- grams with assertion-based verification. In Proceedings of the Sympo- sium on SDN Research, SOSR '18, New York, NY, USA, 2018. Association for Computing Machinery.

[2] K Shiv Kumar, K Ranjitha, PS Prashanth, Mina Tahmasbi Arashloo, U Venkanna, and Praveen Tammana. Dbval: Validating P4 data plane runtime behavior. 2021.

[3] Feamster, N., Rexford, J., Zegura, E.: The road to SDN: an intellectual history of programmable networks. ACM SIGCOMM Comput. Commun. Rev. 44(2), 87–98 (2014)

[4] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese et al., P4: programming protocol-independent packet processors. ACM SIGCOMM Comput. Commun. Rev. 44(3), 87–95 (2014)

[5] Apoorv Shukla, Seifeddine Fathalli, Thomas Zinner, Artur Hecker, and Stefan Schmid. P4consist: Toward consistent p4 sdns. IEEE Journal on Selected Areas in Communications, 38(7):1293–1307, 2020.

[6] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. P4v: Practical veriication for programmable data planes. In Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '18, page 490–503, New York, NY, USA, 2018. Association for Computing Machinery.

[7] Ran Ben Basta, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. 2020. PINT: Probabilistic in-band network telemetry. In *Proceedings of the ACM SIGCOMM Conference.* 662-680.

[8] Sánchez, C., Schneider, G., Ahrendt, W. et al. A survey of challenges for runtime verification from advanced application domains (beyond software). Form Methods Syst Des 54, 279–335 (2019)

[9] Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingolfsdottir. A survey of runtime monitoring instrumentation techniques. Electronic Proceedings in Theoretical Computer Science, 254:15-28, 09 2017

[10] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling innovation in campus networks. ACM SIGCOMM Computer Com- munication Review 38, 2 (2008), 69–74.

[11] P. Bosshart et al., "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN", Proc. ACM SIGCOMM, pp. 99-110, 2013.

[12] George Varghese, Nuno Lopes, Nikolaj Bjorner, Andrey Rybalchenko, NickMcKe-own, Dan Talayco. 2016. Automatically verifying reachability and well-formedness in P4 Networks. Technical Report.

[13] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, C. Raiciu, Debugging P4 programs with Vera, in: Proceedings Of The 2018 Conference Of The ACM Special Interest Group On Data Communication, 2018, pp. 518–532.

[14] Kodeswaran Suriya, Arashloo Mina Tahmasbi, Tammana Praveen, and Rexford Jennifer. 2020. Tracking P4 program execution in the data plane. In Proceedings of the Symposium on SDN Research. Association for Computing Machinery, New York, NY, 117–122.

[15] Srinivas Narayana, Mina Tashmasbi Arashloo, Jennifer Rexford, and David Walker. 2016. Compiling Path Queries. In Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation. USENIX Association, 207--222.

[16] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. 2015. Checking Beliefs in Dynamic Networks. In 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15). USENIX Association, Oakland, CA. 499–512.

[17] Xiang, Qiao et al. “Switch as a Verifier: Toward Scalable Data Plane Checking via Distributed, On-Device Verification.” ArXiv abs/2205.07808 (2022)