P4-InTel: Bridging the Gap between iCF Diagnosis and Functionality

Lucas Castanheira
lbcastanheira@inf.ufrgs.br
Institute of Informatics
UFRGS

Alberto Schaeffer-Filho
alberto@inf.ufrgs.br
Institute of Informatics
UFRGS

Theophilus A. Benson
tab@cs.brown.edu
Brown University

ABSTRACT

Data plane programmability promotes a new kind of computing paradigm in which parts of an application's execution can be offloaded into the network. However, this in-network compute functionality (iCF) adds an extra layer of management complexity for the tracing and debugging of distributed applications. Specifically, current programmable hardware does not provide powerful enough primitives or abstractions to enable in-network tracing. Further, existing distributed application debug solutions do not extend directly into programmable data planes.

In this paper, we take a step back and revisit the fundamental problem by discussing open research questions and challenges towards a comprehensive iCF telemetry and debugging solution which bridges the gap between traditional and iCF-based debugging. To this end, we introduce a system, P4-InTel, which (i) leverages network telemetry to instrument PDPs into monitoring arbitrary trace data, indicated directly on PDP source code using annotations, and (ii) collects and encapsulates this data in a tracing abstraction. This abstraction provides a global vision of an in-network computation's life-cycle in a standard, readable format, which can either be fed to automatic debugging tools, or used by programmers to facilitate troubleshooting.

CCS CONCEPTS

• **Networks → Programmable networks**; **In-network processing**; **Network management**; *Network monitoring*; Programming interfaces.

KEYWORDS

In-Network Compute, Telemetry, Debugging

ACM Reference Format:
Lucas Castanheira, Alberto Schaeffer-Filho, and Theophilus A. Benson. 2019. P4-InTel: Bridging the Gap between iCF Diagnosis and Functionality. In *1st ACM CoNEXT Workshop on Emerging in-Network Computing Paradigms (ENCP '19), December 9, 2019, Orlando, FL, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3359993.3366648

ENCP '19, December 9, 2019, Orlando, FL, USA

ACM ISBN 978-1-4503-7000-4/19/12…$15.00
https://doi.org/10.1145/3359993.3366648

1 INTRODUCTION

Programmable data planes (PDPs) allow the execution of server applications to cross into the network, offloading parts of the computation to PDPs, i.e., in-network compute (iCF) [1]. In light of this development, both industry and researchers have begun actively investigating new designs for classic distributed applications in order to improve performance, scalability, or reliability of these by offloading functionality into the network. NetCache [7], for example, is a system that caches key-value pairs on switches close to server applications, potentially avoiding the long RTTs of a lookup on a remote key-value storage server.

As these new-found approaches near deployment, practical concerns arise about their run-time management, because distributed applications can now cross into the data plane (*i.e.,* iCFs). Specifically, embedding logic into PDPs has added another layer of complexity for tracing and troubleshooting these applications. Current programmable switches do not provide a rich enough abstraction to support traditional tracing [4, 11, 14], and this lack of tracing primitives forces programmers to create their own unique solutions, creating very specific, non-reusable tracing tools to debug their iCFs. More importantly, traces produced by these particular solutions will likely not be interoperable with existing tracing diagnosis frameworks, e.g., Google's Dapper[14]. Orthogonally, existing tracing frameworks do not provide primitives for generating or capturing tracing data from iCFs embedded in programmable data planes.

In this paper, we aim to bridge the gap between traditional network telemetry and distributed tracing frameworks. To this end, we argue for the design of a new framework that tackles executions which cross from the distributed application boundary into the network by capturing trace data from PDPs and presenting them to the application plane through a well-defined, flexible abstraction. We introduce a system, P4-InTel, that (i) leverages network telemetry to instrument PDPs into monitoring arbitrary, user-defined trace data, and (ii) coordinates storage, collection and formatting of this trace data internally, delivering only well-formed context data to any application plane debug tool. We argue that our system will not only simplify tracing of PDP programs, but also, given the interoperability with emerging distributed systems frameworks, we believe our work will simplify management and facilitate the debugging process done by programmers.

The design of P4-InTel faces several unique challenges:

- First, there are **PDP hardware** constraints that prevent sophisticated and overly complex monitoring functions from running in the switch itself. This has been partially addressed by data plane telemetry research that aims to specify customized monitoring code that can be integrated into the

switch code in order to collect metrics of interest [3, 9, 12, 16]. Therein, the use of network telemetry has seen a significant interest in the last few years because of how flexible this telemetry becomes when dealing with PDPs. However, introspecting on iCFs can require specific types of instrumentation that go beyond general telemetry (e.g., establishing causality, ordering of events, consistency between switches, etc).

- Second, **application interface** issues often limit transparent instrumentation and monitoring of distributed systems. Although existing efforts such as *Baggage* contexts [11] offer means of decoupling system instrumentation from the tools used, collecting application context from iCFs is, in essence, different from doing so in traditional distributed applications because language limitations restrict the set of variable and tracing operations that can be performed on a packet.
- Finally, **interoperability** issues make it hard to seamlessly integrate high-level application context data with low-level information from network switches. Thus, means of presenting a unified execution context for iCFs are necessary, with information collected from the data plane (e.g., P4) integrated with application context to support a comprehensive tracing and debugging mechanism.

Workflow: This paper is structured as follows: In Section 2 we provide background information about iCFs, network telemetry, and server tracing; In Section 3 we present the design and prototype of P4-InTel; and, in Section 4, we show an initial evaluation of our system. We conclude with a discussion of open topics (Section 5), related works (Section 6), and final remarks (Section 7).

2 BACKGROUND

We outline below important background on programmable data planes and distributed tracing.

2.1 Programmable Data Planes and iCFs

Until recently, the separation between the application plane (in which computation occurs) and the network data plane (in which communication happens) has been strict. With emerging PDPs, however, parts of server computation can be offloaded to programmable switches (*i.e.,* in-network compute function, or iCF) thus blurring this separation. However, programmability in the data plane hardware has to abide by strict restrictions to provide line rate, *e.g.,* the P4 [2] language does not allow loops in its programs. Furthermore, typical P4 programmable hardware[1], when compared to servers, has very limited memory and does not allow computation to overuse the CPU. Instead, P4 switches implement a once-through pipeline, where the stages and, additionally, their hardware resources (ALUs, SRAM, tables, registers, etc) are shared between the procedures defined in the P4 programs.

Takeaway: Programmable data planes and their language constructs place significant limitations on the set of functionalities that can be supported and the hardware memory resource is significantly limited.

[1] https://barefootnetworks.com/tofino/

2.2 Network Telemetry

Recent works in network telemetry offer more direct (and more efficient) approaches to store and collect stateful data from a PDP, without intervention from the control plane. In P4, telemetry has been widely implemented in two paradigms: (i) using the local storage of P4 switches or (ii) using in-band network Telemetry. Essentially, the local storage approach writes all state information to register arrays (implemented physically by using PDP switch stateful ALUs) and later, collects the data through either custom protocols [3, 16] or, if using the control plane, the switch's API. The alternative is using in-band Network Telemetry (INT) [9], where packet headers carry and aggregate telemetry data as they traverse the network and, once the packets arrive at their destination, these telemetry headers are stripped and forwarded to an external server to be processed.

Both approaches have their shortcomings and their advantages. For example, INT demands more packet real estate in the form of telemetry headers (something that is not acceptable to some applications which maximize the use of the MTU). Conversely, using local storage for telemetry will impact the resources of a PDP application by using more sALUs and SRAM, which is undesirable for applications which already have a high demand for these resources.

Takeaway: Unfortunately, given the nature of P4, we cannot shield programmers from these low-level details regarding packet real estate versus sALU/SRAM usage. Either we implicitly pick one and apply it universally, or provide developers with knobs to specify preferences and explicitly tackle this trade-off.

2.3 Distributed Tracing and Debugging

Amongst common strategies for debugging distributed applications, context propagation enables tracing execution flows by propagating metadata along the flow of an execution, i.e., metadata is added unto the RPC calls and continuously updated as the RPC propagates the flow of control across different processes. Distributed tracing has been shown to reliably ascertain causality between run-time events, which is a foremost concern when introspecting and debugging distributed systems. Consequently, numerous works [4, 8, 11, 14] have been conceived which build upon context propagation, amidst which distributed tracing ranks as the most prevalent.

One such abstraction that has been proposed to trace distributed applications (outside the scope of PDPs) is the *Baggage* context [11]. *Baggages* are logs that follow the execution path of a distributed application RPC calls. A host logs its actions into the baggage and whenever the computation goes to other hosts, via RPCs, a baggage containing data on previous actions is forwarded along within an RPC. At the end of the computation, the baggage has a detailed log of what was done. A Baggage context is general enough to instrument many management tools for distributed applications and has properties that make it safe for use in distributed systems.

Takeaway: Baggage provides a generalized abstraction for tracing distributed applications but lacks primitives for reaching into the data-plane. We argue that we can extend this abstraction to create a solution that coordinates both application and data planes, collecting arbitrary, user-defined trace data on the latter and exposing this in a flexible manner to the former using baggage contexts.

Table 1: Annotation Primitives

Annotation	Description
`@Store(VAR)`	Stores VAR locally at the switch
`@Append(VAR)`	Appends VAR to the Baggage Tag

3 P4-INTEL

We begin by discussing the workflow for P4-InTel (Section 3.1), then discussing the design requirements for P4-InTel (Section 3.2), and conclude with a straw-man approach for P4-InTel (Section 3.3).

3.1 P4-InTel Workflow

In Figure 1, we present a high-level workflow for P4-InTel. We envision that, offline, application developers (step 1) will augment and annotate iCF programs (written in P4) with the set of user-defined variables to export and deploy these annotated iCFs into the network (step 2) . We note that these variables comprise the set of data that can be captured and exported into a baggage context.

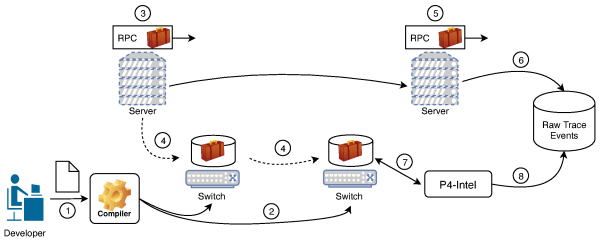

Figure 1: P4-InTel Diagnosis Workflow. Red suitcases represent the *baggage* contexts, i.e., the RPC tracing logs. The servers store the context within RPCs and propagate context in-band within the RPC messages. The switches store the contexts locally and propagate out-of-band, in separate messages, to the external store.

At run-time, the tracing framework will include baggage tags (or headers) into RPC packets (similar to what it does today) (step 3) and propagate to other servers (step 5) via RPCs and finally store them in an external data store (step 6) once RPC-tracing is completed. Additionally, as these RPC traces propagate throughout the network, our data plane framework will capture the appropriate data and store them locally on the switch (step 4) or if required append them to packets. Periodically, P4-InTel will interact with the data plane to export this data to a centralized entity (steps 7-8) which will combine the data captured in the data-plane with the data collected in the RPCs.

3.2 System Design

Challenge #1: Capturing Tracing Data Our approach is distinctly different from a traditional RPC-tracing workflow where all tracing information is captured in-band within the RPC. Due to PDP limitations, we are unable to extract arbitrary P4-program data and embed them into the RPC packets. In particular, while the language enables us to introspect on annotated program state, the hardware limits the size of the information that can be embedded into a packet (as discussed in Section 2).

Solution: Out-of-Band Tracing. There are two clear approaches to addressing this problem. First, to limit the set of annotated data which a developer can introspect on. Second, to export data from the program in an out-of-band manner. In this work, we choose the latter approach because limiting the developer handicaps diagnosis and management. However, this approach introduces additional complexity because the tracing data is fundamentally fragmented with some data stored locally in switch state and others captured in RPC stored in an external data store. To tackle this problem, we design techniques to coordinate orchestration of the data collected in-band with the data collected out-of-band.

Additionally, to support critical contexts we also allow developers to leverage in-band tracing for a limited set of variables; however, we note that this provides very limited contexts.

Challenge #2: Compiling P4-InTel Annotations Traditionally, annotations for tracing are either automatically generated by using language constructs (e.g., Java's AspectJ) or are manually hard-coded into the RPC framework (e.g., Google's Dapper). With iCFs, there is no common RPC language. In fact, iCFs are fundamentally different and given the diversity in functionality, we do not anticipate a common layer for information exchange. However, many emerging verification and diagnosis systems for PDPs leverage manual annotations. In this work, we piggyback on these efforts and argue that in addition to creating verification-oriented annotations, the developers should add annotations for tracing and general diagnosis. Given these annotations, the key challenge is to develop a compiler framework that effectively captures the annotated variables, creates baggage logs and appropriately exports them to the external data store.

Solution: Domain-Specific Compiler and Language Annotations. Our compiler is guided by our language annotations. Using these annotations, the programmer can indicate if the data should be stored and exposed using INT based telemetry (`@Append` annotation) or Local Storage based telemetry (`@Store` annotation). These are shown in Table 1. Using this information our compiler can appropriately create P4 code for using local storage or leveraging network telemetry to augment RPC packets.

(1) Local Storage based Telemetry: Recall that every iCF invocation stems from a packet inserted into the network, and, if that packet is tagged, at every hop the PDP switches log trace data about the packet and its actions. To collect this data, the framework sends a collection packet (*CP*) to retrace the steps of the iCF-invoking packet, collecting the data it left along the way. To enable the framework's collection packets to accurately retrace the steps of iCF invoking packets, the latter will always log, along with trace data, their egress port. As such, the forwarding of *CPs* is done primarily by sending it out the same port the original packet went. However, because the original iCF invoking packet might not have returned to its source (*e.g.,* it was intended to be dropped after invoking its iCF or forwarded to another host), we also devised a fail-safe mechanism that, whenever the outbound port of a packet leads to a host or the packet being dropped, falls back to IP forwarding, being returned to the sender through IP routes.

(2) INT based Telemetry: Contrary to Local Storage, INT uses network packets to carry telemetry data. When we annotate a variable with an `@Append`, our PDP instrumentation will append that variable to the current packet's baggage tag. In case the packet does not return to the sender (along with the collected data on the baggage tag), a fail-safe similar to the previous one also applies. Should the packet's next hop lead to a non-PDP switch, the baggage tag, along with the collected data, is detached from the rest of the packet and forwarded back to the original sender.

Challenge #3: Composing Tracing Data. Finally, during the diagnosis phase, data captured in RPCs and stored within the data plane must be composed together to provide a holistic end-to-end picture of the distributed application's execution across both the server and the network. When a monitored RPC trace completes, the tracing framework interacts with the network devices to collect locally stored baggage information and compose it with the RPC-stored information. The two challenges associated with composing this information are: (1) ensuring that tracing data is interleaved in the correct order, and (2) recreating the rich and complex *Baggage Definition Language* (BDL) data types from the data collected in the data-plane.

Solution: We argue that the switches and the general P4-InTel framework index the stored data by `PacketID`. Note: the `PacketID` will be stored in the RPCs as a part of a context. Thus, whenever an application makes a call to retrieve baggage context, our system will automatically merge the baggage context external in the servers with the context resulting from the iCFs based on the `PacketID`. This ensures that applications that try to resolve contexts will always see the complete baggage across both the data plane and the application plane. We note that by using a unique `PacketID`, the RPC is able to execute across multiple processes and iCFs without loss of general context.

3.3 Strawman Prototype

In Figure 2, we present the architecture for P4-InTel. P4-InTel addresses the three challenges discussed in Section 3.2 using the following components. The *P4-InTel-compiler* is an extension of the traditional P4-compiler and uses the annotations to determine local storage and memory allocations. The *Baggage Context Handler* composes and orchestrates data across the application and data planes. And the *Telemetry Interface* exposes the switch-residing traces for collection.

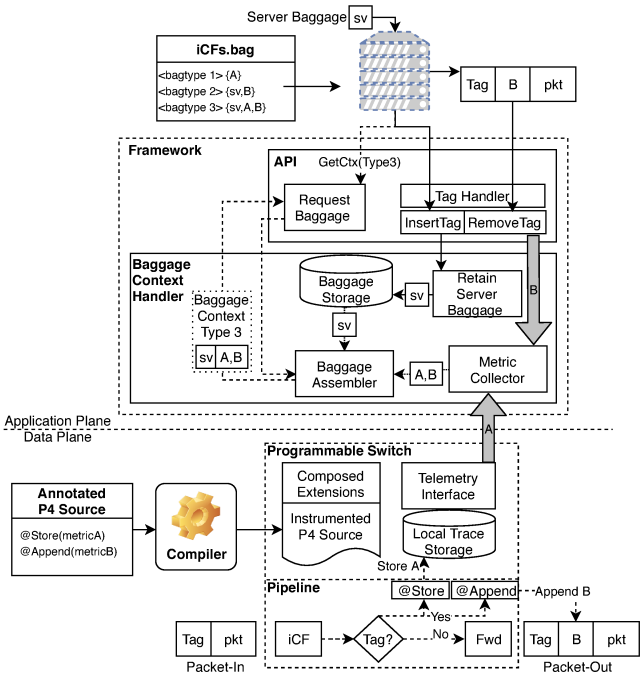

Figure 2: P4-InTel Architecture.

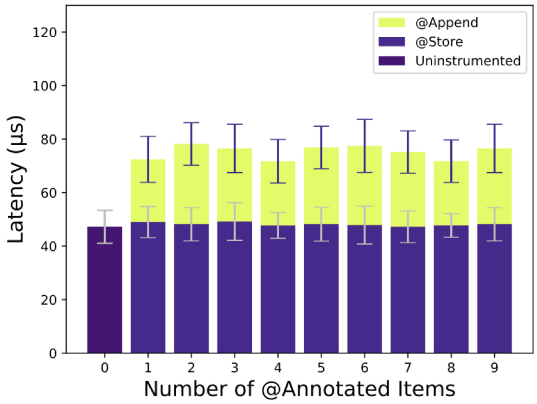

Figure 3: BMv2 Processing Latency

4 EXPERIMENTAL EVALUATION

In this section, we present early experimental evaluation of our proof-of-concept prototype. Because we had no access to a P4 hardware switch, experiments were performed using the BMv2 software switch on an Ubuntu 16.04 virtual machine with 2 cores @ 3.2Ghz and 2GB RAM. We should point out that the following evaluation is somewhat artificial because of the lack of programmable hardware. Our intention is to demonstrate the feasibility of P4-InTel and present a preliminary evaluation in a simulation environment.

In Figure 3, we present the processing latency of collecting an increasing number of 4-byte metrics through both `@Store` and `@Append`. Firstly, we can see the PDP instrumentation has negligible impact on processing latency (as the uninstrumented version of the switch is on par with the tests where the system was instrumented and `@Store` was used). When `@Append` was used, we observed a fixed increase in latency likely due to the procedure of creating the new header.

As can be seen, our annotations have little impact in processing latency. However, for the specific case of INT-based `@Append` annotations, there might be a networking overhead associated with every added metric. As INT headers stack, they leave less space for payloads, decreasing the throughput of applications. Figure 4 shows a scenario where 9 switches are placed in the path of a UDP

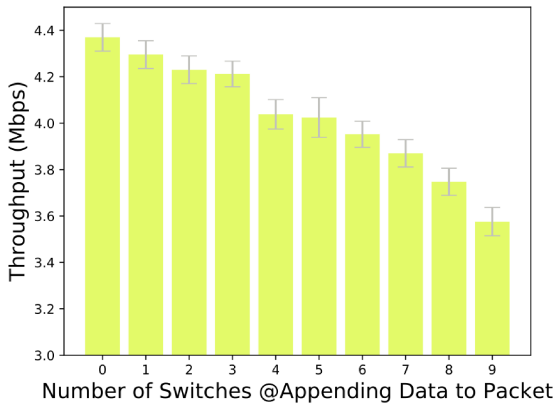

Figure 4: @Append Overhead on Throughput

flow. Switches from 1 to 9 are then incrementally reconfigured to start appending 24 bytes of metrics each to the baggage tag of UDP packets with 240 bytes of payload. Each hop increases the percentage of telemetry bytes when compared to the payload, decreasing efficiency and, consequently, throughput of the flow. It is important to note that the most critical element in this experiment is the BMv2 software switch, with the actual throughput of hardware switches figuring on the Gigabits per second range, and not Megabits. We only alert the reader to the apparent degradation that occurs when using `@Append` to collect too many metrics over long distances. Naturally, the `@Store` primitive is a better solution for these cases.

Memory-wise, the consumed SRAM in switches will be the product of all @Stored variables and a predefined number of slots (i.e., maximum number of packet traces stored at a given time). Appended variables have no impact on SRAM. For the network overhead of telemetry, both Appended and Stored variables cause an impact directly proportional to their respective sizes. Their overall network overhead has a complexity of $O(Hops^2)$, where *Hops* is the number of switches visited by either collection packets (for Stored variables) or tagged packets (for Appended variables). Collection overhead of data happens at a different time than the storing of this data for Stored variables, and concurrently, for Appended variables. No additional tables were introduced by our prototype.

5 DISCUSSION

In this section, we provide a brief discussion of key open issues.

Tracing Non-Trivial Metrics. Some debugging applications may require data that is difficult to calculate in PDPs. While acquiring this data is beyond the scope of our work, we advocate that it should be possible to modularly include code from external solutions that acquire this non-trivial data. We suggest that a compositional mechanism (such as [18]) can be used for non-trivial metrics. An example would be to calculate the entropy of specific flow features, a metric required by some anomaly detectors [10, 13]. Calculating entropy is a difficult undertaking for most PDP programmers. To facilitate the task, a programmer could acquire a specialized P4 module for the calculation of entropy and compose it into his base application, annotating the resulting entropy value.

Resource Allocation. In our current framework, P4-InTel provides no resource management. However, as discussed earlier, switch memory is a crucial resource and, if left unmanaged, developers can create annotations and diagnosis requests that overwhelm switch resources. As part of future work, we plan to explore specific resource allocation and isolation strategies to minimize the impact of an iCF's diagnosis and management efforts on other iCFs on the same device.

6 RELATED WORK

Our work combines network telemetry with diagnosis and debugging of P4 programs. The main research efforts related to these aspects are discussed below.

Network Telemetry FlowStalker [3] is a monitoring system for network telemetry implemented in P4 that runs directly on the data plane. It first captures and stores specific meta-data about flows of interest, which is subsequently collected by the control plane. *Flow [16] efficiently collects features from packets and stores them flexibly on a P4 switch using a dynamic allocation system. Telemetry is used to evict records to the control plane. TurboFlow [15] is a system much like NetFlow [5], except that it is optimized for programmable data planes. TurboFlow is able to generate flow records with rich metadata without sampling.

Our work also relies on telemetry, but our focus is on iCFs, which implies that many of the concerns from the works above, such as the necessity to aggregate high volumes of data directly on the switch, are more relaxed. This allows us to have a very simple yet effective telemetry system on switches. Further, while the works above also have a fixed impact on hardware resources and network resources, our annotations leave this trade-off for the programmer to dose as he sees fit, i.e., allowing more leeway to his iCFs.

P4 Diagnosis Assert-P4 [6] uses annotations directly in P4 source code to annotate invariant conditions as runtime assertions. It then translates P4 code to C and symbolically executes this code. Bugs are discovered if at any point the code reaches an assertion and the invariant is violated. Vera [17] also translates P4 code and performs a symbolic execution of the program. However, Vera is able to identify a range of bugs automatically without assertions. Vera also translates P4 code to an optimized language made to simplify the symbolic execution process, being able to run much faster than Assert-P4.

While most works aimed at debugging P4 programs attempt to debug and guarantee properties about switch code alone, our work encompasses a more comprehensive view of the network, tracing executions from and to servers and their distributed applications.

7 CONCLUSION

In this position paper, we outline a promising approach for iCF telemetry and debugging, primarily aimed at harvesting iCF execution contexts and merging this information with preexisting RPC-tracing contexts. This work presents the first step in a rich line of research on creating holistic end-to-end tracing for in-network compute enhanced distributed applications.

ACKNOWLEDGMENTS

Theophilus A. Benson would like to thank the NSF for award CNS-1749785. Alberto Schaeffer-Filho would like to thank CNPq for research grants 407899/2016-2 and 312091/2018-4. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, and also by NSF CNS-1740911 and RNP/CTIC (P4Sec) grants.

REFERENCES

[1] T. A. Benson. In-network compute: Considered armed and dangerous. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, pages 216–224, New York, NY, USA, 2019. ACM.

[2] 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. *SIGCOMM Comput. Commun. Rev.*

[3] L. Castanheira, R. Parizotto, and A. E. Schaeffer-Filho. Flowstalker: Comprehensive traffic flow monitoring on the data plane using p4. In *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, pages 1–6, May 2019.

[4] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 217–231, Broomfield, CO, Oct. 2014. USENIX Association.

[5] Cisco. Introduction to Cisco IOS NetFlow - a technical overview. https://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.html.

[6] L. Freire, M. Neves, L. Leal, K. Levchenko, A. Schaeffer-Filho, and M. Barcellos. Uncovering bugs in p4 programs with assertion-based verification. In *Proceedings of the Symposium on SDN Research*, SOSR '18, pages 4:1–4:7, New York, NY, USA, 2018. ACM.

[7] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 121–136, New York, NY, USA, 2017. ACM.

[8] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi, V. Venkataraman, K. Veeraraghavan, and Y. J. Song. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 34–50, New York, NY, USA, 2017. ACM.

[9] C. Kim, A. Sivaraman, N. P. Katta, A. Bas, A. Dixit, and L. J. Wobker. In-band network telemetry via programmable dataplanes. 2015.

[10] A. C. Lapolli, J. Adilson Marques, and L. P. Gaspary. Offloading real-time ddos attack detection to programmable data planes. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 19–27, April 2019.

[11] J. Mace and R. Fonseca. Universal context propagation for distributed system instrumentation. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 8:1–8:18, New York, NY, USA, 2018. ACM.

[12] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 85–98, New York, NY, USA, 2017. ACM.

[13] A. Santos da Silva, J. A. Wickboldt, L. Z. Granville, and A. Schaeffer-Filho. Atlantic: A framework for anomaly traffic detection, classification, and mitigation in sdn. In *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, pages 27–35, April 2016.

[14] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[15] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith. Turboflow: Information rich flow record generation on commodity switches. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 11:1–11:16, New York, NY, USA, 2018. ACM.

[16] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 823–835, Boston, MA, July 2018. USENIX Association.

[17] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu. Debugging p4 programs with vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 518–532, New York, NY, USA, 2018. ACM.

[18] P. Zheng, T. Benson, and C. Hu. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, pages 98–111, New York, NY, USA, 2018. ACM.