Are we ready to drive Software Defined Networks? A Comprehensive Survey on Management Tools and Techniques

ELISA ROJAS, Telcaria Ideas S.L.
ROBERTO DORIGUZZI-CORIN, FBK CREATE-NET
SERGIO TAMUREJO, Institute IMDEA Networks
ANDRES BEATO, Telcaria Ideas S.L.
ARNE SCHWABE, University of Paderborn
KEVIN PHEMIUS, Thales Communications & Security
CARMEN GUERRERO, University Carlos III of Madrid

In the context of the emergent SDN paradigm, the attention is mostly directed to the evolution of control protocols and networking functionalities. To this end, network professionals need the right tools to reach the same level –and beyond– of monitoring and control they have in traditional networks. Current SDN tools are developed on an ad hoc basis, for specific SDN frameworks, while production environments demand standard platforms and easy integration. This survey aims to foster the definition of the next generation SDN management framework by providing the readers a thorough overview of existing SDN tools and main research directions.

CCS Concepts: •**Networks** → **Programmable networks; Network management;** *Network monitoring;* Network design principles;

Additional Key Words and Phrases: Software-Defined Networking, Network Maintenance, Debugging, Resource Management, Simulation, Profiling, Monitoring

ACM Reference Format:
Elisa Rojas, Roberto Doriguzzi-Corin, Sergio Tamurejo, Andres Beato, Arne Schwabe, Kevin Phemius and Carmen Guerrero, 201X. Are we ready to drive Software Defined Networks? A Comprehensive Survey on Management Tools and Techniques. *ACM Comput. Surv.* 0, 0, Article 0 (0), 33 pages.
DOI: 0000001.0000001

1. INTRODUCTION

Software-Defined Networking (SDN) has emerged strongly in the last decade, especially since the publication of the first OpenFlow (OF) [McKeown et al. 2008] protocol specifications. The key notion behind SDN is to introduce a separation between the control plane and the data plane of a communication network. The control plane is implemented via a logically centralized component called "the controller".

However, similarly to the heterogeneity reminiscent of the early Internet [Huang and Griffioen 2013], the current SDN ecosystem is extremely fragmented due to the multitude of different controller platforms. Therefore, although SDN introduces new possibilities for network management and configuration [Kim and Feamster 2013] and it solves classical network management problems, it also creates new challenges [Wickboldt et al. 2015]. For example, the management plane is a large and underexplored area, particularly in high-availability designs [Govindan et al. 2016]. That is why

This work is partially supported by: EC FP7 NetIDE, EC H2020 SUPERFLUIDITY and Spanish DRONEXT.
Author's addresses: E. Rojas and A. Beato, Research Department, Telcaria Ideas S.L.; R. Doriguzzi-Corin, Future Networks Department, FBK CREATE-NET; Sergio Tamurejo, Institute IMDEA Networks; Arne Schwabe, Fakultat fur Elektrotechnik, Informatik und Mathematik, University of Paderborn; Kevin Phemius, Thales Communications & Security; Carmen Guerrero, Departamento de Ingenieria Telematica, University Carlos III of Madrid.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 0 ACM. 0360-0300/0/-ART0 $15.00
DOI: 0000001.0000001

addressing SDN management issues is imperative in order to avoid patching SDN later [Wickboldt et al. 2015].

To this purpose, we provide a comprehensive overview and analysis of the SDN tools that are currently available as a concrete part of SDN management and control. The survey covers almost a hundred of tools from different types. Most of them have been designed in the last five years, but we aim to cover the topic from the beginning of the SDN paradigm until the present day.

1.1. SDN tools: An overview

Any piece of software that facilitates the development, deployment and/or maintenance of SDN architecture and, more specifically, of network applications can be classified under the *SDN tool* category. They complete the puzzle of an ideal SDN management framework. When a tool is used at development time, we consider it *offline*, whereas when applied at deployment or run-time, we call it *online*. Offline tools are mainly verifiers or model checkers, but this category may also include simulators; while online tools encompass: loggers, debuggers, profilers, memory managers and emulators. Additionally, there are also parts of the SDN architecture that might be considered tools, such as the communication protocols and channels, or the intrinsic mechanisms of the SDN controllers that allow different SDN applications (or even frameworks) to work together without conflicting with each other. More specifically, we envision the following classes of SDN tools:

— **Composition:** Composition of SDN applications and services let the network support upgrades and expansions, including newer functionalities or coordinating several software modules.
— **Debugging:** Troubleshooting, verification and model checking aim for the same objective: *the network behaves as expected*. Their differences are related to the part of the network that needs to be analyzed (data or control plane) and at which time (development, deployment or run-time).
— **Resource management:** To guarantee an optimal utilization of network resources.
— **Profiling:** To prove that resource management is effective, we need profilers or monitors to measure the network activity and resource consumption.
— **Simulation:** To repeat different scenarios without affecting the production network, simulators and emulators represent an essential tool.

1.2. Related work

Although detailed surveys about SDN exist [Hu et al. 2014b; Jarraya et al. 2014; Nunes et al. 2014; Xia et al. 2015; Hakiri et al. 2014], they basically describe the big picture of SDN, traversing the architecture layers up to down [Singh and Jha 2017], focusing on security [Dargahi et al. 2017; Rawat and Reddy 2017; Khan et al. 2016], energy efficiency [Tuysuz et al. 2017], scalability [Karakus and Durresi 2017; Huang et al. 2017; Oktian et al. 2017], traffic engineering [Mendiola et al. 2017], Northbound Interface (NBI) and Southbound Interface (SBI) interfaces [Masoudi and Ghaffari 2016], wide-area networks [Michel and Keller 2017], transport networks [Alvizu et al. 2017] or compatibility issues, but leaving the Management-Control entity aside.

Some surveys mention the concept of debugging tools [Kreutz et al. 2015; Nde and Khondoker 2016], or focus on specific tools like topology discovery [Khan et al. 2017], but providing just a few examples and without delving into much detail.

In contrast with them, this survey focuses on the state of the art of SDN tools in a generalized manner, as part of the Management-Control entity.

1.3. Contributions and structure of this survey

This paper provides a comprehensive survey of the tools proposed in the literature for the management and control of SDN. We classify and compare them, following different criteria. Finally, we discuss and reach a conclusion about the current status of management tools in SDN.

We start with Section 2, where we describe the SDN architecture as defined by the Open Networking Foundation (ONF), and give an overview on different approaches for interfacing SDN tools and SDN platforms. Section 3 is devoted to composition, a service envisioned to allow mul-

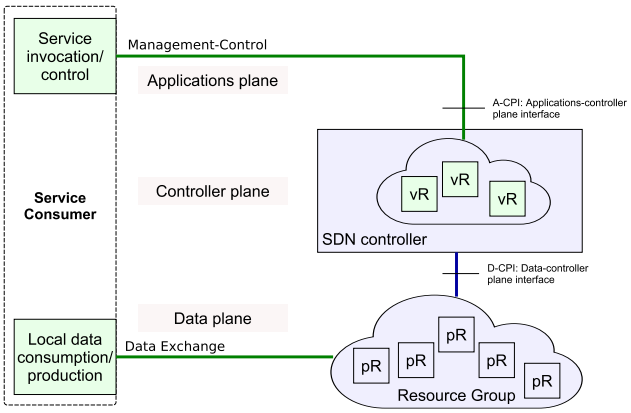

Fig. 1. SDN architecture overview [ONF 2016]

tiple SDN applications or frameworks to cooperate on controlling the same network infrastructure. Section 4 addresses debugging tools, which encompass any diagnostic tool that aims to find network malfunctions. Section 5 describes different approaches for the management of the memory of the SDN-enabled devices, to optimize the usage of such a scarce resource. Section 6 introduces network profilers, proposed to monitor the network resources. Section 7 compares different simulators and emulators that can support the development and testing of SDN applications. Afterward, Section 8 discusses the analysis of all the previous sections and yields future research directions. Finally, Section 9 concludes the paper.

2. TOOLS IN THE SDN ARCHITECTURE

Currently, the ONF definition provided in [ONF 2016] is the reference for most controller frameworks. The architecture (Fig. 1) defines three planes –Data, Controller, and Application–, plus a Service Consumer module which encompasses the Management and Control functions.

The physical Resources (pRs) are confined in the **Data plane**. Communication with the upper plane is performed through the data-controller plane interface (D-CPI), also known as the SBI, which exposes the capabilities of the pRs. *SDN controllers* are located in the **Controller plane** and have a twofold objective: to control the Data plane resources and to orchestrate the requests of the Application plane. Accordingly, the SDN controller virtualizes and orchestrates the virtual Resources (vRs) onto its own underlying pRs. *Services* (such as topology monitoring, statistics or API abstractions to control the Data plane) are offered as a vR from the Controller to the Application plane via the application-controller plane interface (A-CPI), often called the NBI. *SDN applications* are software modules that reside in the **Application plane**, and communicate their desired network behavior to the Controller Plane via the A-CPI.

Finally, **Management and Control** are viewed as a continuum, i.e. as the same entity, which can operate over all the three aforementioned planes. Its minimum functionality is to allocate resources from a resource pool in the Data plane to a particular client in the Controller plane, and to establish reachability information that permits the Data and Controller plane entities to mutually communicate. Management-Control functions are provided by the Operations Support Systems (OSS); which includes features such as: infrastructure maintenance (fault analysis, diagnostics, alarm correlation and management), logging, configuration and service persistence, traffic analysis or initialization parameters[1]. Some of these functions are in the scope of SDN and implemented by means of what we call as SDN tools, presented in the following sections of this paper.

[1]In contrast with this schema, some approaches (e.g. NEOD [Song et al. 2013] and PDEE [Kuklinski 2014]) embed part of the management functions in the Data Plane, through firmware extensions for the network devices.

The Management-Control entity interacts with other planes through the use of **interfaces**. As pointed out in [Jarschel et al. 2014], having open interfaces is crucial in the adoption of SDN. In this sense, many specifications have been proposed for communication between the Management-Control entity and the Data plane, such as **NETCONF** [R. Enns et al. 2011], **SNMP** [J.D. Case et al. 1990] or **OF-Config** [ONF 2014]. However, in case of the Controller plane, standard interfaces have not yet been defined to cope with the requirements of maintenance and management operations. As a consequence, some of the available SDN tools embed their functions inside the Controller plane. In other cases, the Controller plane is enhanced with additional interfaces either defined from scratch, or built on top of existing mechanisms, such as **REpresentational State Transfer (REST)** [Fielding and Taylor 2000; Fielding 2000], **Advanced Message Queuing Protocol (AMQP)** [Vinoski 2006] (including RabbitMQ [RabbitMQ 2007] and ZeroMQ [ZeroMQ 2007]).

Finally, the NBI provides access to network resources from the Application plane and its implementation depends on the SDN platform. As this interface can be also used for management purposes, we provide a quick overview of how to implement different SDN tools leveraging the NBIs of the two most promising open source SDN frameworks at the time, i.e. Open Network Operating System (ONOS) and OpenDaylight (ODL).

ONOS [Berde et al. 2014]: Monitoring and management of ONOS clusters can be performed either via Java interface [ON.Lab 2016c] or REST API [ON.Lab 2016d]. Both interfaces allow to load and unload application bundles, install and uninstall flow rules, obtain the topology, manage devices, etc. Higher level management interfaces include a web GUI and the ONOS CLI [ON.Lab 2016a], an extension of Karaf's CLI built on top of the ONOS Java interface.

Like the CLI, other tools such as debuggers, resource managers, loggers can make use of ONOS interfaces to implement their functions. Especially the Java API allows the implementation of modules for different layers of the ONOS architecture. For instance, debuggers may need to inspect the NBI. In this case the Java API provides interfaces such as *FlowRuleService*, *PacketService* to get all messages generated or received by the running applications. Similarly, resource managers can access traffic or other statistics through the *FlowRuleService* or the *DeviceService*, moreover they can also extend the distributed storage of ONOS to record such statistics with the *StorageService*. However, in some cases even such rich APIs are not sufficient. For instance, a logger willing to inspect the SBIs (e.g. getting OF or NETCONF messages) would need direct access of the *Providers* components, but they do not implement the necessary interfaces. Another example is the composition mechanism currently investigated and developed by authors of one of the ONOS feature proposals [ON.Lab 2016b]. In this case, they make use of existing ONOS interfaces, such as the *FlowRuleService*, to receive flow rules from applications. However, they also need to extend the *FlowRuleManager* component in order to add missing interfaces or to re-program the flow tables.

ODL [Medved et al. 2014]: Monitoring and management in ODL can be performed via RESTful API or via Java APIs generated from Yang [Bjorklund 2010] models. Such APIs are exposed by the Service Adaptation Layer (SAL) to allow developers to implement network applications and plugins (consumers) and connects the consumers to appropriate modules providing services (data providers). Some of the most relevant APIs (in the context of network management) that the ODL core projects expose are: *Topology* to access the network graph containing edges, nodes and their properties, *Flow Programmer* to configure flow entries on the network elements, *Statistics* to retrieve statistics of flow entries and network elements, and *Switch Manager* which exposes the elements of the underlying network, listing their ports and properties.

Accordingly, ODL imposes no limitation to implement interfaces with various SDN tools as far as the model is designed. However, this freedom causes the creation of different custom APIs in the end. Consensus in this regard would help building some APIs as the foundations that could be reused afterward by any SDN tool for ODL.

In the next chapters, we analyze state-of-the-art SDN tools paying special attention to the interfaces they implement. The focus of this survey is to obtain a conclusion regarding the maturity of the Management-Control entity based on that analysis.

3. COMPOSITION OF NETWORK APPLICATIONS

One of the emerging problems in SDN is the heterogeneity regarding network applications, which include SDN applications and services. The idea behind the composition of network applications is to run multiple SDN applications in parallel on the same network, independently of their origin. Composition also involves conflict detection and resolution. The result is a global *network policy*.

Traditionally, network policy management is done manually (network administrators translate high level network policies into low level network configuration commands) and policy changes take a long time to plan and implement. Therefore, problems are typically detected only at run-time when users unexpectedly lose connectivity, security holes are exploited, or applications experience performance degradation [Prakash et al. 2015].

Thus, the objective of composition of network applications is twofold: (i) to allow the coexistence and cooperation among very heterogeneous control programs and (ii) to plan ahead possible conflicts and errors so that they can be detected and solved automatically. To achieve this final objective we envision the three following steps:

(1) **Network partitioning and slicing:** Network administrators should be capable of assigning different slices of the network to the different applications they want to deploy in the network. To achieve this goal, many SDN network hypervisors have already been implemented [Blenk et al. 2015] and we describe the ones related to composition in the following paragraphs.
(2) **Prioritization/Ordering among network applications:** A second step is the actual assembling of the outputs from applications that need to be deployed in the same slice of the network. This requires the definition of criteria and languages, e.g. to define which application has a higher priority. Many current SDN controller frameworks, such as Floodlight [Floodlight 2016a], ODL [Medved et al. 2014] and ONOS [Berde et al. 2014], already provide the capability of defining static priorities for SDN apps to be deployed; however some issues like dynamically changing the priorities, creating more complex behavior (not based only on those priorities) or allowing compatibility of different SDN applications from different frameworks still remain unresolved.
(3) **Conflict detection and resolution:** The third and final step involves detecting and resolving conflicts at run-time. We understand as conflicts the uncertainty of having to merge incompatible policies that resulted from the execution of different SDN applications for the same input, e.g. a drop and a flood action. Conflict resolution can be understood as an enhancement of assembling SDN applications that provides different mechanisms or alternatives to be applied when merging, instead of a single default one.

3.1. Network Partitioning and Slicing approaches

In the SDN domain, **FlowVisor** [Sherwood et al. 2010] can be considered as the first approach to allow multiple network controllers to run side-by-side on top of the same network infrastructure. Instead of allowing all controllers to share the same traffic, FlowVisor partitions the network into smaller slices and gives each controller only the view of its own slice of the network. To achieve this goal, it sits as a centralized module between the network and the SDN controllers. FlowVisor was the first network virtualization hypervisor for SDN, introducing the concept of slicing the networks; many other hypervisors are built based on it and are documented in a comprehensive survey on SDN hypervisors [Blenk et al. 2015]. A followup to FlowVisor is **OpenVirteX** [Al-Shabibi et al. 2014], which introduces the concept of virtual topologies.These two approaches do not cover the scenario where network controllers cooperate to control the same the traffic, therefore they do not implement any assembling or conflict resolution mechanisms.

3.2. Composition approaches

The most relevant works on composition of network applications are based on two basic operators. One is the *parallel* operator, where *new flow* events (e.g. `PACKET_IN` for OF) are relayed to all applications in parallel by the composition component and the resulting actions are then assembled

and applied to the network. The second is the *sequential* operator, where events are sent to applications one after another in a previously defined chain. The resulting actions from one application are merged with the event and then sent to the next controller. Almost all approaches for composition allow arbitrary combinations of these two basic operators. The parallel operator for composition of SDN applications was originally introduced by **Frenetic** [Foster et al. 2010], a high-level language for OF networks. Similar to Frenetic, **NetKAT** [Anderson et al. 2014] is a network programming language based on the so-called Kleene algebra that defines union and sequential operators plus the Kleene star operator to iterate applications. Grounded on Frenetic, **Pyretic** [Monsanto et al. 2013] is a domain-specific language embedded in Python that aims at enabling network programmers to develop SDN applications by leveraging on high-level abstractions. Pyretic enhances Frenetic by introducing (i) the sequential composition, that allows one application's module to operate on the packets already processed by another module and (ii) the concept of topology abstraction, that allows the programmers to limit each module's sphere of influence. Pyretic applications can be executed on top of a modified version of POX [GitHub 2011]. As the Pyretic's interpreter communicates with POX through a socket-based API, it can potentially run on top of any controller platform.

Redactor [Wang et al. 2016a] bases its architecture on the declarative programming language Prolog, used to write the SDN modules. Redactor uses a heuristic approach to resolve conflicts. These modules can be integrated with the Prolog engine in existing controllers afterward.

Based on OpenVirtex, **CoVisor** [Jin et al. 2015] acts as a hypervisor and is placed between the network and multiple controllers. CoVisor speaks OF on both SBI (with the network) and NBI (with the guest controllers). The main goal of CoVisor is to allow applications written for different controller platforms and in different programming languages to cooperate on controlling the same network traffic. In order to achieve this goal, CoVisor defines operators to combine policies of applications running on multiple controllers to produce a single flow table for each physical switch. Moreover, CoVisor exposes a virtual view of the topology to each controller and to the applications running on top of it. These topologies can be very simple, like a one switch topology for a firewall, can provide a "big virtual switch" abstraction, or can mirror the real network for routing applications. In summary, CoVisor assembles the policies of individual applications, written for a virtual network, into a composed policy for the virtual network. Then, it compiles the "virtual" policies into a single one for the physical network. Even though CoVisor came out when OF 1.3 was well established, it only supports OF version 1.0.

NetIDE [Schwabe et al. 2016] provides a run-time Network Engine that allows the composition of multiple network applications from different controllers. The semantics are similar to the ones defined in CoVisor, but NetIDE differs from it in the following aspects: (i) the connection to the network is performed via an SDN platform (e.g. ODL or ONOS) instead of leveraging OpenVirtex, (ii) it supportsOF 1.0 and 1.3, and potentially other protocols such as NETCONF, and (iii) apart from assembling applications, it handles and resolves possible conflicts between them.

Another SDN hypervisor is **FlowBricks** [Dixit et al. 2014]. It is a framework that integrates heterogeneous controllers using only the standardized controller to switch communication protocol. While CoVisor only works with OF 1.0, FlowBricks is designed to support up to OF 1.4 and currently supports all OF 1.1 datapath features, so FlowBricks can work with multiple flow tables, for instance. Similarly to CoVisor, a policy definition configured on FlowBricks specifies how services from controllers are applied to traffic on the datapath. FlowBricks runs on an emulated environment with heavy hacks on the OF switches and cannot be used over standard network hardware.

Corybantic [Mogul et al. 2013] supports composition of network applications by resolving conflicts over specific OF rules. Corybantic acts as a *module* orchestrator, where modules are applications that implement particular network functions and their impact to the network is evaluated in terms of cost and benefits. The *Corybantic Coordinator* implements an iterative approach to evaluate the admission on execution of a particular module. Each round of the iteration is divided in four phases: (i) modules propose changes in the network, (ii) each module evaluates its own proposals in term of cost and benefits, (iii) the Coordinator picks the best proposal and (iv) the modules install the chosen proposal onto the network. Its main disadvantage is that it does not allow the use of different

Table I. Comparison table of the different composition approaches

Properties / Approaches	Description	Type (Levels of composition)			Applications are modified	Interface
		Slicing	Ordering	Conflict		
FlowVisor	Composition defined by users	✓			No	OpenFlow 1.0
OpenVirteX	Composition defined by users	✓			No	OpenFlow 1.0
CoVisor	Composition defined by users		✓		No	OpenFlow 1.0
NetIDE	Composition defined by users		✓	✓	No	OpenFlow 1.0+, NETCONF, etc.
FlowBricks	Composition defined by users		✓		Yes	OpenFlow (modified)
NetKAT	Union of all statements	✓*	✓	†	-	Programming language
Frenetic	Union of all statements		✓	†	-	Programming language
Pyretic	Union of all statements		✓	†	-	Programming language, Custom API
Redactor	Heuristic composition		✓	✓	-	Programming language, Custom API
Statesman	Automatic, with invariant checks		✓	✓	-	Custom API
Corybantic	Modules scores proposals		✓	✓	-	Custom API
Athens	Modules scores proposals		✓	✓	-	Custom API
PGA	Automatic	✓	✓	✓	-	Custom policy language

−: Means *not applicable*.
∗: NetKAT supports slices in its programming language, but not multiple apps running at the same time.
†: The semantic of these programming languages specifies how the operators that assemble policies are resolved, so a conflict in the sense of the other approaches is not possible. A compiler can issue warnings when one policy is ignored in a union.

languages and controller platforms, as CoVisor does, and it requires the specific implementation of the modules to be coordinated.

Like Corybantic, **Statesman** [Sun et al. 2014] composes network applications by resolving conflicts. Statesman defines three views of the network: *observed state*, *proposed state* and *target state*. To prevent conflicts, applications cannot change the state of the network directly. Instead, each application suggests a state to Stateman, in charge of merging (or rejecting) the individual proposals from applications.

The goal of **Athens** [AuYoung et al. 2014] is to ease the coordination and the automatic management of resource conflicts between SDN and cloud controller applications. It proposes a revision of the Corybantic design, but it is essentially a compromise between Corybantic and Statesman, presented above. Basically, Athens sends the current state of the network to each application module. As a reply, all modules synchronously send to the Athens coordinator a set of proposed changes. After that, Athens asks each module to evaluate all proposals (by using the same evaluation method proposed by Corybantic). Based on the evaluation feedback, Athens runs its conflict resolution algorithm to elect the winning proposal, which is eventually implemented onto the network.

Policy Graph Abstraction (PGA) [Prakash et al. 2015] leverages graph–based and "one big switch" abstraction to detect and resolve policy conflicts. The Graph Composer entity generates a conflict-free graph from the input policies, previously provided as graphs. This entity also prompts warnings and errors, suggesting possible fixes. An initial prototype of PGA leverages VeriFlow [Khurshid et al. 2012] to verify whether the policies in the composed graph are correctly realized on the network.

3.3. Summary of the different composition approaches

Table I summarizes and compares the composition approaches, based on:

- **Description:** How the composition is specified. This ranges from fully automatic composition to user-defined composition logic.
- **Type (Levels of composition):**
  - **Slicing:** Network hypervisors, such as FlowVisor, OpenVirtex, force each application module to operate on a disjoint subset, or slice, of the traffic.
  - **Ordering:** Multiple application modules can cooperate on processing the same traffic by ordering their actions.
  - **Conflict:** It indicates whether the approach considers tackling the detection and resolution of conflicts between individual policies generated by different application modules.

— **Applications are modified:** It indicates whether preexisting application modules written for the OF (or other standards) must be modified in order to meet the requirements of the composition framework/approach. This categorization is not applicable to approaches that introduce new programming languages, as existing applications cannot be reused unless they are totally rewritten with the new language.
— **Interface:** APIs used by the framework to implement composition and conflict resolution mechanisms.

3.4. Concluding remarks

Composition of network policies can be executed at different levels of the SDN architecture. Approaches that focus on the Application plane, such as Pyretic, Frenetic and others, introduce new programming languages to support the implementation of applications composed by several modules that jointly manage the network traffic. In particular, the Pyretic interpreter translates the high level instructions into low level messages for the underlying SDN controller.
Other mechanisms, such as FlowVisor, OpenVirteX, Covisor, NetIDE and FlowBricks, place a hypervisor between the Data plane and the application modules. Except NetIDE, which aims at supporting multiple control/management protocols, the others depend on a specific version of the OpenFlow protocol, which limits their application space.

4. DEBUGGING TOOLS

When we convert network behavior into a software, the first concept that comes into our minds as a tool is a debugger. As defined in [Rouse 2016], debugging or troubleshooting is the process of locating and fixing or bypassing bugs (errors) in a computer program code, but also in the engineering of a hardware device. Debugging a program or hardware device starts with a problem, follows with the isolation of the source of the problem, and finally ends with fixing the error. By following this definition, we already foresee two types of SDN debuggers: debuggers for the software or the SDN applications (control plane) and debuggers for the network device or SDN switch (data plane). Another classification considers whether a debugger requires the physical network to be deployed or running (*online debuggers*), or not (*offline debuggers*).

In the following sections, we analyze the different state-of-the-art SDN debuggers based on the first parameter (objective plane). Finally, last section summarizes all of them in a table.

4.1. A brief introduction on model checking

Before starting the analysis, we briefly introduce in this section the concept of model checking, as many of the tools will refer to it. Model checking is an automatic verification technique for finite state concurrent systems. It automatically provides complete proofs of correctness. A model checker systematically explores all possible ways to execute a program (as opposed to testing, which only executes one path depending on the input data). The process for model-checking a software consists basically on three steps:

— **Modeling:** Converts the system into a formalism (e.g. a machine state diagram).
— **Specification:** Determines the correctness properties to be validated by the Model Checker tool (e.g. no loops in a SDN network or the existence of black holes[2]).
— **Verification:** Validates the correctness properties in the modeled system and offers the outcomes obtained (whether or not the property is fulfilled in the system).

As all possible cases are taken into consideration and verified, the obtained results are “absolute”, which means that if the property “no loops in the system” has been validated, the network will never have a loop. However, the most pronounced disadvantage in this technique is the *State Space Explosion* problem, which consists on having a huge state space where the validation of a property is extremely complex and very costly computationally.

[2]A black hole is a place in the network where traffic is silently discarded, without informing the source or destination

4.2. Control plane debugging

NICE [Canini et al. 2012] finds inconsistencies in an SDN application (for instance: black holes in the network produced by an application running on top of an SDN controller). It creates a model from the network topology and the network application. Then, it systematically explores the whole state space of the model and checks the desired correctness properties against it. Eventually, NICE outputs the property violations found along with the traces to deterministically reproduce them. NICE addresses the State Space Explosion issue by means of the Symbolic Execution Engine. The authors consider that the event handlers of the SDN application are the key to explore the whole state space of the model. These handlers must be triggered in order to exercise the different paths of the state space. The Symbolic Engine identifies the packets that trigger these events and feed the network with them. NICE provides a library of correctness properties that can be extended by the user. It was designed for the NOX SDN controller framework [Gude et al. 2008].

Kuai [Majumdar et al. 2014] verifies that an SDN satisfies a specific property. It aims to be reach a higher performance than other SDN verificators by using a simplified version of an OpenFlow switch and using Murphi as the controller language (which implies translating SDN applications previously to the analysis). However, as a full model checker, it still needs to deal with the state-space explosion problem.

VeriCon [Ball et al. 2014] verifies that an SDN application satisfies a set of network-wide invariants (desired correctness properties to be validated) in all admissible topologies and for all possible (infinite) sequences of network events. Unlike NICE [Canini et al. 2012], which creates a finite state model checking why the tool is considered *unsound* (it can never prove the absence of errors in the infinite state SDN application), VeriCon is able to guarantee the absence of errors in SDN applications or to compute a concrete counterexample where a network-wide invariant is violated. It verifies that, for every event executed in an arbitrary topology, the SDN application satisfies the required correctness properties by means of a theorem prover that implements a classical Floyd-Hoare-Dijkstra deductive verification approach.

Verificare [Skowyra et al. 2014] is a platform built to enable formal verification of SDN applications. Verificare has three primary components: a modeling language (VML), a set of formal requirements, and translators for verification tools such as SPIN [Holzmann 2005] or PRISM [Kwiatkowska et al. 2011]. Users need to translate the SDN application, controller and network topology into the equivalent VML models, which are later on composed and checked by Verificare, raising counter-examples if found.

Assertion language for debugging [Beckett et al. 2014] is a language for verifying and debugging SDN applications. Like conventional programming languages, such as C or Python, which make use of assertions to facilitate finding bugs before the final deployment, the proposed assertion language supports debugging SDN applications by allowing programmers to annotate them with C-style assertions. The main contributions of this work are: (i) an assertion-based language to debug and verify dynamic properties of SDN applications and (ii) a verification process that leverages VeriFlow [Khurshid et al. 2012]. This language has been implemented as a debugging library and API atop a modified version of VeriFlow [Khurshid et al. 2012]. The evaluation has been performed in Mininet [Lantz et al. 2010] using Pyretic and POX.

Abstractions for Model Checking SDN Controllers [Sethi et al. 2013] leverages abstraction techniques to prove the correctness of controllers using model checking. Given a network topology, the correctness of an SDN application is proven with an arbitrary number of packets. It addresses the State Space Explosion problem by abstracting the *data state* and the *network state*. Such abstractions are based on keeping only one packet (*concrete packet*) in the system and on injecting packets with arbitrary header values (*environment packets*) in the network. The model size is significantly reduced as a result. Correctness properties that they validate icnlude: no loops and no black holes (both checked in very simple topologies). The main limitation of this approach is its poor scalability.

Toolkit for Automated Sdn TEsting (TASTE) [Lebrun et al. 2014] evaluates the feasibility of the test–driven methodology, inspired by software engineering principles to improve design, con-

figuration and test of SDN. Specifically, the authors propose a methodology for checking the compliance of SDN controllers with given data path requirements (e.g. a data path requirement could be that all outgoing traffic from a private network must pass through a firewall). Another contribution of [Lebrun et al. 2014] is the definition of a formal language called Data Path Requirement Language (DPRL) that allows the reproducibility of network tests.

Authors of **Kinetic** [Kim et al. 2015] propose a domain specific language and an SDN control system that allows operators to express dynamic network policies in an intuitive way. Kinetic is implemented as a Pyretic [Monsanto et al. 2013] module and leverages its high-level abstractions to provide a structured language for expressing network policies in terms of finite state machines (FSMs). Kinetic also uses the Pyretic's composition operators to create large FSMs by combining different smaller ones. The paper reports two different evaluations of Kinetic: (i) a user study to determine the degree of usability of Kinetic for network operators and (ii) a performance evaluation in terms of efficiency in compiling policies into flow rules by varying the number of policies, the size of the network and the rate of events.

Finally, **Automated Bug Removal for SDNs** [Wu et al. 2017] proposes a method for detecting and fixing bugs in SDN applications. It specifically focuses in generating automatic fixes for SDN following the approach of data provenance [Buneman et al. 2001] from databases, which tracks causality, but generalizing it to SDN. The main drawback is that it requires a customized NBI for applications (i.e. a specific SDN framework adapted to the approach).

4.3. Data plane debugging

SOFT [Kuzniar et al. 2012] uses the same technique than NICE (symbolic execution). In contrast, SOFT finds inconsistencies among the implementations of OF agents (software executed in OF switches). SOFT looks for inconsistencies by comparing the behaviors of OF switches from different vendors that may cause malfunctions in the network. It emulates an SDN controller (*Test harness*) capable of injecting symbolic inputs in an OF switch and it looks for a set of inputs which provoke that an OF agent behaves differently than others. Therefore, it exercises all possible paths in the software executed in the OF switch. SOFT has been tested using two publicly available OF agents compatible with the specification 1.0.

Similarly to SOFT, **OFLOPS** [Rotsos et al. 2012] is a software framework for OF switches evaluation. OFLOPS simultaneously emulates an OF controller and the network traffic and lets the user perform different tests to analyze the capabilities and the performance of OF-enabled software and hardware switches. OFLOPS is open source and can be bundled with specialized hardware in the form of the NetFPGA board [Watson et al. 2006] to ensure sub-millisecond-level accuracy of the measurements. OFLOPS presents the behavior and performance of five OF-enabled devices: three hardware switches from different vendors, an Open vSwitch-based [Foundation 2016] software switch, as well as a NetFPGA-based switch.

OFTest [Floodlight 2016c] is a Python-based test framework maintained by the *Project Floodlight* community. OFTest is meant for testing OF switch implementations and their compliance with the OF specification. OFTest is connected to both the control plane and the data plane of the switch. It provides a set of basic pre-configured tests which can be extended to cover more complicated test scenarios.

Similarly to OFTest, **Ryu's OpenFlow Switch Test Tool** [Ryu 2012a] also verifies the degree of compliance of an OpenFlow switch with the OpenFlow specifications. It is integrated in the source code of the Ryu SDN framework. The basic operation of this tool implies registering a flow or meter entry, by generating a specific packet and actions, and processing the expected result. Test scenarios are written in JSON, so that users can easily modify or add new ones.

FlowChecker [Al-Shaer and Al-Haj 2010] detects misconfigurations in OF switches. It enables network administrators/users to: (i) identify inconsistencies across paths within the same or different domains, (ii) validate the correctness of switches' flow tables and (iii) debug reachability and security problems. FlowChecker is written in C/C++ and uses the BuDDy library [Cohen 2004] to encode OF configurations with Binary Decision Diagrams (BDDs) [Akers 1978]. FlowChecker can

be used integrated within OF applications (as a library) running on top of NOX, or as a stand-alone tool that runs FlowChecker and communicates with one or multiple controllers by using a dedicated protocol. In the latter scenario, FlowChecker acts as an independent centralized server application, also called by authors "Master Controller", and receives queries from OF controllers in different domains sliced by the FlowVisor [Sherwood et al. 2010] hypervisor.

VeriSDN [IETF 2013] validates correctness properties (such as the absence of loops). Specifically, VeriSDN is a framework for the formal verification of SDN scenarios based on the process algebra called pACSR, which is an extended version of the packet-based Algebra of Communicating Shared Resources (ACSR) [Brmond-Grgoire et al. 1993], developed for formal verification of real-time embedded systems and cyber-physical systems.

ATPG (Automatic Test Packet Generation) [Zeng et al. 2012] proposes an automated and systematic approach for testing and debugging networks online. ATPG is a framework that automatically generates packets to: (i) test the liveliness of the underlying network (in terms of availability), (ii) verify the consistency of the data plane with respect to configuration specifications and (iii) test performance assertions, such as packet latency. ATPG detects errors by injecting test packets in the network so that every packet processing rule in the data plane is exercised and every link is tested. Differently from common techniques used by network operators, such as levaraging the `ping` command, ATPG is scalable for large networks and tests all links. ATPG operations are grounded on the *header space* network model [Kazemian et al. 2012] and on a packet selection algorithm that computes the minimal set of test packets, so that every forwarding rule can be exercised and covered by at least one test packet. Finally, a fault localization algorithm determines the failing rules or links.

Similarly to ATPG, **BUZZ** [Fayaz et al. 2016] is also a model-based framework to test the correctness of network policy implementations. It is specifically focused on expressiveness and scalability, and it claims to be five orders of magnitude faster than alternative designs. BUZZ has been implemented and evaluated in ODL.

Monocle [Peresini et al. 2015] follows the same principle that ATPG and checks inconsistencies in the data plane with respect to the control plane. It enhances ATPG as it also works efficiently in highly dynamic SDN networks, generating probe packets on a millisecond timescale and detecting misbehaving rules in switches in seconds. It is implemented as a combination of C++ and Python proxies: Multiplexer, responsible for forwarding `PACKET_IN/_OUT` messages, and Monitor, main proxy and responsible for tracking the tables, generating the probes and updating the controller.

Following the same approach, **VeriDP** [Zhang et al. 2016] continuously monitors the control-data plane consistency, using an abstraction of the control plane called *path table*, which is incrementally updated, and packet tagging at the data plane.

Analogously to the `ping` command, **sPing** [Tseng et al. 2017] diagnoses the data plane via packet injection, discovering problems such as: network loops or black holes, and also discovering the link layer information.

RuleScope [Bu et al. 2016] inspects SDN forwarding, generating probe packets and processing them afterward. While it can leverage packet tracing tools like NetSight (described below), it provides a series of monitoring applications implementing specific algorithms for detecting and troubleshooting rule faults. Experiments were performed with the Ryu SDN controller and the Pica8 P-3297 switch.

PathSeer [Aljaedi and Chow 2016] aims to trace packet trajectories in SDN-enabled data centers. PathSeer leverages OpenFlow to rewrite packet headers so that they contain the ingress port number of the switches traversed and, thus, the path followed. It claims to be much more scalable than other approaches, as it does not require to install so many flow rules. Furthermore, probe packets can be injected at any point of the network instead of depending on specific end points.

PathletTracer [Zhang et al. 2014] debugs multiple L2 paths. It focuses on *path tracing*, an operation for SDN troubleshooting that helps the network operators to improve network performance, to validate if a path is available and to allocate resources optimally. To know which path is traversing the packet, PathletTracer associates each given path with an identifier and it leverages unused bits

in a packet's header to carry the identifier across the path. Once a packet arrives to the destination, PathletTracer decodes the identifier to determine the path traversed.

SDN traceroute [Agarwal et al. 2014] is a packet-tracing tool for measuring paths in SDN networks. It leverages the SDN capabilities to overcome the limitations of the well-known tool *traceroute*, which only provides layer 3 path information as it relies on the *time-to-live* (TTL) field in the IP header. SDN traceroute operates regardless of the network layer. It runs as an application on an SDN controller so that it can install flow rules onto the SDN-enabled switches and listen to network events. Like *traceroute*, SDN traceroute injects probe packets to measure network paths. Its algorithm imposes two restrictions. First, SDN traceroute assumes that it can reserve some bits of the packet headers exclusively for its use. These bits must not be used when taking forwarding decisions and must not be modified by any device in the network. Second, SDN traceroute reserves the highest priority value (32,768 in OF).

sTrace [Wang et al. 2016] is a packet-tracing tool for SDN, but specifically focused on large multi-domain SDN networks. It considers other tools, such as SDN traceroute, do not scale well for big networks. sTrace has been implemented for OpenFlow 1.0 and 1.3, and tested with Open vSwitch and Mininet.

SDN-RADAR [Gheorghe et al. 2015] detects network issues by leveraging SDN to identify the most probable under-performing links in the network based on service degradation metrics. It is designed as a run-time application that injects test packets into the network to measure performance degradations and to calculate the most likely links where faults occurred. It also requires specific agents running at different locations in the network, which perform periodic measurements.

Netography [Zhao et al. 2016] defines the concept of *packet behavior* to locate network issues and find out their root causes. It troubleshoots the network by exporting packet behavior with probes, focusing on forwarding errors and performance degradation (latency and packet loss).

4.4. Both (control and data) planes debugging

OFf [Durairajan et al. 2014] is a debugging and testing environment for SDN platforms built on top of fs-sdn [Gupta et al. 2013], a simulation environment for SDN. OFf debugs and tests SDN applications by providing common features such as breakpoints or variable inspection. It allows *trace replay*, which reproduces network activity captured, and *report generation*, which generates a report upon changes happening in the topology or in the flow tables. OFf is designed to work with any controller platform, including POX, ODL and Ryu [Ryu 2012b].

STS [Scott et al. 2013] presents a technique, *retrospective causal inference*, for automatically identifying a minimal sequence of inputs that trigger a bug. It also detects equivalent events It has been applied to five open source SDN control platforms, namely Floodlight, NOX, POX, Pyretic and ONOS. Also, neither ndb [Handigol et al. 2012b] nor OFRewind (see below) address the problem of diagnostic information overload: with millions of packets, picking the right subset to debug can be challenging, and STS programmatically provides the information about what caused the network to enter an invalid configuration in the first place.

FLOWGUARD [Hu et al. 2014a] is a framework for OF-based networks for detecting and solving firewall policy violations. The authors argue that monitoring PACKET_IN messages is not sufficient to detect all firewall policy violations, since violations can also be induced by proactive installation of flow rules, by changes of the network state or by some OF actions such as SET_FIELD. A prototype of FLOWGUARD has been implemented for Floodlight and tested against the Floodlight built-in firewall. Although FLOWGUARD increases the time to inspect the packets with respect to the firewall, the performance overhead is considered acceptable.

NetPlumber [Kazemian et al. 2013] is a real-time policy checking tool based on Header Space Analysis (HSA) [Kazemian et al. 2012] (described below), which shares some authors with ATPG (presented before). NetPlumber sits between the control and data planes, and inspects the control channel (OF messages) to detect network state changes, thus detecting invariant violations such as: loops, reachability problems and black holes, and even checking user-defined policies. The NetPlumber's policy checking mechanism is built around the so-called *plumbing graph* which captures

all the paths derived by the flow installed on the network. The performance of the graph update process has been measured for new flow and link up events on three real-world network: the Google inter-data center WAN, the Stanford University's backbone and Internet2 [Internet2 2016]. The results show that NetPlumber takes longer to update the graph when a link is added. Although a link up/down is usually a rare event, the authors state that NetPlumber is not suitable for networks with a high link up/down rate such as energy-proportional data center networks [Abts et al. 2010].

VeriFlow [Khurshid et al. 2012] checks the network–wide correctness in real–time. It leverages SDN to obtain the state of the network by sitting between the SDN controller and the physical network, checking the validity of network invariants (loops, black holes, path availability, etc.) each time a new rule is added, removed or modified. VeriFlow confines the verification activities to those parts of the network whose actions may be influenced by the new update. Thus, it slices the network into the so-called *equivalence classes* (EC) which are sets of packets that are subject to the same forwarding actions throughout the network. The network state is checked only for those ECs that are affected by the update. VeriFlow represents the network behavior with *forwarding graphs*, where nodes are pairs *(EC, device)* and edges represent a forwarding decision for each (EC, device) pair. Finally, invariants are specified as verification functions that take the forwarding graphs as input and that are used to check potential violations of key network invariants. The evaluation demonstrates that VeriFlow's verification time is linear with the number of ECs involved in the network update. Therefore, VeriFlow has difficulty in verifying invariants in real-time when a large number of ECs is affected by the update (e.g. link failure). VeriFlow was implemented within the NOX controller and recently founded its own company [Systems 2016].

Libra [Zeng et al. 2014] is a fast, scalable tool to detect loops, black-holes, and other reachability failures. It takes as a starting point the modeling approach of NetPlumber, HSA and VeriFlow, by taking and analyzing a snapshot of the forwarding tables, but it focuses on huge network (tens of thousands of switches). They consider snapshots might be inconsistent in large networks with frequent changes to routing state, and they also believe tools should accomplish the performance requirements of modern data center networks. For this reason, they simplify the analysis by assuming packet forwarding based on longest prefix matching. Once a stable snapshot has been captured, Libra check its correctness by dividing the task into smaller, parallel operations, computed with MapReduce [Dean and Ghemawat 2008].

Delta-net [Horn et al. 2017] also competes directly with NetPlumber or VeriFlow, automatically detecting violations in the network, by following program analysis techniques. It is based on the observation of similar forwarding behavior of packets through parts of the networks (while previous works focus on the entire network) and, for this reason, it claims to be ten times faster. It has been implemented and tested with the SDN-IP application in ONOS.

HSA (Header Space Analysis) [Kazemian et al. 2012] is a protocol-agnostic framework proposed to identify failures such as reachability failures, forwarding loops, traffic isolation and others. It consists of a geometric model where packets are points in a *network space* and the network boxes are functions that transform points in the defined network space. Moreover, they show how such a formalism solves the aforementioned network failures in a protocol-agnostic way. Although they prove how HSA can be implemented online by testing it in the Stanford University's backbone network, they do not explicitly mention potential scalability issues. The techniques described above have been implemented as a library called Header Space Library (Hassel).

SHSA (Stateful Header Space Analysis) [Yang et al. 2016] extends HSA to detect and solve invariant violations with stateful middleboxes. After testing it in the Stanford University's backbone network. Authors claim to reach higher efficiency and scalability than HSA.

Authors of **FlowTest** [Fayaz and Sekar 2014] argue that existing testing and verification tools for SDNs often focus only on ensuring that the network meets specific reachability requirements (e.g., no black holes, no loops, etc.). However, such tools are not able to handle many data plane functions (DPF), such as firewalls or load balancers, or complex policy requirements, such as the correct implementation of service chaining policies. The objective of this work is to define the conceptual

foundations for a data plane testing framework able to tackle stateful and dynamic DPFs, and policy requirements. FlowTest is designed with three main logical components:

1. *Test traffic planner*, which generates a test traffic plan, responsible of coordinating the traffic *injectors* to generate traffic traces for testing desired properties.
2. *Injectors*, common hosts connected to the network that run traffic generators or trace injection software, driven by the planner.
3. *The monitoring and validating engines*, to monitor and validate the status of the SDN controller and the data plane.

The authors have implemented an initial prototype of the test traffic planner by using a tool based on Artificial Intelligence (AI) and called GraphPlan [Blum 2001].

OFRewind [Wundsam et al. 2011] provides network behavior record and replay. OFRewind sits between the network and the SDN controller, intercepting and modifying the control messages. While OFRewind takes care of recording and replaying control traffic, it delegates recording and replaying of data traffic to the *DataStore* elements locally attached to the SDN switches. When replaying the control traffic, it emulates an SDN controller toward the SDN switches or, vice-versa. To record data traffic, OFRewind leverages on the control channel to instruct the SDN switches to mirror the traffic to the DataStore elements. Vice-versa, the DataStore elements re-inject the recorded traffic into the network during the replay process.

SDNRacer [El-Hassany et al. 2016] levarages STS to implement a controller-agnostic debugger for production-grade SDN controllers. It detects invariant violations, being able to describe the precise sequence of events that caused them (i.e. the exact pairs of read/write events). Differently from approaches inspecting the control plane, the speed of the analysis in SDNRacer only depends on the trace size, which is more scalable. SDNRacer has been implemented for POX, Floodlight and ONOS.

NetSight [Handigol et al. 2014] captures and builds packet histories and makes them available through an API. A packet history is the route that a packet traverses plus the switch state and header modifications at each hop. NetSight assembles packet histories into *postcards*, event records created whenever a packet traverses a switch. Packet histories can be filtered via a regular-expression-like language, Packet History Filter (PHF). Leveraging on the aforementioned API and PHF, the authors built four applications on top of NetSight: *ndb*, *netwatch*, *netshark* and *nproof*, which are an interactive network debugger, a live network invariant monitor, a network packet history logger and a hierarchical network profiler, respectively. These tools are presented separately in the following paragraphs. NetSight has been developed in C++ and tested with the following controller frameworks: NOX, POX and RipL-POX [GitHub 2012]. Recently, some of the authors of NetSight, together with authors of other debugging tools surveyed in this paper, founded Forward Networks [ForwardNetworks 2016], where the main product is a platform for network assurance.

ndb [Handigol et al. 2012b] is an interactive network debugger which later evolved into the whole NetSight project. ndb provides interactive debugging features for networks, analogous to those provided for software programs by the GNU Project Debugger (GDB) [Foundation 1986]. ndb allows developers to detect and debug wrong network behaviors leveraging on packet histories provided by the NetSight platform. ndb is able to diagnose common bugs such as: reachability errors, race conditions, incorrect packet modifications.

netwatch is a live network invariant monitor. netwatch allows the operator to specify a network behavior in form of invariants and it triggers an alarm whenever a packet violates any invariant. The current supported invariants are: isolation between different groups of hosts, loops, waypoint routing to catch packets that do not go through a specific waypoint (e.g. a proxy) and max-path-length to detect paths that exceed a certain length (e.g. the diameter of the network).

netshark is a Wireshark-like [Wireshark 2016] application that allows users to filter the history of packets. netshark accepts PHF specifications as input and returns the collected packet histories matching the query. A dissector for Wireshark is provided to analyze the results.

Table II. Comparison table of the different debugging tools (1/2 - Control and Data planes)

Tools \ Properties	Description	Type (Targeted plane)	Features				Interface
			Execution time	Modeling	Packet history	Packet injection	
NICE	Finding inconsistencies in SDN applications	Control Plane	Offline	✓			OpenFlow 1.0
Kuai	Finding inconsistencies in SDN applications	Control Plane	Offline	✓			OpenFlow 1.0
VeriCon	Finding inconsistencies in SDN applications	Control Plane	Offline	✓			OpenFlow 1.4
Verificare	Formal verification of SDN applications	Control Plane	Offline	✓			Custom API
Assertion language for debugging	Formal verification of SDN applications	Control Plane	Offline	✓			Custom API
Abstractions for model checking SDN contr.	Proving correctness of SDN controllers	Control Plane	Offline	✓			Custom API
TASTE	Compliance check of SDN controllers	Control Plane	Offline			✓	Custom API
Kinetic	Formal verification of SDN applications	Control Plane	Online	✓			Custom API
Automated Bug removal for SDN	Provides fixes for SDN applications	Control Plane	Online		✓		Custom API
SOFT	Finding inconsistencies in OF agents	Data Plane	Offline				OpenFlow 1.0
OFLOPS	Analyzing capabilities of OF switches	Data Plane	Offline				OpenFlow 1.0
OFTest	Compliance check of OF switches	Data Plane	Offline			✓	OpenFlow 1.0, 1.1
Ryu's Switch Test Tool	Compliance check of OF switches	Data Plane	Offline			✓	OpenFlow 1.0, 1.3, 1.4
FlowChecker	Detects misconfigurations in OF switches	Data Plane	Online	✓			OpenFlow 1.0, Custom API
VeriSDN	Formal verification of SDNs	Data Plane	Online	✓			Custom API
ATPG	Consistency of data plane with control plane (via packet injection)	Data Plane	Online	✓		✓	OpenFlow 1.0
BUZZ	Consistency of data plane with control plane (via packet injection)	Data Plane	Online	✓		✓	OpenFlow 1.0+
Monocle	Consistency of data plane with control plane (via packet injection)	Data Plane	Online	✓		✓	OpenFlow 1.0+
VeriDP	Consistency of data plane with control plane (via packet injection)	Data Plane	Online	✓		✓	OpenFlow 1.0+
sPing	Inspects SDN behavior (via packet injection)	Data Plane	Online			✓	OpenFlow 1.0, 1.3
RuleScope	Inspects SDN forwarding (via packet injection)	Data Plane	Online			✓	OpenFlow 1.0+
PathSeer	Inspects SDN forwarding (via packet injection)	Data Plane	Online			✓	OpenFlow 1.0+
PathletTracer	Inspects SDN forwarding (without packet injection)	Data Plane	Online		✓		OpenFlow 1.0
SDN Traceroute	Inspects SDN forwarding (via packet injection)	Data Plane	Online		✓	✓	OpenFlow 1.0+
sTrace	Inspects SDN forwarding (via packet injection)	Data Plane	Online		✓	✓	OpenFlow 1.0, 1.3
SDN-RADAR	Inspects SDN performance (via packet injection)	Data Plane	Online		✓	✓	Custom API
Netography	Inspects SDN behavior (via packet injection)	Data Plane	Online		✓	✓	Custom API

Table III. Comparison table of the different debugging tools (2/2 - Both planes)

Tools \ Properties	Description	Type (Targeted plane)	Features				Interface
			Execution time	Modeling	Packet history	Packet injection	
OFf	Debugging and testing of SDN controllers and switches (trace replay and breakpoints)	Both	Offline		✓	✓	OpenFlow 1.0+
STS	Debugging and testing of SDN controllers and switches (provides right subset to debug)	Both	Offline		✓	✓	OpenFlow 1.0+
FLOWGUARD	Detects and solves firewall policy violations	Both	Online	✓			OpenFlow 1.0+
NetPlumber	Detects and solves invariant violations (loops, blackholes, etc.)	Both	Online	✓			OpenFlow 1.0+
VeriFlow	Detects and solves invariant violations (loops, blackholes, etc.)	Both	Online	✓			OpenFlow 1.0+
Libra	Detects and solves invariant violations (loops, blackholes, etc.)	Both	Online	✓			OpenFlow 1.0+
Delta-net	Detects and solves invariant violations (loops, blackholes, etc.)	Both	Online	✓			OpenFlow 1.0+
HSA	Detects and solves invariant violations (loops, blackholes, etc.)	Both	Online	✓			Custom API
SHSA	Detects and solves invariant violations (loops, blackholes, etc.)	Both	Online	✓			Custom API
FlowTest	Checks correct implementation of functions and policies	Both	Online			✓	Custom API
OFRewind	Debugging and testing of SDNs (replay of data and control traffic)	Both	Online		✓	✓	OpenFlow 1.0
SDNRacer	Debugging and testing of SDNs (packet histories)	Both	Online		✓		OpenFlow 1.0+
NetSight	Debugging and testing of SDNs (packet histories)	Both	Online		✓		Custom API

Finally, ***nprof*** is a network profiler and, as such, it will be described in Section 6 devoted to profilers.

4.5. Summary of the different debugging tools

Tables II and III summarize and compare the different debugging tools, based on:

— **Description:** Concise description of the tool.
— **Type (Targeted plane):** Whether analysis and verification focus on the data plane (SDN switches), the control plane (SDN applications) or both (overall SDN behavior).
— **Features:**
  — **Execution time:** Two types of tools: *offline*, which fulfill their purpose when the network is not being executed, and *online*, which accomplish their function while the SDN network is running (either at deployment or at run-time).
  — **Modeling:** This parameter indicates whether the tool creates a model for later verification or not.
  — **Packet history:** Tools that keep track of the information from the packets that traverse the network. In some cases, by tracking the traffic crossing a certain point and in other cases by recording information about nodes traversed by a packet and the modification of its header fields.
  — **Packet injection:** Tools that meet this parameter inject a certain number of specific packets in the analyzed network to troubleshoot misbehaviors in the network.
— **Interface:** The interface required by the tool between control and data planes.

4.6. Concluding remarks

Debugging tools are the most miscellaneous group of SDN tools. They follow different approaches and focus on different parts of the SDN architecture. Many tools in this category are based on OpenFlow and support multiple versions of such protocol, thus they can be used in combination with most of the popular SDN controller platforms and SDN-enabled switches. However, several other tools implement custom Application Programming Interfaces (APIs). Those tools urge a deep analysis on *what* parts of the SDN they aim to debug and, as a consequence, on *how* to model a common interface.

5. MEMORY MANAGEMENT

Memory management is a critical part of any computer Operating System (OS), as it is the process that controls and coordinates the computer memory to optimize overall system performance. The terms *memory management* usually refer to operations such as memory allocation/deallocation, virtualization, protection, mapping and swapping. Like the computer OS, SDN controllers have the ability to observe and control hardware resources (i.e. network elements) while providing programmatic interfaces to the applications to access those resources. For these reasons, SDN controllers are often referred as Network Operating Systems (NOSs). Nevertheless, most of the open source NOSs available nowadays force the developers of SDN applications to take care of memory management tasks, like cleaning the memory of the switches from unused flow rules or setting appropriate idle and hard timeouts to the installed rules. More practically, SDN applications install the forwarding rules into the switch memory, mainly Ternary Content-Addressable Memory (TCAM). However, this memory has a finite capacity and allows to accommodate only a few thousand wildcard flow rules, while recent studies have shown that data centers can have up to 10,000 network flows per second per server rack today [Benson et al. 2010]. Since TCAMs are power hungry, expensive and require significant silicon space, increasing their size is not a viable solution to reduce the risk of reaching the full capacity. In this context, we can divide the memory management operations in two different categories: (i) deletion of unused flow rules from the switches' memory and (ii) optimization of the memory usage.

5.1. Memory cleaning

A flow rule can be classified as *unused* for two main reasons: the application that installed the rule has been deactivated or uninstalled from the NOS or the rule is never matched by the network traffic. Current NOSs do not foresee any mechanism to automatically remove them. Such a behavior is potentially harmful and may affect the stability of the network; in fact, the rules that have not been removed may match part of the incoming traffic, thus leading to undesired network actions and preventing newly installed rules to work properly. Recent NOSs, such as **ONOS**, implement mechanisms to purge entries on a per application basis, despite not automatically, while most controllers leave this duty to the developers of SDN applications.

The problem of collecting and removing rarely matched flow rules has been tackled by the authors of **FRESCO** [Shin et al. 2013]. FRESCO is a programming framework for advanced security. Among other components, FRESCO offers a *resource controller* that monitors OF devices. The resource controller performs two main functions: (i) the *switch monitor*, periodically collects switch status information, such as the number of empty flow entries, and stores the collected information in the switch status table, (ii) the *garbage collector* checks the switch status table to monitor whether the flow table in an OF switch is nearing capacity, and then identifies and evicts the least used flow rules.

The OF specifications [ONF 2013] define two specific mechanisms, namely *Eviction* and *Vacancy events*, which can be used by the developers for the control of the memory utilization to avoid getting the memory full and the possible service outages that may happen consequently. **Eviction** enables the OF switches to automatically eliminate the flow entries of lower importance. Such an eviction mechanism is handled by the SDN applications that can (i) enable/disable it on a per table

basis and (ii) configure it by setting the importance of each flow entry. **Vacancy events** introduces a mechanism enabling the SDN applications to get an early warning based on a capacity threshold chosen by the SDN developer. This allows the applications to react in advance and avoid the memory full condition.

5.2. Memory optimization

Despite current NOSs lack of an automatic memory management system, several approaches have been proposed for an optimal usage of the memory resources (i.e. flow table space) of the network devices.

The goal of **CacheFlow** [Katta et al. 2014] is to give network applications the illusion of an arbitrarily large switch memory. It is achieved by defining a hardware-software hybrid switch design that relies on rule caching mechanisms. Architecturally, CacheFlow consists of a component interposed between the controller and the OF hardware switches. CacheFlow receives the OF commands from the controller and uses the OF protocol to distribute the rules to the underlying switches. During this process, CacheFlow selects a set of important rules from among the rules given by the controller to be cached in the TCAM of the hardware switches, while redirecting the cache misses to the software switches inside CacheFlow.

The swapping mechanism is one of the two functions of the **Memory Management System** (MMS) for SDN controllers proposed in [Marsico et al. 2017]. This mechanism monitors the occupancy of the TCAM of SDN-enabled switches by intercepting the `TABLE_FULL` OpenFlow error messages and the `TABLE_STATUS` OpenFlow events with reason `VACANCY_DOWN`. The `TABLE_FULL` error is used to detect when the TCAM is full, while `TABLE_STATUS` events indicate that the remaining space in the TCAM has decreased below a pre-defined threshold. Based on this OpenFlow events, the swapping mechanism moves (swaps out) the least used flow entries from the TCAM of the switches to a database maintained by the MMS. Since the MMS intercepts all the `PACKET_IN` OpenFlow messages, it automatically (and transparently to the SDN applications) re-installs (swaps in) the previously swapped out flow entries onto the TCAM when they are needed again to forward the traffic. The authors implemented the MMS and the swapping mechanism for the ONOS platform by using the Java NBI. However, in another work [Doriguzzi-Corin et al. 2016b], they provide requirements and specifications for implementing the MMS for other well-known SDN platforms such as OpenDaylight, Floodlight, Beacon and Ryu.

SmartTime [Vishnoi et al. 2014] employs an adaptive heuristic to compute idle timeouts for the flow rules. It aims to optimize TCAM utilization (e.g. via eviction of flow rules) and, at the same time, to reduce the number of table misses (and, as a consequence, the controller load). Its strategy is based on the following features: a small initial timeout, a rapid ramp up for frequent flows, a maximum idle timeout, a timeout reduction for short flows that repeat often but after a long gap, and proactive eviction based on a threshold.

Tag-in-Tag [Banerjee and Kannan 2014] aims at providing a high level compaction of the flow entries in the TCAM memories and reducing the TCAM power consumption. Tag-In-Tag achieves these goals by replacing the OF entries stored in the TCAM memories with two layers of tags. One tag (referred as PATH TAG (PT)) exploits the availability of a unique path for individual flows from the ingress switch to the egress switch that can be computed a priori. The second one (referred as FLOW TAG (FT)) allows finer identification of the flows to enable flow specific actions. The Tag-In-Tag concept is based on commonly observed phenomena of networks: (i) a flow takes a path, (ii) the paths are deterministic set (all source-destination paths are known a priori) and (iii) multiple flows can take the same path. Through various experiments, authors show that the Tag-In-Tag approach can accommodate 15 times more flow entries in a fixed size TCAM whereas power consumption per-flow is reduced by 80% compared to an "unoptimized" SDN-enabled switch.

Authors of **DevoFlow** [Curtis et al. 2011] propose a modification of the OF model where part of the control is delegated back to the switches, while the controller maintains control over only targeted *significant flows*. By modifying the *action* of wildcard rules, it promotes the use of the exact-match lookup table, thus reducing the use of the TCAM. As this adaptation requires an en-

Table IV. Comparison table of the different memory management approaches

Properties / Approaches	Description	Type	Mechanism	Interface
FRESCO	Eviction of the least used flow rules	Memory Cleaning	Flow rule eviction	FRESCO API
Eviction	Eviction of the least important flow rules	Memory Cleaning	Flow rule eviction	OpenFlow 1.4+
Vacancy Events	Notification when the TCAM is reaching full capacity	Memory Cleaning, Memory Optimization	"Memory full" warning	OpenFlow 1.4+
CacheFlow	Arbitrarily large virtual flow tables	Memory Optimization	Flow rule swapping	OpenFlow 1.0
MMS	Arbitrarily large virtual flow tables	Memory Optimization	Flow rule swapping	ONOS NBI
SmartTime	Heuristic to compute efficient idle timeouts	Memory Optimization, Memory Cleaning	Idle timeout optimization	Floodlight NBI
Tag-in-Tag	Replacement of flow rules with shorter tags	Memory Optimization	Flow rule compaction	N/A
DevoFlow	Leveraging exact match rules to save TCAM space	Memory Optimization	Flow rule cloning	N/A

hancement of the switch devices, DevoFlow was evaluated in a simulated environment. The results show that DevoFlow uses 10–53 times fewer flow table entries at an average switch, and uses 10–42 times fewer control messages.

5.3. Summary of the different memory management approaches

Table IV summarizes and compares the memory management approaches, based on:

— **Description:** The basic idea behind the proposed approach, in short.
— **Type:** Two main categories: memory cleaning and memory optimization. Both have the objective of saving TCAM memory space for newer rules and of improving the performance of the network. *Memory cleaning* includes mechanisms conceived to remove the flow rules that meet certain criteria (e.g. low traffic counters). Mechanisms in the *Memory optimization* category aim at saving TCAM space without deleting the rules but improving the way this space is used.
— **Mechanism:** Brief description of the technical solution proposed by each approach. *Eviction* means the action of removing the flow rules from the TCAM memory. Other approaches aim at saving TCAM space by compacting the rules, by tampering with the idle timeouts of by moving the rules to a different (often slower) memory. *Warnings* is a mechanism defined in the OF specifications v1.4+ (called *Vacancy Events*) that enables the controller to react in advance before the TCAM gets full.
— **Interface:** The interface between the tool and the SDN controller. This information is not available for *Tag-in-Tag* and *DevoFlow*, since such works focus on the description of a mechanism without providing any concrete implementation detail.

5.4. Concluding remarks

Eviction and vacancy events APIs are available in OpenFlow 1.4 or newer, while CacheFlow only works in SDN environments based on OpenFlow 1.0. That is, they impose strict constraints on SDN controllers and SDN-enabled devices. On the other hand, FRESCO, MMS and SmartTime leverage the NBI of the controller that hosts them. However, although their implementation is based on different APIs, they require a relatively small set of messages and services to accomplish the management of the switches' memory. In summary, such tools need to: (i) be notified on new flow arrivals, (ii) modify of the switches' flow table, (iii) collect flow statistics from switches, (iv) receive flow removal notifications, and (v) receive notifications related to the status of the TCAM capacity.

6. PROFILING

In software engineering the term *profiling* is known as the performance analysis of a program. Commonly, profiling a program refers to gather relevant data, such as the execution time or its memory consumption. The collection of data, as opposed to static code analysis, is carried out while the program is being executed. A profiler can provide different outcomes including an execution trace or a statistical summary. Another term used in reference to profiling is monitoring, although the latter implies a passive behavior and the former an active one, where the current scenario could be modified based on the gathered information. In this section, a summary of the most relevant profilers for SDN environments is provided.

OFCBenchmark [Jarschel et al. 2012] is a multi-thread OF controller benchmark tool that analyzes the performance of SDN controller platforms by generating requests for packet forwarding rules and watching for responses from the controller. It improves Cbench [Sherwood and Yap 2011], a single-thread benchmarking tool for controller, with improved scalability, modularity and the ability to provide fine-grained performance statistics. The authors compare OFCBenchmark with Cbench in terms of performance by measuring the throughput of the NOX controller. Although OFCBenchmark implements advanced features, its performance results are comparable with the ones produced by Cbench.

SPIRIT [Kang et al. 2015] is an SDN profiler that automatically discovers bottlenecks in SDN applications. SPIRIT connects to the NBI of the SDN controller to collect profiling data of the SDN application under testing. At the same time, SPIRIT records the CPU load of the machine where the controller is running. The collected data is then analyzed for discovering any critical path in the execution flow of the SDN application. A prototype of SPIRIT has been implemented as a proof-of-concept and used to profile Floodlight and ONOS applications.

The **NetIDE profiler** tackles the problem of profiling SDN environments by leveraging the NetIDE Network Engine architecture [Doriguzzi-Corin et al. 2016a]. It comprises an *Application profiler* and a *Network Profiler*. The former provides the execution time of network applications at different granularity levels (from application modules to specific software functions), while the latter retrieves network statistics such as the current network load. Although the NetIDE profiler uses a dedicated interface (based on the NetIDE API), it can be potentially used with any control platform thanks to the adaptors provided by the Network Engine platform. The code is publicly available at [GitHub 2016].

To better balance the monitoring overhead and the anomaly detection accuracy, the author of **OpenWatch** [Zhang 2013] proposes a prediction-based algorithm that dynamically change the granularity of measurement along both spatial and temporal dimensions. OpenWatch starts by collecting coarse-grained data from the switches, then the collected information is compared with the data obtained previously. If an anomaly is detected, it iteratively adjusts the wildcard rules and reports fine-grained information to the anomaly detection applications. Additionally, in case of anomaly, the reporting frequency is increased.

nprof: nprof is network profiler included in the NetSight [Handigol et al. 2014] platform (presented in Section 4.4). It focuses on the data plane and profiles network links to understand the traffic characteristics and routing decisions that determine the link utilization. nprof combines topology information and packet histories to show which switches are injecting traffic to a specific link and how much. Furthermore, nprof is able to identify how subsets of traffic are being routed across the network. This information helps to understand how to distribute the traffic load in the network.

OpenNetMon [van Adrichem et al. 2014] is a passive flow-based monitoring system. It collects samples of traffic and estimates per-flow QoS metrics such as throughput and packet loss. It is implemented as a module for the POX controller.

Sonata [Gupta et al. 2016] presents an architecture for refined active monitoring. Sonata allows operators to express network monitoring queries that are efficiently partitioned among the network switches, reducing the overall data rate and, therefore, ensuring scalable traffic rates of several

terabits per second. Sonata's framework has been implemented in Ryu and OpenFlow 1.3, but is planned to be extended and optimized with P4 [Bosshart et al. 2014] in the future.

FlowSense [Yu et al. 2013] is a monitoring tool for OF-based networks that takes advantage of the control channel to provide high accuracy link utilization monitoring with zero measurement cost. Instead of polling the devices to retrieve traffic statistics, FlowSense relies on OF messages such as `PACKET_IN` and `FLOW_REMOVED` sent by the switches to the controller. Based on its design, FlowSense works in reactive OF deployments, where switches generate control messages every time a new flow arrives or a flow entry expires. On the other hand, there are some scenarios where the proposed approach fails. For instance, when there is little or no control traffic or when the input port field is wildcarded.

Payless [Chowdhury et al. 2014] is a network monitoring framework for SDN that operates on top of the OF controllers, leveraging the controller's NBI to collect and aggregate network statistics. Moreover, Payless exposes to applications a uniform and high-level RESTful API for expressing monitoring requirements. Like FlowSense described above, Payless intercepts `PACKET_IN` and `FLOW_REMOVED` messages to keep track of flow installations and removals. The authors provide a comparison between Payless and FlowSense, and they demonstrate that Payless can achieve higher accuracy of statistics collection than FlowSense. Payless has been implemented as an application for Floodlight.

Following a similar approach to Payless, **PathMon** [Wang et al. 2016b] enhances it by providing the flexibility of querying path-specific flow statistics at any aggregation levels.

SDN Interactive Manager [Heleno Isolani et al. 2015] is an OF-based monitoring software which: (i) monitors the resource consumption and control channel load, (ii) presents aggregated statistics and (iii) supports the configuration of network parameters that affect the analyzed metrics. The SDN Interactive Manager connect to the controller's RESTful NBI and it is accessible by the users through a GUI. A prototype of the SDN Interactive Manager has been implemented for the Floodlight controller.

The work **Network State Collection Methods** [Aslan and Matrawy 2016] does not introduce any specific SDN tool. Instead, the authors provide an analysis of active and passive network state collection mechanisms and their impact on SDN applications. The analysis focuses on OF-based mechanisms for collecting network state information, which involve the use of the OF API to keep track of control messages (passive mode) and to collect flow, port or other statistics (active mode). Through a series of experiments, the authors demonstrate that in case of low-variation traffic, where flows are comparable in byte counts, the application based on passive state collection performs better than the one that relied on active state collection. On the other hand, the performance of the application that relies on active state collection is mainly dependent on the polling periods: as the polling period increases, the performance degrades.

6.1. Summary of the different profiling tools

Table V summarizes and compares the profiling tools, based on:

— **Description:** Concise description of the tool.
— **Type:** Only meaningful for data plane profilers. Active profilers send messages or actively configure the network devices, passive profilers silently monitor the control channel.
— **Target:** Either data plane or the control plane (controller and SDN applications).
— **Interface:** The interface between the tool and the SDN controller. This information is not available for *OpenWatch* and *Network State Collection Methods*, since the former focuses on the algorithm rather that implementing a tool for SDN controllers, the latter presents an overview of active and passive network state collection mechanisms.

6.2. Concluding remarks

Control plane profilers use the SBI to stress the controller and to measure how they perform when handling "new flow" messages. This is only partially true for SPIRIT, which also connects to the

Table V. Comparison table of the different profiling tools

Tools \ Properties	Description	Type	Target	Interface
cbench	Benchmarking of SDN controllers	-	Control Plane	OpenFlow 1.0
OFCBenchmark	Benchmarking of SDN controllers	-	Control Plane	OpenFlow 1.0
SPIRIT	SDN application profiler	-	Control plane	Floodlight NBI ONOS NBI
NetIDE profiler	Application monitoring and data plane statistics collection	Active	Control and data planes	NetIDE API
OpenWatch	Monitoring tool that balances overhead and accuracy	Active	Data plane	N/A
nprof	Link utilization monitor	Active	Data plane	NetSight API
OpenNetMon	End-to-end QoS monitor	Active	Data plane	POX NBI
Sonata	Optimized network monitoring queries	Active	Data plane	Sonata API
FlowSense	Link utilization monitor	Passive	Data plane	Not specified controller NBI
Payless	Framework that combines active and passive monitoring	Active/Passive	Data plane	Floodlight NBI
PathMon	Path-specific flow statistics collection	Active/Passive	Data plane	Floodlight NBI
SDN Interactive Manager	Control channel monitor Data plane monitor	Active/Passive	Data Plane	Floodlight NBI
Network State Collection Methods	Analysis of active/passive methods for monitoring the data plane	Active/Passive	Data plane	N/A

−: Means *not applicable*.

controller's NBI to monitor the applications. The NBI of the SDN controller is also used by data plane profilers to collect statistic counters from the SDN switches or to intercept network events such as new flow arrivals or flow rule expirations. A different approach is proposed by the NetIDE profiler, which is potentially compatible with any SDN controller thanks to the NetIDE platform. An open question remains though: *how to effectively profile the network without affecting it?*

7. SIMULATORS AND EMULATORS

Network emulators and simulators allow researchers and network practitioners to evaluate the behavior of networks when subjected to a given workload. With the introduction of the OpenFlow protocol, well-known simulators have been extended with additional components to provide support to OF-based experiments. At the same time, many SDN-enabled emulators have been developed based on software switches, such as Open vSwitch (OvS) [Foundation 2016], CPqD's ofsoftswitch13 [Fernandes and Rothenberg 2014] or Indigo Virtual Switch (IVS) [Floodlight 2016b].

Mininet [Lantz et al. 2010] is a network emulator that provides a rapid prototyping workflow for SDN, by combining lightweight virtualization with an extensible CLI and API on one physical machine. The different nodes in Mininet are simply shell processes with their own network namespace, such as interfaces, ports, and routing tables. The switches shaping the network are software-based OF switches. Once the network is created, Mininet includes a network-aware command line interface (CLI) that allows developers to control and manage the entire network, interacting with it running commands on hosts, verifying switch operation, and even inducing failures or adjusting link connectivity. Hosts can also execute any other command installed in the OS and accessible by the shell. In addition, Mininet provides the opportunity to use a Python API to create custom experiments, topologies, and node types.

As originally implemented, it did not provide any assurance of performance fidelity and then **Mininet-HiFi** [Handigol et al. 2012a] improved it. Those enhancements were included from the release 2.0.0 in Mininet, a major upgrade that expanded Mininet's scope from functional testing

to performance testing. Mininet also has a Cluster Edition prototype [Lantz and O'Connor 2015], although it is considered experimental and MaxiNet (below) is recommended instead. Another clustering examples for Mininet have been also described in **Mininet-CE** [Antonenko and Smelyanskiy 2013] and DOT (below). Finally, **Datacenter in a box** [Teixeira et al. 2013] and **SDDC** [Darabseh et al. 2015] are two different proposals for an SDN data center experimental framework.

When emulating large networks with both high link bandwidths and high traffic volume, the computational complexity of the emulation overwhelms today's computers. In this way, **MaxiNet** [Wette et al. 2014] fixes those limitations of Mininet, using multiple physical machines for large-scale SDN emulations. The whole process of mapping and deploying the network to be emulated onto the physical environment is transparent to the user, since MaxiNet is an abstraction layer connecting multiple, unmodified Mininet instances running on different workers. A centralized API is provided for accessing this cluster of Mininet instances and GRE tunnels are used to interconnect nodes emulated on different workers. Therefore, MaxiNet works as a front end for Mininet that sets up all Mininet instances, invokes commands at the nodes and sets up the tunnels required for proper connectivity. MaxiNet also includes traffic generators: **DCT^{2}Gen** [Wette and Karl 2015] emulates the traffic behavior of data centers and **netSLS** [Wette et al. 2015] combines Hadoop's Yarn Scheduler Load Simulator (SLS) [Foundation 2014] with MaxiNet to emulate Hadoop network traffic based on artificial or real world job traces.

Distributed OF Testbed (DOT) [Roy et al. 2014] proposes a highly scalable emulator for SDN that provisions the emulated network across a cluster of machines. Unlike Mininet and MaxiNet, DOT provides guaranteed compute and network resources for the emulated components (such as switches, hosts and links).

OFNet [Shankar 2016] is an recent SDN emulator that aims to bring the capabilities of Mininet plus some monitoring and traffic generator tools, as the author considers it is difficult to debug SDN networks just via pinging, as it is usually done with Mininet. OFNet is an open source project currently distributed as a virtual machine image, but the code will also be available soon.

Virtual Network Overlay(ViNO) [Bemby et al. 2015] is an orchestration service that creates arbitrary network topologies with OvS switches and VMs. The overlay interconnection between VMs is provided through VXLAN encapsulation [Mahalingam et al. 2014]. Similarly to Mininet, the user can specify the network topology using a Python-based Domain Specific Language (DSL). Differently from Mininet and MaxiNet, ViNO sits on top of the OpenStack platform and it is particularly focused on migrating VM containers across heterogeneous platforms with minimal downtime, especially meaningful for data center networks. For simulating load in servers they use JMeter [Foundation 1999] in their tests.

EstiNet [Wang et al. 2013] combines the advantages of both simulation and emulation. In a network simulated by EstiNet, each simulated host can run the real Linux operating system, and any real application program can run on a simulated host without any modification. The advantage of the EstiNet's approach over emulators such as Mininet, is that the controllers can correctly control the switches based on the simulation clock, which can be faster or slower than the real time. The authors have used EstiNet to perform functional validation and performance evaluation of several NOX/POX components and protocols such as the Learning Bridge and the Spanning Tree Protocols.

fs-sdn [Gupta et al. 2013] is a simulator based on the *fs* [Sommers et al. 2011] simulation platform that was developed for realistic test and validation of standard networks. *fs* is a Python-based tool that uses discrete-event simulation techniques for synthesizing the network measurements and the measurements it produces are accurate down to the timescale of one second. *fs-sdn* extends *fs* by incorporating the POX controller for prototyping and evaluating SDN-based applications.

OMNeT++ [OpenSim 2001; Varga and Hornig 2008] is a C++ based discrete event simulator for modeling communication networks, multiprocessors and other distributed or parallel systems. To simulate SDN environments in OMNeT++, the OF components are integrated using the INET Framework [Klein and Jarschel 2013], where an OF switch and a basic controller are available, as well as OF messages, such as: `PACKET_IN`, `PACKET_OUT` or `FLOW_MOD`. OMNeT++ implements two modular OF nodes: an OF switch, which highlights the separation of the data and the control

Table VI. Comparison table of the different SDN simulators/emulators

Tools \ Properties	Description	Type	Scalability	Interface
Mininet	Open vSwitch-based SDN network emulator	Emulator	∼100 nodes	OpenFlow 1.0, 1.2, 1.3
MaxiNet	Mininet-based large-scale SDN network emulator	Emulator	∼3200 nodes	OpenFlow 1.0, 1.2, 1.3
DOT	Open vSwitch-based large-scale SDN network emulator	Emulator	Not specified	OpenFlow 1.0, 1.3
OFNet	SDN emulator including traffic generator	Emulator	Not specified	OpenFlow (version not specified)
ViNO	Orchestrator of virtual networks focused on VM migration	Emulator	Not specified	Not specified
EstiNet	OpenFlow emulator and simulator	Emulator/Simulator	Thousands nodes	OpenFlow 1.0, 1.3.4
fs-sdn	Extension of the *fs* simulator	Simulator	∼100 nodes	POX NBI
OMNeT++	Discrete event simulator with OpenFlow support	Simulator	Unlimited	OpenFlow 1.0
ns-3	Discrete event simulator with OpenFlow support	Simulator	Unlimited	OpenFlow 0.8.9 (1.3 with third-party plugins)

plane, and an OF controller, which provides public methods to send OF messages to the connected OF switch, while the actual controller behavior is implemented in a separate module.

ns-3 [ns 3 2011] is a C++ based discrete event simulator. ns-3 simulations can use OF switches, which are configurable via OF API and designed to express basic use of the OF protocol by maintaining virtual flow tables and TCAM memories to provide OF-like results. ns-3 implements its own OF controller which simulates the behavior of a real controller. External modules can be used to extend ns-3, such as OFSwitch13 [Chaves 2017] which brings compatibility with OF 1.3.

7.1. Summary of the different simulators and emulators

Table VI summarizes and compares the simulators and emulators, based on:

— **Description:** Concise description of the tool.
— **Type:** This parameter classifies the tools in two types, simulators and emulators.
— **Scalability:** Scalability of the tools in terms of number of nodes simulated/emulated.
— **Interface:** The interface between the tool and the SDN controller. In case of *OMNeT++* and *ns-3*, it refers to interfaces between internal OF-enabled nodes simulating controller and switches.

7.2. Concluding remarks

Several emulators have appeared to ease experimentation in the OpenFlow domain. Most of them support OpenFlow from version 1.0 to version 1.3.4, hence working with most of the SDN controllers available. If we exclude *fs-sdn*, which incorporates the POX controller to enable OpenFlow 1.0 experiments, the other simulators do not interface with regular SDN controllers. They instead simulate the SDN controller's behavior with specific internal modules (or nodes).

8. ARE WE READY TO DRIVE SOFTWARE DEFINED NETWORKS?

Probably the answer is *not yet*. But this should be considered from an optimistic point of view, as a motivation to foster research in this area. In this chapter, we briefly summarize the main directions

to be investigated toward the next generation SDN management framework.

8.1. Immature interfaces

Along the article, we have introduced the concept of SDN tool and discussed its role in the management of SDN-enabled networks. Together with the classification of the tools, we have analyzed the interfaces they use to interact with the different planes of the SDN architecture. Specifically, we have found that the OpenFlow protocol and Controller plane NBIs are the most used interfaces. However, while the latter approach limits the applicability of the tool to a specific SDN controller platform (NBIs are not compatible with each others across different platforms), using a SBI such as OpenFlow as a management interface implies forcing the whole SDN environment (data plane included) to stick to a specific control protocol. This is particularly questionable when protocols such as OpenFlow will presumably be substituted in the future by new ones, like P4.

Many other approaches overcome the limitations of existing interfaces by defining custom protocols and data models, although not generic enough to cover all tools, which proves the lack of consensus.

Tool design criteria is diverse because the management-control functions are defined quite generally by the ONF. At the same time, the NBI and SBI are still under evolution. As a result, developers do not have a clear picture of where to deploy their tools and, furthermore, these developments are prone to be deprecated as these interfaces (and their associated protocols) are immature or limited (e.g. OpenFlow still lacks many desirable features). Aside from deciding whether using one or more interfaces, research on the evolution of these interfaces and their requirements is imperative for effective and long-lasting management designs.

8.2. The SDN toolbox

Currently, a wide range of SDN tools exist (and we envision more are yet to come). However, they are being implemented following miscellaneous ideas and requirements, thus associated to specific architecture models or particular SDN controller platforms. Therefore, most of them can be hardly adopted in a production environment where the maintenance of the network is a critical task, and where patching the components of the SDN network to merge the different approaches is usually not a viable option.

As the reason behind using specific SDN platforms is the interfaces immaturity mentioned previously, a first step for their evolution could be creating an SDN toolbox to model them. The motivation is to leave SDN platforms as black boxes (free of implementation or standardization) that might (or might not) accomplish the definition of these evolved interfaces for management. Similarly, new tools developed might (or might not) follow this design, but at least if they do, they will be independent and not anchored to a single SDN framework.

After the analysis completed in this survey, we have elaborated a list of tools and minimum requirements that we consider fundamental for a generalized SDN toolbox, shown in Table VII. Each parameter is explained in the following:

— **Features:** Set of the most common features associated with the specific tool. They could be considered the minimum properties a tool of that type should accomplish.
— **Interfaces:** Frequently used interfaces leveraged to develop these tools. For the interfaces, we are pointing at the planes currently defined by the ONF, following the SDN architecture. For example, an interface in the Data plane would have access to the information related to network devices, such as flow tables or packets.
— **Data models:** Communication via the previous SDN interfaces implicitly requires the definition of the data models that will be exchanged through them. These models aim to represent information in a standardized way, so that not only the communication is feasible between SDN tools and platforms, but also among tools from different developers, for example. For instance, OpenFlow defines a model to represent a packet, but not for network device resources.

Table VII. Generalized list of SDN tools: Features and Requirements

Tools \ Properties	Features	Interfaces	Data models	Remaining questions
Composition	Network slicing Conflict resolution	Control plane	Common functionality primitives	How to move toward automatic composition?
Debugging	Behavior verification Checking forwarding App debugging and fine-tuning	Application, Control and Data planes	Packet model Flow table model	How to define a process/flow for SDN debugging?
Resource management	Scalability Energy consumption	Control and Data planes	Resource definition	How to define thresholds for underused/overused resources?
Profiling	Optimization of network applications	Control and Data planes	Statistics model App performance model	How to measure the network without affecting it?
Simulation	Behavior verification App debugging and fine-tuning	Application and Control planes	Network app model SBI protocol model	How to be as close to real networks as possible?

— **Remaining questions:** Apart from the interfaces and associated data models to be defined, some questions are still unanswered and require active investigation.

8.3. Toward the next generation SDN management framework

We envision the following challenges and future research directions for the standardization of the management functions, to drive and help SDN environments thrive:

(1) ***Create a generalized list of SDN tools:*** The reason for this list is to establish a starting point to work on, as currently the different developments are commingled. We found out that currently there is not even a clear distinction among types of tools (e.g. some tools with same functionality are named differently). Therefore, setting up an initial list would help classifying the requirements and splitting design tasks. Although this survey already introduces an initial list, it requires further discussion across different standardization groups.

(2) ***Cooperation among different open source SDN communities:*** If the most popular open source SDN frameworks reached an agreement for interfaces and data models, the evolution toward a standardized management framework would follow easily. To achieve this, the most popular SDN frameworks (at the time of writing this article, ODL and ONOS) should proactively work on some common criteria. However, currently they are evolving based on external (independent) petitions.
For example, ONOS is currently evolving the platform based on work brigades, in which any community member can participate. If somebody wants to integrate an SDN tool, could do it by proposing a brigade. But the decisions at this brigade might not be the same if the same person tries to integrate it in ODL, as their communities are isolated. In fact, this dilemma goes beyond, as ideally both SDN frameworks could merge if their communities would proactively communicate, which might occur in the near future.

(3) ***Coordination with production environments and telcos:*** Currently, companies applying SDN in their networks are usually customizing SDN frameworks on their own. An effort should be made in order to list common requirements for SDN management, so that different SDN communities could take the token and work together on it.
The problem behind is that usually telcos are unwilling to reveal their products and deployments, and particularly in this case where SDN is an emergent technology. Furthermore, many companies still prefer to build (and sell afterward) their own close and opaque solutions, which burdens the evolution toward a standardized management framework.

9. CONCLUSIONS

In this paper, we provided an overview of the management and operational tool that facilitates the development, deployment and/or maintenance of SDN-based networks. We started by classifying the tools based on their features and objectives. Then we presented the current SDN architecture and

interfaces, and how we envision the role of the tools in it. Afterward, we provided a short description for each tool, a comparison and a conclusion for each category. Finally, we discussed issues, challenges, and future research directions regarding management in SDN. The joint conclusion is that management functions in SDN are still set aside as a secondary requirement and, therefore, they need further standardization efforts in the forthcoming years; particularly the *SDN tool* concept should be developed. In the meantime, we expect that this comprehensive survey on management could guide different stakeholders to understand and evolve the future of SDN management.

ACKNOWLEDGMENTS

This work is partially supported by the following projects: EC FP7 NetIDE [NetIDE 2016] (G.A. 619543), EC H2020 SUPERFLUIDITY (G.A. 671566) and Spanish DRONEXT (G.A. TEC2014-58964-C2-1-R).

REFERENCES

Dennis Abts, Michael R. Marty, Philip M. Wells, Peter Klausler, and Hong Liu. 2010. Energy Proportional Datacenter Networks. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*.

Kanak Agarwal, Eric Rozner, Colin Dixon, and John Carter. 2014. SDN Traceroute: Tracing SDN Forwarding Without Changing Network Behavior. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*.

S. B. Akers. 1978. Binary Decision Diagrams. *IEEE Trans. Comput.* 27, 6 (1978), 509–516.

Ali Al-Shabibi, Marc De Leenheer, Matteo Gerola, Ayaka Koshibe, Guru Parulkar, Elio Salvadori, and Bill Snow. 2014. OpenVirteX: Make your virtual SDNs programmable. In *Proceedings of the third workshop on Hot topics in software defined networking*.

Ehab Al-Shaer and Saeed Al-Haj. 2010. FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*.

Amer Aljaedi and C. Edward Chow. 2016. Pathseer: a centralized tracer of packet trajectories in software-defined datacenter networks. In *2016 Principles, Systems and Applications of IP Telecommunications (IPTComm)*. 1–9.

Rodolfo Alvizu, Guido Maier, Navin Kukreja, Achille Pattavina, Roberto Morro, Alessandro Capello, and Carlo Cavazzoni. 2017. Comprehensive survey on T-SDN: Software-defined Networking for Transport Networks. *IEEE Communications Surveys Tutorials* PP, 99 (2017), 1–1.

Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

Vitaly Antonenko and Ruslan Smelyanskiy. 2013. Global Network Modelling Based on Mininet Approach. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*.

M. Aslan and A. Matrawy. 2016. On the Impact of Network State Collection on the Performance of SDN Applications. *IEEE Communications Letters* (2016).

Alvin AuYoung, Yadi Ma, Sujata Banerjee, Jeongkeun Lee, Puneet Sharma, Yoshio Turner, Chen Liang, and Jeffrey C. Mogul. 2014. Democratic Resolution of Resource Conflicts Between SDN Control Programs. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*.

Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. 2014. VeriCon: Towards verifying controller programs in software-defined networks. In *ACM SIGPLAN Notices*.

S. Banerjee and K. Kannan. 2014. Tag-In-Tag: Efficient flow table management in SDN switches. In *Network and Service Management (CNSM), 10th International Conference on*.

Ryan Beckett, Xuan Kelvin Zou, Shuyuan Zhang, Sharad Malik, Jennifer Rexford, and David Walker. 2014. An Assertion Language for Debugging SDN Applications. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*.

S. Bemby, Hongbin Lu, K.H. Zadeh, H. Bannazadeh, and A. Leon-Garcia. 2015. ViNO: SDN overlay to allow seamless migration across heterogeneous infrastructure. In *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*.

Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*.

Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. 2014. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*.

M. Bjorklund. 2010. YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF). RFC 6020 (Proposed Standard). (Oct. 2010). http://www.ietf.org/rfc/rfc6020.txt

Andreas Blenk, Arsany Basta, Martin Reisslein, and Wolfgang Kellerer. 2015. Survey on Network Virtualization Hypervisors for Software Defined Networking. *CoRR* (2015). http://arxiv.org/abs/1506.07275

Avrim Blum. 2001. Graphplan. (2001). http://www.cs.cmu.edu/~avrim/graphplan.html

Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95. http://doi.acm.org/10.1145/2656877.2656890

Patrice Brmond-Grgoire, Insup Lee, and Richard Gerber. 1993. ACSR: An algebra of communicating shared resources with dense time and priorities. In *CONCUR'93*. Lecture Notes in Computer Science, Vol. 715. Springer Berlin Heidelberg, 417–431.

K. Bu, X. Wen, B. Yang, Y. Chen, L. E. Li, and X. Chen. 2016. Is every flow on the right track?: Inspect SDN forwarding with RuleScope. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. 1–9.

Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. 2001. *Why and Where: A Characterization of Data Provenance*. 316–330.

Marco Canini, Daniele Venzano, Peter Peresini, Dejan Kostic, Jennifer Rexford, and others. 2012. A NICE Way to Test OpenFlow Applications. In *NSDI*.

Luciano Jerez Chaves. 2017. OpenFlow 1.3 module for ns-3. (2017). http://www.lrc.ic.unicamp.br/ofswitch13/ofswitch13.pdf

Shubhajit Roy Chowdhury, M Faizul Bari, Rizwan Ahmed, and Raouf Boutaba. 2014. Payless: A low cost network monitoring framework for software defined networks. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*.

Haim Cohen. 2004. The BuDDy Library & Boolean Expressions. (2004). http://www.drdobbs.com/the-buddy-library-boolean-expressions/184401847

Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. 2011. DevoFlow: Scaling Flow Management for High-performance Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*.

A. Darabseh, M. Al-Ayyoub, Y. Jararweh, E. Benkhelifa, M. Vouk, and A. Rindos. 2015. SDDC: A Software Defined Datacenter Experimental Framework. In *Future Internet of Things and Cloud (FiCloud), 3rd International Conference on*.

Tooska Dargahi, Alberto Caponi, Moreno Ambrosin, Giuseppe Bianchi, and Mauri Conti. 2017. A Survey on the Security of Stateful SDN Data Planes. *IEEE Communications Surveys Tutorials* PP, 99 (2017), 1–1.

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. http://doi.acm.org/10.1145/1327452.1327492

Abhishek Dixit, Kirill Kogan, and Patrick Eugster. 2014. Composing heterogeneous SDN controllers with Flowbricks. In *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on*.

R. Doriguzzi-Corin, P. A. A. Gutierrez, E. Rojas, H. Karl, and E. Salvadori. 2016a. Reusability of software-defined networking applications: A runtime, multi-controller approach. In *2016 12th International Conference on Network and Service Management (CNSM)*. 209–215.

Roberto Doriguzzi-Corin, Domenico Siracusa, Elio Salvadori, and Arne Schwabe. 2016b. Empowering Network Operating Systems with Memory Management Techniques. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium*.

Ramakrishnan Durairajan, Joel Sommers, and Paul Barford. 2014. Controller-agnostic SDN Debugging. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*.

Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, and Martin Vechev. 2016. SDNRacer: Concurrency Analysis for Software-defined Networks. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 402–415.

Seyed K Fayaz and Vyas Sekar. 2014. Testing stateful and dynamic data planes with FlowTest. In *Proceedings of the third workshop on Hot topics in software defined networking*.

Seyed K. Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. 2016. BUZZ: Testing Context-Dependent Policies in Stateful Networks. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 275–289.

Eder Leao Fernandes and Christian Esteve Rothenberg. 2014. OpenFlow 1.3 Software Switch. *Salao de Ferramentas do XXXII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos SBRC* (2014), 1021–1028.

Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. Dissertation.

Roy T. Fielding and Richard N. Taylor. 2000. Principled Design of the Modern Web Architecture. In *Proceedings of the 22Nd International Conference on Software Engineering*.

Floodlight. 2016a. Floodlight OpenFlow Controller. (2016). https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller

Floodlight. 2016b. Indigo Virtual Switch. (2016). https://floodlight.atlassian.net/wiki/display/indigodocs/Indigo+Virtual+Switch+Documentation

Floodlight. 2016c. OFTest. (2016). https://floodlight.atlassian.net/wiki/spaces/OFTest

ForwardNetworks. 2016. Forward Networks. (2016). https://www.forwardnetworks.com

Nate Foster, Michael J. Freedman, Rob Harrison, Jennifer Rexford, Matthew L. Meola, and David Walker. 2010. Frenetic: A High-level Language for OpenFlow Networks. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*.

Apache Software Foundation. 1999. Apache JMeter. (1999). http://jmeter.apache.org/

Apache Software Foundation. 2014. Hadoop: Yarn Scheduler Load Simulator (SLS). (2014). https://hadoop.apache.org/docs/r2.4.1/hadoop-sls/SchedulerLoadSimulator.html

Free Software Foundation. 1986. GDB: The GNU Project Debugger. (1986). https://www.gnu.org/software/gdb/

Linux Foundation. 2016. OvS: Open vSwitch. (2016). http://openvswitch.org/

G. Gheorghe, T. Avanesov, M.-R. Palattella, T. Engel, and C. Popoviciu. 2015. SDN-RADAR: Network troubleshooting combining user experience and SDN capabilities. In *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*.

GitHub. 2011. POX Controller. (2011). https://github.com/noxrepo/pox

GitHub. 2012. Ripcord-Lite for POX: A simple network controller for OpenFlow-based data centers. (2012). https://github.com/brandonheller/riplpox

GitHub. 2016. NetIDE Profiler (GitHub). (2016). https://github.com/fp7-netide/Tools/tree/master/profiler

Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. 2016. Evolve or Die: High-Availability Design Principles Drawn from Google's Network Infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. 58–72.

Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. 2008. NOX: Towards an Operating System for Networks. *SIGCOMM Comput. Commun. Rev.* 38, 3 (2008), 105–110.

Arpit Gupta, Rüdiger Birkner, Marco Canini, Nick Feamster, Chris Mac-Stoker, and Walter Willinger. 2016. Network Monitoring As a Streaming Analytics Problem. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets '16)*. 106–112. http://doi.acm.org/10.1145/3005745.3005748

Mukta Gupta, Joel Sommers, and Paul Barford. 2013. Fast, Accurate Simulation for SDN Prototyping. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*.

Akram Hakiri, Aniruddha Gokhale, Pascal Berthou, Douglas C. Schmidt, and Thierry Gayraud. 2014. Software-Defined Networking: Challenges and research opportunities for Future Internet. *Computer Networks* 75 (2014), 453 – 471.

Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. 2012a. Reproducible Network Experiments Using Container-based Emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*.

Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Maziéres, and Nick McKeown. 2012b. Where is the Debugger for My Software-defined Network?. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*.

Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. 2014. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*.

Pedro Heleno Isolani, Juliano Araujo Wickboldt, Cristiano Bonato Both, Juergen Rochol, and Lisandro Zambenedetti Granville. 2015. Interactive monitoring, visualization, and configuration of OpenFlow-based SDN. In *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*.

G. Holzmann. 2005. The SPIN Model Checker: Primer and Reference Manual. *Addison-Wesley* (2005).

Alex Horn, Ali Kheradmand, and Mukul Prasad. 2017. Delta-net: Real-time Network Verification Using Atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 735–749.

F. Hu, Q. Hao, and K. Bao. 2014b. A Survey on Software-Defined Network and OpenFlow: From Concept to Implementation. *IEEE Communications Surveys Tutorials* 16, 4 (2014), 2181–2206.

Hongxin Hu, Wonkyu Han, Gail-Joon Ahn, and Ziming Zhao. 2014a. FLOWGUARD: Building robust firewalls for software-defined networks. In *Proceedings of the third workshop on Hot topics in software defined networking*.

Shufeng Huang and J. Griffioen. 2013. Network Hypervisors: Managing the Emerging SDN Chaos. In *Computer Communications and Networks (ICCCN), 2013 22nd International Conference on*.

T. Huang, F. R. Yu, C. Zhang, J. Liu, J. Zhang, and Y. Liu. 2017. A Survey on Large-Scale Software Defined Networking (SDN) Testbeds: Approaches and Challenges. *IEEE Communications Surveys Tutorials* 19, 2 (2017), 891–917.

IETF. 2013. VeriSDN: Formal verification for software defined networking (SDN). (2013). https://www.ietf.org/proceedings/87/slides/slides-87-sdnrg-6.pdf

Internet2. 2016. Internet2 Home Page. (2016). http://www.internet2.edu

Y. Jarraya, T. Madi, and M. Debbabi. 2014. A Survey and a Layered Taxonomy of Software-Defined Networking. *IEEE Communications Surveys Tutorials* 16, 4 (2014), 1955–1980.

Michael Jarschel, Frank Lehrieder, Zsolt Magyari, and Rastin Pries. 2012. A flexible OpenFlow-controller benchmark. In *Software Defined Networking (EWSDN), 2012 European Workshop on*.

M. Jarschel, T. Zinner, T. Hossfeld, P. Tran-Gia, and W. Kellerer. 2014. Interfaces, attributes, and use cases: A compass for SDN. *Communications Magazine, IEEE* 52, 6 (2014), 210–217.

J.D. Case et al. 1990. Simple Network Management Protocol (SNMP). RFC 1157 (Historic). (May 1990). http://www.ietf.org/rfc/rfc1157.txt

Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. 2015. CoVisor: A Compositional Hypervisor for Software-Defined Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*.

H. Kang, S. Lee, C. Lee, C. Yoon, and S. Shin. 2015. SPIRIT: A Framework for Profiling SDN. In *2015 IEEE 23rd International Conference on Network Protocols (ICNP)*. 417–424. DOI: http://dx.doi.org/10.1109/ICNP.2015.49

Murat Karakus and Arjan Durresi. 2017. A survey: Control plane scalability issues and approaches in Software-Defined Networking (SDN). *Computer Networks* 112 (2017), 279 – 293.

Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. 2014. Infinite CacheFlow in Software-defined Networks. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*.

Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. 2013. Real Time Network Policy Checking Using Header Space Analysis. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*.

Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*.

Suleman Khan, Abdullah Gani, Ainuddin Wahid Abdul Wahab, Mohsen Guizani, and Muhammad Khurram Khan. 2017. Topology Discovery in Software Defined Networks: Threats, Taxonomy, and State-of-the-Art. *IEEE Communications Surveys Tutorials* 19, 1 (2017), 303–324.

Suleman Khan, Abdullah Gani, Ainuddin Wahid Abdul Wahab, Ahmed Abdelaziz, Kwangman Ko, Muhammad Khurram Khan, and Mohsen Guizani. 2016. Software-Defined Network Forensics: Motivation, Potential Locations, Requirements, and Challenges. *IEEE Network* 30, 6 (2016), 6–13.

Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2012. VeriFlow: Verifying Network-wide Invariants in Real Time. *SIGCOMM Comput. Commun. Rev.* 42, 4 (2012), 467–472.

Hyojoon Kim and N. Feamster. 2013. Improving network management with software defined networking. *Communications Magazine, IEEE* 51, 2 (2013), 114–119.

Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. 2015. Kinetic: Verifiable Dynamic Network Control. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*.

Dominik Klein and Michael Jarschel. 2013. An OpenFlow Extension for the OMNeT++ INET Framework. In *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*.

D. Kreutz, F. M. V. Ramos, P. E. Verssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. 2015. Software-Defined Networking: A Comprehensive Survey. *Proc. IEEE* 103, 1 (2015), 14–76.

S. Kuklinski. 2014. Programmable management framework for evolved SDN. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*.

Maciej Kuzniar, Peter Peresini, Marco Canini, Daniele Venzano, and Dejan Kostic. 2012. A SOFT way for OpenFlow switch interoperability testing. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*.

M. Kwiatkowska, G. Norman, and D. Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *23rd International Conference on Computer Aided Verification*.

Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (Hotnets-IX)*.

Bob Lantz and Brian O'Connor. 2015. A Mininet-based Virtual Testbed for Distributed SDN Development. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*.

David Lebrun, Stefano Vissicchio, and Olivier Bonaventure. 2014. Towards Test-Driven Software Defined Networking. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*.

M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. 2014. *Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks*. RFC 7348. RFC Editor. http://www.rfc-editor.org/rfc/rfc7348.txt http://www.rfc-editor.org/rfc/rfc7348.txt.

Rupak Majumdar, Sai Deep Tetali, and Zilong Wang. 2014. Kuai: A model checker for software-defined networks. In *2014 Formal Methods in Computer-Aided Design (FMCAD)*. 163–170.

Antonio Marsico, Roberto Doriguzzi-Corin, and Domenico Siracusa. 2017. An Effective Swapping Mechanism to Overcome the Memory Limitation of SDN Devices. In *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management*.

Rahim Masoudi and Ali Ghaffari. 2016. Software Defined Networks: A survey. *Journal of Network and Computer Applications* 67 (2016), 1 – 25.

N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review* 32, 2 (April 2008), 69–74.

J. Medved, R. Varga, A. Tkacik, and K. Gray. 2014. OpenDaylight: Towards a Model-Driven SDN Controller architecture. In *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*. 1–6.

A. Mendiola, J. Astorga, E. Jacob, and M. Higuero. 2017. A Survey on the Contributions of Software-Defined Networking to Traffic Engineering. *IEEE Communications Surveys Tutorials* 19, 2 (2017), 918–953.

Oliver Michel and Eric Keller. 2017. SDN in wide-area networks: A survey. In *2017 Fourth International Conference on Software Defined Systems (SDS)*. 37–42.

Jeffrey C. Mogul, Alvin AuYoung, Sujata Banerjee, Lucian Popa, Jeongkeun Lee, Jayaram Mudigonda, Puneet Sharma, and Yoshio Turner. 2013. Corybantic: Towards the Modular Composition of SDN Control Programs. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*.

Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. 2013. Composing Software Defined Networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*.

G. N. Nde and R. Khondoker. 2016. SDN testing and debugging tools: A survey. In *2016 5th International Conference on Informatics, Electronics and Vision (ICIEV)*. 631–635.

NetIDE. 2016. An integrated development environment for portable network applications. (2016). http://www.netide.eu/

ns 3. 2011. ns-3 simulator. (2011). https://www.nsnam.org/

B. A. A. Nunes, M. Mendonca, X. N. Nguyen, K. Obraczka, and T. Turletti. 2014. A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks. *IEEE Communications Surveys Tutorials* 16, 3 (2014), 1617–1634.

Yustus Eko Oktian, SangGon Lee, HoonJae Lee, and JunHuy Lam. 2017. Distributed SDN controller system: A survey on design choice. *Computer Networks* 121 (2017), 100 – 111.

ONF. 2013. OpenFlow Switch Specification 1.4.0. (2013). https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf

ONF. 2014. OpenFlow Management and Configuration Protocol 1.2. (2014). https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow-config/of-config-1.2.pdf

ONF. 2016. SDN architecture - Issue 1.1. (2016). https://www.opennetworking.org/wp-content/uploads/2014/10/TR-521_SDN_Architecture_issue_1.1.pdf

ON.Lab. 2016a. ONOS CLI Commands. (2016). https://wiki.onosproject.org/display/ONOS/The+ONOS+CLI

ON.Lab. 2016b. ONOS Composition Mode. (2016). https://wiki.onosproject.org/display/ONOS/Composition+Mode

ON.Lab. 2016c. ONOS Java API. (2016). http://api.onosproject.org/

ON.Lab. 2016d. ONOS REST API. (2016). https://wiki.onosproject.org/display/ONOS/REST

OpenSim. 2001. OMNeT++ Home Page. (2001). https://omnetpp.org/

Peter Peresini, Maciej Kuzniar, and Dejan Kostic. 2015. Monocle: Dynamic, fine-grained data plane monitoring. In *Proceedings of the 11th International on Conference on emerging Networking EXperiments and Technologies*. ACM.

Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. 2015. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*.

R. Enns et al. 2011. Network Configuration Protocol (NETCONF). RFC 6241 (Proposed Standard). (June 2011). http://www.ietf.org/rfc/rfc6241.txt

RabbitMQ. 2007. RabbitMQ official website. (2007). https://www.rabbitmq.com/

D. B. Rawat and S. R. Reddy. 2017. Software Defined Networking Architecture, Security and Energy Efficiency: A Survey. *IEEE Communications Surveys Tutorials* 19, 1 (2017), 325–346.

Charalampos Rotsos, Nadi Sarrar, Steve Uhlig, Rob Sherwood, and Andrew W Moore. 2012. OFLOPS: An open framework for OpenFlow switch evaluation. In *Passive and Active Measurement*.

Margaret Rouse. 2016. Debugging definition. (2016). http://searchsoftwarequality.techtarget.com/definition/debugging

A. R. Roy, M. F. Bari, M. F. Zhani, R. Ahmed, and R. Boutaba. 2014. Design and management of DOT: A Distributed OpenFlow Testbed. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*.

Ryu. 2012a. OpenFlow Switch Test Tool. (2012). https://osrg.github.io/ryu-book/en/html/switch_test_tool.html

Ryu. 2012b. Ryu SDN framework. (2012). http://osrg.github.com/ryu/

Arne Schwabe, Pedro A. Aranda Gutiérrez, and Holger Karl. 2016. Composition of SDN Applications: Options/Challenges for Real Implementations. In *Proceedings of the 2016 Applied Networking Research Workshop (ANRW '16)*. 26–31. DOI: http://dx.doi.org/10.1145/2959424.2959436

Colin Scott, Andreas Wundsam, Sam Whitlock, Andrew Or, Eugene Huang, Kyriakos Zarifis, and Scott Shenker. 2013. *How Did We Get Into This Mess? Isolating Fault-Inducing Inputs to SDN Control Software*. Technical Report UCB/EECS-2013-8. EECS Department, University of California, Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-8.html

Divjyot Sethi, Srinivas Narayana, and Sharad Malik. 2013. Abstractions for model checking SDN controllers. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*.

Ganesh H. Shankar. 2016. OFNet. (2016). http://sdninsights.org/

Rob Sherwood, Michael Chan, Adam Covington, Glen Gibb, Mario Flajslik, Nikhil Handigol, Te-Yuan Huang, Peyman Kazemian, Masayoshi Kobayashi, Jad Naous, and others. 2010. Carving research slices out of your production networks with OpenFlow. *ACM SIGCOMM Computer Communication Review* 40, 1 (2010), 129–130.

Rob Sherwood and KK Yap. 2011. Cbench controller benchmarker. *Last accessed, Nov* (2011).

Seungwon Shin, Phillip A Porras, Vinod Yegneswaran, Martin W Fong, Guofei Gu, and Mabry Tyson. 2013. FRESCO: Modular composable security services for software-defined networks. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*.

Sanjeev Singh and Rakesh Kumar Jha. 2017. A Survey on Software Defined Networking: Architecture for Next Generation Network. *Journal of Network and Systems Management* 25, 2 (2017), 321–374.

Richard Skowyra, Andrei Lapets, Azer Bestavros, and Assaf Kfoury. 2014. A Verification Platform for SDN-Enabled Applications. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*.

J. Sommers, R. Bowden, B. Eriksson, P. Barford, M. Roughan, and N. Duffield. 2011. Efficient network-wide flow record generation. In *INFOCOM, 2011 Proceedings IEEE*.

Sejun Song, Sungmin Hong, Xinjie Guan, Baek-Young Choi, and Changho Choi. 2013. NEOD: Network Embedded On-line Disaster management framework for Software Defined Networking. In *IM*.

Peng Sun, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, Ming Zhang, and Ahsan Arefin. 2014. A Network-state Management Service. In *Proceedings of the 2014 ACM Conference on SIGCOMM*.

Veriflow Systems. 2016. Veriflow. (2016). http://www.veriflow.net/

J. Teixeira, G. Antichi, D. Adami, A. Del Chiaro, S. Giordano, and A. Santos. 2013. Datacenter in a Box: Test Your SDN Cloud-Datacenter Controller at Home. In *Software Defined Networks (EWSDN), 2013 Second European Workshop on*.

Fan-Hsun Tseng, Kai-Di Chang, Shang-Chuan Liao, Han-Chieh Chao, and Victor C.M. Leung. 2017. sPing: a user-centred debugging mechanism for software defined networks. *IET Networks* 6 (March 2017), 39–46(7). Issue 2.

Mehmet Fatih Tuysuz, Zekiye Kubra Ankarali, and Didem Gzpek. 2017. A survey on energy efficiency in software defined networks. *Computer Networks* 113 (2017), 188 – 204.

N. L. M. van Adrichem, C. Doerr, and F. A. Kuipers. 2014. OpenNetMon: Network monitoring in OpenFlow Software-Defined Networks. In *2014 IEEE Network Operations and Management Symposium (NOMS)*. 1–8.

András Varga and Rudolf Hornig. 2008. An Overview of the OMNeT++ Simulation Environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*.

S. Vinoski. 2006. Advanced Message Queuing Protocol. *Internet Computing, IEEE* 10, 6 (2006), 87–89.

Anilkumar Vishnoi, Rishabh Poddar, Vijay Mann, and Suparna Bhattacharya. 2014. Effective Switch Memory Management in OpenFlow Networks. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*.

Ming-Hung Wang, Shao-You Wu, Li-Hsing Yen, and Chien-Chao Tseng. 2016b. PathMon: Path-specific traffic monitoring in OpenFlow-enabled networks. In *2016 Eighth International Conference on Ubiquitous and Future Networks (ICUFN)*. 775–780.

Shie-Yuan Wang, Chih-Liang Chou, and Chun-Ming Yang. 2013. EstiNet openflow network simulator and emulator. *Communications Magazine, IEEE* 51, 9 (September 2013), 110–117.

Wen Wang, Wenbo He, and Jinshu Su. 2016a. Redactor: Reconcile network control with declarative control programs in SDN. In *2016 IEEE 24th International Conference on Network Protocols (ICNP)*. 1–10.

Yangyang Wang, Jun Bi, and Keyao Zhang. 2016. A tool for tracing network data plane via SDN/OpenFlow. *Science China Information Sciences* 60, 2 (2016).

G. Watson, N. McKeown, and M. Casado. 2006. NetFPGA: A tool for network research and education. In *Workshop on Architecture Research using FPGA Platforms*.

P. Wette, M. Draxler, A. Schwabe, F. Wallaschek, M.H. Zahraee, and H. Karl. 2014. MaxiNet: Distributed emulation of software-defined networks. In *Networking Conference, 2014 IFIP*.

Philip Wette and Holger Karl. 2015. DCT2Gen: A traffic generator for data centers. *Computer Communications* (2015).

Philip Wette, Arne Schwabe, Malte Splietker, and Holger Karl. 2015. Extending Hadoop's Yarn Scheduler Load Simulator with a highly realistic network & traffic model. In *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*. 1–2.

J. Wickboldt, W. De Jesus, P. Isolani, C. Both, J. Rochol, and L. Granville. 2015. Software-defined networking: management requirements and challenges. *Communications Magazine, IEEE* 53, 1 (2015), 278–285.

Wireshark. 2016. Wireshark. (2016). http://wireshark.org

Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2017. Automated Bug Removal for Software-Defined Networks. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 719–733.

Andreas Wundsam, Dan Levin, Srini Seetharaman, and Anja Feldmann. 2011. OFRewind: Enabling Record and Replay Troubleshooting for Networks. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*.

W. Xia, Y. Wen, C. H. Foh, D. Niyato, and H. Xie. 2015. A Survey on Software-Defined Networking. *IEEE Communications Surveys Tutorials* 17, 1 (2015), 27–51.

Yufan Yang, Xinlin Huang, Shang Cheng, Shiyun Chen, and Peijin Cong. 2016. SHSA: A Method of Network Verification with Stateful Header Space Analysis. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*. 232–238.

Curtis Yu, Cristian Lumezanu, Yueping Zhang, Vishal Singh, Guofei Jiang, and Harsha V Madhyastha. 2013. Flowsense: Monitoring network utilization with zero measurement cost. In *Proceedings of Passive and Active Measurement Conference*.

Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Automatic Test Packet Generation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*.

Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. 2014. Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 87–99.

ZeroMQ. 2007. ZeroMQ official website. (2007). http://zeromq.org/

Hui Zhang, Cristian Lumezanu, Junghwan Rhee, Nipun Arora, Qiang Xu, and Guofei Jiang. 2014. Enabling Layer 2 Pathlet Tracing Through Context Encoding in Software-defined Networking. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*.

Peng Zhang, Hao Li, Chengchen Hu, Liujia Hu, Lei Xiong, Ruilong Wang, and Yuemei Zhang. 2016. Mind the Gap: Monitoring the Control-Data Plane Consistency in Software Defined Networks. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. ACM, 19–33.

Ying Zhang. 2013. An adaptive flow counting method for anomaly detection in SDN. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*.

Yusu Zhao, Pengfei Zhang, and Yaohui Jin. 2016. Netography: Troubleshoot your network with packet behavior in SDN. In *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*. 878–882.

Received ...; revised ...; accepted ...