GitFlow: Flow Revision Management for Software-Defined Networks*

Abhishek Dwaraki†, Srini Seetharaman‡, Sriram Natarajan⋆, and Tilman Wolf†
†Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA USA
‡Infinera Corporation, Sunnyvale CA USA
⋆Deutsche Telekom, Silicon Valley Innovation Center, Mountain View, CA USA

ABSTRACT

Our work addresses the problem of revision control for flow state management in SDN-enabled networks, so that the underlying data plane might be able to provide better state protection, provenance, ease of programmability, and support for multiple applications. Inspired by the revision control tools in the software development world, we propose an abstraction and a system called GitFlow, which provides flow state revisioning in the SDN context. The core idea of GitFlow is to run a repository server to maintain the authoritative copy of the flow configuration state and track additional meta data for each evolving snapshot of the flow state. When multiple applications make incremental commits to state, our system also automates conflict resolution by rebasing new flow state with committed flow state. We envision that our revision control abstraction will provide safety in the data plane and better programmability in the control plane.

General Terms

Design, Management

Categories and Subject Descriptors

C.2.6 [**Computer-Communication Networks**]: Internetworking—*Routers*; C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network Management*

1. INTRODUCTION

Software-Defined Networking (SDN) offers an opportunity to both the control and data planes of today's networks to become programmable and easily evolvable. The most popular way to realize SDN has been to adopt devices that support a standardized application programming interface (API), for example OpenFlow [7], for programming network behavior in the data plane through external software. This network programmability brings with it several challenges: migrating to new architectures, interoperating with legacy networks, coordinating control plane state in a centralized fashion, supporting different authors to the state, and restricting anomalous or malicious behaviors.

While the aspect of multiple co-existing authors editing flow state of a shared resource has been previously investigated [6, 8, 13, 17] from the perspective of conflict detection or resource isolation, the issue of *tracking* all changes to the flow state has not yet been examined sufficiently. In software development, this process of managing changes to shared data is referred to as *revision control*[1]. We see a need for similar revisioning and state change management rapidly manifesting in software-defined networks.

Traditionally, revision control in the networking world has been limited to managing CLI configurations of data plane devices. The state of the flows and the control plane, however, is inaccessible. In the context of an SDN-enabled network, we have access to much more state than was previously possible. Based on existing software revision control techniques, we propose a system called GitFlow that provides flow state revisioning in the SDN context. GitFlow underlies all flow-level state management that a switch undertakes, and provides safety at the data plane and better programming discipline in the control plane.

Additionally, our framework can be easily extended for several other purposes: tracking control plane evolution (similar to past works on AS path evolution [2]); providing network level accountability (similar in motivation to past works on AIP [1] or packet provenance [14]); preventing flow space conflicts across different authors (without being as restrictive as the isolation efforts of FlowVisor [16] or overlap prevention of Frenetic [8]); and extracting network state in a form conducive for automated troubleshooting (beyond methods made possible by header space analysis [5] and NDB [9]). This

*This work is based upon work supported by the National Science Foundation under Grant No. 1421448.

SOSR2015 June 17-18 2015, Santa Clara, CA, USA

ACM 978-1-4503-3451-8/15/06..$15.00
http://dx.doi.org/10.1145/2774993.2775064.

[1]The popular revision control software `git` provides author tracking, versioning and timestamping of changes, immutability of past alterations, automated conflict resolution, and annotations.

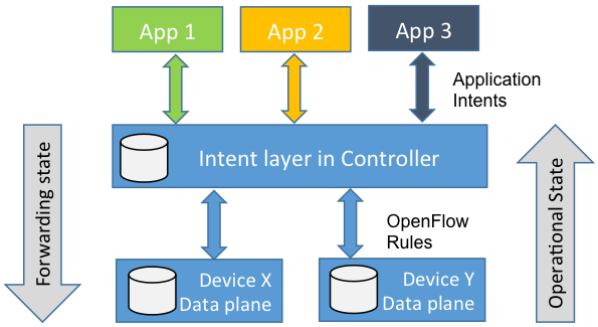

Figure 1: Programmability model in SDNs.

makes the GitFlow abstraction a vital part of any SDN programming workflow.

In this paper, we investigate flow rule revisioning, so that SDN-enabled networks can provide a higher level of provenance, security, ease of programmability and support for multiple applications. The remainder of this paper is organized as follows. Section 2 describes the model and design goals flow management in software-defined networks. In Section 3, we present our proposal inspired by the `git` version control system. In Section 4, we illustrate applications of our GitFlow approach to describe it better. Section 5 concludes the paper.

2. BACKGROUND

Software-Defined Networking allows external network applications to directly access the state of the data plane through well-defined APIs. The data plane device type (virtual or physical), flow table types (e.g., VLAN table, MAC table) and size, and supported feature set (match fields and supported actions) vary depending on the deployment scenario (e.g., edge-gateways [11], wide-area networks [15]). Typically, the packet processing functionality is achieved via a match-action pairs that span a multi-table pipeline. As the flow table evolves, through route updates, gratuitous ARP replies or new tenants addition, multiple tables and their related attributes (e.g., meters attached to each flows) may be updated or modified by different applications/authors. All these characteristics have implications on the network state management.

A typical model of SDN solution is shown in Figure 1. One or more applications, either directly through the API or through an abstracted intent layer, edit the configuration state in the data plane, e.g., the flow table. Correspondingly, the data plane exports to the external applications certain operational state, e.g., statistics, relevant for the programming logic. We adopt this layered, decoupled model as our SDN programming model.

Such decoupling of the control and data planes in the context of SDN can be challenging when it comes to managing network state. Thus, in this paper we only focus on the flow rule configuration state. Table 1 correlates each of SDN's programmability characteristics to a certain state management requirement. The safety and mutability requirements have been least investigated in the past, and we adopt it as the main focus of this paper. In the subsequent sections, we describe our state management system and illustrate its use through examples.

Table 1: State management requirements

Characteristics	State Requirements
Decoupled network control	Consistency
Dynamicity in application	Mutability
Co-existing applications/authors	Safety
Network troubleshooting	Provenance

3. GITFLOW

In this section, we discuss our proposal for state management to improve network programmability.

3.1 State Management Abstraction

The existence of multiple authors attempting to operate on a shared resource (network state) dictates the following requirements for concurrent access and accountability:

- *Author or application tracking*: The system should possess the ability to exactly identify and report the author that was responsible for the change of network state.
- *Versioning and change tracking*: Transformation and evolution of network state can be tracked, providing valuable information that can be used for planning and provenance reasons.
- *State safety*: Preventing the alteration of network state that has been committed by a different author brings in safety that ensures deterministic system behavior. It is, however, important to allow the same author to alter actions for previously established match rule.
- *Conflict resolution and rebasing*: The mutability requirement in statement management brings with it a need to resolve conflicts or overlaps in flow space to a feasible extent. Rebasing is the process of merging changes to an extent allowable by a pre-specified policy, based on the level of conflict.
- *Annotations*: The applications or authors can also include comments or other metadata to be associated with the change.

In the software development world, the above features are commonly available in a revision control software, such as `git` [3]. By making an analogy between source code and network state, we envision that adopting a layer of revision control for managing network state becomes vital to the programmability of the network.

3.2 GitFlow Architecture

We architect the abstraction described above in a manner similar to that used by `git` by adopting an "authoritative" GitFlow server that keeps up-to-date state that has been committed and programmed onto the flow tables. In reality, this server and the authoritative copy of the state it protects can be placed in two locations:

- Co-located on the data plane devices: each data plane device can host a separate GitFlow server and give it direct access to the flow table. With this approach, the state and its management are distributed. Each controller and its applications will act as GitFlow clients that conduct transactions with each of these servers. If an operator were to make flow state edits on the device, they will use a client to achieve that.
- Located external to the device: the GitFlow server is hosted external to the device, in a centralized location. All readers and writers to the flow state will use a GitFlow client to access it. The flow state in the server is always kept in-sync with the flow table in the data plane devices.

To avoid the overhead on switch CPU and to keep state management simpler to reconstruct on a network-wide scale, we prefer the latter approach of the GitFlow repository being external to the devices in a centralized server[2]. The implementation of this server becomes even simpler if one were to host it at a controller instance. For the rest of the paper, we work with the architecture choice of the GitFlow server being co-located with a controller instance.

In this approach, all readers (often switch) and writers (often controller), similar to `git`'s distributed approach, have their own local object database and staging indexes. Our architecture requires the writers to use a *flowmod monitor* module (running in the background) to monitor and analyze flow-modification messages received by the SDN/OpenFlow agent, as well as to version network state information that passes through it. The changes are then pushed to the GitFlow server. The pushed changes are serialized to disk regularly. Figure 3 provides an architectural overview of this process. When a change is made and committed to the set of past changes, we create a new "snapshot", which is a copy of the flow state that is versioned and uniquely identified.

When the system initializes, the server sets up the revision control framework by creating an empty master branch as part of the GitFlow server. Any data plane device that connects to the controller triggers the fork of a branch from the master's initial state. This new branch is then tagged with the datapath-id associated with the corresponding device. The branching behavior is illustrated in Figure 2. The device then clones this branch locally. The staging index and repository local to the device are used to store flow modifications occuring as a result of programming changes effected directly on it. The local changesets are, then, pushed to the GitFlow server on the controller to keep the network state in sync. It is foreseeable that there can be policies to regulate how frequently state changes are pushed upstreeam. An instance of the *flowmod monitor* running on the switch handles these operations.

The flow information and metadata that is important to track changes across versions is represented as a facile data table in Table 2.

[2]The centralized server can be sufficiently replicated and prevented from being a weak link in the SDN programming.

Table 2: Configuration state model

Type	Config data	Example
Timestamp	{time}	1425884076.342
Author	{app_uuid}	...1681e6b88ec1
Version	{change_uuid}	...a79d110c98fb
Table	{table_id}	5
Flow rule	{priority, match, action}	priority: 32768, match: tp_dst=22, action: drop
Timeout	{idle, hard}	idle: 10 hard: 0
Metadata	{OF config input}	vxlan_remote_ip= 1.1.1.5

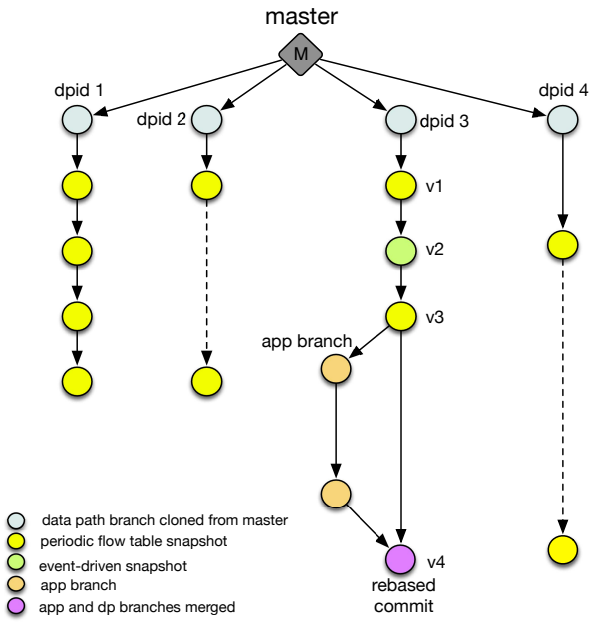

Figure 2: GitFlow Branching Model

Revision control can be performed at two levels of granularity: 'per-flow' and 'flow-table'. Based on the application and operational environment, it is understandable that an user might pick a different granularity to perform state management at. The controller can choose to make available a snapshot at either granularity across its other instances, or the other instances can choose to directly "pull" from the GitFlow repository. This ensures that all the instances are consistent with the underlying network state.

Although our paper and the Table 2 primarily deal with managing flow state, there are other network states worth preserving, especially the operational state pertaining to liveness of switch resources. For instance, there are OpenFlow flow configurations, such as failover group actions, that specify actions conditional on the liveness of ports (logical and physical). During troubleshooting or provenance inspection, such actions may be incorrectly interpreted if the operational state of the ports is not correlated with the configuration state. To address this need, we add an additional agent for track-

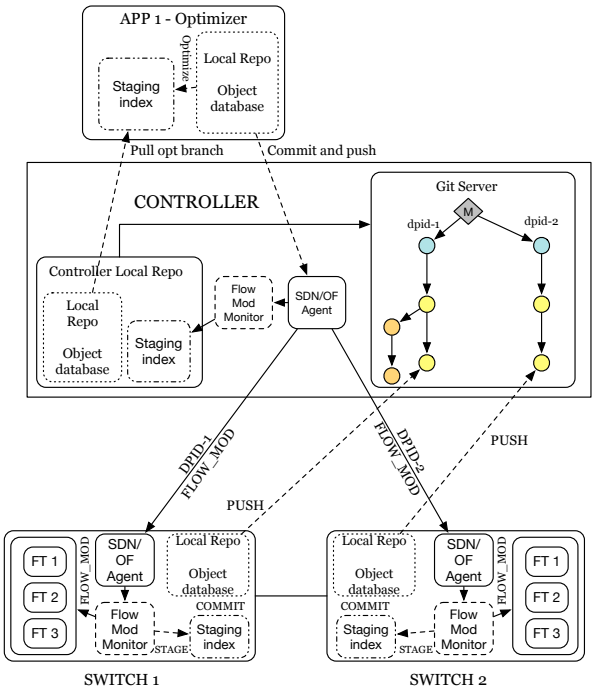

Figure 3: GitFlow Architecture

ing this operational state and archiving it to a centralized store alongside the GitFlow repository.

3.3 Conflict Resolution and Rebasing

During flow state programming, it is common to experience conflict with the flow space of an existing rule. In the past, SDN implementations based on OpenFlow relied on external mechanisms or implementations, such as FortNOX [13], FlowVisor [17] or that used in [10], to perform conflict resolution or provide flow space isolation. Recently, the OpenFlow specification [12] included an option to mark a flag called `OFPFF_CHECK_OVERLAP` on a `FLOW_MOD` operation to force the data plane to verify if the current `FLOW_MOD` overlaps with an existing rule on the match set; in the event of an overlap, the switch must refuse the addition and respond with an OpenFlow error message. This overlap check, however, is too restrictive and insufficient when frequent flow state programming from multiple authors (applications) becomes common.

Revision control and rebasing (locally replaying and merging changes), as used by `git`, can be an important tool for programmers. In the GitFlow architecture, we include the logic proposed in Algorithm 1 to attempt local conflict resolution to a feasible extent. The algorithm takes as input some merging policies, to specify what the action should be when GitFlow detects a conflict and is unable to automatically resolve. Our approach relies on building a set of flow-spaces, overlapping and non-overlapping, for the new flow modification and the existing rule that it potentially overlaps with. Non-overlapping spaces are accepted right away. The overlapping space is checked against the current rule a second time to determine the set relation between the two and the requisite action to be performed.

Algorithm 1 Conflict Resolution and Rebasing

```
1: procedure resolve_rebase(dpid, flow_mods)
2:     for all flow_mod ∈ flow_mods do
3:         S ← current snapshot of flow space
4:         compare flow_mod to S
5:         if disjoint(flow_mod,S) then
6:             accept flow_mod
7:         else
8:             compute overlap(flow_mod, S)
9:             X ← non-overlapping space
10:            Y ← overlapping space
11:            Z ⊂ S, match(Z) = match(Y)
12:            accept X
13:            if (Y, Z) is exact_overlap then
14:                action(Y) ← action(Z)
15:            else if Y ⊃ Z then
16:                apply superset merge policy
17:            else
18:                apply subset merge policy
19:            end if
20:        end if
21:        commit flow_mod
22:    end for
23: end procedure
```

The approach will need further extension to make the rebasing author-aware, i.e., only allowing merges based on the author privileges or based on whether changes are overwriting the same author's past commits. A `git`-like approach offers user authentication and privilege tracking to specify what source code can be modified by which user. We reserve all author-aware rebasing for future work.

4. GITFLOW USE CASES

To illustrate the value of GitFlow, we discuss the following scenarios where versioning the flow table can be useful for network operations and explain the practical value of our abstraction:

- Understanding the evolution of network state maintained in the flow table.
- Tracing and identifying security issues or misconfigurations in the network state.
- Adopting state extraction for network troubleshooting.

4.1 Tracking Flow Table Evolution

As discussed above, flow tables evolve over time through additions, deletions, and modifications. Current switch implementations and controller applications have limited capabilities to track these changes. Applications are required to periodically query the network state maintained in the switch and parse that information to comprehend the updates made to each flow. This approach is complex and might require state management in application logic. GitFlow can provide basic snapshotting and inspection features to efficiently track

the evolution of the flow table state. Here we discuss some simple versioning features to highlights its value:

- *gitflow commit*: The *commit* feature takes a snapshot of the changes made to the flow table from the previous committed state. When the controller programs flow modification messages (e.g, either a single or bundle of flows), GitFlow considers this as a new commit. When flows are programmed in the flow table, the committed changes are merged to the master copy and recorded as a version of the network state. A commit highlights valuable information on changes made to the flow table and eases an application or administrator to track the evolution of the flow table.
- *gitflow diff*: With *diff*, GitFlow can compare and highlight the updates between commits or between the current master state and any previously snapshotted version. This helps in comprehending the modifications made to individual flows, specific tables and the updated actions.
- *gitflow tag*: Tagging is a critical component in GitFlow. Flow modifications (commits) can be annotated with tags to carry metadata information about the updates. For example, a security update from a firewall application can add annotations specific to the application and provide descriptions about the updates.
- *gitflow grep*: *grep* searches for the specified input pattern (e.g., all entries matching on port 80, or tagged commits) either within a single commit or across commits.

4.2 Investigating Security Issues

Multiple interfaces (sets of controllers, side-channel access via SSH logins, malicious applications with flow-level access) with write access to the flow table can introduce security issues in the data plane. A security violation or malicious update of any manner can impact the packet forwarding functionality, thereby bringing down the network. For example, network operators can login to the switch and introduce simple flow updates that can undermine the network policies maintained by the applications. Similarly, multiple controllers in EQUAL roles can introduce conflicting flow behavior that can violate the security policies programmed in the switch. Existing security enforcement kernels are built as an intermediate layer between the controller and the switches to detect such violations, however, they lack the feature to track the local changes made in the data plane (e.g., via SSH access). We exhibit some GitFlow features that can complement such security offerings:

- *gitflow blame*: The *blame* feature provides information about the changes and the entity/user that modified the interested flow entries. Additional annotations such as timestamps, impacted tenant information, updates (revisions) to individual flows are provided to detail the interested changes.
- *gitflow bisect*: *bisect* identifies the changes that introduced a bug or violation between two versions of the flow table. In addition, the options provided with *bisect* can help operators backtrack the life of a packet in the data path (e.g., set of tables which processed the packet in the multi-table pipeline).
- *gitflow reset*: While debugging a flow related security issue, an application (or operator) can make use of *reset* feature to revert the current flow table to a specific secure state. For example, a compromised controller entity can introduce a flow update to re-direct data traffic to the master controller (via send to controller action); if the operator identifies such a flow update as a security violation, the reset command can help reset the flow table state to the last known working state and discard all changes made to flow table since the previous committed version.

4.3 Revision Control in Troubleshooting

Past work on network troubleshooting [4,5,9,18] highlight how flow space can prove handy to identify issues. GitFlow allows extracting different snapshots of the flow space, thereby empowering troubleshooting. This section attempts to address some important questions posed by Heller et al. [4] in their analysis for troubleshooting SDNs and how GitFlow can assist in these scenarios.

1. **How can we integrate program semantics into network troubleshooting tools?** The paper discusses network equivalents to *gdb, valgrind, gprof*, but interestingly, does not talk about revision management. Network specific versions of these tools help identify errant network states, but revision control can ensure that these situations do not occur again. Troubleshooting information can be stored as version metadata that can assist in avoiding faulty network configurations from occuring again.
2. **How can we integrate troubleshooting information into network control programs?** The presence of multiple versions of network state actually help troubleshooting tools revert to a working version in the case of misconfigurations, failures or such. The use-cases presented in this section reinforce this point.
3. **What abstractions are useful for troubleshooting?** Exposing and abstracting the network state is in itself an interesting proposition. Previously, the absence of accountability was a major driver in not exposing network state to applications running on the controller. With revision control, we have an extra layer of backup to make sure the exposed network state is not misused.

As networks grow to be more autonomous and self-healing in nature, automated troubleshooting tools become a focal point of operations. Flow-level revision control can potentially aid these tools exercise a more intricate level of inspection than just investigate changes to the flow tables.

These illustrative use-cases represent the necessity for versioning across different disciplines such as security and troubleshooting. We believe that GitFlow makes a

strong case towards filling an existing gap in network state management for software-defined networks.

5. CONCLUDING REMARKS

Based on the premise that SDN can also benefit from source code development and troubleshooting tools in the software world, we proposed GitFlow as an important abstraction for flow state management. We show several examples where adopting revision control in the network programming workflow allows us to achieve higher level of safety, provenance, ease of programmability, and support for multiple applications.

While the use cases discussed above are mostly investigating single switch or single table scenarios, we envision and expect our revision control system to provide substantial support in network-wide and multi-table state management. Our revision control framework can be used to correlate state across multiple switches to identify related disruptive changes, or across multiple flow tables to identify the exact sequence of rules matched for a packet.

As part of our future work, we will implement GitFlow in a realistic SDN deployment setting. The usefulness of its features will be evaluated against the presented use cases and overhead implications investigated.

GitFlow, as an abstraction, is easily extensible and allows for adding intelligence on how the local copy of the state is mutated. Our framework can be reused for other purposes, including batch transactions, state rollbacks, and clustering consistency. We can also extend GitFlow to address network state optimizations, such as conserving flow table space. As the flow table state evolves over time, it is critical to maintain the optimal state in the forwarding agent as well as handle the limitation of flow table size (i.e., TCAM limitation). Efficient aggregation schemes address this problem by reducing the number of flow entries, but still maintaining the same forwarding behavior. GitFlow can facilitate such optimizations with efficient resolution features.

6. REFERENCES

[1] Andersen, D. G., Balakrishnan, H., Feamster, N., Koponen, T., Moon, D., and Shenker, S. Accountable internet protocol (AIP). In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication* (Seattle, WA, USA, 2008), SIGCOMM '08, ACM, pp. 339–350.

[2] Dhamdhere, A., and Dovrolis, C. Twelve years in the evolution of the internet ecosystem. *IEEE/ACM Trans. Netw. 19*, 5 (Oct. 2011), 1420–1433.

[3] Git-A Distributed Version Control System. `http://git-scm.com`.

[4] Heller, B., Scott, C., McKeown, N., Shenker, S., Wundsam, A., Zeng, H., Whitlock, S., Jeyakumar, V., Handigol, N., McCauley, J., Zarifis, K., and Kazemian, P. Leveraging SDN layering to systematically troubleshoot networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking* (Hong Kong, China, 2013), HotSDN '13, ACM, pp. 37–42.

[5] Kazemian, P., Varghese, G., and McKeown, N. Header Space Analysis: Static Checking for Networks. *NSDI* (2012), 113–126.

[6] Khurshid, A., Zou, X., Zhou, W., Caesar, M., and Godfrey, P. B. VeriFlow: Verifying Network-Wide Invariants in Real Time. *NSDI* (2013), 15–27.

[7] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev. 38*, 2 (Mar. 2008), 69–74.

[8] N. Foster, et al. Frenetic: A network programming language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (2011).

[9] N. Handigol, et al. Where is the debugger for my software-defined network? In *Proceedings of the first HotSDN workshop* (New York, New York, USA, Aug. 2012), pp. 55–60.

[10] Natarajan, S., Huang, X., and Wolf, T. Efficient conflict detection in flow-based virtualized networks. In *Computing, Networking and Communications (ICNC), 2012 International Conference on* (2012), IEEE, pp. 690–696.

[11] Natarajan, S., Ramaiah, A., and Mathen, M. A Software defined Cloud-Gateway automation system using OpenFlow. *CLOUDNET* (2013), 219–226.

[12] Openflow specification. `https://www.opennetworking.org/sdn-resources/technical-library`.

[13] Porras, P., Shin, S., Yegneswaran, V., and Fong, M. A security enforcement kernel for OpenFlow networks. In *Proceedings of the first HotSDN Workshop* (2012).

[14] Ramachandran, A., Bhandankar, K., Tariq, M. B., and Feamster, N. Packets with provenance. In *Proceedings of the ACM SIGCOMM Poster* (2008).

[15] S. Jain, et al. B4: Experience with a globally-deployed software defined wan. *SIGCOMM Comput. Commun. Rev. 43*, 4 (Aug. 2013), 3–14.

[16] Sherwood, R. Can the production network be the testbed. In *USENIX OSDI* (2010).

[17] Sherwood, R., Gibb, G., and Yap, K. K. Flowvisor: A network virtualization layer. *Open Networking Foundation* (2009).

[18] Wundsam, A., Levin, D., Seetharaman, S., and Feldmann, A. OFRewind: enabling record and replay troubleshooting for networks. In *Proceedings of the 2011 USENIX ATC* (June 2011).