Where is the Debugger for my Software-Defined Network?

Nikhil Handigol[†], Brandon Heller[†], Vimalkumar Jeyakumar[†],
David Mazières, and Nick McKeown
Stanford University
Stanford, CA, USA
{nikhilh,brandonh,jvimal,nickm}@stanford.edu, http://www.scs.stanford.edu/~dm/addr/
[†] These authors contributed equally to this work

ABSTRACT

The behavior of a Software-Defined Network is controlled by programs, which like all software, will have bugs – but this programmatic control also enables new ways to *debug* networks. This paper introduces `ndb`, a prototype network debugger inspired by `gdb`, which implements two primitives useful for debugging an SDN: *breakpoints* and *packet backtraces*. We show how `ndb` modifies forwarding state and logs packet digests to rebuild the sequence of events leading to an errant packet, providing SDN programmers and operators with a valuable tool for tracking down the root cause of a bug.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operation; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Design, Algorithms, Reliability

Keywords

Interactive, Network Debugging, Software-Defined Networks

1. INTRODUCTION

Networks are notoriously hard to debug. Network operators only have a rudimentary set of tools available, such as `ping` and `traceroute`, passive monitoring tools such as `tcpdump` at end-hosts, and `netflow` at switches and routers. Debugging networks is hard for a reason: these tools try to reconstruct the complex and distributed state of the network in an ad-hoc fashion, yet a variety of distributed protocols, such as L2 learning and L3 routing, are constantly changing that state.

In contrast, Software-Defined Networks (SDNs) are programmatically controlled: network state is managed by logically centralized control programs with a global network view, and written directly into the switch forwarding tables using a standard API (e.g. OpenFlow [10]). In this new world, we can start to debug networks like we debug software: write and execute control programs, use a debugger to view context around exceptions (errant packets), and trace sequences of events leading to exceptions to find their root causes. SDNs provide the opportunity to rethink how we debug networks, from the development of control programs, all the way to their deployment in production networks.

Inspired by `gdb`, a popular debugger for software programs, we introduce `ndb`, a debugger for network control programs in SDNs. Both `gdb` and `ndb` execute a slightly modified version of the original control flow, while maintaining its original semantics. Our goal for `ndb` is to help pinpoint the sequence of events leading to a network error, using familiar debugger actions such as *breakpoint, watch, backtrace, single-step, and continue.*

This paper focuses on the first primitives we have implemented in `ndb`: *breakpoint* and *backtrace*. When debugging a *program* `gdb` lets us pause execution at a breakpoint, then shows the history of function calls leading to that breakpoint. Similarly, a *packet backtrace* in `ndb` lets us define a packet breakpoint (e.g., an un-forwarded packet or a packet filter), then shows the sequence of forwarding actions seen by that packet leading to the breakpoint, like this:

```
packet [dl_src: 0x123, ...]:
      switch 1: { inport: p0, outports: [p1]
                  mods: [dl_dst -> 0x345]
                  matched flow: 23 [...]
                  matched table version: 3 }
      switch 2: { inport: p0, outports: [p2]
                  mods: []
                  matched flow: 11 [...]
                  matched table version: 7 }
      ...
      switch N: { inport: p0
                  table miss
                  matched table version: 8 }
```

The information in a packet backtrace helps SDN pro-

HotSDN'12, August 13, 2012, Helsinki, Finland.
Copyright 2012 ACM 978-1-4503-1477-0/12/08 ...$15.00.

grammers resolve logic bugs, helps switch implementers resolve protocol compliance errors, and helps network operators submit detailed bug reports to vendors. We make the following contributions:

Introduce packet backtrace and show its value. §2 shows how the information in a packet backtrace can help solve a range of bug types.

Show the feasibility of backtrace in hardware. Our prototype is a usable `ndb` with a few caveats. §3 describes the prototype, requirements for deterministic packet backtrace, and protocol features that would ease future implementations. §4 examines practical concerns and ways to address them.

Describe packet backtrace extensions. §5 shows natural extensions to `ndb`: forward trace, generalized breakpoints, and controller integration.

We conclude with a discussion in §7 of the generality of packet backtrace and avenues for future network debugging tools.

2. NETWORK DEBUGGER IN ACTION

To illustrate the value of a network debugger, we walk through three bugs encountered while creating an in-network load balancer [5]. All three represent errors commonly seen by SDN programmers: (1) a race condition when installing switch flow entries, (2) a controller logic error, and (3) a bug in a switch implementation. Each bug provides a concrete example where the information from a breakpoint-triggered packet backtrace helped a programmer to narrow down the source of a bug.

Load Balancer. The load-balancing controller has two main functions: server location discovery and flow routing. To locate servers, each server periodically injects packets to a known UDP port, and each first-hop switch sends these packets to the controller. To route flows, the controller chooses a server and a path to it, then pushes flow entries to switches along the way.

Symptoms. Three bugs appeared: (1) packets matching no forwarding rule in the middle of the network, (2) servers discovered at the wrong location, and (3) servers that could not connect to clients.

Using `ndb`. Breakpoints[1] are specified using a `tcpdump`-like interface. A breakpoint is a filter defined on packet headers that applies to one or more switches. When a packet triggers a breakpoint along its path, the user sees a backtrace that lists the forwarding details for each hop encountered by the packet on its path, including inputs (flow table state, input packet, and ingress ports) and outputs (matched flow entry, packet modifications, and egress ports).

Bug 1. We instructed `ndb` to return backtraces for packets that matched *no* flow entries. The first backtrace highlighted a packet that was forwarded properly until its last hop, where it failed to match any flow entries. This suggests either a misconfigured flow entry or an incomplete path (e.g. a race condition where differences in control channel delays lead to partially inserted paths, as explained more in [1, 12]). By inspecting the log of control channel messages for the switch, we found that a matching entry was on its way and added shortly after our packet backtrace. With sufficient evidence that the bug was a race condition, not invalid logic, the programmer chose to work around the bug by sending these rare packets directly to their destination through the control channel.

Bug 2. To find the server location bug, we added a breakpoint to match UDP location discovery packets at each interior switch. The first backtrace showed a flow entry on an ingress switch that forwarded UDP packets. The flow entry pointed to the source of the logic error: in order to route application-generated UDP packets between hosts, the controller installed a high-priority flow entry that matches all UDP packets originating from the given host. This particular flow entry also matched and forwarded the location-discovery packets, causing them to show up at an internal switch. The bug was fixed by changing the code to install flow entries with a more specific match.

Bug 3. To investigate why a specific client and server could not connect, we added breakpoints to view all data packets coming to the server. The backtrace showed that packets were reaching the server along the right path, but they were being corrupted along the way. One switch was corrupting the packet's source MAC and the destination server rejected packets from unknown senders. The information in the backtrace made the story clear. Remembering a mailing list post with similar symptoms [11], we updated the software switch and the connectivity problem went away.

Test Cases. These three examples served as our first test cases for the `ndb` prototype. We recreated each scenario in Mininet [7] with a custom controller and a chain topology. In each case, `ndb` produced a correct backtrace with the full context around an errant packet, allowing us to immediately find the bug.

General Applicability and Limitations. We wondered if `ndb` was providing useful information for debugging, so we informally surveyed SDN developers to find which bugs, in their experience, were most common. Their war stories were of two types: performance issues (non-line-rate forwarding or traffic imbalance) and forwarding correctness (reachability and isolation). While `ndb` can't diagnose performance bugs, its strength is in diagnosing bugs that affect the correctness of forwarding, including control logic errors, network race conditions, configuration errors, unexpected packet formats, and switch implementation errors. For the majority of bugs that were painful enough for developers to recall, the information in a packet backtrace produced by `ndb` would help to isolate one of these categories as the culprit.

[1] Our "breakpoints" do not pause packets, so perhaps "tracepoint" is a better name. We keep the breakpoint terminology for its familiarity in program debugging.

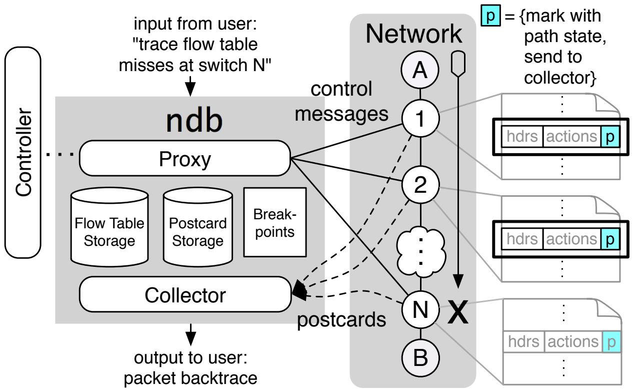


Figure 1: Architecture of ndb, showing a packet backtrace for Bug 1 (Network Race Condition) on a chain topology.

3. PROTOTYPE

Should we record the path taken by a world traveler from the stamps in their passport, or from the postcards they send home? This analogy summarizes our choice when implementing packet backtrace. It is conceptually simple to "stamp" every packet with the flow entry it matches at every hop; it is then easy to reconstruct the path followed by the packet. But this approach is impractical in hardware networks. Commodity switches do not support adding this state within a packet. Even if they did, the packet would become too long and the added state would interfere with forwarding.[2] Instead, we choose to send a small "postcard" every time a packet visits a switch. This approach avoids the problems of stamping, but requires postcards to uniquely identify a packet and include a matched flow entry to reconstruct that packet's path.

In our implementation, a postcard is a truncated copy of the packet's header, augmented with the matching flow entry, switch, and output port. Figure 1 shows the main code components of `ndb`: (1) Proxy, which modifies control messages to tell switches to create postcards, and (2) Collector, which stores postcards and creates backtraces for breakpoint-marked packets. Note that postcards are generated and collected entirely in the datapath; they never enter the control channel. Each piece must scale to high flow entry and postcard rates, without affecting production traffic; §4 addresses these practical concerns of bandwidth, processing, and storage.

3.1 Main Functions

`ndb` must create postcards, collect the postcards, and construct backtraces. These functions could be implemented in multiple ways, and we picked the choices most implementable today on OpenFlow 1.0. Each choice highlights a requirement for producing correct, unambiguous and complete packet backtraces on an arbitrary network topology, which a future, more production-ready `ndb` could attain. For simplicity, we assume single-table switches; for practicality, we assume no time synchronization between switches or the collector(s).

```
# Assume all packets on outport C are routed to the collector.
# Interpose on the control channel:
while True:
  M = next message
  S = switch targeted by M
  if M is a flow modification message:
    F = flow entry specified by M
    S.version += 1
    tag_actions = []
    for action in F.actions:
      if action == Output:
        tag = pack_to_48bits(one_byte(S.id),
                             two_bytes(action.port),
                             three_bytes(S.version))

        tag_actions.append(SetDstMac(tag), Output(port=C))
    F.actions.append(tag_actions)
  S.send_message(M)
```

Figure 2: Algorithm in the Proxy component of ndb to modify control messages to create postcards.

Creating Postcards. To create postcards, `ndb` appends each flow entry modification message with actions to tag and output a copy of the original packet — one per original output port. The tag encodes three values: the switch ID, the output port, and a version number for the matching flow entry. The version number is a counter that increments for every flow modification message. `ndb` uses the destination MAC address field to write hop state, as per the pseudocode in Figure 2. To modify each message, `ndb` proxies the control channel, leaving controllers and switches unmodified.

Collecting Postcards. At the collector, `ndb` checks incoming postcards against a list of breakpoint filters, and stores the vast majority that do not trigger a breakpoint. Later, when constructing a backtrace, `ndb` retrieves those postcards created and sent by the breakpoint-triggering packet. Finding these postcards requires packets to be uniquely identifiable, even though switch actions can modify packet headers. To identify postcards, `ndb` looks at immutable header fields, such as the IPID and TCP sequence numbers, that not modified by forwarding rules in switches.

To store and retrieve postcards efficiently `ndb` uses a hash table called the *path table*. Each key is a combination of all immutable header fields in a packet, and each value is a list of collected postcards. To minimize in-memory state, `ndb` stores postcards in a circular buffer and evicts them after a delay equal to the maximum time for a packet to traverse the network. When a postcard times out, the corresponding data is removed from the path table. For simplicity, our prototype collector is a single server that receives all postcards.

Constructing a Backtrace. When a packet triggers a breakpoint at the collector, `ndb` must reconstruct the back-

[2] It would be easy in a SDN simulator, emulator or software-only network, but our goal is for backtrace to work in any SDN.

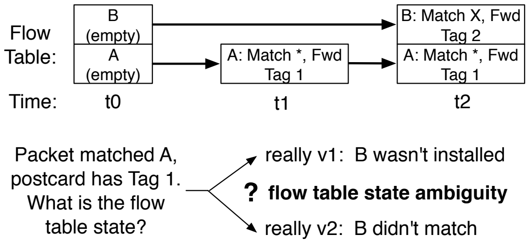

Flow Table:
B (empty)
A (empty)
A: Match *, Fwd Tag 1
B: Match X, Fwd Tag 2
A: Match *, Fwd Tag 1
Time:
t0
t1
t2
Packet matched A, postcard has Tag 1. What is the flow table state?
really v1: B wasn't installed
? flow table state ambiguity
really v2: B didn't match

Figure 3: Simple two-entry flow table highlights a possible ambiguity that cannot be resolved by knowing just the flow entry a packet matched.

trace from the set of matching postcards. Doing this without timestamps or explicit input ports requires topology knowledge to know the (switch, input port) opposite a (switch, output port) tuple. With topology knowledge, the set of postcards define a subgraph of the original network topology. The returned backtrace path is simply the topological sort of this subgraph, formed after a breakpoint triggers.

3.2 Resolving Ambiguity

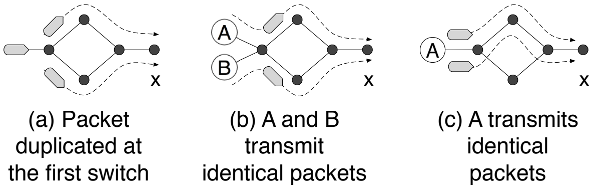
Our implementation is not perfect. In some cases, flow table and packet ambiguity may prevent ndb from identifying an unambiguous packet backtrace.

Flow Table Ambiguity. Postcards generated by the tagging algorithm (§3.1) uniquely identify a switch, flow entry, and output port(s). In most cases, a backtrace constructed from these postcards provides sufficient information to reason about a bug. However, a developer cannot reason about the *full* flow table state when postcards specify only the matching flow entry. This gap in knowledge can lead to the ambiguity shown in Figure 3. Suppose a controller inserts two flow entries A and B in succession. If ndb sees a postcard generated by entry A, it may be that entry B wasn't installed (yet), or that entry B did not match this packet. Moreover, a switch can timeout flow entries, breaking ndb's view of flow table state. To resolve such ambiguities and produce the state of the *entire* flow table at the time of forwarding, ndb should observe and control every change to flow table state, in three steps.

First, to prevent switch state from advancing on its own, ndb must emulate timeouts. Hard timeouts can be emulated with one extra flow-delete message, while soft timeouts require additional messages to periodically query statistics and remove inactive entries.

Second, ndb must ensure that flow table updates are ordered. An OpenFlow switch is not guaranteed to insert entries A and B in order, so ndb must insert a barrier message after every flow modification message.

Third, ndb must version the entire flow table rather than individual entries. One option is to update the version number in *every* flow entry after *any* flow entry change. This bounds flow table version inconsistency to one flow entry, but increases the flow-mod rate by a factor equal to the number of flow entries. A cleaner option is atomic flow updates. Upon each state change, ndb could issue a transaction to atomically update all flow entries to tag packets with the new version number, by swapping between two tables or by pausing forwarding while updating. Our preferred option is to have a separate register that is atomically incremented on each flow table change. This decouples versioning from updates, but requires support for an action that stamps packets with the register value.



(a) Packet duplicated at the first switch

(b) A and B transmit identical packets

(c) A transmits identical packets

Figure 4: Ambiguous scenarios for packet identification. X marks the breakpoint.

Packet Ambiguity. Identifying packets uniquely within the timescale of a packet lifetime is a requirement for deterministic backtraces. As shown in Figure 4, ambiguity arises when (a) a switch duplicates packets within the network, (b) different hosts generate identical packets, or (c) a host repeats packets (e.g. ARP requests). For example, in Figure 4(a), a packet duplicated at the first switch triggers breakpoint X, but backtraces along the upper and lower paths are equally feasible. In these situations, ndb returns all possibilities.

To evaluate the strategy of using non-mutable header fields to identify packets, we analyzed a 400k-packet trace of enterprise packet headers [8]. Nearly 11.3% of packets were indistinguishable from at least one other packet within a one-second time window. On closer inspection, we found that these were mostly UDP packets with IPID 0 generated by an NFS server. Ignoring these removed all IP packet ambiguity, leaving only seven ambiguous ARPs. This initial analysis suggests that most of the packets have enough entropy in their header fields to be uniquely identified. We leave dealing with ambiguous packets to future work.

3.3 SDN Feature Wishlist

Implementing our ndb prototype revealed ways that future versions of OpenFlow (and SDN protocols in general) could be modified to simplify the creation of future network debuggers. We make three concrete suggestions:

Atomic flow table updates. As discussed in §3.2, tagging postcards with just the matching flow entry leads to ambiguous flow table state. An unambiguous backtrace would require concurrent updates – either across multiple entries within a single flow table or over a flow entry and a register. We recommend atomic update primitives for future SDN protocols.

Layer-2 encapsulation. The current version of ndb overwrites the destination MAC address field with postcard data (§3.1), but this choice obscures modifications to this field, and possibly bugs [11]. Another approach is to encapsulate postcard data. We could use a VLAN tag, but OpenFlow 1.0 only supports a single layer of VLAN tags, preventing its use on already-VLAN-tagged networks. OpenFlow 1.1+ allows multiple VLAN tags, but that gives only 15 bits to store the data for one hop. Support for layer-2 encapsulation, such as MAC-in-MAC, would remove this limitation.

Forwarding actions. ndb could benefit from additional forwarding actions: (1) WriteMetadataToPacketField: To generate a backtrace without knowing the network topology, ndb would need the inport and outport(s) for every switch hop. If the inport field in a flow entry is wildcarded, ndb must expand the wildcard and insert input-port-specific flow entries to correctly tag packets, wasting limited resources. Writing the packet's input port number into its header would help ndb create backtraces without topology knowledge. Similarly, writing the contents of a version register to a packet decouples versioning from postcard actions. (2) TruncatePacket: Our prototype creates full sized postcards, since OpenFlow does not support an action to truncate packets in the datapath. Truncating packets would enable postcard collection to scale.

4. FREQUENTLY ASKED QUESTIONS

Will ndb melt my network? If postcards are sent over a separate "debug port", then the original (i.e., not-to-collector) packets and paths are *unaffected*. Even if postcards travel in-band to share links with the original packets, we can minimize the impact by placing postcards in low-priority queues.[3] Duplication of packets for postcard generation does increase the use of internal switch bandwidth, but should only matter for highly utilized switches with small average packet sizes.

In the control path, ndb, *as implemented*, creates no additional (expensive) flow insertions for the switch to process and should also be able to process control messages even for fairly large networks. Recent work has demonstrated OpenFlow controllers that scale to millions of messages per second, beyond the capabilities of entire networks of hardware switches [15].

Can my network handle the extra postcard bandwidth? Recall that *every* packet creates a postcard at *every* hop. For an estimate of the bandwidth costs, consider the Stanford campus backbone, which has 14 internal routers connected by 10Gb/s links, two Internet-facing routers, a network diameter of 5 hops, and an average packet size of 1031 bytes. Minimum-size 64 byte postcards increase traffic by $\frac{64B}{1031B} \times 5(hops) = 31\%$. The average aggregate bandwidth is 5.9Gb/s, for which collector traffic adds 1.8Gb/s. Even if we conservatively add together the peak traffic of each campus router, the sum of 25Gb/s yields only 7.8Gb/s of collector traffic.

Can the collector handle the deluge of postcards? At Stanford, the collector must be provisioned to handle a rate of 7.8Gb/s, which equals 15.2M postcards/second. Prototype software routers have shown that a Linux server can forward minimum-size packets at 40Gb/s [2]. In other words, we could collect postcards for every packet header in every switch of the campus backbone using a single server.

Can a server store the entire path table? The memory required to store the postcards is determined by the maximum queueing delay in the network and the maximum postcard rate. Storing postcards at Stanford for a second requires less than 1Gbyte of memory.

Can the collector be parallelized? Collection is embarrassingly parallel. Postcards can be load-balanced across collectors at the granularity of a switch, port, or single flow entry. At the extreme, a collector could be attached to (or embedded inside) each switch. The reconstruction process would have to change slightly, as multiple collectors would need to be queried for each triggered breakpoint.

Can we reduce the rate of postcards? A programmer may be interested only in a subset of traffic; e.g. UDP packets destined to a specific IP address. Creating postcards for a subset of traffic greatly reduces demands on the collector. However, if packets may be transformed in arbitrary ways inside the network, it is hard to safely design a filter. In the example above, if IP addresses are rewritten, or worse, if TCP packets can change to UDP packets, postcards have to be created for *all* TCP and UDP packets. In such cases, ndb can benefit from techniques such as Header Space Analysis [6] to identify which forwarding rules may trigger the current set of breakpoints.

5. EXTENDING NDB

Our ndb prototype can continuously record single-path packet backtraces for multiple breakpoints, in parallel. There are a number of ways to extend this:

Forward Trace. A backtrace connects from a breakpoint back to its source; a forward trace connects from a breakpoint to its destination. The algorithm described in Figure 2 actually handles this case already, providing combined forward-and-backward paths for all packets.

Flexible Breakpoints. Because breakpoints are implemented by the collector, rather than the switches, we can implement filters on both packet fields and their paths. Example filters include port ranges, path lengths, switch hops, and anything that can be implemented efficiently at the collector. One can even imagine creating a language to define queries that supports tcpdump-like header match semantics over subsets of network paths.

Integrating Packet Backtrace with the control plane. Currently, packet backtrace is limited to the network; it cannot directly point to a line of code that modified or inserted a flow entry. Integrating packet backtrace mechanisms into a controller, adding an API by which controllers can integrate themselves, or combining ndb with gdb

[3] We would need to use a 'postcard bit' (e.g. in the type of service field) to prevent generating postcards of postcards.

would allow a packet backtrace to show the relevant code paths, events, and inputs involved in forwarding a packet. For example, storing program backtraces as values, keyed by the flow entries they produce, would support an efficient controller-integrated backtrace.

6. RELATED WORK

Anteater [9] and Header Space Analysis [6] statically analyze the dataplane configuration to detect connectivity and isolation errors. Languages such as Frenetic [4] and Nettle [16] abstract aspects of SDNs to improve code correctness and composability. Consistent Updates [12] adds flow and packet consistency primitives. NICE [1] combines model checking and symbolic execution to verify controller code. The above techniques model network behavior, but bugs can creep in at any level in the SDN stack and break an idealized model. `ndb`, on the other hand, takes a more direct approach – it finds bugs that manifest themselves as errantly forwarded packets and provides direct evidence to help identify their root cause.

OFRewind [17] is a tool for recording and playing SDN control plane traffic. While `ndb` also records flow table state via a proxy and logs packet traces, they differ in their overall approach to debugging. OFRewind aids debugging via scenario re-creation, whereas `ndb` helps debugging via live observation. `ndb` also benefits from many techniques used for efficient traffic monitoring and robust IP traceback in ISPs. Trajectory sampling [3] proposed a hash function to sample packets consistently at all routers, for direct traffic monitoring. Further, `ndb` can use sampling and memory reduction techniques for IP traceback [13, 14] to reduce bandwidth and memory costs for debugging an SDN.

7. DISCUSSION

SDN as an architecture provides structured state and well-defined semantics, which enables new ways to debug networks. This paper showed one approach — packet backtrace triggered by a breakpoint — to be useful, feasible, and scalable. But we wish to stress the generality of the *concept* of interactive debugging, not just the specifics of a prototype. `ndb` appears useful across a range of:

Network Types: Unlike today's tools, `ndb` doesn't focus on a single layer or protocol.

Control Applications: `ndb` makes no assumptions about the controller(s) and imposes no language, primitive, or framework restrictions on them.

Human Roles: `ndb` benefits switch vendors, framework developers, application writers, and even network operators.

Still, many practical questions remain: What are the scaling limits of the current design? Where should `ndb` functionality be placed? What is the right user interface? What switch changes would help to build a future `ndb` *right*? As shown in §3.2, currently used SDN control protocols have limitations for debugging; the semantics for modifying forwarding state and packets could be more flexible. Other communities have found support for easier debugging so valuable that they have standardized hardware support for it, such as JTAG in-circuit debugging in embedded systems and programmable debug registers in x86 processors. As a community, we should explore whether to augment hardware or whether to use software workarounds with caveats like those in §3. We believe that `ndb` is only one of many SDN-specific tools that will make networks easier to debug, benefiting networking researchers, vendors, and operators.

Acknowledgments

This work was funded by NSF FIA award CNS-1040190, NSF FIA award CNS-1040593-001, Stanford Open Networking Research Center (ONRC), a Hewlett Packard Fellowship, and a gift from Google.

References

[1] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A nice way to test openflow applications. In *NSDI*. USENIX, 2012.

[2] M. Dobrescu, N. Egi, K. Argyraki, B. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *ACM SOSP*, volume 9. Citeseer, 2009.

[3] N. G. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. *IEEE/ACM Trans. Netw.*, 2001.

[4] N. Foster, R. Harrison, M. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. *ACM SIGPLAN Notices*, 46(9):279–291, 2011.

[5] N. Handigol and others. Aster* x: Load-balancing web traffic over wide-area networks. *GENI Engineering Conf. 9*, 2010.

[6] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *NSDI*. USENIX, 2012.

[7] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *HotNets*. ACM, 2010.

[8] LBNL/ICSI Enterprise Tracing Project. `http://www.icir.org/enterprise-tracing/download.html`.

[9] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteater. SIGCOMM '11, New York, NY, USA, 2011. ACM.

[10] The openflow switch. `http://www.openflow.org`.

[11] [ovs-discuss] setting mod-dl-dst in action field of openflow corrupts src mac address. `http://openvswitch.org/pipermail/discuss/2012-March/006625.html`.

[12] M. Reitblatt, N. Foster, J. Rexford, and D. Walker. Consistent updates for software-defined networks: change you can believe in! In *HotNets*. ACM, 2011.

[13] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical network support for ip traceback. SIGCOMM '00, New York, NY, USA, 2000. ACM.

[14] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Strayer. Hash-based ip traceback. SIGCOMM '01, New York, NY, USA, 2001. ACM.

[15] A. Tootoonchian and Y. Ganjali. Hyperflow: A distributed control plane for openflow. In *Workshop on Research on enterprise networking (INM/WREN)*. USENIX Association, 2010.

[16] A. Voellmy and P. Hudak. Nettle: Functional reactive programming of openflow networks. *PADL, Jan*, 2011.

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