Toward Taming Policy Enforcement for SDN in The RIGHT Way: Or Can We?

Kai Bu[⋆,▷], Minyu Weng[∗], Junze Bao[⋆], Zhenchao Lin[⋆], Zhikui Xu[⋆]
[⋆]College of Computer Science and Technology, Zhejiang University
[∗]College of Electrical Engineering, Zhejiang University
([▷]corresponding author: kaibu@zju.edu.cn)

***Abstract*—This paper explores a RIGHT framework for reliable policy enforcement in Software-Defined Networking (SDN). Current SDN uses overlapping rules with common matching packets. Even if a packet's expectant rule is inactive, it might hit another rule and experience incorrect yet unnoticed processing. This leads to inconsistency between control plane and data plane, that is, unreliable policy enforcement. RIGHT advocates three adaptations to respectively mitigate, detect, and correct packet processing errors. It is challenging for RIGHT to maintain both accuracy and efficiency. We explore lightweight modifications to current SDN policy enforcement toward better reliability. For example, we decouple rules and priorities through tagging to mitigate matching ambiguity. We also use exact-match rules to efficiently, correctly process packets in the same micro-flow. RIGHT can remedy mis-forwarded packets as well. We expect that a comprehensive design and deployment of RIGHT helps ensure correct per-packet processing in real time for SDN.**

I. Introduction

SDN, where one controller to rule them all. Software-Defined Networking (SDN) enables flexible network management using a centralized controller to take over otherwise distributed management functions [1]. These functions are realized through applications like routing, firewall, and load balancing [2]. Each application translates management policies to rules understandable to SDN switches. The controller then populates rules to switches, which accordingly process network traffic. To guarantee policy enforcement correctness, the controller should verify rules against configuration errors [3]–[5] or security loopholes [6] before issuing them to switches.

Out of controller, out of control. Once rules are heading toward switches, the controller loses control over policy enforcement in that switches may not always conform to populated rules as the controller intends. Possible causes are, for example, rule installation failure [7], [8] and rule priority fault [9], [10]. They make some packets follow unexpected rules and breach management policies. Several methods aim to detect the existence of such policy breach [7], [8], [10]–[12]. These methods, however, do not delve into how to fix detected rule faults. Finding a policy breach on data plane is certainly not the ultimate goal of network management. Instead, we need to ensure reliable policy enforcement on data plane.

Don't like it? Change it. Reliable policy enforcement for SDN certainly requires more efforts beyond existing detection methods. We cannot simply re-update detected faulty rules either. The newly issued rules might experience faults again or even affect other installed rules. The latter surprising case is because SDN switches emulate rule priority using memory location [13]. Upon each rule update, a switch needs to guarantee that a higher-priority rule has higher memory location. This usually induces rule relocation across the entire memory on switches. In summary, current SDN is susceptible to unreliable policy enforcement but has intrinsic barriers to an ideal fix without modification. Given the importance of reliable policy enforcement in network management, we decided to see how we could adapt current SDN framework toward better reliability. RIGHT is the result of that experiment.

Quest for the RIGHT change. This paper explores the design of RIGHT (*pRIority Goes to Hardware TCAM*), a framework for reliable policy enforcement in SDN. RIGHT incorporates three main components that respectively mitigate, detect, and correct forwarding errors at packet level in real time. We are not proposing any radical new techniques for building RIGHT; we explore and leverage prior wisdoms toward an important goal of SDN's interest—reliable policy enforcement. Specifically, three principles and benefits of RIGHT design are highlighted as follows.

1) *Decouple rules and priorities to mitigate forwarding errors.* The major reason for stealthy mis-forwarding is overlapping rules. No matter whether a packet's expectant rule is inactive [7], [8], [11] or priority-swapping with another one [9], [10], as long as the packet matches a rule, a switch will process the packet as if all was well. RIGHT retrofits rules and packet headers such that a packet matches at most one rule on en-route switches. This not only mitigates rule priority fault but also eases revealing rule installation failure.

2) *Verify forwarding correctness at packet level.* Since the ingress switch processes original packets, we cannot alter its rules' matching fields. In other words, the ingress switch is still forwarding-error prone. RIGHT enables a second-hop switch to verify every incoming packet's forwarding correctness together with the controller. An efficient solution should not repeat verifying packets in the same micro-flow. If a packet is mis-forwarded, RIGHT follows the subsequent correction scheme to re-direct it to the correct forwarding path.

3) *RIGHT the wrong, for mis-forwarded packets as well.* Once a mis-forwarded packet is detected, the controller will 1) re-direct it for correct forwarding and 2) issue new rules to prevent packets in the same micro-flow from mis-forwarding. Enforcing such correction scheme, RIGHT helps guarantee processing correctness of packets forwarded both before and

after it detects corresponding forwarding errors.

Roadmap. Section II reviews causes for unreliable policy enforcement in SDN. Section III outlines how RIGHT adapts current SDN framework toward reliable policy enforcement. Sections IV-V explore the RIGHT design. Finally, Section VI concludes the paper.

II. Background

In this section, we first walk through how SDN enforces network management policies. We then review causes and countermeasures of policy inconsistency between control plane and data plane.

A. Policy Enforcement as Controller Wishes

To enforce high-level network management policies, the centralized controller transforms them to rules understandable to underlying switches. Policy-to-rule transformation is conducted by various control applications such as routing, firewall, and load balancing. The controller then populates rules to corresponding switches. Since a rule specifies which network traffic flow it matches and acts on, switches can accordingly process flows per their matching rules. Moreover, switches store rules using fast Ternary Content Addressable Memory (TCAM), which should be limited in space due to being expensive and power-hungry [14]. The controller thus usually populates rules to switches in a reactive fashion.

Figure 1 exemplifies SDN enforcing a policy that requires packets from the 10.10.0.0/16 subnet be forwarded along switches sw1, sw3, and sw4. This policy enforcement process is triggered by the arrival of the first packet pkt1 from the 10.10.0.0/16 subnet (step a). Assume that sw1 caches no rules for processing pkt1. It needs to query the controller by a PacketIn message with pkt1's header encapsulated (step b). The controller invokes the routing application to transform the 10.10.0.0/16-policy to rules corresponding to switches sw1, sw3, and sw4 along the forwarding path (step c). A rule specifies a matching field and an action field. First, the matching field specifies the condition that a rule's matching packets should satisfy. Leveraging don't care bits ("*") that match both one and zero, a rule aggregates multiple exact-match rules. For example, the rule of sw1—src=10.10.*.*—matches packets with source IP addresses ranging from 10.10.0.0 to 10.10.255.255. Second, the action field regulates how a switch processes a rule's matching packets. The rule of sw1, for example, matches packets from the 10.10.0.0/16 subnet and forwards them to sw3. Given that a packet might match multiple rules, a rule specifies also a priority field to address matching ambiguity [1]. If matching multiple rules, a packet should follow the highest-priority one. The controller populates rules to switches by FlowMod messages (step d). Switches sw1, sw3, and sw4 then follow the rules to forward 10.10.0.0/16-packets without querying the controller (step e).

B. Things outside Controller Gone Wild

Although rules from the controller's viewpoint can be guaranteed to faithfully represent policies [4], [5], [15], they may not accordingly take effect on switches. We reason about various causes for rule inconsistency, some of which have already been substantiated on commercial SDN switches.

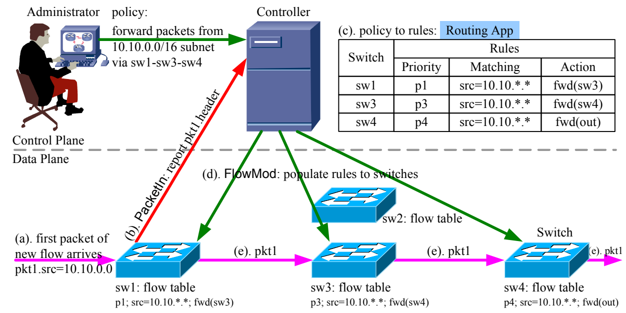


Fig. 1. SDN policy enforcement.

Rule-update message loss. The controller isolates batches of control messages using barrier commands. A switch should complete processing all messages received before a barrier request prior to those received after the barrier request. After processing all prior–barrier-request messages, the switch sends the controller a barrier reply, which ensures the controller of successful message process. Barrier commands can indicate message receipt only at batch level instead of desired message level. Loss of any rule-update message will induce rule inconsistency. A possible cause for message loss is control channel congestion [16].

Asynchronous rule activation. It is hard for the controller to synchronize rule installation on distributed switches. Both controller scheduling and control-channel status affect when rules reach switches. Upon arrival of rule update messages, switch scheduling decides when to install certain rules. Scheduling of rule installation on some commercial switches is different from the sequence instructed by rule update messages [9]. Furthermore, switches usually send barrier replies to the controller before rules take effect [17]. This further exacerbates rule inconsistency between control plane and data plane, especially in dynamic networks [11].

Rule priority violation. A recent measurement study reveals that switches from some vendors deviate from what the controller intends during rule installation [9]. For example, HP 5406zl trims priorities before installing rules to hardware and treats rules installed later as higher-priority ones. According to a test with two rules [9], the one installed later always wins over packets matching both rules. HP 5406zl is thus susceptible to priority violation and rule inconsistency.

Rule installation failure. Even if all the above causes were avoided, switches may still fail to successfully install rules on hardware. Potential causes could be switch software bugs [7] or even hardware errors [18].

C. Bear or Change?

Previous efforts focus mainly on testing rule inconsistency [7], [8], [10]–[12]. The primary idea is exercise rules using test packets. For example, assume that a switch caches only one rule, which forwards packets from the 10.10.0.0/16 subnet via port 1. To test whether this rule is active, we feed the switch with a test packet with source IP address set as, for

example, 10.10.10.10. If we can catch the test packet at port 1, we regard the 10.10.*.*/16-port1 rule as functional. None of existing solutions generates sufficient test packets to exhaust all possible rule inconsistency causes. ATPG [7] tests rule activeness via rule reachability; but even if a rule is inactive, a test packet may still undergo the same path and ATPG misses detecting the inactive rule [8]. Against this issue, ProboScope [8] and Monocle [11] test rule activeness at rule level. They, however, ignore rule priority violation. In practice, it is challenging to achieve real-time test due to the potential nonpolynomial complexity of generating sufficient test packets [10]. Our RuleScope [10] takes the first step toward detecting rule priority violation; but it still faces challenges to real-time monitoring for dynamic networks.

Finding out a rule inconsistency is, however, not the ultimate goal of network management. Networks should process traffic as exactly what the controller/policy wishes. We cannot just simply notify the controller that some rules are inactive or faulty on switches. The controller already did what it thought right. How can it correct the wrong? Will the correction messages encounter rule inconsistency again? What if some traffic undergone undesirable forwarding paths prior to the detection of rule inconsistency? What if an entire short-lived flow has already traversed an unexpected route upon detection? All of these concerns require way more efforts to address than does simply detection.

Then a natural question is how to adapt SDN against rule inconsistency. If we bear rule inconsistency causes in mind from the beginning of configuring SDN, can we prevent rule inconsistency? If it cannot be fully prevented, how can we detect rule inconsistency due to various causes described in Section II-B? For detected rule inconsistency, how can we redirect the affected packets back to the expected paths? That is, how to correct rule inconsistency? For those who hate to bear SDN rule inconsistency, let us set out on the journey toward finding a RIGHT change.

III. The RIGHT Change

In this section, we outline how RIGHT adapts SDN framework toward reliable policy enforcement. We will explore its design specifics in Section IV.

A. Goals

The ultimate goal of exploring RIGHT changes for current SDN design is reliable policy enforcement. Through mapping reliable policy enforcement to rule consistency across control plane and data plane, we further break the above goal into the following three goals.

Goal 1: Prevent or mitigate causes of rule inconsistency. Among such causes investigated in Section II-B, message losses and hardware/software faults are hard to prevent. We could adopt certain acknowledgement schemes against message losses and test/verification schemes against hardware/software faults. For asynchronous rule activation, we could design reasonable rule update scheduling along switches and related acknowledgement schemes for successful rule installation. However, none of the above schemes are trivial; some may associate with a large overhead [17], [19]. In this paper, we attack priority violation from the protocol design perspective. Surprisingly, the scheme we adopt not only mitigates priority violation but also eases revealing rule inconsistencies due to the aforementioned other three causes.

Goal 2: Detect rule inconsistency at packet level. Since causes for rule inconsistency are hard to eliminate, another key step toward reliable policy enforcement is packet-level forwarding correctness verification. The stringent requirement of packet-level verification is because SDN enables a packet to likely match more than one rule. Even if the expectant rule for processing a packet is faulty, the packet might still hit another rule. In this case, the forwarding error is stealthy and hardly exposed to the controller. Furthermore, not all packets will experience mis-forwarding over faulty rules [7], [8]. We cannot simply use forwarding correctness of some packets as the evidence of rule consistency. A desirable SDN framework should detect rule inconsistency at packet level in real time.

Goal 3: Remedy forwarding errors induced by rule inconsistency. Network management solicits correct packet forwarding (Section II-C). SDN should integrate mis-forwarding correction schemes for network administrators to fully embrace its associated flexibility. We expect that such correction schemes cannot benefit only post-remedy packets; packets undergone mis-forwarding are also worthy of remedy.

Toward achieving these goals, RIGHT advocates the following three changes to current SDN forwarding.

B. Changes

Change 1: Decouple rules and priorities against priority violation. To prevent priority violation, an ideal solution should resolve rule overlap/ambiguity in matching fields instead of currently leveraged priority fields. This is because packet headers contain what matching fields specify but not priority values. If we could introduce a unique bound between a packet and its expectant matching rule, it will match at most one rule in switch flow table. Such solution needs to modify both packet headers and rule matching fields.

The idea is highlighted here, while more details are given in Section IV-B. The ingress switch of a packet rewrites packer header such that the packet uniquely matches a rule on subsequent en-route switches. To support this, the controller needs to modify action fields for ingress-switch rules and matching fields for en-route–switch rules. If a packet with re-written header hits no rule on an en-route switch and is reported to the controller, it eases revealing asynchronous rule activation and rule installation failure.

Change 2: Verify forwarding correctness at packet level on second-hop switches. Although the ingress switch rewrites packet headers to decouple rules and priorities for subsequent en-route switches, it does not alter forwarding decisions. In other words, the ingress switch might still experience rule inconsistency. The second-hop switch thus has to assure its subsequent switches that what it sticks to is correct. That is, the second-hop switch should verify whether incoming packets

from the ingress switch follow expectant rules and, if not, trigger the controller for correction.

It is, however, quite challenging for the second-hop switch to achieve accurate yet efficient packet-level verification. If we simply let the second-hop switch direct every incoming packet to the controller for verification, the overhead would be too high. Section IV-C will explore efficient solutions, some of which may sacrifice verification accuracy with a controllable, slight probability.

Change 3: How to RIGHT the wrong? Once a mis-forwarded packet is detected, we should get it back on track. A correction scheme may not simply update corresponding rules on the ingress switch; rule update might cause rule inconsistency again. On the other hand, it is neither practical to let the controller direct mis-forwarded packets to their expectant second-hop switches due to overhead concern. Section IV-D will explore feasible schemes.

IV. Design

In this section, we explore possible RIGHT design choices along with various challenges. For simplicity, *we temporarily base our discussions on a network with only one ingress switch and one egress switch*. (We will generalize RIGHT design toward more complex networks with multi-role switches in Section IV-E.) As later discussions will show, RIGHT design may raise quite a few challenges. A comprehensive design certainly solicits more research/practice efforts.

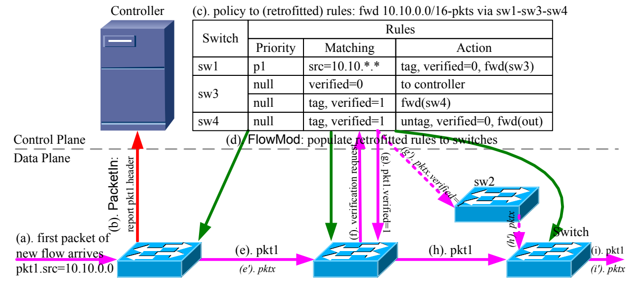
A. RIGHT at First Glance

Figure 2 instantiates our RIGHT framework for reliable policy enforcement. Based on current SDN framework in Figure 1, RIGHT incorporates three adaptations. First, RIGHT retrofits packets and rules such that each packet matches at most one rule on each en-route switch (step c). This confines mis-forwarding to only the ingress switch. Second, the second-hop switch thus needs to corporate with the controller to verify whether incoming packets from the ingress switch follow expectant rules (step f). Third, in case a packet is mis-forwarded, the controller should re-direct it to the correct forwarding path (step g').

One may already perceive certain challenges of implementing RIGHT as in Figure 2. For example, should the second-hop switch direct every packet to the controller for verification (step f)? Should the controller be responsible for directing every mis-forwarded packet to the expectant switch (step g')? We next present RIGHT design specifics as well as how we try to address corresponding challenges.

B. Retrofit Packets and Rules

RIGHT retrofits packets and rules to eliminate rule priority violation and ease revealing rule inactivation on en-route switches. Packet retrofitting rewrites packet headers via tagging on the ingress switch. Tagging can exploit certain unused bits in a packet header [20]. Specifically, we instrument each ingress-switch rule with an additional tagging operation in the action field. Tags should be unique across rules on the



Switch	Rules		
	Priority	Matching	Action
sw1	p1	src=10.10.*.*	tag, verified=0, fwd(sw3)
sw3	null	verified=0	to controller
	null	tag, verified=1	fwd(sw4)
sw4	null	tag, verified=1	untag, verified=0, fwd(out)

Fig. 2. (Basic version of) RIGHT policy enforcement. (Steps e', f, g', h', and i' instantiate mis-forwarding detection and correction.)

ingress switch. Then en-route switches match packets using only tags, leading to no matching ambiguity. Moreover, if an en-route–switch rule is inactive, the en-route switch will direct its matching packet to the controller because the inactive rule is the only one to process the packet. This eases revealing rule inactivation on en-route switches in comparison with conventional SDN where the packet might hit another rule and leave rule inactivation unnoticed. Finally, the egress switch needs to untag packets before steering them out of the network.

Handling drop rules on the ingress switch is worthy of emphasis here. If packets are dropped on the ingress switch, we have no chance to verify whether the ingress switch treats them right. Inspired by ProboScope [8], we defer the drop action to a second-hop switch. Specifically, the controller splits a drop rule on the ingress switch to two rules. One is for the ingress switch; it tags supposed-to-drop packets and forwards them to a second-hop switch. The other is for the second-hop switch; it drops the packets only if the controller verifies the processing correctness of the ingress switch. For simplicity, we omit drop rules in what follows.

C. Verify Forwarding Correctness

Since the ingress switch still uses overlapping rules, it is critical for the second-hop switch to verify the forwarding correctness of incoming packets before delivering them to the next hop. The verification should involve the controller, which is aware of all packets' expectant forwarding paths. Figure 2 illustrates a straightforward yet expensive verification scheme. It requires that the second-hop switch direct every packet to the controller for verification (step f). To this end, the ingress switch specifies each packet a *verified* bit while tagging. It initially zeros the verified bit to indicate that a packet is not verified. Then the second-hop switch accordingly caches a rule for forwarding *verified=0*-packets to the controller. Using a packet's tag, the controller can deduce which rule on the ingress switch processes the packet and therefore verify its forwarding correctness. After verification, the controller sets a packet's verified bit as one and directs it back to data plane (step g or g'). Upon receiving *verified=1*-packets, switches no longer need to query the controller for verification; they simply process them using rules with matching tags.

It is challenging to achieve efficient packet-level verification without touching SDN integrity. While actively pursuing a comprehensive solution, we next discuss three possible efficiency enhancement techniques.

Omit verifying packets processed by "isolated" rules. The verification process concerns with whether a packet follows the expectant rule among multiple overlapping ones. If an *isolated* rule on the ingress switch does not overlap with the others, packets it processes may not experience forwarding errors. Such packets therefore need no verification on the second-hop switch. In light of this, a rule overlapping with no other rule on the ingress switch can directly set its matching packets' verified bit as one. This, however, is not a radical efficient solution as SDN benefits from overlapping rules against TCAM space constraint.

Introduce exact-match rules to track verified packets. After verifying the forwarding correctness of a packet, packets in the same microflow need no further verification. We thus can construct an exact-match rule with extracting certain fields from the verified packet's header as the matching field, that is, (tag, verified=0, other header fields used for matching on the ingress switch). For the next-hop switch to omit verifying corresponding packets, the newly constructed exact-match rule's action should be (verified=1, forward to next hop) instead of directing packets to the controller. We then could introduce an exact-match lookup table to accommodate the preceding exact-match rules [16]. This scheme might modify the conventional SDN switch design, requiring multi-table processing [21]. That is, one for the above exact-match rules and one for the conventional rules. A packet first goes through the former table to check whether its associated micro-flow is verified and only if no will the packet enter the latter table.

Leverage Bloom filters to compress exact-match rules. When exact-match rules for verified microflows cost too much space, we could hash them into a Bloom filter toward space efficiency. The Bloom filter can be embedded in a small, fast memory like SRAM [22]. Upon arrival at the second-hop switch, a packet is first checked against the Bloom filter. If it is "in", it need not be directed to the controller for verification. Instead, the second-hop switch first changes the packet's verified bit to one and then processes it using TCAM rules. This scheme requires more modifications to SDN switches. Furthermore, the intrinsic false positives of Bloom filter technique might leave mis-forwarded packets undetected (with a controllable, slight probability).

D. Remedy Mis-forwarding

To guarantee forwarding correctness, the controller needs to direct any detected mis-forwarded packet to its expectant forwarding path. Figure 2 shows a basic scheme where the controller directly forwards a mis-forwarded packet $pktx$ to the correct second-hop switch sw2 (step g'). This scheme works well for a small amount of mis-forwarded packets. But when mis-forwarded packets are of a large scale, directing all of them to the controller is impractical. Large-scale packets will, for example, fatigue control channel bandwidth and incur forwarding delay.

We can again introduce exact-match rules to prevent most mis-forwarded packets from rushing to the controller. For a verified mis-forwarded packet, the corresponding exact-match rule has matching field as (tag, verified=0, other header fields used for matching on the ingress switch) and action field as (verified=1, forward to the correct second-hop switch). Such rule requires a link from the wrongly chosen second-hop switch to the correct one. Now a subsequent packet will traverse at most three flow/lookup tables on the second-hop switch—an exact-match lookup table matching verified mis-forwarded microflows, an exact-match lookup table matching verified correctly forwarded microflows, and a TCAM flow table matching other flows.

E. RIGHT at Second Sight: Multi-role Switch

We now consider adapting RIGHT to more complex networks with multi-role switches. A multi-role switch is on different hops along multiple forwarding paths. Take switch sw3 in Figure 2 for example. It is now on the second hop of the illustrated forwarding path sw1-sw3-sw4. It can also serve as an ingress switch if it connects to some end-hosts. Moreover, it may become a $(n \geq 3)$th-hop or egress switch under certain network topologies. In the most complex case, a switch could be playing the above three roles at the same time. Although this sounds complicated, we observe that it introduces no particular changes to the explored RIGHT design. The controller follows the same way as in Sections IV-B-IV-D to configure each switch along a forwarding path. In particular, ingress and $(n \geq 3)$th-hop switches require only current SDN switches whereas second-hop switches need cache exact-match rules and support multi-table processing. We therefore suggest designating fewer second-hop switches during network topology planning such that the network can benefit from RIGHT at minimum cost.

V. Discussion

A. Rule Update Cost

Current SDN uses priorities to reconcile overlapping rules. Since packet headers contain no such priority information, TCAM rules match against only packet headers while using memory location to emulate priority. Specifically, the highest-priority rule has the highest memory location in the TCAM [13]. To process a packet, TCAM matches it against all rules in parallel. Among all matched rules, the one with the highest memory location dominates the forwarding decision. Handling priority in such way, a rule update may force relocating some other rules. Our recent measurements show that relocating rules leads to heavy rule update overhead on TCAM. Although many efforts have been made to optimize rule update at the controller end [23], few delve into the switch TCAM side.

Introducing unique tags across rules promises not only fewer forwarding errors but also lower rule update cost. After retrofitting packets and rules (Section IV-B), RIGHT enables non-overlapping rules within each en-route switch. It thus needs no priority to avoid matching ambiguity therein. When the controller updates a new rule to a switch, the switch can place the rule anywhere empty on TCAM, minimizing rule update cost.

B. Flow Table Size

For a tagged packet, RIGHT uses only the tag as its corresponding rule's matching field. A tag is much shorter than currently used tuples comprising several packet header fields [1]. RIGHT thus can store more tag-rules than traditional rules on a switch. This promises a larger flow table, albeit on the same TCAM, and finer-grained network management policies [24].

C. Forwarding Delay

Verifying forwarding correctness at the second-hop switch induces forwarding delay (Section IV-C). In comparison with unverified forwarding, some packets may match through multiple flow tables or traverse an extra round-trip between the controller and switch.

First, such forwarding delay can be somewhat compensated by simplified management of retrofitted rules (Section V-A). When a switch installs only rules with unique tags, it no longer needs the tedious rule relocation to emulate rule priority upon rule update. Corresponding packets then experience less buffering time. This helps decrease average forwarding delay, especially for mice flows.

Second, the benefit of reliable forwarding may outweigh the downside of forwarding delay. The induced latency is necessary for intervening in SDN forwarding toward error detection and correction. Furthermore, it works well for dynamic networks with frequent rule updates. Normally, it is more challenging to test forwarding behavior of dynamic networks [11]. And no matter how fast a test method is, it cannot benefit mis-forwarded packets prior to error detection.

D. Robustness

As with any SDN forwarding investigation methods [7], [8], [10], [11], [25], we find it a bit paradoxical to ensure RIGHT's robustness against packet loss. On the one hand, packet loss is one of the causes for forwarding errors. On the other hand, RIGHT relies on reliable message delivery to enforce reliable forwarding. Without mounting additional acknowledgement schemes, RIGHT's forwarding reliability is also susceptible to packet loss. Instead of making every interaction between the controller and switches reliable, RIGHT requires reliable communication between the controller and second-hop switches. Many choices are in literature but they are beyond the scope of this short paper.

VI. Conclusion

We have presented RIGHT, a new SDN forwarding framework toward reliable policy enforcement. We first reason about the root causes for unreliable forwarding in current SDN. RIGHT then incorporates lightweight modifications to current SDN forwarding to mitigate, detect, and remedy misforwarding. Although still facing various design challenges, we are actively embarking on RIGHT adventure through emulation and implementation. We expect that RIGHT promise correct per-packet processing in real time for SDN.

Acknowledgment

This work is supported in part by the National Science Foundation of China under Grant No. 61402404. We would also like to sincerely thank IEEE SDDCS 2016 chairs and reviewers for their helpful feedback.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[2] C. Monsanto, J. Reich, N. Foster, J. Rexford, D. Walker *et al.*, "Composing software defined networks," in *USENIX NSDI*, 2013, pp. 1–13.

[3] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks." in *USENIX NSDI*, 2012, pp. 113–126.

[4] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis." in *USENIX NSDI*, 2013, pp. 99–111.

[5] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *USENIX NSDI*, 2013.

[6] P. Porras, S. Cheung, M. Fong, K. Skinner, and V. Yegneswaran, "Securing the software-defined network control layer," in *NDSS*, 2015.

[7] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic test packet generation," in *ACM CoNEXT*, 2012, pp. 241–252.

[8] M. Kuzniar, P. Peresini, and D. Kostic, "Proboscope: Data plane probe packet generation," Tech. Rep., 2014.

[9] ——, "What you need to know about sdn flow tables," in *PAM*, 2015.

[10] K. Bu, X. Wen, B. Yang, Y. Chen, L. E. Li, and X. Chen, "Is every flow on the right track?: Inspect sdn forwarding with rulescope," in *IEEE INFOCOM*, 2016.

[11] P. Peresini, M. Kuzniar, and D. Kostic, "Monocle: Dynamic, fine-grained data plane monitoring," in *ACM CoNEXT*, 2015.

[12] I. Pelle, T. Lévai, F. Németh, and A. Gulyás, "One tool to rule them all: a modular troubleshooting framework for sdn (and other) networks," in *ACM SOSR*, 2015.

[13] D. Shah and P. Gupta, "Fast updating algorithms for tcams," *IEEE Micro*, no. 1, pp. 36–47, 2001.

[14] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with difane," in *ACM SIGCOMM*, 2010, pp. 351–362.

[15] W. Zhou, D. Jin, J. Croft, M. Caesar, and P. B. Godfrey, "Enforcing customizable consistency properties in software-defined networks," in *USENIX NSDI*, 2015, pp. 73–85.

[16] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: scaling flow management for high-performance networks," in *ACM SIGCOMM*, 2011, pp. 254–265.

[17] M. Kuzniar, P. Peresini, and D. Kostić, "Providing reliable fib update acknowledgments in sdn," in *ACM CoNEXT*, 2014, pp. 415–422.

[18] A. Bremler-Barr, D. Hay, D. Hendler, and R. M. Roth, "Peds: a parallel error detection scheme for tcam devices," *IEEE/ACM Transactions on Networking*, vol. 18, no. 5, pp. 1665–1675, 2010.

[19] M. Dobrescu and K. Argyraki, "Software dataplane verification," in *USENIX NSDI*, 2014, pp. 101–114.

[20] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags," in *USENIX NSDI*, 2014.

[21] H. Pan, H. Guan, J. Liu, W. Ding, C. Lin, and G. Xie, "The flowadapter: Enable flexible multi-table processing on legacy hardware," in *ACM HotSDN*, 2013, pp. 85–90.

[22] M. Yu, A. Fabrikant, and J. Rexford, "Buffalo: Bloom filter forwarding architecture for large organizations," in *ACM CoNEXT*, 2009.

[23] X. Jin, J. Gossels, J. Rexford, and D. Walker, "Covisor: A compositional hypervisor for software-defined networks," in *USENIX NSDI*, 2015.

[24] A. S. Iyer, V. Mann, and N. R. Samineni, "Switchreduce: Reducing switch state and controller involvement in openflow networks," in *IFIP Networking*, 2013, pp. 1–9.

[25] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *USENIX NSDI*, 2014, pp. 71–85.