國立臺灣大學電機資訊學院資訊工程學系
碩士論文
Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Taiwan University
Master Thesis

利用生成測試封包偵測軟體定義網路資料平面上的 IP 前綴不符

Detecting IP Prefix Mismatches on SDN Data Plane by Test Packet Generation

董書博
Shu-Po Tung

指導教授：蕭旭君博士
Advisor: Hsu-Chun Hsiao, Ph.D.

中華民國 110 年 6 月
June, 2021

doi:10.6342/NTU202101229

doi:10.6342/NTU202101229

國立臺灣大學碩士學位論文
口試委員會審定書

利用生成測試封包偵測軟體定義網路資料平面上的 IP 前綴不符

Detecting IP Prefix Mismatches on SDN Data Plane by Test Packet Generation

本論文係董書博君（學號 R08922074）在國立臺灣大學資訊工程學系完成之碩士學位論文，於民國 110 年 6 月 25 日承下列考試委員審查通過及口試及格，特此證明

口試委員：蕭旭君

鄭欣明　（指導教授）林忠緯

系主任　洪士灝

doi:10.6342/NTU202101229

doi:10.6342/NTU202101229

誌謝

我要感謝我的指導教授蕭旭君老師一直以來的指導。在網路安全實驗室的 4 年中，我獲益良多。此外，我也要感謝我的家人的支持與鼓勵。

董書博謹致

2021 年 6 月

doi:10.6342/NTU202101229

doi:10.6342/NTU202101229

Acknowledgements

I would like to thank my advisor, Prof. Hsu-Chun Hsiao, for her guidance. I have learned a lot from my 4 years in the Network Security Lab. In addition, I would like to thank my family for their support and encouragement.

Shu-Po Tung

June 2021

doi:10.6342/NTU202101229

doi:10.6342/NTU202101229

摘要

軟體定義網路將資料平面與控制平面分開到不同的設備上以集中管理網路。然而，配置錯誤、硬體上的錯誤或攻擊者都可能導致封包在資料平面上的實際行為和控制平面所定義的規則不同。過去提出的方法透過傳送測試封包來檢驗資料平面是否正確。但他們通常致力於減少測試封包的數量或生成封包的時間以提高效能，因此只假設了簡單的轉發錯誤。本論文識別一個新的錯誤叫做 IP 前綴不符，這個錯誤沒有辦法被過去提出的工具完全檢測到。我們提出了一個封包生成演算法，並且證明我們的方法在最壞的情況下依然可以在每輪的檢測中找到至少一個前綴不符。因此，只要不斷檢測並修復這些錯誤，最終所有錯誤都可以被發現。此外，我們實驗顯示我們的方法有著較好的性能：即使一個交換機包含 50% 的錯誤規則，我們的方法也可以在平均兩輪檢測中找到所有前綴不符。

關鍵字：軟體定義網路，資料平面安全，測試封包生成，匹配欄位錯誤，IP 前綴

doi:10.6342/NTU202101229

doi:10.6342/NTU202101229

Abstract

Software Defined Network separates the data plane and the control plane to different devices for centralizing the network management. However, the actual data-plane behavior of the packets may not match the control-plane rules due to misconfiguration, hardware errors, or attacks. Prior methods verify the data plane's correctness by sending test packets. However, these tools often assume simple forwarding errors, and focus on reducing the packet counts or the packet generation time to improve performance. This thesis identifies a new error type called IP prefix mismatch, which cannot be fully detected by previous tools. We propose an algorithm to generate test packets and prove that our method can find at least one prefix mismatch in each round of detection in the worst case. Therefore, by continuously detecting and fixing them, all errors can be found eventually. Moreover, our experiment shows a much better average-case performance: even if a switch contains 50% of erroneous rules, our method can find all prefix mismatches in an average of two detection rounds.

Keywords: Software Defined Network, Data Plane Security, Test Packet Generation, Match Field Error, IP Prefix

doi:10.6342/NTU202101229

doi:10.6342/NTU202101229

Contents

doi:10.6342/NTU202101229

doi:10.6342/NTU202101229

List of Figures

doi:10.6342/NTU202101229

doi:10.6342/NTU202101229

List of Tables

doi:10.6342/NTU202101229

doi:10.6342/NTU202101229

Chapter 1

Introduction

Software-Defined Network (SDN) is a network architecture that separates the control plane and data plane to different devices, which enables flexible network management. Network devices such as switches or routers forward packets to destinations based on routing rules installed by the network administrators. A routing rule includes a *matching header field* like IP addresses or port numbers and a *forwarding action* deciding where packets match this rule should go. On the data plane, network devices parse and extract the packet header fields and then try to match the predefined rules to transfer packets. However, the actual behaviors may not follow the expected rules due to misconfigurations, hardware errors [14], or even attacks [19]. It is challenging to verify whether network devices violate rules because network administrators have to troubleshoot hundreds and thousands of rules in the network manually.

Probe-based fault detection is a type of scheme to detect inconsistencies between desired network states and the data plane. Its main idea is to send testing packets (probes) with different headers to switches to check if packets match each rule correctly. These schemes can verify the correctness of all rules by monitoring where the packets are forwarded. There are two common optimization strategies for these schemes: the first strategy is to minimize the number of probes because probes share the bandwidth with regular traffic, reducing probe numbers can mitigate the bandwidth overhead. For example, tools such as ATPG [25] and SDNProbe [12] try to maximize the usage of each probe, covering as many rules as possible. The second strategy is to minimize the time to generate probes,

doi:10.6342/NTU202101229

as a fast generation algorithm is desirable for dynamic networks. Prior work likes Monocle [18], RuleChecker [28], and Pronto [29] has shown that finding a unique header in IP space to traverse multiple rules can be modeled as an NP-complete problem, meaning that there is no polynomial solution for now to solve this optimization problem. Therefore, they proposed approximation solutions to reduce the calculation time.

However, prior work mostly considers that forwarding actions of a rule may go wrong. A forwarding action error is that packets match a legitimate match field but are sent to incorrect destinations, such as misdirection and dropping. Detecting each of these errors is an independent event because, except for packets that match the faulty rule, others would not be sent to the wrong destination. As a result, to test if a rule is faulty, one can send a packet that matches the rule and then track its destination. Apparently, this approach has some limitations. When a switch is broken or controlled by attackers, not only the action fields of rules can go wrong. For example, packets may disobey the match field installed by users due to the vulnerability on SDN [7]. However, prior work cannot properly detect errors on the match field because they only focus on forwarding action errors. We summarize the limitations of previous work in the following aspects.

- The threat model defined by prior work is overly simplified. Their approaches only focus on detecting simple forwarding errors on action fields.
- Unlike forwarding action errors, match field errors would affect other benign rules, which leads to a more complex problem. Prior work did not consider this situation.

In this work, we propose a data plane testing system that verifies the correctness of a rule and detects errors on fields that could cause incorrect forwarding behaviors. We focus on detecting errors on IP match fields, one of the header matching fields that is wildly used in the real world. Our solution extends the prior probe-based fault detection scheme so that it can detect errors on both action fields (i.e., dropping and misdirection) and match fields (i.e, IP mismatch) without installing external functions on switches. We found that it needs three probes per rule to guarantee the IP match field of each rule is correct. Finally, we show that our solution can detect faults with zero false negatives.

In summary, this work makes the following contributions:

doi:10.6342/NTU202101229

- We identify a data plane error called prefix mismatch, which has never been discussed in prior work.
- We design a method to detect prefix mismatch with no false negatives and show the correctness of this method.
- We evaluate our system with real and synthetic network datasets, and the results show that multiple prefix mismatches can be detected in two rounds on average.
- We show that our solution is easy to deploy and can be easily optimized by previous work.

doi:10.6342/NTU202101229

doi:10.6342/NTU202101229

Chapter 2

Background

2.1 Software Defined Network

A typical Software Defined Network (SDN) is composed of three main components: controller, SDN switch, and Openflow protocol.

Controller: The controller is a device that is distinct from SDN switches. It controls the flow of packets by installing rules into switches. Because packet routing is centralized to the controller, SDN switches only need to follow the instructions requested by the controller with no additional maintenance.

OpenFlow[16]: OpenFlow is a protocol that is used to transmit information between the controller and SDN switches. It provides a common interface for different controllers to communicate with SDN switches. For example, the controller can install or delete rules on switches by *FLOW_MOD* message defined by OpenFlow, or it can get the statistical data of rules such as the number of packets that have matched a rule.

SDN switch: There are multiple flow tables in an SDN switch. Packets will go through flow tables in sequence and be forwarded to different output ports by flow entries (rules) on the flow tables. Table 2.1 shows the components of a flow entry. Each entry consists of three main fields which directly affect the forwarding behaviors: match, priority, and action. The match field can determine what kinds of packets a rule can match. OpenFlow supports more than thirty types of match fields such as MAC addresses, IPv4 addresses, TCP port numbers, etc. The action field decides how to handle the matching packets.

doi:10.6342/NTU202101229

Match field	Priority	Counters	Instructions (Actions)	Timeouts	Cookie	Flags

Table 2.1: Components of a flow entry.

(a) A sample network including three rules on a switch.

(b) The flow direction of each probe.

Figure 2.1: An example of probe-based fault detection scheme.

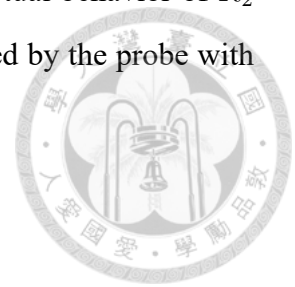
These packets could be directed to other flow tables or be sent to different switches. Finally, unlike traditional switches that match packets according to *longest prefix match*, SDN switches follow the priority field, which strictly defines the order of all rules. In other words, if a packet matches multiple entries, it selects the entry with the highest priority. According to the OpenFlow specification [3], the behavior of SDN switches is undefined when there are multiple entries with the same priority. By default, this situation is prevented by *OFPFF_CHECK_OVERLAP* bit on *FLOW_MOD* messages.

2.2 Probe-based Fault Detection Schemes

Probe-based solutions proactively send test packets to detect faulty flow entries in the network. As we mentioned before, prior work assumes that errors are only on forwarding actions, and match fields of all entries are benign. These schemes generate probes according to the benign match fields and check if a rule is faulty by tracking the destination of each test packet. Figure 2.1 shows an example of how probe-based schemes work. There are three rules on a switch in the network. To check the correctness of rules, these tools generate probes to cover every rule. In this case, if all probes are sent from $P1$, generating

doi:10.6342/NTU202101229

three test packets is enough to verify all rules respectively. If the actual behavior of R_2 is $goto\ P3$ instead of $goto\ P2$, for example, this error can be detected by the probe with header $10.1.1.234$ because of the incorrect destination.



doi:10.6342/NTU202101229

doi:10.6342/NTU202101229

Chapter 3

Problem Definition

In this work, we propose an automatic tool that extends prior probe-based schemes to detect data plane errors. We focus on detecting rules with faulty IP match fields, which has never been discussed in probe-based solutions. Specifically, according to CIDR [5], an IP address is composed of a prefix that is a set of fixed values representing a unique network block and a suffix that indicates the number of bits of the prefix. For example, the prefix of $192.168.1.0/24$ is $192.168.1$ or $11000000.10101000.00000001$. The remaining 8 bits after the prefix can be arbitrary values, indicating the hosts under it. We attempt to narrow down the issue of IP field error into a *prefix mismatch* problem. Our goal is to find this type of error by comparing the expected rules and the actual behaviors tested by probes. In the rest of the sections, we will demonstrate our work can fully detect IPv4 prefix mismatch.

3.1 Threat Model

In our scenario, benign users manage the controller and already know the expected rules on all switches in the network. They also deploy our detection system on the controller, which means that the controller is trustworthy. Rules on a switch are assumed to be *optimized*. That is, they should be reachable by packets. It is a reasonable assumption because users should prevent unreachable rules from being installed on a switch. On the other hand, data plane errors can occur on any switch in the network, resulting in unexpected forwarding

doi:10.6342/NTU202101229

behaviors. However, we assume that errors only occur in *existing* flow entries. In other words, there is no additional rule generated by malicious switches. We leave this issue to future work and will discuss it in Section 6.

Because IP spaces are allocated according to CIDR, IP match fields of rules are based on prefixes in most cases. Therefore, we focus on detecting prefix mismatches on the match field in this work. Formally, an IPv4 match field of a rule r contains a 32-bit IP value V_r and a 32-bit mask M_r that represent the bits this rule should match. If the header of an incoming packet H_i satisfies the equation $H_i \wedge M_r = V_r \wedge M_r$, this packet matches the rule r. In practice, the mask contains a *continuous* sequence of 1's starting from the left-most bit [5]. We do not discuss the case of discontinuous masks because this kind of mask is commonly used in ACL, and prior work [29] has proposed methods to verify ACL rules. The binary representation of r is denoted as $V_r = \{b_0^r, b_1^r, ..., b_{31}^r\}$ from left to right, and we can define the prefix of the rule as the following equation.

$$P_r = \begin{cases} \{b_0^r, b_1^r, ..., b_n^r\}, n = |M_r|, & |M_r| \neq 0 \\ \emptyset, & |M_r| = 0 \end{cases}$$

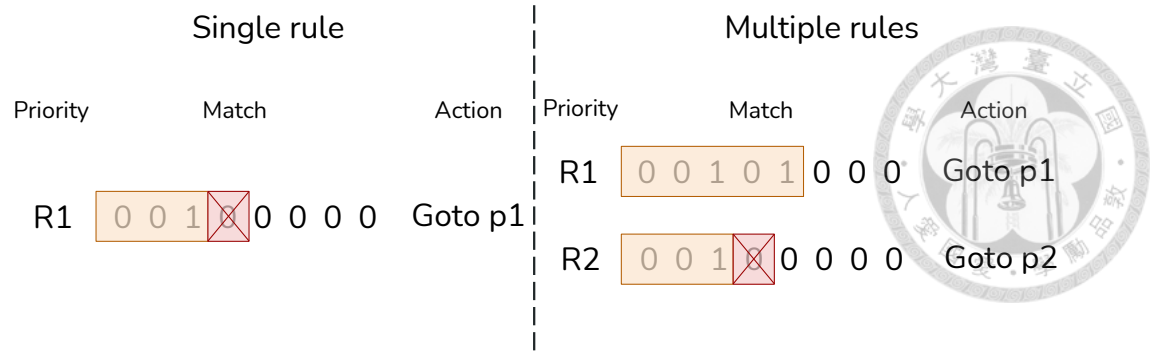
Where $|M_r|$ is denoted as the number of consecutive 1's from the left-most bit of M_r. Assume that the actual match field of the rule r on the data plane is $V_{r'}$ with the prefix $P_{r'} = \{b_0^{r'}, b_1^{r'}, ..., b_k^{r'}\}$. we can categorize prefix mismatches into three types:

- **Wrong value:** $\exists i, b_i^r \neq b_i^{r'}$, where $0 \leq i \leq n$ and $P_r, P_r' \neq \emptyset$.
- **Shrinkage:** $|P_r'| < |P_r|$, and $\forall i, b_i^r = b_i^{r'}$, where $0 \leq i \leq k$ and $P_r \neq \emptyset$.
- **Expansion:** $|P_r'| > |P_r|$, and $\forall i, b_i^r = b_i^{r'}$, where $0 \leq i \leq n$.

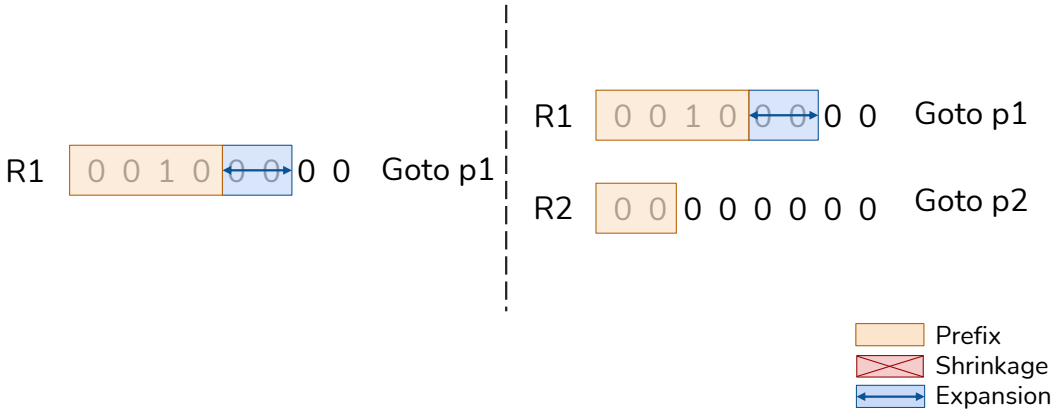
3.2 Prefix Mismatch Examples

To simplify the description, this section will show examples of prefix mismatches on an 8-bit field and explain how prior work fails to detect them. Assume that there is a match field x with value $V_x = 00100000$ and mask $M_x = 11110000$. The prefix of x is $P_x = 0010$.

doi:10.6342/NTU202101229



(a) Prefix shrinkage



(b) Prefix expansion

Figure 3.1: Examples of prefix mismatches of a match field with value 00100000 and mask 11110000.

First of all, we discuss the case of wrong prefix values. Assume that the prefix value of x is incorrect (e.g. $P_{x'} = 0110$). Recalling the prior probe-based solutions, because they send probes without considering the match field errors, the packet headers would be constructed under the legitimate prefix $P_x = 0010$. Therefore, these probes always fail to match $P_{x'} = 0110$, which means that this error can always be detected by packets generated by prior work. It is worth noting that the length of $P_{x'}$ can be arbitrary. For example, $P_{x'} = 1$ and $P_{x'} = 011100$ are both cases of wrong prefix values. However, this error is still detectable by packets under $P_x = 0010$ because there are always some bits in $P_{x'}$ that are different from P_x according to the threat model.

In this work, we focus on prefix shrinkage and expansion because they may not be detected by prior work. Figure 3.1(a) shows an example of prefix shrinkage. In this case, $P_{x'}$ shrinks to 001. Similarly, since the testing packets generated by previous tools are

doi:10.6342/NTU202101229

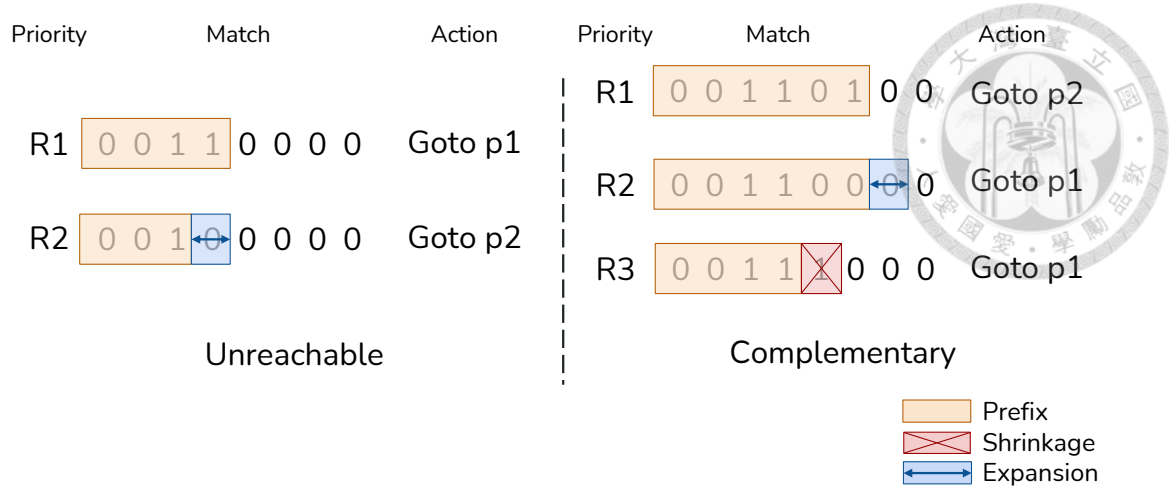


Figure 3.2: Benign cases with prefix mismatches.

always under $P_x = 0010$, it has no chance to detect the extra part $0011xxxx$ covered by $P_{x'}$. In the prefix expansion example in Figure 3.1(b), $P_{x'} = 001000$. Regardless of the number of rules in a switch, previous work cannot fully detect prefix expansion because they do not know which part of the match field is faulty. If the probe header is 00100010, which follows the original prefix P_x and is also under $P_{x'}$, this rule will be labeled as benign by the controller and thus becomes a false negative. Prefix mismatches have several potential risks: prefix expansion would cause a black hole by reducing the available IP range. Prefix shrinkage would affect other rules and lead to unexpected forwarding results. Moreover, because prefix shrinkage may allow more traffic to pass through than the original match field, there might be privacy leaks.

3.3 Benign Cases

We define that rules are benign as long as the forwarding results are correct, in other words, the prefix mismatches do not affect the actual behaviors. Therefore, instead of finding all prefix mismatches, our goal is to find out all unexpected behaviors caused by prefix mismatches. We show some special cases of prefix mismatches that are defined as benign in Figure 3.2: the left side of the figure means that $R2$ has expanded, but the

doi:10.6342/NTU202101229

available part of $R2$ is still the same as before. That is,

$$R2' - R1 = 0010xxxx - 0011xxxx = R2 - R1 = 001xxxx - 0011xxxx = 0010xxxx$$

The other side of the figure indicates that the prefix shrinkage on $R3$ covers the prefix expansion on $R2$. For example, if there is a packet with header 00110010, it would be forwarded to $p1$ by $R3$. Although the packet matches the wrong rule, its forwarding behavior is correct. On the other hand, because the rest of the extra part on $R3$ is overlapped by $R1$, the prefix shrinkage does not affect the actual routing. Both of these cases can forward all packets properly. As a result, we only consider the prefix mismatches that could affect the forwarding behaviors in the following statements.

3.4 Desired Properties

- **Zero false negative.** Given multiple prefix mismatch errors in a network, a detection method should be able to detect at least one of the errors in each testing round, and eventually, all errors can be found and fixed by the controller.
- **Scalability.** A detection method should find out all errors in a few testing rounds even when the number of rules is large with a high probability.

doi:10.6342/NTU202101229

doi:10.6342/NTU202101229

Chapter 4

System Design

In this chapter, we will introduce our probe-based detection mechanism that targets to find out match field inconsistency in the flow entries. In the rest of the sections, we first provide the basic idea of detecting prefix mismatches (§4.1) and then introduce the workflow of our system and each step in details (§§4.2-4.4). Finally, we discuss the situation when there are multiple errors and present a comprehensive analysis (§4.5).

4.1 Detection Method

4.1.1 Prefix expansion

To detect prefix expansion, we generate two probes under *each half* of the target match field. The main idea of this method is to ensure that at least one of the probes will go wrong when the prefix of an entry expands. As shown in Figure 4.1, for example, to test an entry with match field $10.0.0.0/28$, we choose two disjointed parts of IP ranges: $10.0.0.0/29$ and $10.0.0.8/29$ to generate test packet headers. Because the faulty IP range can be any part smaller than the original one when a prefix expands, we have no idea which part of the match field becomes incorrect. Therefore, we test two maximum subnets of the target so that if the prefix expands to one side, the packet on the other side must experience a faulty behavior. To be more precise, the prefix of $10.0.0.0/28$ denotes as $P_{10.0.0.0/28} = 00001010...00$, where $|P_{10.0.0.0/28}| = 28$. The minimum prefix expansion

doi:10.6342/NTU202101229

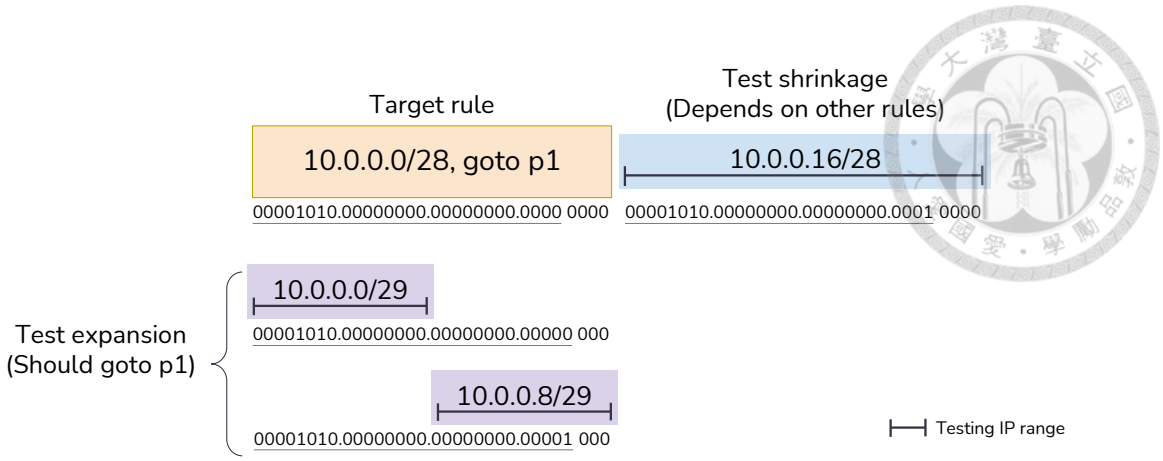


Figure 4.1: An example of our detection mechanism. Detect prefix mismatches on $10.0.0.0/28$.

of the rule can be either $P^l_{10.0.0.0/28'} = 00001010...000$ or $P^r_{10.0.0.0/28'} = 00001010...001$, where $|P^l_{10.0.0.0/28'}| = |P^r_{10.0.0.0/28'}| = 29$. Because all other prefix expansions are subnets of $P^l_{10.0.0.0/28'}$ or $P^r_{10.0.0.0/28'}$, we can detect the error by the probe under one of the minimum prefix expansion.

4.1.2 Prefix shrinkage

When the prefix of an entry shrinks, the size of the IP range is at least two times larger than before because the number of arbitrary bits in the match field increases at least by one. Our target is to find the minimum prefix shrinking range so that no matter how large the extra IP range is, we can always send probes under faulty parts. In Figure 4.1, because the minimum prefix shrinkage of this entry is $10.0.0.0/27$, or denoted as $P_{10.0.0.0/28'} = 00001010...0, |P_{10.0.0.0/28'}| = 27$, all prefix shrinkages are supernets of $P_{10.0.0.0/28'}$. Therefore, the probe under $10.0.0.16/28$, which is *the other half* of the target rule $10.0.0.0/28$ and in the range of the minimum prefix shrinkage, can always detect the fault when the prefix shrinks. It is worth noting that when we test prefix shrinkage, the expected behavior depends on other entries with lower priority. If the test packet under $10.0.0.16/28$ matches an entry that also forwards packets to $p1$, we cannot detect the prefix shrinkage by comparing the different forwarding behaviors. As a result, we need to ensure that when

doi:10.6342/NTU202101229

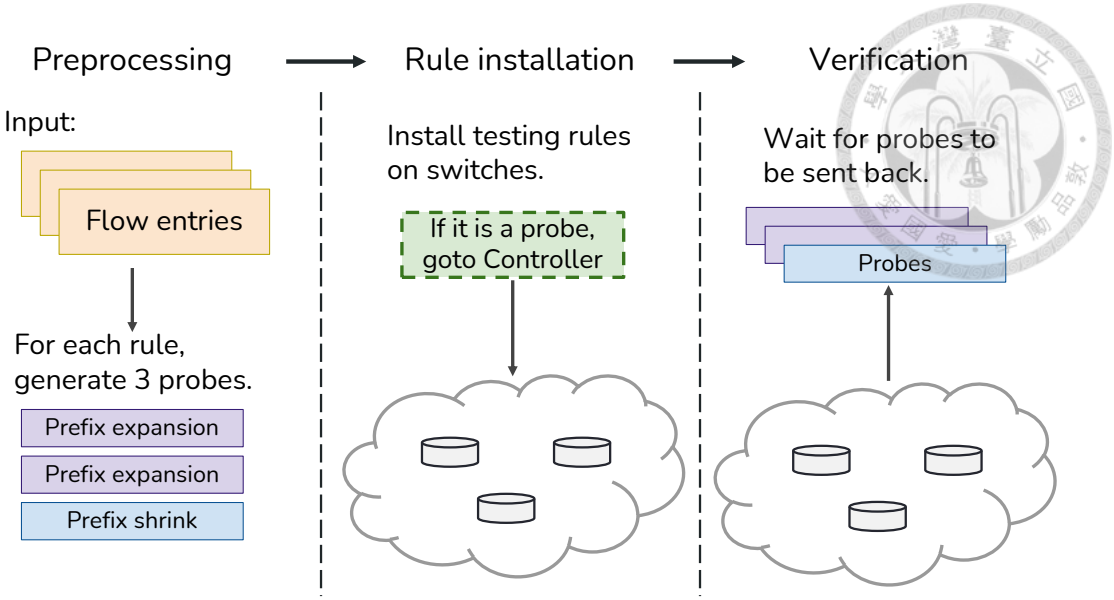


Figure 4.2: Workflow of our detection system.

testing prefix shrinkage, the expected behavior of the probe must not be the same as the action field of the target rule. We will describe more details in Section 4.3.

4.2 System Overview

As Figure 4.2 shows, there are three steps in our system, namely, preprocessing, rule installation, and verification. In the preprocessing step, the system generates test packets according to flow entries on each switch. Because the IP ranges of flow entries with higher priorities may overlap with lower ones, it is difficult to find an available header that avoids other rules and exactly matches the target rule. We leverage an SAT solver, which is widely used in previous work [12], [18], [28], to find available headers. After generating test packets, the next step is to check the forwarding behaviors of these packets by installing *testing rules* on switches. The purpose of testing rules is to catch the probes that behave correctly and then send them back to the controller. If test packets are forwarded to the wrong destination or dropped by switches because of faulty entries, these misdirected packets cannot go back to the controller. In the end, the controller can check whether all entries work properly by waiting for test packets to be sent back.

doi:10.6342/NTU202101229

4.3 Preprocessing

Preprocessing is the most important step in our system. The purpose of this step is to generate packets for each entry to test prefix expansion and shrinkage. However, the problem of finding an available header for an entry is hard to solve because we need to avoid matching rules with higher priorities. It is proven to be a well-known NP-complete problem [23]. In practice, we can use an SAT solver such as PicoSAT [4] to calculate available headers for each rule.

Algorithm 1 Generate headers (prefix expansion)

Input: A set of rules of a switch $R = \{r_1, r_2 ... r_n\}$
Output: A set of headers, H_{expand}, used to test prefix expansion.

1: **function** GENERATE_HEADERS($header, H_{total}$)
2: **if** $header.mask = 32$ **then**
3: **return** $header, null$
4: **end if**
5: $header_l, header_r \leftarrow Split(header)$ ▷ split the header into 2 maximum subnets
6: $h_l^{solved} \leftarrow (header_l \wedge \neg H_{total}).solveSAT()$ ▷ get a header by SAT solver
7: $h_r^{solved} \leftarrow (header_r \wedge \neg H_{total}).solveSAT()$
8: **if** $h_l^{solved} = null \wedge h_r^{solved} = null$ **then**
9: **return** $null, null$
10: **else if** $h_l^{solved} = null$ **then**
11: **return** GENERATE_HEADERS($header_r, H_{total}$)
12: **else if** $h_r^{solved} = null$ **then**
13: **return** GENERATE_HEADERS($header_l, H_{total}$)
14: **else**
15: **return** $h_l^{solved}, h_r^{solved}$
16: **end if**
17: **end function**
18: sort R in decreasing priority order
19: $H_{total} \leftarrow null$
20: **for** each $r \in R$ **do**
21: $h_l, h_r \leftarrow$ GENERATE_HEADERS($r.header, H_{total}$)
22: $H_{expand}.add((h_l, h_r))$
23: $H_{total} \leftarrow H_{total} \vee r.header$
24: **end for**

Algorithm 1 shows the process to generate headers to test prefix expansion. The input is a ruleset of a switch; the output is a set of packet headers. The algorithm first sorts all rules by priority (line 18) and initializes an empty set H_{total} to store rules that have been processed (line 19). Line 20-25 iterates each rule to generate two available headers

doi:10.6342/NTU202101229

by function GENERATE_HEADERS. In this function, we first split the target rule into two equal subnets (line 5) to obtain minimum prefix expansion ranges, and then obtain headers for each subnet by an SAT solver (line 6-7), where $header \wedge \neg H_{total}$ is a set that avoids any IP range with higher priority but matches the target range $header$. If both h_l^{solved} and h_r^{solved} are unsatisfiable, it means that the target rule is unreachable (line 8-9). On the contrary, if there are solutions under both subnets (line 14-15), the results will be returned. Note that the function must obtain two available headers before returning the results except for the case of $header.mask = 32$ (line 2-4). As we mentioned above, an IP range is unreachable if $header \wedge \neg H_{total}$ is unsatisfiable. When we send a probe to an unreachable subnet, this packet never matches the target rule. In this case, the probe becomes useless to detect prefix expansion, which leads to false negatives. Therefore, we have to avoid the unreachable part and split the reachable subnet again (line 10-13). In the worst case, the function runs at most 32 times for an IPv4 rule.

Algorithm 2 Generate headers (prefix shrinkage)

Input: A set of rules of a switch $R = \{r_1, r_2 ... r_n\}$, an array H_P storing the union of rules of each output port

Output: A set of headers H_{shrink}, used to test prefix shrinkage.

```
1:  function GENERATE_HEADERS(header, H_total, H_port)
2:      if header.mask = 0 then
3:          return null
4:      end if
5:      h_e ← Flip(header)                    ▷ flip the right most bit covered by mask
6:      h_e^solved ← (h_e ∧ ¬H_port ∧ ¬H_total).solveSAT()
7:      if h_e^solved = null then
8:          return GENERATE_HEADERS(header ∨ h_e, H_total, H_port)
9:      else
10:         return h_e^solved
11:     end if
12: end function
13: sort R in decreasing priority order
14: H_total ← null
15: for each r ∈ R do
16:     h_e ← GENERATE_HEADERS(r.header, H_total, H_P[r.port])
17:     H_shrink.add(h_e)
18:     H_total ← H_total ∨ r.header
19: end for
```

The algorithm of generating headers to check prefix shrinkage is shown in Algorithm

doi:10.6342/NTU202101229

2. The input of this algorithm includes a ruleset of a switch and an array H_P that stores the union of rules of the different output ports. H_P can be obtained in linear time by iterating all rules once. Line 13-19 is the same as Algorithm 1, in each loop, we calculate a unique header for a rule and then store the rule into H_{total}. In the function, line 5 flips the rightmost bit of the rule covered by the mask to get the minimum prefix shrinkage range. Before solving SAT, we define our constraints as finding a header that avoids rules with higher priority and whose expected behavior (output port) is not the same as the target's (line 6). If the formula is unsatisfiable, the extra IP range from the prefix shrinkage either is unreachable or has the same behavior as the target. Line 7-10 keeps shrinking the prefix until getting a header under a minimum *available* prefix that the target rule can shrink. If a prefix is equal to zero, it means that there is no extra IP range that can be caused by prefix shrinkages. Therefore, We do not have to generate packets for this rule (line 2-4).

4.4 Rule Installation & Verification

To track where test packets are forwarded, we install additional testing rules [12, 28] to collect packets back to the controller, which has the ability to analyze the network and make decisions. In practice, some prior work [25, 29] requires benign hosts to send and receive test packets. In this work, we simulate this scenario by sending packets from the controller and installing testing rules on switches to receive packets. Therefore, we do not consider errors in testing rules. Though we do not simulate benign hosts, our method can still detect prefix mismatches properly. We discuss the rule installation and verification process in the following two aspects:

- **Prefix expansion.** To check whether a prefix expands, we need to ensure both packets generated by Algorithm 1 are forwarded to the correct destination. Therefore, we install the testing rule on the destination switch so that if probes are sent to the wrong switch, the controller would not receive them from the testing rule.

- **Prefix shrinkage.** When testing the prefix shrinkage of a rule, we send a probe whose destination is different from the target rule. We install the testing rule on

doi:10.6342/NTU202101229

the destination switch of the target rule so that if the controller receives the probe from this switch, we know that the probe is directed to the wrong destination by the expanded prefix.

4.5 Multiple Errors

In this section, we will prove that our solution can always detect at least an error even if there are multiple prefix mismatches. Assume that there are N rules with decreasing priority order on a switch denoted as $\{r_1, r_2, ..., r_N\}$.

Theorem 1 *The first prefix shrinkage (with the highest priority) on a switch is always detectable by our solution.*

Proof: We denote the first prefix shrinkage as r'_s, where $1 \leq s \leq N$. If there is no other errors (i.e. prefix expansions) above r'_s, Algorithm 1 guarantees that the generated probe pkt can avoid all other rules with higher priorities and match r'_s, denoted as,

$$pkt \in H = r'_s \wedge \neg \bigcup_{i=1}^{s-1} r_i$$

Therefore, our system can detect r'_s as if the switch contains only one error. On the other hand, assume that a set of rules containing prefix expansions is above the first prefix shrinkage, denoted as E. The actual available IP space for the probe of testing r'_s becomes,

$$H' = r'_s \wedge \neg(\bigcup_{i=1, i \notin E}^{s-1} r_i \vee \bigcup_{j \in E} r'_j)$$

According to the threat model, if a prefix P expands, the expanded prefix P' always satisfies two conditions: the first condition is $|P'| > |P|$. The second one is that all bits from the left-most bit to the $|P|$-th bit in P' are the same as P. These conditions guarantee that the IP range of the expanded prefix is always a subset of the original one. Therefore,

$$\begin{array}{rlcl} & (\bigcup_{i=1,i\notin E}^{s-1} r_i \vee \bigcup_{j\in E} r'_j) & \subset & \bigcup_{i=1}^{s-1} r_i \\ \Rightarrow H' = & [r'_s \wedge \neg(\bigcup_{i=1,i\notin E}^{s-1} r_i \vee \bigcup_{j\in E} r'_j)] & \supset & (r'_s \wedge \neg \bigcup_{i=1}^{s-1} r_i) \quad = H \end{array}$$

doi:10.6342/NTU202101229

Because the IP range of H' is a superset of H, our algorithm can still generate test packets to detect the first prefix shrinkage by H without being affected by the prefix expansions with higher priorities.

At this point, we have proved that our method can at least detect the first prefix shrinkage on a faulty switch. We can repeat the detection process and keep repairing the first prefix shrinkage until fixing them all. The next step is to prove that we can properly detect prefix expansions. Assume that we already fix all detected prefix shrinkages.

Theorem 2 *If a switch only contains prefix expansions, the last prefix expansion (with the lowest priority) is always detectable by our solution.*

Proof: Assume the last prefix expansion as r'_e, where $1 \leq e \leq N$. Algorithm 2 generates probes under the following constraint,

$$H = r'_e \wedge \neg \bigcup_{i=1}^{e-1} r_i$$

We denote the set of prefix expansions above the last prefix expansion as E. The actual IP range affected by other prefix expansions can be denoted as $H' = r'_e \wedge \neg(\bigcup_{i=1, i \notin E}^{e-1} r_i \vee \bigcup_{j \in E} r_j)$. Because Theorem 1 shows that the IP range of a prefix expansion is a subset of the original one, H' is a superset of H. Therefore, we can still detect the last prefix expansion by probes under H.

The above proofs show that, in any case, our solution can detect at least one prefix mismatch. Therefore, we can *gradually* find out prefix mismatches on a switch by repairing some detected errors in each testing round and apply our solution again. Eventually, all errors can be fixed. Chapter 5 will show detailed performance evaluation.

doi:10.6342/NTU202101229

Chapter 5

Evaluation

5.1 Implementation

We implement our system on a Ryu controller [6], and all modules are written in Python. The SAT solver we use in the packet generation module is PicoSAT [4]. To simulate the SDN network, we generate and manage Open vSwitch [2] instances by Mininet [1] on a server with a 2.53 GHz Intel Xeon CPU.

Dataset. We run the experiments on two datasets to demonstrate that our system can apply to real networks and satisfy scalability. The first one is the Stanford backbone network, which includes configurations of 16 backbone switches. It is a public dataset that was widely used in previous work [11, 18, 24, 25, 28, 29]. To evaluate the scalability of our system, we leverage Classbench [22], a tool to generate synthetic rule sets according to real flow distributions and configurations. However, because Classbench was no longer maintained years ago, in practice, we test our system by a recently published tool called Classbench-ng [15], which is based on Classbench but can generate rules closer to recent network configurations.

Evaluation metrics. We do not evaluate the performance of our system by the detection delay of finding all errors because the delay strongly depends on the packet sending rate and the network bandwidth. Evaluating delay time on a virtual network environment generated by Mininet is biased. Therefore, we define an evaluation metric as **the number of rounds to detect all prefix mismatches**. Although prefix mismatches could overlap

doi:10.6342/NTU202101229

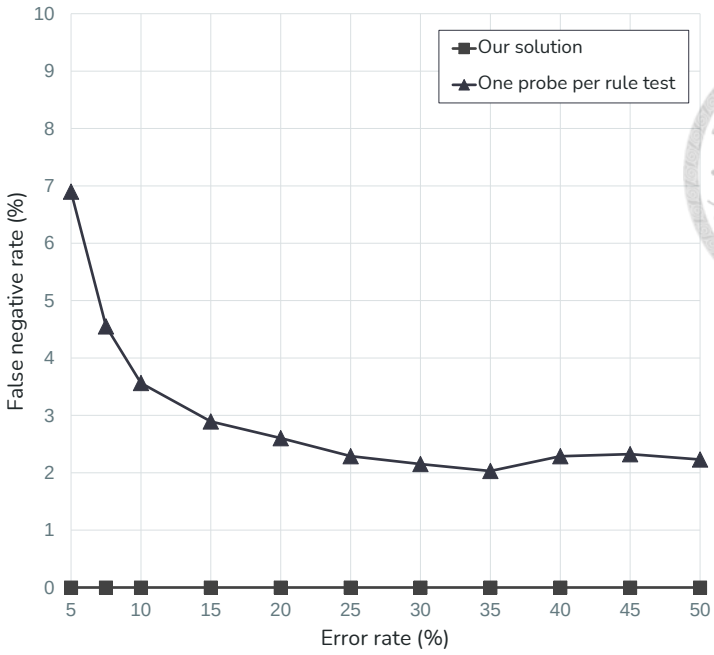


Figure 5.1: False-negative rates of detection of prefix mismatches.

each other and result in false negatives on some rules, we have shown in Section 4.5 that our method can always detect at least one error. We assume that the users can *fix detected errors* in each detection round. Eventually, all prefix mismatches will be found. *The repairing process* is defined as that users would collect information of misdirected probes in each round and attempt to fix the corresponding rules no matter they are true or false positives. After fixing all detected faults, users can start the next detection round. On the other hand, we also evaluate **packet generation time** to show the delay the system needs to prepare for the detection.

5.2 Comparison with Prior Schemes

We implement a naive method to detect forwarding action errors by testing each rule by a probe. To test a rule, this method randomly generates a packet under the legitimate match field. We compare this method with our approach in the following two experiments. We test these methods on a ruleset with about 200 rules. Each data point is an average of 100 tests.

False negative rates: In this experiment, there are both prefix shrinkages and prefix

doi:10.6342/NTU202101229

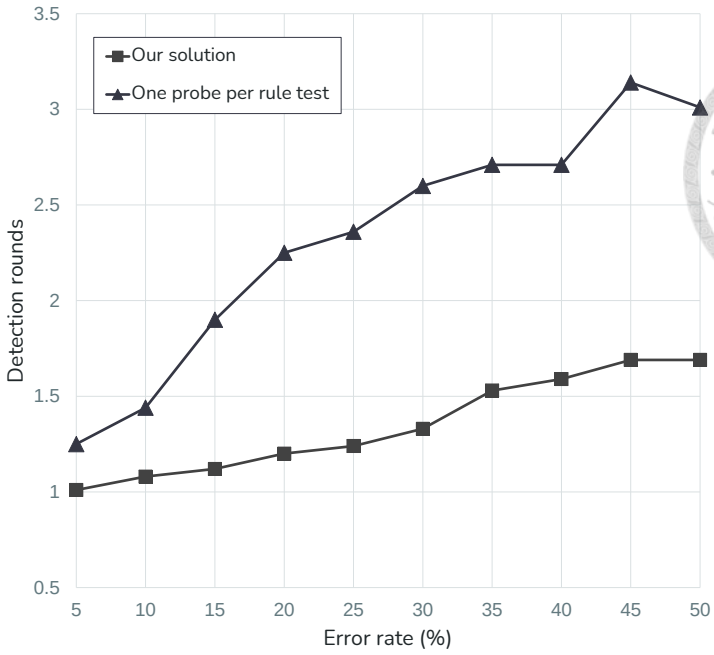
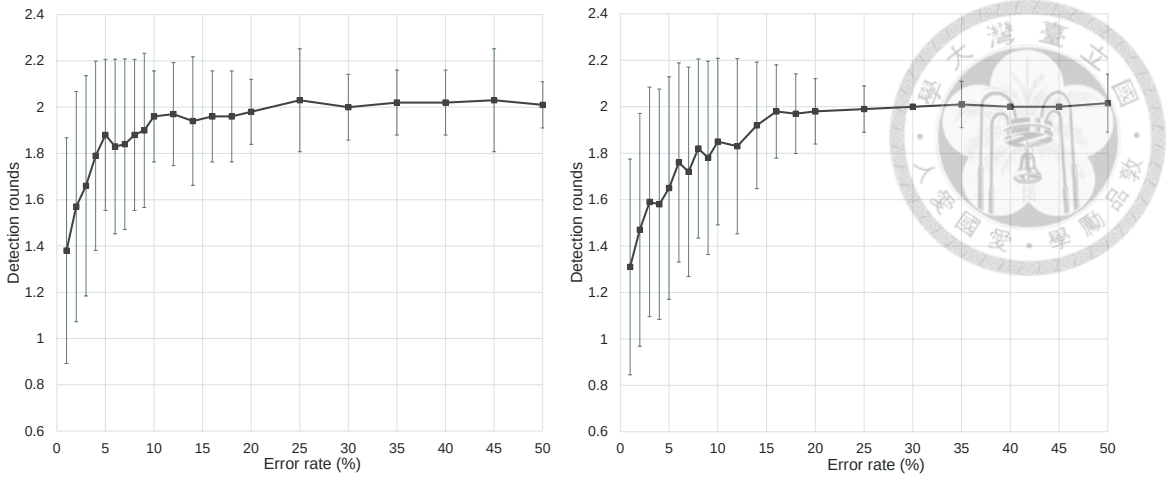


Figure 5.2: Detection rounds for different error rates on a switch containing only prefix expansions.

expansions on a switch. In each test, we repeat the detection process until all errors are detected or none can be detected. The result in Figure 5.1 shows that our method can detect all prefix mismatches with zero false negative rates. It is worth noting that the naive method has lower false negative rates of detection for higher error rates. It is because that when there are more errors, the chance of a probe experiencing a fault also increases. However, this method still cannot detect all prefix mismatches.

Detection rounds: Section 3.2 shows that prior work cannot *fully* detect prefix expansions because the generated probe may test under the benign IP range of a rule instead of the faulty part. That is, prior schemes cannot guarantee that errors would be detected in every round. In this experiment, we assume that there are only prefix expansions on a switch. We evaluate the average detection rounds a method needs to detect and fix all prefix expansions. Figure 5.2 shows that our approach costs 2 times fewer rounds than the prior method to detect and fix all prefix expansions.

doi:10.6342/NTU202101229



(a) 843 rules (b) 2458 rules

Figure 5.3: Detection rounds for different error rates.

5.3 Performance Evaluation

Different numbers of errors: To show the scalability, we test our method with larger rulesets. We randomly make the prefix of some rules shrink or expand and use *the repairing process* mentioned above to fix detected errors. As shown in Figure 5.3, we can see that even if the error rate is up to 50%, all errors can be repaired in an average of two rounds, which means that most of the time, false negatives caused by multiple prefix mismatches can be detected and fixed in a few detection rounds. Even when the number of rules is up to 2000, the average detection rounds is not greater than 2.

The entropy of output ports: The other factor that could affect the performance is the output ports of rules. Our method detects errors by comparing the difference between the actual behaviors and the expected rules. The probes may experience a false negative when two errors with the same output ports overlap each other. On the contrary, our method can detect all errors in one round when all rules on a switch have different output ports because overlap parts caused by faulty rules would always have different behaviors from the original rules. Figure 5.4 shows the rounds to detect errors on a fixed size ruleset with varying entropy values of output ports. We can see that the lower the entropy value is, the higher the detection rounds are, which means that the probability of false negatives in one round increases when the output ports of rules are mostly the same. However, we can see

doi:10.6342/NTU202101229

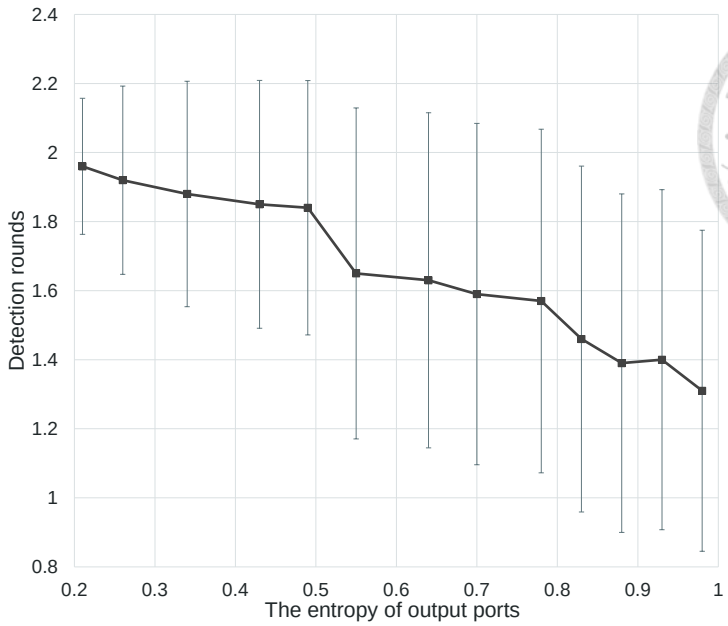


Figure 5.4: Detection rounds for varying entropy values of output ports. (200 rules, 10 % error rate.)

	Prefix expansion		Prefix shrinkage	
Number of rules	Generation time	Avg. number of recursions	Generation time	Avg. number of recursions
2458	13.05	1.02	11.86	2.10
869	4.87	1.00	4.70	1.82
203	1.34	1.00	1.86	4.37
202	1.18	1.00	0.81	1.39
145	0.95	1.01	0.57	1.35

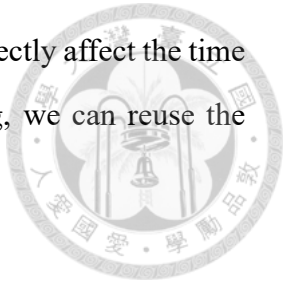
Table 5.1: Packet generation time (sec).

that our method can still detect and fix all errors in 2 rounds on average.

Packet generation: Table 5.1 summarizes the packet generation time and the average number of recursions to generate probes. We can see that it takes about one recursion on average to generate probes for prefix expansions per rule. As we mentioned before, only when one of the split subnets is unreachable, Algorithm 1 repeats the recursion function to find two reachable headers. Therefore, if the average number of recursions is close to one, it means that rules seldom overlap each other. On the other hand, generating headers to test prefix shrinkage should ensure the expected output of probes is different from the target action. If too many adjacent rules have the same output, it will slow down the

doi:10.6342/NTU202101229

packet generation process of Algorithm 2 to find an available probe. As shown in the third and fourth rows of Table 5.1, the number of recursions would directly affect the time to generate packets. Although packet generation is time-consuming, we can reuse the result in each detection round to reduce the time overhead.



doi:10.6342/NTU202101229

Chapter 6

Discussion

6.1 Optimization

In this work, we propose a novel approach to detect prefix mismatches on switches. However, the number of probes is three times larger than the ruleset, which is heavy overhead in the network. Though we leave the optimization to future work, we show that our solution can still be optimized by previous work. To reduce the number of test packets, we need to consider the topology information. Section 2.2 has shown the basic method of prior probe-based schemes [12, 28, 29]. In practice, these schemes calculate a path for each probe to travel as many rules as possible to reduce duplicated testing. To take advantage of these tools, we provide the network topology and a *composite* ruleset for each switch. This ruleset is used to make prior work generate three probes for each rule as if there were three rules. To generate a *composite* ruleset, we preprocess the original ruleset by splitting each rule into three rules. Two of them are used to test prefix expansions, and the other is used to test prefix shrinkages. We can leverage Algorithm 1 to find the maximum *available* subnets for each rule. These subnets can be converted to two disjointed rules with the same output as the original one. On the other hand, to generate rules for prefix shrinkages, we can use Algorithm 2 to generate a header set that can probe the extra part caused by prefix shrinkages. We can leverage H_{port} mentioned in Algorithm 2 to decide the output of the header set. With this ruleset, the number of packets can be optimized by prior probe-based tools.

doi:10.6342/NTU202101229

6.2 Other Errors

6.2.1 Forwarding action errors

As discussed in Section 1, forwarding action fields are frequently considered and detected in the prior probe-based work [18, 20, 29]. In general, forwarding errors mean that packets match a legal match field but the destination is unexpected. Prior tools detect this error by sending a test packet that matches the target rule. If this packet does not go to the expected destination, there are forwarding errors on the rule. Because packets generated by Algorithm 1 are always under the match field of the target rule, we can detect all forwarding errors on a switch. To show that our system can find hybrid errors without false negatives, we need to ensure the theorems mentioned in Section 4.5 still work even with the interference of forwarding errors. Assume that there are N rules with decreasing priority order on a switch denoted as $\{r_1, r_2, ..., r_N\}$.

The first prefix shrinkage on a switch is detectable. According to Theorem 1, the available IP range of detecting the first prefix shrinkage r'_s can be denoted as $H = r'_s \wedge \neg \bigcup_{i=1}^{s-1} r_i$. We define the set of all other errors (i.e. prefix expansions, forwarding errors) above the first prefix shrinkage as E'. Because the match field of a rule with forwarding errors remains the same and the IP range of an expanded prefix is shown to be a subset of the original one in Theorem 1, the IP range of testing r'_s affected by E' can be denoted as,

$$H' = r'_s \wedge \neg(\bigcup_{i=1, i \notin E'}^{s-1} r_i \vee \bigcup_{j \in E'} r_j) \supseteq H$$

Therefore, our method can still detect the first prefix shrinkage by H. On the other hand, if the first prefix shrinkage also contains forwarding errors, it can be easily detected because probes generated by Algorithm 1 would be forwarded to incorrect destinations.

If a switch only contains prefix expansions and forwarding errors, the last prefix expansion is detectable. Assume that the last prefix expansion is r'_e and the set of errors above r'_e is E'. E' includes both prefix expansions and forwarding errors. The IP range affected by E' has been proven to be a superset of H. Therefore, we can still detect the

doi:10.6342/NTU202101229

last prefix expansion by probes generated by Algorithm 2.

As a result, our system can still gradually fix all prefix mismatches even if there are hybrid faults.

6.2.2 Priority reordering & rule missing

Priority reordering and rule missing have been discussed in [28]. We can see that rule missing is a special case of prefix expansion because both reduce the available IP range of a rule. Therefore, probes generated by Algorithm 1 can detect rule missing. Even when there are multiple errors, the properties discussed in Section 4.5 remain the same.

Priority reordering changes the order of rule matching, which would make rules partially faulty. To detect this error, the system needs to consider the dependency between rules to avoid false negatives. However, we show that if priority reordering only occurs in a prefix-based ruleset, our solution can detect it without false negatives. A prefix-based ruleset is a set that every rule follows the CIDR [5], dividing IP blocks by prefixes. Assume that some rules swap on a switch. Because the total number of rules is the same as before, we can consider all affected rules as hybrid errors that include prefix mismatches and forwarding errors. When two rules swap, the match field of these rules should become wrong prefix value, prefix expansion, or prefix shrinkage. The action fields could be incorrect either. We have already proved that we can detect hybrid errors without false negatives in the previous section. Therefore, our method can properly detect priority reordering in a prefix-based ruleset.

6.2.3 Other match field errors

OpenFlow supports match fields for protocols such as TCP, UDP, VLAN, and ARP. In this work, we only focus on detecting errors on the IP match field. Because the IPv6 network space is also classified by prefixes, we can extend our solution to support IPv6 prefix mismatches detection. To detect IPv6 match fields, two parts of the algorithms would be affected. The first part is the SAT solving time will increase because we need to satisfy the formula with 128 instead of 32 variables. The second part is the maximum number of

doi:10.6342/NTU202101229

recursions in Algorithms 1 and 2 will become 128 in the worst case. Despite the reduced performance of packet generation, we can still detect IPv6 prefix mismatches without false negatives.

6.2.4 Additional rules

Rule insertion is not discussed in this work because it is quite difficult for test packets to probe an additional rule within the 32-bit IP field. For example, if a malicious rule inserted in a switch only reroutes a specific IP address to different destinations, the only chance we detect the rule is to send probes with the same address. To solve this issue, a possible solution is to gather statistical data on switches and track whether the traffic is as expected [27].

6.2.5 More complex settings

The OpenFlow specification [3] provides several sophisticated functions for users to dynamically adjust their networks. One can design a complex flow control on multiple flow tables in a switch. For example, the *set-field* action can modify the packet headers. These functions make probe-based solutions more difficult to detect inconsistency on the data plane. One of the limitations of this work is that we cannot detect rules with arbitrary bitmasks. It is because that prefix-based bitmasks align every network block and thus simplifies the detection process. To address this issue, instead of probe-based solutions, static analysis [17] can reach better performance on detecting misconfiguration of rules with arbitrary bitmasks.

doi:10.6342/NTU202101229

Chapter 7

Related Work

There are several related studies of securing the data plane by proactively sending test packets. Most of those studies aims to optimize the detecting process by reducing the network overhead or speeding up the packet generation. For example, SDNProbe [12] minimizes the packet numbers by solving the packet generation problem on directed acyclic graphs. To lower the traffic caused by probes, the work proposed by Chi et al. [20] periodically sends test packets instead of testing all flow entries at once. Pronto [29] encodes flow entries as Atomic Predicates [24] to reduce the time to generate test packets. Because these tools assume that only forwarding actions of packets could be wrong, they cannot properly detect the header mismatching. RuleChecker [28] focuses on detecting rule missing and swapping, which eliminates false negatives by considering the dependency between rules. However, because prefix mismatches would result in IP ranges never seen before, probes generated according to original rule sets cannot detect them.

Some studies discover unexpected network behaviors by collecting statistical data from network devices. They model the current network states with some mathematical metrices. By comparing the calculated result with a pre-defined threshold, they can determine whether traffic is abnormal or not. The additional bandwidth caused by this kind of work is less than the bandwidth caused by sending probes. However, their threat models have additional constraints. For example, forging statistical data is not allowed so that the calculated results would not be affected by attackers. Moreover, to define a reasonable threshold, these tools require some trusted data as ground truth first. SPHINX [8] assumes

doi:10.6342/NTU202101229

that there are a set of honest switches in the network and uses a similarity metrix to detect malicious devices. FlowMon [9] shows that the critical parameters should be computed when the network contains no faults. FOCES [27] also defines the benign volume vectors before anomalies occur in the network. In short, comparing to probe-based schemes, threshold-based solutions need more requirements to reach better performance.

Instead of monitoring the network to detect anomalies, some work attempts to enforce routing rules by cryptographic operations. [21] installs Path Validation Component on each switch, which uses symmetric keys to verify forwarding information embedded in the headers. By checking the MAC values, switches can notify the controller if packets are invalid. REV [26] leverages a similar approach to enforce rules, but it proposes the compressive MAC to reduce the bandwidth between the controller and switches. The crypto-based solution can avoid spoofing information by adversaries. However, to deploy this solution to the network, one needs to install additional functions on each network device, which is hard to complete in a relatively large network.

Another research area statically analyzes rules and finds the potential risks within them. For example, HSA [11] checks rule headers to detect loops and black holes in networks and to verify the reachability between hosts. To apply to rapidly changing networks, several real-time checking tools have been proposed [10, 13, 24]. NetPlumber [10] is a tool based on HSA. It maintains plumbing graphs and dynamically updates them to check network policy in real-time. VeriFlow [13] generates Equivalence Classes (ECs) by slicing the network. It shows that when the network updates, only a small number of ECs would be affected. Therefore, it can track the network policy even if there are dynamic updates. Yang et al. [24] reduce the time and space to analyze rules by dividing rules into Atomic Predicates. Pan et al. [17] define a stronger threat model that can check the misconfiguration on IP match fields with arbitrary bitmasks. Although static analysis can detect misconfiguration, which is efficient and never affects the network traffic, this work cannot detect the inconsistency between rules and actual behaviors.

doi:10.6342/NTU202101229

Chapter 8

Conclusion

Sending test packets effectively detects the inconsistency between actual forwarding behaviors and expected routing rules on the data plane. To simplify testing the data plane, prior work could only detect limited types of errors. In this work, we identify a new type of error called prefix mismatches, which cannot be fully detected by prior probe-based schemes. Our solution can efficiently detect this error without false negatives even in a large ruleset. Furthermore, we conduct a comprehensive analysis of the correctness of our approach. We show that our method complements prior approaches that detect forwarding action errors and priority reordering.

doi:10.6342/NTU202101229

doi:10.6342/NTU202101229

Bibliography

[1] Mininet. *http://mininet.org/*.

[2] Open vswitch. *http://www.openvswitch.org/*.

[3] Openflow switch specification. *https://opennetworking.org/*.

[4] Picosat. *http://fmv.jku.at/picosat*.

[5] Rfc 4632. *https://datatracker.ietf.org/doc/html/rfc4632*.

[6] Ryu framework. *https://ryu-sdn.org/*.

[7] J. Cao, R. Xie, K. Sun, q. li, G. Gu, and M. Xu. When match fields do not need to match: Buffered packets hijacking in sdn. *Network and Distributed System Security Symposium*, 2020.

[8] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann. SPHINX: detecting security attacks in software-defined networks. *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[9] A. Kamisiński and C. Fung. Flowmon: Detecting malicious switches in software-defined networks. *Proceedings of the 2015 Workshop on Automated Decision Making for Active Cyber Defense*, page 39–45, 2015.

[10] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. *Proceedings of the 10th*

doi:10.6342/NTU202101229

USENIX Conference on Networked Systems Design and Implementation, page 99–112, 2013.

[11] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, 2012.

[12] Y. Ke, H. Hsiao, and T. H. Kim. Sdnprobe: Lightweight fault localization in the error-prone environment. *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 489–499, 2018.

[13] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, page 49–54, 2012.

[14] M. Kuźniar, P. Perešíni, and D. Kostić. What you need to know about sdn flow tables. *Passive and Active Measurement*, pages 347–359, 2015.

[15] J. Matoušek, G. Antichi, A. Lučanský, A. W. Moore, and J. Kořenek. Classbench-ng: Recasting classbench after a decade of network evolution. *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, page 204–216, 2017.

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

[17] H. Pan, Z. Li, P. Zhang, K. Salamatian, and G. Xie. Misconfiguration checking for sdn: Data structure, theory and algorithms. *2020 IEEE 28th International Conference on Network Protocols (ICNP)*, pages 1–11, 2020.

[18] P. Peresini, M. Kuzniar, and D. Kostic. Dynamic, fine-grained data plane monitoring with monocle. *IEEE/ACM Trans. Netw.*, 26(1):534–547, 2018.

doi:10.6342/NTU202101229

[19] G. Pickett. Staying persistent in software defined networks. *Black Hat Briefings*, 2015.

[20] Po-Wen Chi, Chien-Ting Kuo, Jing-Wei Guo, and Chin-Laung Lei. How to detect a compromised sdn switch. *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*, pages 1–6, 2015.

[21] T. Sasaki, C. Pappas, T. Lee, T. Hoefler, and A. Perrig. Sdnsec: Forwarding accountability for the sdn data plane. *2016 25th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–10, 2016.

[22] D. E. Taylor and J. S. Turner. Classbench: A packet classification benchmark. *IEEE/ACM Transactions on Networking*, pages 499–511, 2007.

[23] X. Wen, K. Bu, B. Yang, Y. Chen, L. E. Li, X. Chen, J. Yang, and X. Leng. Rulescope: Inspecting forwarding faults for software-defined networking. *IEEE/ACM Transactions on Networking*, 25(4):2347–2360, 2017.

[24] H. Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. *2013 21st IEEE International Conference on Network Protocols (ICNP)*, pages 1–11, 2013.

[25] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, page 241–252, 2012.

[26] P. Zhang, H. Wu, D. Zhang, and Q. Li. Verifying rule enforcement in software defined networks with rev. *IEEE/ACM Transactions on Networking*, 28(2):917–929, 2020.

[27] P. Zhang, S. Xu, Z. Yang, H. Li, Q. Li, H. Wang, and C. Hu. Foces: Detecting forwarding anomalies in software defined networks. *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 830–840, 2018.

doi:10.6342/NTU202101229

[28] P. Zhang, C. Zhang, and C. Hu. Fast testing network data plane with rulechecker. *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, pages 1–10, 2017.

[29] Y. Zhao, H. Wang, X. Lin, T. Yu, and C. Qian. Pronto: Efficient test packet generation for dynamic network data planes. *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 13–22, 2017.

doi:10.6342/NTU202101229