ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Hildur Ólafsdóttir

Supporting Header Field Re-writing for Policy-bound flows in a Software Defined Network

Semester Thesis MA-2012-13
May 2013 to July 2013

Tutor: Dr. Bernhard Ager
Supervisor: Prof. Dr. Bernhard Plattner

Abstract

The emergence of Software Defined Networking (SDN) testbeds has provided researchers with an opportunity to develop and test new network solutions. However this research is impeded by a lack of real user traffic. This is an understandable problem since the testbed operators can't provide users with a guarantee that their privacy won't be invaded or that their quality of service won't be degraded if they send their data through the testbed. Previously a Privacy and Availability Layer (PAL) has been introduced that solves this problem. The PAL offers users that volunteer their traffic to a testbed both privacy and quality of service guarantees. The PAL does however lack support for re-writing header fields which is a feature offered in OpenFlow.
We have extended the PAL to provide support for header field re-writing. In this thesis we discuss the problems with supporting header field re-writing such as keeping track of what is user related header space and preventing re-writing which results in different flows becoming indistinguishable from each other. We present our solution which involves lookup tables to keep track of user related header space. We also present tests that confirm the correctness of our implementation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Publicly accessible Software Defined Networking (SDN) testbeds have cropped up all around the world where researchers can work with SDN and develop new exciting protocols and network solutions. However most of these testbeds suffer from a lack of real user traffic. This is unfortunate since it may be difficult to truly replicate real traffic for experiments that require it. However it is understandable that users are reluctant to hand over their data since there are several privacy and quality of service concerns to consider. A experimenter in an SDN testbed has a lot of control over the traffic and a malevolent researchers could intercept, manipulate, and redirect communication. Network outages could be a result of a malevolent researchers or just an unfortunate mistake but nonetheless result in a degradation of the user experience.
Previously Kotronis et al introduced a Privacy and Availability Layer (PAL) [5] that seeks a compromise between the users' concerns and the experimenters' needs. The PAL offers users a chance to specify which parts of their traffic they are willing to make available. Additionally a user can specify certain constraints, e.g., keeping the traffic payload private. However the PAL has some proposed functions that have not yet been implemented. One such feature that we will focus on in this thesis is support for header field re-writing. This is a feature that researchers are quite likely to want to utilize and thus supporting it is an important step in convincing testbed operators to adopt the PAL.
The goal of this project is to add functionality to the PAL that would allow re-writing without jeopardizing the privacy of users data or degrading the user's quality of service. This is not straight-forward since by allowing header field re-writing it becomes harder to keep track of what is user traffic. In addition, headers can be re-written in ways that can cause different flows to become indistinguishable from each other. This can both be exploited by a cunning attacker to compromise user privacy and quality of service. In this thesis we present our solution which involves lookup tables to keep track of user related header space and we also present tests that confirm the correctness of our implementation.

Chapter 2

Background

To effectively be able to talk about what is needed to support header field re-writing we first need to gain an understanding of our working environment and the PAL, which we will be expanding. In this section we give a description of the PAL and present an overview of both header space analysis, which the PAL relies on to conduct its computations, and OpenFlow.

2.1 SDN testbeds and OpenFlow

A basic SDN testbed consist of a network of switches and physical hosts that are often densely connected. However each experimenter may only see part of the network topology. This topology is defined by a network slicing hypervisor such as FlowVisor [1] which is a proxy between the experimenters' controllers and the switches. A typical setup is depicted in Figure 2.1.

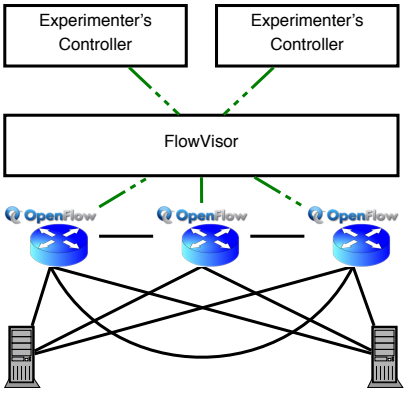

Figure 2.1: Typical SDN testbed setup as presented by Vasileios Kotronis et al [5].

OpenFlow is an open source communications protocol that allows remote control of forwarding tables in network switches, routers, and access points. The PAL is limited to OpenFlow [6] version 1.0 [9] though its methodology is adaptable to newer OpenFlow versions. Each OpenFlow switch has a flow table. This flow table contains a set of flow entries which determine how to handle packets. These flow entries are managed by a controller which can send instructions to insert a new entry or to modify or delete an existing entry. Hereafter we will refer to the insertion, modification or deletion of a flow entry as a flow table modification.
Each flow entry contains header values to match against packets, a priority, and a set actions to apply to matching packets. The actions can for example be to forward a packet out a specified port, drop the packet

or to re-write certain header fields. Additionally each flow entry contains a hard and idle time out and when one of these time outs expires the flow entry is removed automatically.
All packets processed by the switch are compared against the flow table. If a matching entry is found, any actions for that entry are performed on the packet. If no match is found, the packet is forwarded to the controller for instructions on what to do with the packet.
The value of a header field can be wildcarded which means that no value is specified and all packets will positively match this field regardless of their value in this field. An IP field can be partly wildcarded, i.e. it can be a subnet. However when these fields are set by a re-write action it must be set to a single IP address. If more than one rule matches a packet the highest priority rule is applied. If multiple entries have the same priority, the switch is free to choose which rule is applied. A flow entry with no wildcarded field always has the highest priority.

2.2 PAL

To achieve the goal of providing security and availability guarantees a Privacy and Availability Layer (PAL) is introduced along with gateways that control the injection of user traffic into the testbed [5]. The PAL consists of a proxy that is placed between the experimenters' controllers and the network slicing hypervisor. The PAL observes all control plane traffic and has the power to manipulate it to ensure that the users traffic safety is not compromised. *User related traffic* is defined as traffic that originates or is destined for a user that has allowed the use of his traffic for the testbed. A user can specify how different parts of his traffic should be treated and these specifications are referred to as a policy. Figure 2.2 shows a testbed setup with a PAL layer introduced.
To ensure the safety of user traffic the PAL may need to reject a flow table modification if it would cause a policy violation. The PAL rejects a flow table modification by simply not forwarding it from the controller to the switch. However in the event that a flow entry is removed because of a time out the PAL is not able to stop it. To prevent a policy violation the PAL instead short-cuts all traffic that was affected by the removed flow entry directly to the internet. We will also refer to this as rejecting the flow modification. It can also deny the controller access to the traffic payload by replacing it with a nonce before forwarding a packet.
The PAL maintains a virtual copy of each switch's flow table. Using these virtual flow tables it can trace the path through the network that the traffic will take. If along this path a violation of the policy is discovered the flow modification is rejected . This procedure of tracing the traffic's path is hereafter referred to as a path validation. This path validation is only performed when the changes affect user related traffic.
This thesis will not delve deeper into the workings of the PAL unless it's relevant to the subject at hand. For more detailed coverage of the PAL we refer the reader to the paper On Bringing Private Traffic into Public SDN Testbeds [5] and the original code base [1]. Some bug fixes and re-factoring were made to the PAL that were not directly related to implementing re-writing support. These changes are detailed in appendix A.

2.3 Header Space

For the PAL to be able to determine if a rule violates a policy we must both be able to determine what rules affect what traffic and then determine its path through the network. To perform calculations on the traffic we use header space algebra proposed by Kazemian et al [3]. We view protocol headers as a sequence of ones and zeros ignoring the protocol-specific meanings associated with header bits. A header space is a subspace in the $\{0,1\}^L$ space, where L is the upper limit of the header length.
A wildcard expression is a sequence of L bits, where L is the length of the header, where each bit can be either 0, 1 or x. When a bit is a x it means that it can either take the value 0 or 1. Wildcard expressions are used as the building blocks of a header space. Indeed a headerspace can always be expressed as a union of wildcard expressions. Header space algebra can tell us how different header spaces relate to each other, for example their intersection, union, difference or the complement of a header space [3].

[1] `https://bitbucket.org/vkotronis/of_privacy_proxy`

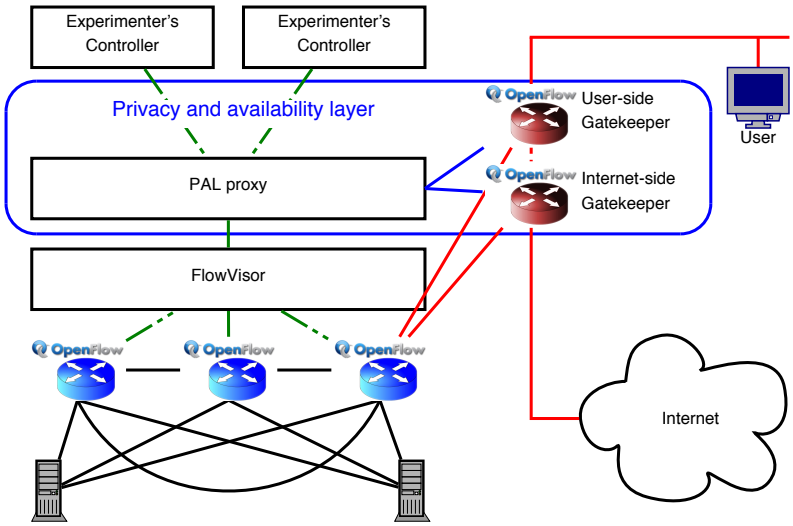

Figure 2.2: Testbed setup with a PAL introduced as presented by By Vasileios Kotronis et al [5].

The match-specifications of the flow entries can be seen as a headerspace consisting of a single wildcard expression where a wildcarded field in the header is represented as a string of x's that is the length of the field. The traffic that a policy governs can also be viewed as a headerspace, by taking the union of wildcard expressions that represent the traffic. Using header space algebra we can calculate if the header space of a flow entry intersects with the header space of a policy and is thus governed by the policy. To perform the algebra calculations we use the library Hassel [2].

[2] https://bitbucket.org/peymank/hassel-public/wiki/Home

Chapter 3

Design

The PAL evaluates flow table modifications by traversing all possible paths that the corresponding header space can travel through the testbed. This is of course only performed if the flow entry that was inserted, modified or deleted impacted user related header space. The PAL relies on the ability to be able to identify what is user related and what policy it is governed by. This is not straightforward when the header space can be changed as we will explain later. Our challenge thus is to enable the PAL to keep track of what is user related header space for every switch in the testbed. To simplify our discussion we will hereafter refer to all possible header space that can arrive at a switch as the *active header space* on this switch. This is defined by all the flow entries that are already installed on the switches throughout the network and how they manipulate and forward traffic.
To keep track of all the user related active header space in the testbed we introduced a lookup table, a LUT, for each switch. A lookup table contains all the active user related header space for each switch, the original header space and their associated policies. The PAL uses the lookup tables to decide when a path validation is necessary. The path validation is also used to maintain the lookup tables. We will now take a look at the challenges and how the path validation and the lookup tables work.

3.1 Challenges

Introducing re-writing of header space causes two issues that need to be dealt with that we will now explore further. First the ability of the PAL to reject flow table modifications if they cause a policy violation is dependent on its ability to identify that a flow entry affects user-related traffic and the governing policy. Up until now this has been straightforward, the header space of the flow entry is examined and compared to that of all policies, and if there is a match the governing policy has been found. However if re-writing actions are allowed identifying user related header space is no longer possible by simply inspecting the header space since a header space may change on its way through the testbed and may no longer match that of the policy governing it. Only on the gateways where traffic enters the testbed can we be sure that the header space the same as the policies.
Lets take a simple example where we assume that the testbed topology is that of Figure 3.1. Lets assume that the header space $hs1$ coming into switch S1 is user related but header space $hs2$ is not. Switch S1 has a flow entry whose actions specify re-writing $hs1$ to $hs2$ and forwarding it to switch S2 and another flow entry whose actions specify re-writing $hs2$ to $hs1$ and forwarding it to switch S3. Now the header space $hs1$ is user related header space for switch S1 and header space $hs2$ is user-related for switch S2. If the flow entries on switch S1 are changed what is user related on switches S2 and S3 may also change.
The second consequence of allowing re-writing of header spaces is that it is possible to re-write different header spaces in such a way that they are indistinguishable from each other. This becomes a problem if these header spaces are then routed to the same switch since there will be no way to separate the traffic again.
Additionally the PAL needs to ensure that traffic that enters the testbed also leaves leaves the testbed with its original header. Otherwise this could cause loss of connectivity for users or someone might try to intercept

the traffic by routing it somewhere else after it leaves the network. However, when header space is re-written it is not as straightforward since different traffic may have been re-written so that it appears that the correct traffic is leaving the testbed when that is infact not the case.

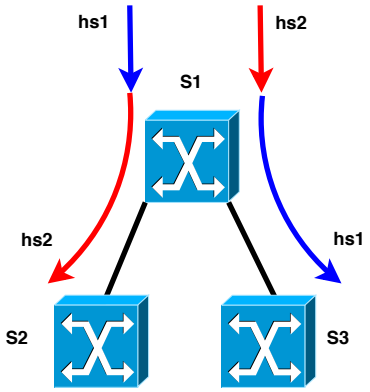

Figure 3.1: The network topology of a testbed. On switch S1 header space hs1 is user related, on switch S2 header space hs2 is user related and there is no user related header space on switch S3.

3.2 Path validation

When a flow entry is inserted, modified or deleted, the PAL first checks if any part of the header space is user-related and if so a path validation needs to be performed on the part of the header space that is user related. To check if the header space is user related the PAL compares the header space of the flow entry to the entries of the affected switch's lookup table. If the switch is a gateway it additionally checks the part of the header space that is not in the LUT against that of currently active policies and if a match is found then a new entry is put into the LUT.
During path validation every possible path that the header space could take is followed. This means that for each switch that is visited during path validation the relevant header space is matched against the flow entries of the switch. Then the actions that would be taken on the actual switch are performed on the header space and a list is kept of what header space is forwarded and where. The path validation then visits each switch in the list and thereby traverses its way through all possible paths in a depth first manner.
Notice that the header space that is forwarded to a switch may be different from the header space of the previous switch. This may happen if the flow entry only forwards part of the original header space or if re-write actions are applied. This requires the PAL to be able to manipulate the header space to emulate the subsequent actions on the switch.

3.3 The task of the lookup table

The lookup tables (LUTs) contain what is user related active header space and the associated policies for each switch. Active header space, is as we said before, all header space that can reach the switch. We need to keep a list of the user-related header space even if the switch has no flow entries that govern this traffic because the traffic can still arrive and flow entries that do govern it may be added later. In that event we must be able to identify it as user related.
During path validation the lookup table of each visited switch is updated with the header space that was forwarded to it. This header space is clearly user related otherwise it would not be part of the path validation. Additionally the original header space that entered the testbed is also stored in each LUT entry.
In the event that the flow table modification that initiated the path validation is rejected these changes to the lookup tables must be reverted. We facilitate this by implementing an interface supporting transactions. This

enables committing changes or rolling back the state of the lookup tables to the last commit.
Table 3.1 shows a summary of the information stored in a LUT entry. Flow entries are stored in a dictionary list and thus any search in a lookup table has a time complexity of O(n) where n is the number of LUT entries. In the remainder of the section we will elaborate on the scenarios that can arise when a flow table modification is performed and why this information is necessary.

header space	The header space that can arrive at the switch
Original header space	The original header space as it was when it entered the testbed
Priority	An integer number specifying the priority of the flow entry or entries governing the header space (field 1). Has the value None if no flow entry is governing the header space.
Children	List of lookup table entries that depend on this entry. If the entry is updated or deleted its children may need to be updated or deleted as well.
Parents	List of lookup table entries that this entry depends on. This entry can't be deleted if it still has at least one parent. If this entry is split up into two or more entries the parents must be informed.
Header field value before re-writing	Keeps a list of header fields and their original value before they were re-written.

Table 3.1: Summary of information in a LUT entry

3.3.1 Flow entry insertion

When a flow entry is added its header space is matched against entries in the lookup table. If there is no positive match then the flow entry can be accepted but otherwise there are a few scenarios that can happen. We will refer to the header space of the new flow entry that matches user related header space as $hs_{matching}$. If there are flow entries already in the flow table that cover all or part of $hs_{matching}$ then the new flow entry may only affect part or none of the user related header space. We will refer to the intersection between an already existing flow entry and $hs_{matching}$ as $hs_{intersection}$. The remaining header space is referred to as $hs_{difference}$. There are four basic scenarios that depend on the respective priorities of the flow entries.

- If no existing flow entries cover the header space $hs_{matching}$ then a path validation needs to be performed on the entire matching header space.
- If the new flow entry has a lower priority than pre-existing flow entries that govern $hs_{intersection}$ then the new flow entry only governs $hs_{difference}$ and a path validation needs only to be performed on this header space.
- If the new flow entry has the same priority as pre-existing flow entries we can't know which flow entry the switch will use. Therefore we need to perform a path validation on the entire header space $hs_{matching}$.
- If the flow entry has a higher priority than the pre-existing flow entries then we need to perform a path validation on $hs_{matching}$. Additionally LUT entries on other switches that were affected by the change may need to be updated or deleted.This is because they may no longer accurately depict what is active user related header space.

Of course there might be the case that one flow entry may cause all scenarios or a subset of them. This happens if it covers the header space of several flow entries where some have a higher, lower or of equal priority as the new flow entry.
Additionally if this switch is a gateway it needs to match the part of the flow entry's header space that is not in its lookup table against all policies. If there is a match this needs to be added to the lookup table and a

path validation needs to be done. To know which scenario a new flow entry falls under, the entries in the LUT contain the priority of the flow entry that governs it.
We need to be able to update LUT entries when the flow entry that caused them is superseded (or modified or deleted). To simplify this we want to easily find all the entries that were affected by a flow entry. During the path validation each affected LUT entry keeps track of what LUT entry in the previous switch corresponds to it, we will refer to this as the *parent* entry. Additionally each entry also keeps a list of all its *children*. Note that a LUT entry may have more than one parent e.g. in the event that the same header space arrives from different sources. Also entries added after the gateway matches a flow entry to a policy have no parent.

3.3.2 Deletion, modification and removal due to time out

We define an *active flow entry* as an entry that could affect how incoming packets are handled. This excludes flow entries that are superseded by flow entries of higher priority.
Once a LUT entry has been added to a lookup table this entry should not be deleted unless traffic with its header space is no longer being routed to this switch. In essence that means that as long as an entry has a parent it should not be deleted. However the entry may be updated or split up into multiple entries, e.g, to reflect the priorities of the flow entries on the switch.
In the event that an active flow entry is modified or removed, that concerns user-related header space, updates on the corresponding LUT entries may need to be performed. There are four basic scenarios that can happen to a LUT entry that is affected.

- The LUT entry has no parents and the switch has no other flow entries that govern the header space. Then the LUT entry can be deleted.
- The LUT entry has no parents but the switch has other flow entries that govern at least part of the header space. Then the LUT entry is updated and possibly split up into multiple entries to reflect this. The remaining header space that is not governed by any flow entry does not need a LUT entry.
- The LUT entry has at least one parent but the switch has no flow entries that govern the header space. Then the priority of the LUT entry is set to None.
- The LUT entry has at least one parent and the switch has flow entries that govern at least part of the header space. Then the LUT entry is updated and possibly split up into multiple entries to reflect this. The remaining header space that is not governed by any flow entry is put in a LUT entry with the no priority set.

As we said a LUT entry may need to be split up. This happens if there are different flow entries of various priorities that govern parts of the header space. For each priority level of the remaining flow entries, a LUT entry is put into the lookup table. This entry has the header space that these flow entries govern that is not already governed by higher priority flow entries. If an entry needs to be updated its children may no longer be its children since it is no longer certain that the relevant header space is routed the same way. Thus the children and their children's children and so forth may need to be deleted if they have no other parent.
As discussed earlier the PAL performs a path validation slightly before a flow entry is about to be removed because of a time out. When this path validation is finished the LUT changes need to be rollbacked since there might be changes between then and the actual removal of the flow entry. Then when the flow entry is actually removed the state of the lookup tables are updated to reflect this removal.

3.3.3 Lookup table updates during path validation

As we discussed earlier lookup tables are updated during path validation and now we will further elaborate on some scenarios that can happen. When the path validation arrives at a switch this switch may already have LUT entries that covers part or the entire the header space.
The path validation can be skipped for the part of the header space that is covered since it has previously

been evaluated and we only need to update the LUT entries. If there are flow entries that govern the remaining header space (that is not covered) the appropriate entries are put into the lookup table and the path validation continues. In the case that there is header space that was neither covered by LUT entries or flow entries an entry is put into the LUT with no priority set. This ensures that when a flow entry matching that header space is added, it will be correctly identified as concerning user related header space.
When we update the LUT entries that were already in the lookup tables they may need to be split up into two entries. One with the header space that intersects with the incoming traffic. This entry should add the parent of the incoming traffic to its parent list since it now is coming from this source as well. The other entry will then contain the remaining header space and will only contain the original parent list.

3.3.4 Example 1

Let's take a simple example to demonstrate how this works. Let hs_{TCP} be a user-related header space, for example all TCP traffic and hs_{SSH}, hs_{SMTP} and hs_{other} are header spaces that are a subset of hs_{TCP} but do not themselves intersect. For example hs_{SSH} is all TCP traffic with transport layer destination port 22, hs_{SMTP} is all TCP traffic with transport layer destination port 25 and hs_{other} is is all TCP traffic which doesn't have transport layer destination port 22 or 25. Figure 3.3 shows a visual representation of theses header spaces. Now let the network be that of Figure 3.2 and the flow entries are those given in Table 3.2a then the resulting lookup tables of the switches would be that of Table 3.2b. Because switch S2 doesn't have a flow entry governing hs_{other} this entry doesn't have a priority. Traffic matching this header space could still be forwarded to switch S2 from switch S1 and when a flow entry governing this header space is added it will be matched against this entry and correctly identified as user related.

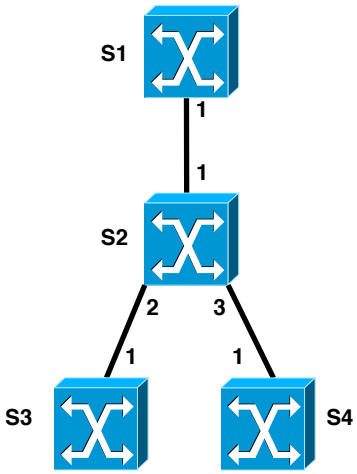

Figure 3.2: The network topology of the testbed

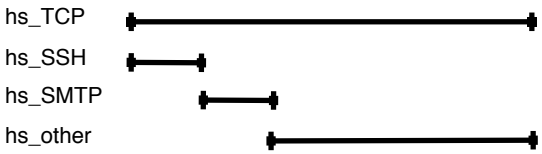

Figure 3.3: Visual representation of header spaces hs_{TCP}, hs_{SSH}, hs_{SMTP} and hs_{other}

3.3.5 Example 2

Now let say that a new flow entry is added to switch S2 that governs hs_{TCP}, has a priority of 2 and it action is to output the traffic through port 1. Now this flow entry superseded the flow entry that previously governed hs_{SSH} but not the one that governs hs_{SMTP} and it also affects hs_{other}. The result will be that of Table 3.2.

Switch	header space	Priority	Actions
S1	hs_{TCP}	1	Output port: 1
S2	hs_{SSH}	1	Output port: 1
	hs_{SMTP}	3	Output port: 2

(a) OpenFlow flow entries of switches

Switch	Entry	header space	Priority	Children (Switch, Entry)	Parent (Switch, Entry)
S1	1	hs_{TCP}	1	{S2, 1}, {S2, 2}, {S2, 3}	
S2	1	hs_{SSH}	1	{S3, 1}	{S1, 1}
	2	hs_{SMTP}	3	{S4, 1}	{S1, 1}
	3	hs_{other}			{S1, 1}
S3	1	hs_{SSH}			{S2, 1}
S4	1	hs_{SMTP}			{S2, 1}

(b) LUT entries of switches

Switch	Entry	header space	Priority	Children (Switch, Entry)	Parent (Switch, Entry)
S1	1	hs_{TCP}	1	{S2, 1}, {S2, 2}, {S2, 3}	
S2	1	hs_{SSH}	2	{S3, 1}	{S1, 1}
	2	hs_{SMTP}	3	{S4, 1}	{S1, 1}
	3	hs_{other}	2		{S1, 1}
S3	1	hs_{SSH}			{S2, 1}
S4	1	hs_{SMTP}			{S2, 2}

Table 3.2: LUT entries of switches

3.4 Attack vectors

Observant readers may have discovered that in its current form this system is still vulnerable to two types of attack. It is possible to re-write two different flows in such a way that they become indistinguishable and hosts can still inject traffic. We will elaborate on this int he following sections.

3.4.1 indistinguishable header space

One consequence of allowing header spaces to be re-written is that different header spaces can be re-written to be indistinguishable from each other. This becomes a problem if these header spaces are then routed so they end up on the same switch. Obviously there will be no way to recover the original header space of this traffic. This could both lead to a loss of connectivity and a savvy attacker can utilize it to intercept user traffic.
To visualize this let the network topology be that of Figure 3.4. Switch S1 has a flow entry that specifies forwarding all traffic with destination IP address 10.0.0.1 to switch S2 and a flow entry that specifies forwarding all traffic with destination IP address 10.0.0.2 to switch S3. The switch S2 has a flow entry that forwards all traffic with destination IP address 10.0.0.1 to switch S4. Switch S3 has a flow entry that re-writes the destination IP address 10.0.0.2 to 10.0.0.1 and forwards to switch S4. If wwitch S4 then forwards this traffic further there will be no way to distinguish traffic destined for 10.0.0.1 from traffic destined for 10.0.0.2. If a controller has a host outside the testbed which should be receiving traffic with IP destination address 10.0.0.1 but not 10.0.0.2 it could be intercepting traffic from the user that should legitimately be receiving it.
To solve this we store the original field value before it was re-written. When a flow reaches a switch which is already receiving the same header space from another source these original field values are compared. If these original values are not the same a policy violation is declared.
However this doesn't solve one remaining problem. If a field that is wildcarded is re-written to take one specific value this could cause flows to become indistinguishable. For example re-writing all IP addresses

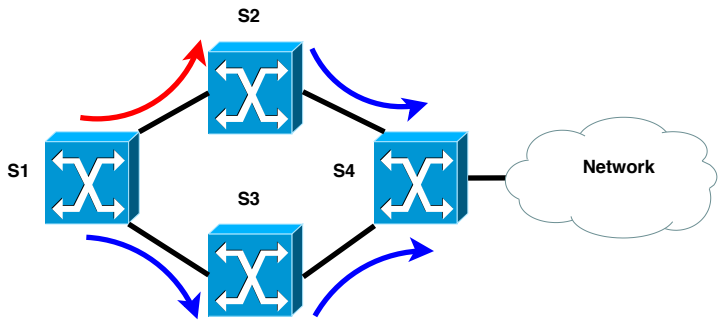

Figure 3.4: The network topology of the testbed. The blue flow is represents traffic with destination IP address 10.0.0.1 and the red flow traffic with destination IP address 10.0.0.2

in a certain subnet into one IP address. To solve this we simply disallow the re-writing of wildcarded fields. There are a few scenarios where such a flow entry may not actually cause an issue. For example if a switch has a flow entry which specifies to re-write a wildcarded field to a specific value but never receives any traffic that matches this flow entry there will not be any issue. There also won't be a problem if the switch only receives traffic where this field has a specific value. The the flow entry will simply re-write one value to another. For example a switch has a flow entry to re-write all IP addresses in the 10.0.0.0/24 subnet into one IP address 10.0.0.1. If this switch never receives any traffic in this subnet or only traffic with exactly one IP address no harm is done since no merging of traffic is takes place. However clearly then having such a wide flow entry that governs traffic that it will not receive is not necessary.
We allow MAC addresses to be exempt from the above restrictions since re-writing of them will not affect user-related header spaces.

3.4.2 Host injection

Observant readers may have discovered that in its current form this system will not necessarily prevent a host from injecting traffic that will wrongfully send traffic to a user. Since even if we keep track of user related header space that comes from the gateways there is a possibility for a host inside the testbed to inject traffic and a complicit controller to later re-write it to appear to be traffic that is approved to be sent to a user. Even if we keep track of all user related header space it would not solve this problem. For example let the network have the topology of Figure 3.5. The gateway, gw, is expecting traffic with header space $hs1$ and has a rule to forward this to a user. Switch S1 receives a rule that specifies to re-write traffic coming from the host to that of $hs1$ and forward it to the gateway. Since the legit traffic has not yet reach switch S2 from the network there is no information indicating that this is user related and thus there is nothing to indicate that there is an issue with accepting this rule.

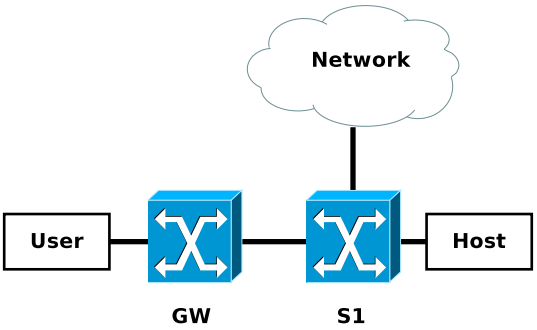

Figure 3.5: The network topology

To prevent this from happening we need to keep track of all active header space coming in to a switch and what the original header space of that traffic was. To achieve this every flow entry insertion or change causes a path validation. However those that are not currently user related do not have a policy attached to them. When such a header space reaches a gateway, the original header space is used to assess whether allowing this traffic would cause a policy violations. This was implemented as an optional feature since it is more expensive and host injection is a limited attack vector.

Chapter 4

Correctness Validation

To ensure that the new implementation acts as expected we ran several tests which were all successful leading us to be confident that the lookup tables and their functionality in the PAL is correct. In this chapter we provide general descriptions and some examples. For a full list and more detail we refer the reader to the code base [1]. We use the library POX [10] to write our controller to perform the tests and Mininet [7] to create a virtual network.
In our test we are using two types of policies, Allow and NoSniff. These were previously defined by Kotronis et al [5]. The NoSniff policy is designed for sensitive data, it specifies that the traffic governed by this policy should never be forwarded to a host in the network and denies the experimenter access to packet payload. The Allow policy on the other hand is designed for traffic that is not sensitive and allows experimenters to access packet payload, including forwarding packets to hosts. In all tests switch S1 acts as the gateway into the testbed.

4.1 Re-write SMTP traffic to appear to be SSH

We tested whether the PAL correctly rejects flow entries that re-wrote header space if this would cause a policy violation. For this test we assume that the topology of the network is that shown in figure 4.1. Here host 1 is a user who has specified that all SSH traffic (TCP traffic with transport layer destination port 22) is not sensitive and should fall under the Allow policy but all other traffic should fall under the NoSniff policy. This is reasonable policy since SSH traffic is already encrypted so an attacker doesn't gain much information from the payload.
To test if our system rejects traffic that violates this policy three flow entries are inserted into our testbed. Lets define TCP traffic from host 1 to host 3 with destination port 22 (SSH traffic) as headerspace hs_{SSH} and TCP traffic from host 1 to host 3 with destination port 25 (SMTP traffic) as headerspace hs_{SMTP}. We chose SMTP traffic in this case since it is not encrypted and if the payload of this traffic were to come into the hands of an attacker it would compromise the privacy of the user.
A flow entry is inserted on switch S2 and S3 to forward packets matching hs_{SSH} out port 2 and a flow entry on switch 1 to re-write hs_{SMTP} to hs_{SSH} and forward it out port 2. These header spaces are defined in Table 4.1a and the flow entries are shown in Table 4.1b.This test was performed multiple times where the order in which each switch received its flow entry was altered. In all cases the PAL rejected the last flow entry as expected.
Additionally we ran similar tests where the flow entry to re-write the header space was insert onto either switch S2 or S3.

[1] https://bitbucket.org/hildur/of_privacy_proxy_with_re-write_support

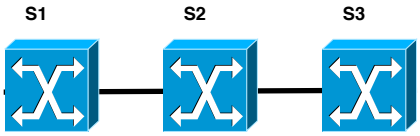

Figure 4.1: For test 4.1 we use a network topology consisting of 3 switches connected serially.

Header space	Definition
hs_{SSH}	TCP traffic from host 1 to host 3 with destination port 22
hs_{SMTP}	TCP traffic from host 1 to host 3 with destination port 25

(a) Header space definitions

Switch	Header space	Actions
S1	hs_{SMTP}	re-write hs_{SSH} to hs_{SMTP}forward to S2
S2	hs_{SSH}	forward to S3
S3	hs_{SSH}	forward to Host 3

(b) OpenFlow flow entries of switches

Table 4.1: OpenFlow flow entries and definitions of header spaces for test 4.1

4.2 Indistinguishable header space

We tested whether the PAL would correctly reject flow entries that result in different header spaces becoming indistinguishable from each other. For these tests we assume that the topology of the network is that shown in Figure 4.2.

4.2.1 Merging

We tested whether the PAL correctly reject flow entries that result in different header spaces being re-written and merged at the same switch such that they become indistinguishable from each other.
The relevant header space is defined in Table 4.2a and flow entries specified in Table 4.2b are inserted. If these flow entries are all allowed it will cause traffic with IP destination address 10.0.0.4 and 10.0.0.2 to be indistinguishable. However the PAL rejected the last flow entry as expected. If the order in which the flow entries are inserted are shuffled the last one will always be rejected since it will create a state where a policy violation occurs.

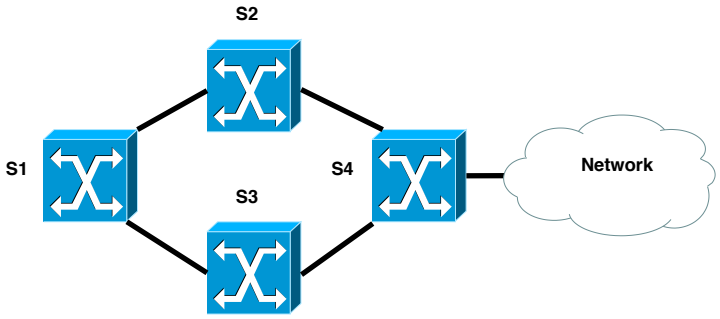

Figure 4.2: For tests 4.2.1 and 4.5.1 we use a network topology consisting of 4 switches connected in a circle.

Header space	Definition
hs_{subnet}	Traffic with destination IP address in the 10.0.0.0/24 subnet
hs_{IP1}	Traffic with destination IP address 10.0.0.4
hs_{IP2}	Traffic with destination IP address 10.0.0.2

(a) Header space definitions

Switch	Header space	Actions
S1	hs_{subnet}	forward to switches S2 and S3.
S2	hs_{IP1}	forward to switch S4.
S3	hs_{IP2}	set destination IP address to 10.0.0.4 forward to switch S4.

(b) OpenFlow flow entries of switches

Table 4.2: OpenFlow flow entries and definitions of header spaces for test 4.2.1

4.2.2 Wildcards re-written into one value

We tested whether the PAL would correctly reject flow entries that re-wrote wildcarded fields into one value. A flow entry was insert onto a switch that specified to re-write all destination IP addresses in the 10.0.0.0/24 subnet into one single IP address. As expected this flow entry was rejected.

4.3 Correct state of lookup tables

We tested whether the lookup tables are correct after the insertion, modification or deletion of certain flow entries. We begin by testing what happens when flow entry is inserted that governs part of or all of a header space that a pre-existing flow entry governed. There are three scenarios:

- The new flow entry governs a narrower header space, i.e., a header space that is a subset of the pre-existing flow entry's header space.
- The new flow entry governs a wider header space, i.e., a header space of the pre-existing flow entry is a subset of the new flow entry's header space.
- The header space of the new and pre-existing flow entry intersect but neither is a subset of the other.

For each scenario the priority of the new entry could either be lower, the same or higher than that of the pre-existing one. We tested all these scenarios and will now describe them. For these tests we assume that traffic with source IP address in the 10.0.0.0/24 subnet is user related and that the topology of the network is that shown in Figure 4.3. When we refer to the wider flow entry we are referring to a flow entry that governs header space hs_{subnet} and when we refer to the narrower flow entry we are referring to a flow entry that governs header space hs_{IP}.

4.3.1 Insertion of narrower flow entry

For this test we use the header spaces defined in Table 4.3. We insert onto switch S1 a flow entry governing header space hs_{subnet} (all traffic with source IP address in the 10.0.0.0/24 subnet) that specified to forward this traffic to switch S2. Then we insert on switch S1 a flow entry governing header space hs_{IP1} (all traffic with source IP address 10.0.0.1) that specified to forward this traffic to switch S3. We present how we expect the lookup tables of the switches to look after the first flow entry is added in Table 4.4a. Now we expect three possible outcomes depending on the priority of the new entry with regards to the pre-existing one:

- **If the new flow entry has a lower priority**
  There should be no changes to any LUT in the network since this flow entry will not affect any traffic.

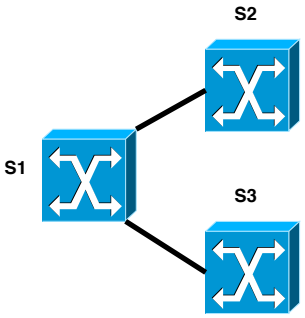

Figure 4.3: For test 4.2.2 we use a network topology consisting of 3 switches connected serially with the gateway in the middle.

- **If the new flow entry has the same priority**
  A new LUT entry should be added with header space hs_{IP} to switch S3's LUT as shown in 4.4b since now traffic with header space hs_{IP} may be forwarded to switch S3. However the same traffic may be forwarded to switch S2 so the entries in switch S2's LUT still stand.

- **If the new flow entry has a higher priority**
  The LUT entries on both switch S1 and S2 need to be updated to reflect that any traffic with header space hs_{IP} is no longer forwarded to switch S2 but to switch S3. Additionally a new entry in switch S3's LUT is added with header space $hs_{difference}$ to reflect that that this traffic is still begin forwarded to switch S2 as shown in Table 4.4c.

Header space	Definition
hs_{subnet}	Traffic with source IP address in the 10.0.0.0/24 subnet
hs_{IP}	Traffic with source IP address 10.0.0.1
$hs_{difference}$	Traffic with source IP address in the 10.0.0.0/24 subnet except 10.0.0.1

Table 4.3: Header space definitions for tests 4.3.1 and 4.3.2

4.3.2 Insertion of wider flow entry

For this test we use the header spaces defined in Table 4.3. We insert a flow entry governing header space hs_{IP} (all traffic with source IP address 10.0.0.1) that specified to forward this traffic to switch S3. Then we insert a flow entry governing header space hs_{subnet} (all traffic with source IP address in the 10.0.0.0/24 subnet) that specified to forward this traffic to switch S2. Now we expect three possible outcomes depending the priority of the new entry with regards to the pre-existing one:

- **If the new flow entry has a lower priority**
  All previous entries stay the same since header space hs_{IP} is still forwarded to switch S3. Additional entries are added to reflect that the header space $hs_{difference}$ is forwarded to switch S2.

- **If the new flow entry has the same priority**
  All previous entries stay the same since header space hs_{IP} is still forwarded to switch S3. Additional entries are added to reflect that the header space hs_{subnet} is forwarded to switch S2.

- **If the new flow entry has a higher priority**
  All traffic with header space hs_{subnet} will be forwarded to switch S2. To reflect this all LUT entries that were created because of the narrower flow entry should be deleted from the network and new entries should be added indicating that hs_{subnet} is forwarded to switch S2.

Switch	Entry	Headerspace	Priority	Children (Switch, Entry)	Parent (Switch, Entry)
S1	1	hs_{subnet}	X	{S2, 1}	
S2	1	hs_{subnet}			{S1, 1}
S3					

(a) LUT tables before the second flow entry is added and when Y<X.

Switch	Entry	Headerspace	Priority	Children (Switch, Entry)	Parent (Switch, Entry)
S1	1	hs_{subnet}	X	{S2, 1}, {S3, 1}	
S2	1	hs_{subnet}			{S1, 1}
S3	1	hs_{IP}			{S1, 1}

(b) LUT tables when Y=X.

Switch	Entry	Headerspace	Priority	Children (Switch, Entry)	Parent (Switch, Entry)
S1	1	$hs_{difference}$	X	{S2, 1}	
	2	hs_{IP}	Y	{S3, 1}	
S2	1	$hs_{difference}$			{S1, 1}
S3	1	hs_{IP}			{S1, 2}

(c) LUT tables when Y>X.

Table 4.4: LUT entries of switches for test 4.3.1. Switch S1 has a flow entry specifying forwarding traffic with header space hs_{subnet} to switch S2 and a flow entry specifying forwarding traffic with header space hs_{IP} to switch S3. The first flow entry has priority X and the second priority Y.

4.3.3 Insertion of intersecting flow entry

For this test we use the header spaces defined in Table 4.5. The two header spaces hs_{dstIP} and hs_{port} intersect but neither is a subset of the other. We insert a flow entry governing header space hs_{dstIP} that specified to forward this traffic to switch S2. Then we insert a flow entry governing header space hs_{port} that specified to forward this traffic to switch S3. Now we expect three possible outcomes depending the priority of the new entry with regards to the pre-existing one:

- **If the new flow entry has a lower priority**
  The pre-existing flow entry controls hs_{dstIP} and the new flow entry controls $hs_{port} - hs_{dstIP}$. All previous entries stay the same and additional entries are added to reflect that $hs_{port} - hs_{dstIP}$ is forwarded to switch S3 .
- **If the new flow entry has the same priority**
  The header space $hs_{dstIP} \cap hs_{port}$ could either be controlled by the new or the pre-existing flow entries. All previous entries stay the same and additional entries are added to reflect that the header space hs_{dstIP} is controlled by the new flow entry.
- **If the new flow entry has a higher priority**
  All LUT entries that were created because of the pre-existing flow entry should be deleted and/or updated to reflect that now the pre-existing flow entry only controls the header space $hs_{dstIP} - hs_{port}$. Additionally entries reflecting that the new flow entry controls the header space hs_{dstIP} and it is thus forwarded to switch S2.

4.3.4 Deleting narrower flow entry

For this test we use the header spaces defined in Table 4.3. First we insert a narrower rule and then a wider one just as in test 4.3.2. That is we insert a flow entry onto switch S1 governing header space hs_{IP} that

Header space	Definition
hs_{dstIP}	TCP traffic with source IP address 10.0.0.1 and destination IP address 10.0.0.4
hs_{port}	TCP traffic with source IP address 10.0.0.1 and transport layer destination port 22
$hs_{dstIP} \cap hs_{port}$	TCP traffic with source IP address 10.0.0.1, destination IP address 10.0.0.4 and transport layer destination port 22
$hs_{dstIP} - hs_{port}$	TCP traffic with source IP address 10.0.0.1, destination IP address 10.0.0.4 and transport layer destination is not port 22
$hs_{port} - hs_{dstIP}$	TCP traffic destination IP address 10.0.0.4, transport layer destination port 22 and not with source IP address 10.0.0.1

Table 4.5: Header space definitions for test 4.3.3.

specified to forward this traffic to switch S2. Then we insert a flow entry governing header space hs_{subnet} that specified to forward this traffic to switch S3. The wider flow entry had a lower priority and thus only governed header space $hs_{difference}$. Then we removed the narrower flow entry. Now the wider flow entry governs the entire header space hs_{subnet} and the lookup tables should reflect this. The flow entry was deleted both by explicitly deleting it and by making it time out.

4.4 Pre-existing flow entries

For this test we use the header spaces defined in Table 4.3 and assume the topology of the testbed is that shown in Figure 4.4. We want to test that pre-existing flow entries on a switch are correctly identified when a flow table modification elsewhere in the network causes them to govern user-related header space. We insert a flow entry that governs hs_{IP} on switch S2 before any such traffic is being sent there from S1 (the gateway). Then we insert a flow entry on switch S1 that governs hs_{subnet} that forwards it to switch S2. Now we expect that the there will be two entries in the lookup Table of switch S2. The first entry with header space of hs_{IP} with a priority set since there is a flow entry governing this header space. The second with header space $hs_{difference}$ with no priority set since there is no flow entry on switch S2 that governs this traffic.

4.5 Correct state of parent and children list

We tested whether the children and parent list of the LUT entries were correct after the insertion and subsequent modification or deletion of the same flow entries. For these tests we assume that the topology of the network is that shown in Figure 4.2. In these test we will only be dealing with one undefined header space that will be referred to as hs.

4.5.1 Modify path

We insert a flow entry on switch S1 which forwards the header space hs to switch S2. Both S2 and S3 have a flow entry that forwards this same header space to switch S4. In this state any traffic with headers space hs will be routed from switch S1, to switch S2 and finally to switch S4. We expect the children and parent list of the switches to be that of Table 4.6a.
Then we modify the flow entry on switch S1 to forward header space hs to switch S3 instead of switch S2. In this state any traffic with headers space hs will be routed from switch S1, to switch S3 and finally to switch S4. We expect the children and parent list of the switches to be that of Table 4.6b.

4.5.2 Multiple parents

We tested whether the PAL would correctly handle receiving the same header space from more than one source. We insert a flow entry on switch S1 which forwards the header space hs to both switches S2 and S3. Both S2 and S3 receive a flow entry that forwards this same header space to switch S4. We expect that the

Switch	Entry	Headerspace	Children (Switch, Entry)	Parent (Switch, Entry)
S1	1	hs	{S2, 1}	
S2	1	hs	{S4, 1}	{S1, 1}
S3				
S4		hs		{S2, 1}

(a) LUT entries when traffic with headers space hs is be routed from switch S1, to switch S3 and finally to switch S4

Switch	Entry	Headerspace	Children (Switch, Entry)	Parent (Switch, Entry)
S1	1	hs	{S2, 1}	
S2				
S3	1	hs	{S4, 1}	{S1, 1}
S4		hs		{S3, 1}

(b) LUT entries when traffic with headers space hs is be routed from switch S1, to switch S3 and finally to switch S4

Table 4.6: LUT entries of switches for test 4.5.1.

resulting LUT entry at switch S4 has two parent.
We then delete only the flow entry on switch S3 thus resulting in switch S4 only receiving header space hs from switch S2. We expect that the LUT entry on switch S4 only has one parent. We then delete the flow entry on switch S2 thus resulting in switch S4 not receiving header space hs. We expect that the LUT entry on switch S4 will be deleted.

4.6 Host injection

We also tested whether a host could inject traffic via re-writing of header space. For this test we assume that the topology of the network is that shown in Figure 4.4 and that host 1 is a user with IP address 10.0.0.1. The user has specified that all traffic with source IP address 10.0.0.2 is not sensitive and should fall under the Allow policy but all other traffic should fall under the NoSniff policy. The header spaces are defined in Table 4.7a and the flow entries in Table 4.7b
If these flow entries were allowed host 3 could inject traffic that would end up with host 1 thus violating the policy. However the PAL rejected the last flow entry as expected. If the order in which the flow entries are inserted are shuffled the last one will always be rejected since it will create a state where a policy violation occurs.

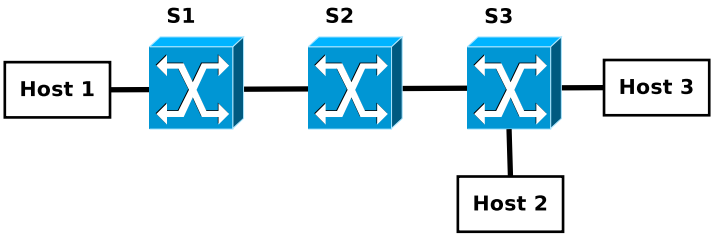

Figure 4.4: For test 4.6 we use a network topology consisting of 3 switches and 3 hosts.

Header space	Definition
hs_{not_host}	Traffic with destination IP address 10.0.0.1 and source IP address 10.0.0.2
hs_{host}	Traffic with destination IP address 10.0.0.1 and source IP address 10.0.0.3

(a) Header space definitions

Switch	Header space	Actions
S1	hs_{not_host}	Forward to host 1
S2	hs_{not_host}	Forward to switch S1
S3	hs_{host}	Set source IP address to 10.0.0.2 and forward to switch S2

(b) OpenFlow flow entries of switches

Table 4.7: OpenFlow flow entries and Header space definitions for test 4.6

Chapter 5

Summary

We have in this thesis presented our implementation for supporting header field re-writing in the PAL. To achieve our goal we have explored the various issues inherent to allowing header field re-writing such as keeping track of what is user related. This is not trivial since what is active user related header space is specific to each switch and is dynamic. Active user-related header space on each switch can change by the insertion, modification or deletion of a flow entry anywhere in the network. Allowing header field re-writing also opens the door to re-writing header spaces is such a way that different flows can become indistinguishable from each other. A savvy attacker can use this to cause loss of connectivity or to intercept user traffic.
We presented our solution which involves keeping track of active user related header space in lookup tables that are specific to each switch. We also explored the various scenarios that needed to be solved for this solution to work. This included going through what happens when a flow entry was inserted, modified or deleted. We showed that care must be taken to ensure that all the lookup tables are correct at each time since modifying one switch's flow table could have a rippling affect on other switches' lookup tables. We also demonstrate that our implementation works as expected by going over some tests that we performed.
In the future an evaluation of the PAL's performance under heavy load should be performed in our opinion. This may shed light on what performance improvements can be made.
Adding support for header field re-writing is an important step in convincing testbed operators to adopt the PAL. And hopefully the guarantees provided by the PAL will encourage users to volunteer their traffic which would give researchers better data to assess their solutions. Though the support of header field re-writing brings the PAL closer to being complete it still lacks support for various special functionalities and in our opinion the next step is to further expand the PAL.

Appendix A

Unexpected challenges

In the course of the projects some errors in the previous code were discovered that caused delays. However this also provided an opportunity to dive into the code and thus understand it better. This appendix elaborates on what needed to be altered in the original code.

A.1 Header space matching

Both the way policy header space was being created and how it was later compared to incoming traffic was heavily faulty and contained several errors. This caused incoming flow table modifications to be labelled as user related only in very rare specific circumstances. However flow table modifications that concerned user related header space were not flagged and therefore were allowed without scrutiny. In the cases were it was flagged as user related it was always treated like it was governed by the NoSniff policy even though it was under another policy. We fixed this with a complete overhaul of the code that created user policy header spaces and the way they are stored and matched against incoming flow entries. In the process we made the policies more general. Now it is possible to specify desired values for all fields in an OpenFlow header. Previously only the transport layer port could be specified.

A.2 Inconsistent flow tables

Four separate errors caused the simulated flow tables in the PAL and the real flow tables on the switches to become inconsistent. This resulted in calculations being based on faulty data.
The first error was in the code that performs flow table modifications. For each switch, the PAL maintains a simulated flow table that is kept in sync with the real table on the switch. The PAL must be able to abort a flow table modification and restore the simulated flow table to its original state before the modification started. To implement this functionality, the code maintains an additional table, called the *test table*. All modifications are done on the test table, and then if the modifications are accepted, the test table becomes the new real table via an assignment:

```
table = test_table
```

In Python, such an assignment just copies the value of test_table to table, but because the value is a reference to an object, the effect was that both variables point to the same object after this assignment, i.e. the test flow table. During the next flow table modification, changes to the test table will also be updating the main table.
Furthermore, the rollback was also implemented similarly, just by "resetting" the test table to the original table, by repeating the mistake.

```
test_table = table
```

The second error affected the handling of flow entry modifications and deletions. When a flow entry was to be modified or deleted the PAL tried to find a matching entry in its simulated flow table. However the function to find this match was being called with a wrong parameter causing no match to be found except for a rare case were this parameter incidentally was the same as the correct one. If no match was found the modification was handled as a flow entry insertion on the simulated flow table causing it to be added alongside the entry it was suppose to modify. The actual flow table on the switch however handled it correctly and modified the entry. In the event of a deletion of a flow entry the PAL simply ignored it but still forwarded the packet with the flow entry deletion to the switch which removed the entry. This caused the PAL to have entries in its table that had been deleted from the switch.
The third error affected how flow entry removals due to time outs were handled. When a flow entry was removed due to a time out it was only removed from the simulated table if it was user related. This caused an inconsistency whenever a flow entry was removed that was not user related.
The fourth error affected the priority of certain flow entries. In OpenFlow a flow entry with no wildcarded fields always has the highest priority, we will refer to this as an exact match. However a controller can send a flow table modification with an exact match and specify a priority that is lower than the highest priority. The PAL will receive this modification and see this low priority and then forward it to the switch. When this flow table modification is received by the switch the priority is set to the highest value regardless of what the original priority was. The PAL was not accounting for this scenario and thus using the incorrect low priority when inserting the flow table modification in its simulated flow table. An attacker could abuse this by for example:

- First inserting a flow entry that is not an exact match and will not cause a path violation.
- Then inserting a flow entry with an exact match that governed a subset of the first flow entry's header space. The attacker would specify the priority of this flow entry to be lower than the priority of the first flow entry.

When the PAL received the second flow entry it would conclude that this flow entry would never be used since it has a lower priority than another flow entry that governed the same headers space. Because of this the second flow entry would not be examined and any path violation it could cause would not be discovered. All these errors were fixed to ensure a consistent state between the simulated and the real flow tables of the switches.

A.3 Premature return from path validation

A error was found in the path validation process that was causing the path validation to return a positive result prematurely. This was in certain circumstances causing flow entries that should have been rejected to be accepted. At each switch in the path validation the PAL iterates over each flow entry (that affects the relevant header space) on that switch by priority. A positive result can not be returned until all flow entries have been examined. However if two or more flow entries were of the same priority and the first one was deemed not to be in violation the later flow entries were not examined. Thus if any of them were causing a violation it was not discovered in the path validation. This was fixed to ensure that all flow entries were examined.

A.4 Cosmetic re-factoring

In addition to the fixing error some cosmetic re-factoring was performed. This was done both to make error fixing easier and to make maintenance of the code easier in the future.

Appendix B

Original Problem

Bringing Private Traffic to Public SDN Testbeds

The combined efforts on exploring the possibilities of Software Defined Networking (SDN) for new network applications have led to the emergence of publicly accessible SDN testbeds all around the world. Examples include OFELIA and other FIRE-related activities, FIBRE, GENI, and JGN-X. However, these testbeds often run as stand-alone islands and have only limited possibilities to exchange traffic with the Internet for safety and privacy reasons. Safety considerations come into play when thinking about the damage, e.g., network outages, that can result from an experiment going bad. Moreover, an experimenter in an SDN testbed has a vast amount of control over the traffic, including the possibility to intercept, manipulate, and redirect communication. This immense power of the experimenter raises immediate privacy and availability concerns when thinking about having user traffic inside an SDN testbed. Nonetheless, experimenters would like to test out their inventions with user traffic for different reasons, e.g., investigating how a system performs under a real-world work load.

We are currently investigating how a compromise between the users' concerns and the experimenters' needs can be found. Ideally, an experimenter should be able to describe the type of traffic he needs for his experiment, and a user should be able to specify which parts of his traffic he is willing to make available under certain constraints. Such constraints could include keeping the traffic payload private, anonymizing endpoint addresses, or not passing some traffic through the testbed at all. Moreover, network availability should be guaranteed to the user whenever possible. This still leaves the question why users would be willing to share part of their traffic. However, building such a meeting point for experimenters and users allows to create a market place, where in addition to voluntarily donated traffic, experimenters can offer advantageous network features to users or even pay users in order to get access to the interesting parts of their traffic.

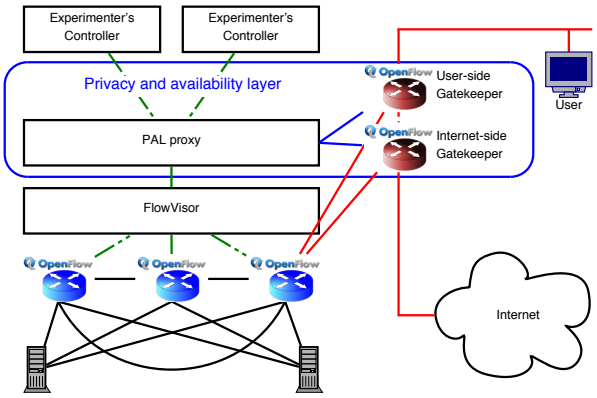

Figure B.1: Privacy in a testbed.

Prior work in constructing policing layers includes FlowVisor [11], VeriFlow [4], and Header Space Analysis (HSA)[2]. We build on HSA in order to enable injecting user traffic into a public testbed with privacy guarantees.

Tasks

We have implemented a prototype policing layer offering a "no-sniff" privacy guarantee. The goal is to complete the functionality of this prototype for eventual deployment on our OFELIA [12] island. To achieve this, we have several implementation tasks open, including enabling header-space rewriting inside the testbed, implementation of specific policies, an arbiter assigning user traffic to experiments, building a user web-interface for policy specification, and validation of the result.

Your work includes:

1. Studying related work on Software Defined Networking and OpenFlow.
2. Implementing support for header-space rewriting. This includes getting familiar with the Hassel library [2] and the OpenFlow protocol [8].
3. Writing a project report.

Requirements

Good programming skills in Python, analytical thinking, creativity. This thesis offers practical and theoretical tasks, including the development of SDN controller software and validation tools.

Bibliography

[1] About Flowvisor. https://github.com/OPENNETWORKINGLAB/flowvisor/wiki.

[2] KAZEMIAN, P., CHANG, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real time network policy checking using header space analysis. In *Proc. of USENIX NSDI, to appear* (2013).

[3] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header space analysis: static checking for networks. In *Proc. of USENIX NSDI* (2012).

[4] KHURSHID, A., ZOU, X., ZHOU, W., CAESAR, M., AND GODFREY, P. B. VeriFlow: verifying network-wide invariants in real time. In *Proc. of USENIX NSDI, to appear* (2013).

[5] KOTRONIS, V., SCHATZMANN, D., AND AGER, B. On bringing private traffic into public sdn testbeds, 2013.

[6] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. Openflow: enabling innovation in campus networks. *ACM SIGCOMM CCR* (Mar. 2008).

[7] About mininet. https://github.com/mininet/mininet/wiki/Documentation.

[8] ONF documents. https://www.opennetworking.org/about/onf-documents.

[9] Openflow Switch Specification v1.0.0. http://www.openflow.org/documents/openflow-spec-v1.0.0.pdf.

[10] About POX. http://www.noxrepo.org/pox/about-pox/.

[11] SHERWOOD, R., GIBB, G., YAP, K.-K., CASADO, M., MCKEOWN, N., AND PARULKAR, G. Can the production network be the testbed. In *Proc. of USENIX OSDI* (2010).

[12] OFELIA. http://www.fp7-ofelia.eu/.