SUMMARY In OpenFlow, data and control plane are decoupled from switches or routers. While the data plane resides in the switches or routers, the control plane might be moved into one or more external servers (controllers). In this article, we propose verification mechanisms for the data plane functionality of switches. The latter consists of two parts: (1) Flow-Match Header part (to match a flow of incoming packets) and (2) action part (e.g., to forward incoming packets to an outgoing port). We propose a mechanism to verify the Flow-Match Header part of the data plane. The mechanism can be executed at the controller, or on an additional device or server (or virtual machines) attached to the network. Deploying a virtual machine (VM) or server for verification may decrease the load of the controller and/or consumed bandwidth between the controller and a switch. We propose a heuristic to place external verification devices or VMs in a network such that the verification time can be minimized. Verification time with respect to consumed resources are evaluated through emulation experiments. Results confirm that the verification time using the proposed heuristic is indeed shortened significantly, while requiring low bandwidth resources.

key words: OpenFlow, in-band, out-of-band, verification

1. Introduction

OpenFlow [1] decouples the control plane from the data plane of switches or routers and embeds it into one or more external servers (controllers). The core idea of OpenFlow is to provide programmability of the data plane from the controllers using the OpenFlow protocol. In fact, the controllers program the data plane by “adding/modifying/deleting” entries in the FlowTables (forwarding table) of switches.

An entry in a FlowTable contains: (1) Flow-Match Header, which defines a flow, (2) actions, which define how packets should be forwarded (i.e., forward to an output port or to a different FlowTable) and (3) some additional fields such as priority, statistics and cookie identifier (Fig. 1). When a packet arrives at an OpenFlow switch, it is matched against the Flow-Match Header (wildcarded or exact match) of the entries of the FlowTable. If a match is found, the statistics of that entry is updated and the actions are performed. If two or more matches are found, the actions of the highest priority number entry are performed. If no match is found, the packet (a part thereof) is forwarded to the controller. Thereafter, the controller determines how the packet can be handled. It may return the packet to the switch indicating the forwarding port, or it may add a Flow Entry in the switch to forward the packet.

The research challenge that we consider in this article is the verification of matching of incoming packets with the Flow-Match Header of a Flow Entry. There can be two causes of incorrect or no matching of packets: (1) bugs in OpenFlow switch implementation and (2) errors in FlowTable configuration. Bugs in OpenFlow switch implementation [2] may be caused by bugs in the hardware or software part of switches. The errors in FlowTable configuration may be caused by: (1) bugs in controller software for the addition of a Flow Entry and/or (2) presence of a high priority error-prone Flow Entry (added manually or by a controller) that gives a match with the incoming packets [3]. The objective of verification is to find this incorrect or no matching and hence, to find the packet-headers that cannot be matched or can be matched incorrectly with the Flow-Match Header of a Flow Entry. In the absence of this verification, it may be difficult to find which packets cannot be delivered or can be delivered incorrectly by a switch.

Most of the existing verification tools such as HSA (Header Space Analysis) [4], Anteater [5], VeriFlow [6] detect matching issues by analyzing configuration of switches. However, without sending real packets, it is difficult for these tools to find software or hardware bugs in flow matching. Recently, the automatic test packet generation (ATPG) tool [7] is proposed to verify the network by transmitting test packets. ATPG verifies only one of the packet headers that gives a match with a wildcarded Flow Entry, whereas matching of all the other packet headers remains untested. In the (short) demo paper [8], we demonstrated a mechanism to verify this matching functionality of a switch, where the controller is used for verification. However, this mechanism requires modification of the Flow Entries, which may not be acceptable in operational networks. In this article, we propose a mechanism to verify the Flow Entries without modifying them.

Our mechanism transmits test packets to verify that all the packet-headers match correctly with the Flow-Match
Header or not. However, if the test packets have to match with the Flow Entries of data packets, these would need additional bandwidth to be reserved for test packets on the links corresponding to the outgoing actions of the matched Flow Entries. To overcome the challenge (additional bandwidth requirement), we forward the test packets through duplicated Flow Entries, which either drop or forward the test packets to the controller (instead of forwarding these on the outgoing links).

Our mechanism copies all the Flow Entries of a FlowTable to a new table (i.e., any other FlowTable of a switch), and the new table is verified then by sending test packets and performing a verification activity. As the Flow-Match Headers of the Flow Entries in the new table is the exact copy of the Flow-Match headers in the original table, the assumption is that if a flow matching failure is reported by the new table, the failure is also present in the original table. Moreover, as the action part (verification of the action part is already covered by ATPG) of the Flow Entries in the new table is different from the corresponding Flow Entries in the original table, all the test packets can be forwarded through the new table without being sent on the outgoing action of the original Flow Entries. Therefore, the objective of copying Flow Entries is to: (1) maintain the same set of Flow-Match Headers in the new table (as the original table) for finding FlowTable specific matching issues [3], and (2) prevent forwarding of test packets through outgoing links (actions of Flow Entries in the original table) of a switch and therefore, decreasing the bandwidth requirements of those links for verification. Furthermore, as the mechanism does not verify the original Flow Entries, there may be some cases when our mechanism is not able to verify matching functionality for some software or hardware bugs. These cases are described in Sect. 2.1.

Our mechanism can be executed by the controller. However, it requires computational resources (to generate test packets) of the controller and the bandwidth resources between the controller and a switch (which are also utilized for the other controller activities) to perform verification. Therefore, to decrease the controller requirements for these resources, we consider a network in which servers (custom machines) can be attached to a switch or router to transmit or receive packets [9]. For verification, either these servers can be used directly for verification or these can host virtual machines (VM) to perform verification.

Deploying a VM or server over the network infrastructure may increase the cost, as it requires additional resources (memory, CPU etc.,) to be reserved for verification. Nevertheless, the VM may need to transmit test packets to a switch through the data path links of other switches, as all the switches may not have a direct link with the VM. Therefore, if the VM is placed at a location from where test packets need to travel a long path or same links to reach all or some switches, the bandwidth requirements of all these links for verification will increase. Hence, we also propose a heuristic to efficiently place the VMs or servers and efficiently select a path between switches and the VMs so that the verification time is reduced and resource requirements are distributed over many links in the network.

We perform extensive emulation experiments on the Fed4Fire testbed facility provided by iMinds [10]. In our experiments, verification using the controller and a VM (or server) is emulated. Software matching errors are emulated by translating some packets of a flow to incorrect headers. This leads to matching of these packets with an incorrect Flow Entry. In our results we show the verification time (the time required to find these matching errors) with respect to resources available in the network. The experiments validate the proposed approach and evaluate the trade-off between the verification time and the required resources, enabling the user of the mechanism to choose which bandwidth to reserve for given verification time. Additionally, our proposed heuristic is compared for different types of topologies which vary with the number of switches and degree of meshedness. The results show that the network can be verified in limited time, when a VM is placed using our proposed heuristics.

The mechanism is implemented as part of the verification functionality of the integrated prototype of the UNIFY project [9]. Section 2 provides errors in Flow-Matching Functionality and Sect. 3 presents our proposed mechanism, Sect. 4 presents in-band or out-of-band verification, Sect. 5 and Sect. 6 present emulations and results, and then finally Sect. 7 concludes.

2. Errors in Flow-Matching Functionality

In OpenFlow switches, there can be two types of Flow Entries: (1) exact match and (2) wildcarded Entries [11]. In case of the exact match entries, all the matching-header fields are specified for a Flow Entry. However, in case of the wildcarded entries, some of fields can be wildcarded for a Flow Entry. Finding matching issues in the exact match entries is simple, as these entries can match only one type of a flow. However, as wildcarded flows can match many different flows, it is complex to verify matching of these flows in OpenFlow networks. Our mechanism can verify the matching of these wildcarded flows. However, the similar mechanism can be used to verify the exact match flows. In this section, we describe the Flow-Matching bugs that can be present in an OpenFlow switch and the errors that can be identified by our proposed verification mechanism.

2.1 Software or Hardware Bugs in Flow Matching

The proposed mechanism can be used to find errors due to configuration issues and software or hardware bugs of flow matching. These bugs can be present at any block of software or hardware programming [2]. Figure 2 depicts the functional blocks of OpenFlow switches such as HP, NEC [13], [14], and indicates where flow matching occurs. It also shows in which blocks matching-related bugs can be present, and which errors can be detected through our mechanism. Here, errors mean which packet-headers are not matched correctly or can be matched incorrectly by
the Flow-Match Header due to a software or hardware bug.

Figure 2 shows matching in both software and hardware. In fact, many switches such as HP, NEC contain FlowTables in software as well as in hardware. Usually, the software table contains the full set of Flow Entries, while the hardware table contains a subset of the Flow Entries. When a packet arrives at a switch (Input Arbiter in Fig. 2), the packet is stored in the input queue and the packet-header is first extracted by the header extractor and then the packet header is matched against all the entries (TCAM or SRAM) present in hardware. If a match is found, the Packet-Editor forwards the packet to the output port/queue from the input queue. If no match is found, the packet is forwarded to software which translates again the packet into headers. If a matching entry is found in software, the entry is installed in the hardware FlowTable and the packet is forwarded through the Packet-Editor (Fig. 2). Otherwise, it is forwarded to the controller to define its action.

All the bugs in this packet forwarding are listed in Fig. 2. Some of these are also listed below:

1. In the header extractor (block 1 in Fig. 2), there can be hardware bugs related to extracting of some header fields (such as specific MAC address). In this case, the same set of packet-headers may be extracted incorrectly. Our mechanism can find the errors generated due to these hardware bugs.

2. In the match lookup in TCAM or SRAM (block 2 in Fig. 2), there can be bugs related to specific TCAM bits. For example, if a TCAM bit is supposed to be x, but it is always 1, then all packets with a 0 at that location will be ignored. If this issue is present in all TCAM entries, the errors can be reported correctly by our mechanism. However, if this issue is specific to some entries, it cannot be correctly identified by the mechanism. This is because test packets used in verification may be matched against an error-free or error-prone TCAM entry.

3. In the software part of packet translation (block 3 in Fig. 2), there may be some bugs related to the translation of some header fields. The errors due to these bugs can be detected by our mechanism.

4. In the hash table lookup in software (block 4 in Fig. 2), bugs may occur related to calculating the hash for specific headers. The errors due to these bugs can be identified using our mechanism.

5. In the Flow Entry addition from software to hardware (block 5 in Fig. 2), there may be bugs related to installing Flow Entries containing specific header fields. These errors also can be found through our mechanism.

3. Verification Mechanism

Our mechanism uses a header field such as EtherType (or VLAN ID) for differentiating test packets from data packets and assumes that this header field (e.g., EtherType) is wildcarded in the Flow-Match Header part of Flow Entries.

The mechanism performs three steps for verification: (1) flow duplication, (2) test packet generation, and (3) matching errors identification. In the flow duplication step, the mechanism duplicates the Flow Entries from a FlowTable to another FlowTable. In the test packet generation step, the mechanism generates and transmits test packets that can match with the Flow-Match Header of duplicated Flow Entries. In the matching errors identification step, the mechanism calculates the matching errors either by reading the counters (statistics) of the duplicated Flow Entries or by comparing the sent and received test packets.

The flow duplication step can be performed by the controller, as it only needs to insert additional Flow Entries in the switches for verification. However, for test packet generation and matching errors identification, either the controller or a VM (or server, see Introduction) can be used. It reduces the load of the controller (i.e., due to generation, transmission, or reception of many test packets) for performing verification. We describe now all the steps in detail:

3.1 Flow Duplication Step

In the flow duplication step, for verifying FlowTable x (Fig. 3), the controller copies all the flows from FlowTable x to FlowTable y (x and y are ≥ 0 and y > x). The Flow-Match Header in Fig. 3 is an IP address field, but it can be any field (one or more) of the Flow-Match Header of Flow Entries. Figure 3 shows that the Flow-Match Header of all the duplicated flows in FlowTable y is same as the Flow-Match Header part of the original flows in FlowTable x. However, the difference lies only in the action part. The action of all these duplicated flows is DROP, CONT, or VM. “DROP” means drop all the packets that match with the Flow Entry. “CONT” or “VM” means send all the matched packets to the controller or to the VM respectively.

Moreover, the controller inserts an additional Flow Entry (5th entry in FlowTable x in Fig. 3) in FlowTable x to forward all the test packets to FlowTable y (action=Table:y). To match all test packets with this entry, the entry contains a higher priority (P+) number than the priority (P) number of the existing Flow entries in FlowTable x and the Flow-Match Header contains the EtherType (E) of test packets and...
3.2 Test Packet Generation Step

In this step, the controller or VM transmits test packets to the switch whose entries are needed to be verified. For this, the controller or VM can transmit all the test packets that can give a match with the Flow-Match Header of a Flow Entry. However, if all the fields of the Flow-Match Header are wildcarded, there is a need to transmit lots of test packets (e.g., $2^{32}$ different packets if there is a wildcard in all bits of the IP address field), increasing the bandwidth requirements and time for verifying the Flow Entries. To decrease the bandwidth requirement, we propose to transmit only those test packets that give a match with a partially wildcarded field and in a fully wildcarded field, we suggest to fill randomly generated values. This is proposed because if a field is partially wildcarded (such as wildcards in last 8 bits of IP address), there is a complex software implemented for matching a Flow Entry, as a part of a field is needed to be matched with the incoming packets [15]. This may lead to a matching error in any of these flows. However, if a field is fully wildcarded, matching implementation will be very simple (i.e., just ignore or do not match the header of the field [15]). Therefore, if these are tested with the limited number of headers, these can be considered as error free.

In our mechanism, the test packets are generated in the form of packet-out messages. Packet-out messages [12] are defined in OpenFlow to send packets from the controller through the FlowTables or to an outgoing port of a switch. We use these messages to transmit test packets to the switch (under verification). Figure 4A describes all the fields of the packet-out messages according to OpenFlow version 1.5. Figure 4B describes these fields for generating a test packet.

In Fig. 4B, the packet-out message contains the ingress port of the Flow-Match Header in the In-Port location and the test packet-header (i.e., all the other matching fields) in the Data[] location. As the action of the packet out messages is TABLE in Fig. 4B, it is matched against the Flow Entries of the FlowTables when it is received by the switch. Therefore, the matching errors can now be found using the Matching Error identification step.

3.3 Matching Error Identification

For this step, we describe two methods: (1) binary search and (2) packet-reception. The binary search method applies the well known binary search algorithm to find the matching errors. The packet-reception method receives the sent test packets and from the unreceived/received test packets, the method finds the matching errors. The advantage of the binary search method is that it reduces the upstream bandwidth to receive the test packets. However, the disadvantage is that it takes more time to find matching errors.

3.3.1 Binary-Search Method

For the binary-search method, the action of all the Flow Entries in FlowTable y is “DROP”. The method is described in Fig. 5. Our explanation for the method is given for the Flow-Match Header as 10.1.1.* and the matching error is present in the packet that contains the IP address as 10.1.1.65. However, this explanation is applicable for more than one matching fields or errors.

To find the matching error in IP 10.1.1.65, the controller first transmits all test packets (i.e., containing the IP address from 10.1.1.0 to 10.1.1.255) to the switch (BS1 in Fig. 5). The controller then finds the number of errors by subtracting the number of sent test packets (256) from the increase in the counters of the Flow Entry (i.e., under verification) of FlowTable y. As the Flow Entry cannot match one IP address (i.e., 10.1.1.65). Therefore, the controller at this time knows
that there is one matching error. However, it does not know
that which flow (i.e., 10.1.1.65) cannot be matched through
the Flow Entry. Therefore, to find this flow, the controller
now transmits the first half of the test packets (i.e. 10.1.1.0
to 10.1.1.127, BS2 in Fig. 5) and then finds the matching
erors by the same formula (i.e., subtracting the number of
sent packets from the counter increments). The controller
will again find one error and therefore, it again sends the first
half of the test packets (i.e. 10.1.1.0 to 10.1.1.63, the third
row in Fig. 5) and finds no error. As there is no error in this
first half, it now knows that the error is in the second half.
Therefore, the controller now switches to the second half
(i.e., 10.1.1.64 to 10.1.1.127, BS3) and then process is re-
peated until it reaches to the last row of Fig. 5 (BSn) in which
it transmits only one test packet i.e., the test packet contain-
ing IP 10.1.1.65 and finds the error using the same formula.
The controller then reports IP (10.1.1.65) as a matching er-
ror.

3.3.2 Packet-Reception Method

For this method, the action of all the Flow Entries in
FlowTable y is CONT or VM. If this step is performed by
the controller, the action is CONT. If this step is performed
by a VM, the action is VM. Due to this action, the test pack-
ets are sent back to the controller or to the VM in the form
of packet-in messages [12] after matching with the Flow-
Match Header of a Flow Entry. If a test packet is not re-
ceived back by the controller or the VM, it declares that it is
due to a matching issue present in the respective Flow Entry
and therefore, reports this matching issue.

To find matching with an incorrect Flow Entry, the
packet-reception method depends on the flow cookie identi-
fier [12] (Fig. 1), which is a unique 32 bit number associ-
ated with each Flow Entry in a switch. When a test packet
is sent to the controller or VM back after matching a Flow
Entry, the flow cookie identifier of the matched Flow Entry
is copied in the packet-in message. The controller or VM
receives this message. If it finds that the flow cookie identi-
fier in the message is not same as the cookie identifier that
is associated with the Flow Entry that must have a match with
the test packet, the method declares that it is due to match-
ing with an incorrect Flow Entry. This may happen because
the received test packet has found a match with a Flow En-
try that actually shouldn’t match with the test packet header
(e.g., due to configuration errors or bugs in the switch im-
plementation).

4. Out-of-Band or In-Band Verification

Out-of-band means that control traffic is sent on a separate
channel (or network). In-band means that control traffic is
sent on the same infrastructure as the data plane [16]. As
described earlier, the controller can perform all the steps
of verification (section 3) using out-of-band or in-band net-
works (without the VM part in Fig. 6A and Fig. 6B). How-
ever, in this case, the bandwidth requirement of the control
network will increase due to the transmission of the large
number of test packets for verification. Furthermore, if an
in-band network is used (Fig. 6B) for the controller com-
munication, the bandwidth requirement of the link between
the controller and the switch (switch C in Fig.6B) which
connects the data network with the controller, will increase
substantially as this link will be used for the communication
between many switches and the controller. Nevertheless,
this also increases the computational requirement of the con-
troller to perform verification. Therefore, we propose that
the steps – test packet generation and matching-error identi-
fication – which increase the computational and bandwidth
requirement of the controller significantly, can be performed
by a VM.

Instead of using additional controllers, we propose to
use light-weight VMs for distributing the load of the con-
troller. This is because the resources of a VM can be re-
leased once verification is completed, whereas in case of us-
ing additional controllers, these controllers may need to be
present in the network all the time just for verification. This
may be a wastage of the controller resources in the network.

For performing verification through VMs, we assume
that there are some servers in the network that have capabil-
ity to create VMs [9]. However, if a server has a capability
to transmit test packets and can perform the verification ac-
tivity, the server can be used directly for verification. In
emulation, the controller creates VMs, makes a connection
between a VM and a switch, and establishes paths between
switches and the VM (Fig. 6). Currently, OpenStack neutron
with the OpenDayLight [17] or Floodlight controllers [18]
defines the API to create VMs from the controller.

We propose that the VM would be part of the control
plane and should establish an OpenFlow session with the
switches in order to perform the verification activity. As
not every switch is directly connected to a cluster of servers
capable of hosting VMs (we assume this to be only possi-
ble for some switches, which have for example a direct data
center link), the VM needs to establish an OpenFlow session
path with these switches through other switches in the data
plane. In this proposal, the VM works like an additional
The first capability is required because the Flow-Match Header part of a Flow Entry may contain an ingress port as a matching field [12]. Without having the first capability, the VM may need to send test packets through additional links in order to match the Flow Entry with the test packets, requiring more bandwidth for verification. This can be explained through Fig. 6. Suppose that switch A has a Flow Entry, containing port 1 (i.e., port AB) in the ingress port of the Flow-Match Header. For verifying this entry, the VM needs to send a test packet to switch A through path VAD-CBA. This requires bandwidth to be reserved in links AD, DC, CB, and BA for verification. However, if there is an OpenFlow session between the VM and switch A through path VA, the VM can directly send a test packet to switch A through link VA in the form of a packet out message that contains port 1 (or any port) in its ingress port field. This decreases the bandwidth required for verification.

The second capability is required because it is possible that VM needs to verify two or more switches at the same time to decrease the verification time. However, if there is no OpenFlow session between the VM and switches, the verification of two or more switches at the same time may be difficult to implement or if it is implemented, the verification time will increase significantly. This can also be explained through Fig. 6. In Fig. 6, the VM transmits test packets to B through A, as switch A is the only node connecting VM to the switch topology. If the VM wants to send the test packets to switch A and B simultaneously, switch A should have a way to distinguish which test packets are sent to it and which test packets are sent to switch B. This may be possible if the test packet specific entries are present in A to forward the test packets of switch B. However, as these entries are first needed to be verified before the entries in switch B, it will increase the verification time. However, if there is OpenFlow sessions between the VM and switches, the VM can send test packets to switch A and B at the same time by sending them through their OpenFlow session paths, decreasing the verification time.

4.1 OpenFlow Session Path Selection in In-Band Networks

Suppose that the OpenFlow session paths between a VM and switches are along the data plane switches and contain the same subset of data plane links. In this case, as the VM can verify one or more switches at the same time (second capability, described above), the bandwidth requirement of these subset of links will increase significantly for verification. In addition, if bandwidth for verification is limited for a link, some switches have to wait for other switches to complete verification. This will increase the verification time, as some switches have to wait for other switches in the verification path to complete verification. Therefore, we propose to distribute the load of verification into many links (load balancing/load distributed approach) and therefore, the verification paths for different switches may not contain the same subset of links, decreasing the waiting time of switches.

For the load distributed approach, we assume that there is limited bandwidth (let say b) available in each switch link for verification. This bandwidth is assumed to be equal for each link. Furthermore, as the VM links (the link between VM and switch A in Fig. 6) are the links which will be used to create OpenFlow session paths for many switches, we assume that the VM links have enough bandwidth (equal to the bandwidth required to verify multiple switches at the same time) for verification.

We consider a network of N switches denoted by \( v_i \) connected through a link set \( e_{ij} \) and link cost \( c_{ij} \), where \( e_{ij} \) is the link incident to nodes \( v_i \) and \( v_j \), and \( c_{ij}(\geq 1) \) is the cost associated with \( e_{ij} \). In addition, VM has a direct connection with \( m \) (1 \( \leq m \leq N \)) switches in the network. The edge between the VM and switch \( k \) is denoted by \( e_{ik} \) and the cost of the link between the VM and switch \( k \) is denoted by \( c_{ik} \). The cost of each link (switch links and the link between the VM and a switch \( k \)) is 1 by default. However, the cost of a link \( e_{ij} \) including the VM links \( c_{ik} \) will increase depending on the load on the link due to selection of the link for OpenFlow session paths. Let \( P \) represents a path between the VM and switch \( t \), where

\[
x_{ij} = \begin{cases} 
1 & \text{the path } P \text{ traverses } e_{ij} \\
0 & \text{otherwise}
\end{cases}
\]

We assume that \( P \) is a path that contains no cycle. This path is not required to be the shortest path between VM and \( t \) pair. In our mechanism we choose a path for establishing an OpenFlow session for which the cost \( \sum_{ij} c_{ij} x_{ij} \) is minimal. If a session path for a switch is selected, which goes through a link \( e_{ij} \) (including a VM link), the cost of that link \( (e_{ij}) \) will increase by the load that the session path will generate on the link. This load is represented by the number of test packets that the session path needs to transmit for verifying Flow Entries. The number of test packets is equal to the number of packet-headers that a partially wildcarded field of Flow Entries in the switch can match (i.e., \( \sum_{f} F_f \)), where

\[ F_f \text{ is the number of different packet-headers that the Flow-Match Header of Flow Entry } f \text{ can match.} \]

Suppose that the maximum value of the number of test packets that an OpenFlow session transmits is \( F_{Max} \). Then, the cost of link \( e_{ij} \) will increase by:

\[
c_{ij} = c_{ij} + \frac{\sum_{f} F_f}{F_{Max}}
\]  

(1)

For selection of a path to a switch, this cost will be taken into account.
4.2 VM Placement

For VM placement, we rely on “betweenness centrality” which is the number of paths from all nodes to all other nodes that crosses a given switch. We believe that this is a good indication where the VM should be placed in the network. Because if we place a VM in a switch (i.e., a direct connection between the VM and the switch) with the highest “betweenness centrality”, the VM will become close to many switches and OpenFlow session paths of many switches may contain different links. In this case, due to the second capability (discussed above), switches can be verified in the limited time. For example, if a VM is placed at switch E (switch E has the highest “betweenness centrality” in Fig. 6), the VM will become close to each switch and the session paths for switch A, B, C, D will contain different links from switch E, decreasing the verification time of these links for verification. However, in a network, there can be only some switches that have capability to place a VM (i.e., having a direct connection). Therefore, for VM placement, we consider only those switches to calculate the “betweenness centrality” and place the VM in the switch that has the highest “betweenness centrality”.

Using our emulations, we show that the “betweenness centrality” is a good indication of the placement of VMs. The future work is to place more than one VM in the network such that verification time can be reduced. For this, we can partition the network into k partitions and place the VM in each partition with the switch having the highest “betweenness centrality”.

5. Emulation

We performed emulations on the Fed4fire testbed facility at iMinds [10]. The testbed is a generic test environment for advanced network, distributive software and service evaluation. It consists of 100 physical nodes interconnected by a non-blocking 1.5 Tb/s Force10 Ethernet switch. Each node has 4 CPU cores and 4 GB RAM. We generated emulated pan-European topologies using the nodes of the testbed and performed extensive verification experiments.

Figure 7 shows one of the emulated pan-European topologies that contains 16 OpenFlow switches connected with each other in a mesh fashion. Each switch of the topology has also provided a dedicated interface to a switched Ethernet LAN (not shown in Fig. 7), which establishes an out-of-band connection with a single controller (the controller is shown in Fig. 7).

We perform three different ranges of experiments — (1) (out-of-band) controller-induced verification, (2) (in-band) VM-induced verification, (3) validation on multiple topologies — in our testbed. Each of these will be handled in more detail in the following subsections. For our emulation, we implemented the proposed verification mechanism in the Floodlight controller that uses OpenFlow version 1.3 [18] in its implementation. In addition, we used Open vSwitch [15] for running OpenFlow in the switches of the emulated topology. In the experiments, software matching errors are emulated by translating some packets of a flow to incorrect headers. In our emulations, we find these errors and find verification time. In all the experiments, multiple switches are verified at the same time.

5.1 Controller-Induced Verification Experiment

In the controller-induced experiment, the controller makes an out-of-band connection with switches and performs all the steps of verification. Flow-Match Header errors are detected either via the packet-reception or via the binary-search method.

Figure 8 shows the emulation methodology using traffic captured on the controller when the binary-search or packet-reception method is used for verification. Traffic is shown from second −100 to 200, when 1 Mbps is available in between each switch and the controller for verification. In emulation, each switch in the emulated pan-European topology first establishes an OpenFlow session over the TCP session with the controller. The spikes in Fig. 8 from −100 to −90 seconds are due to traffic exchanged between the controller and switches to establish OpenFlow sessions. At emulation time −50 seconds, the controller adds Flow Entries in each switch to forward data traffic. The Flow Entries contain the ingress port, source IP address and destination IP address as matching fields. The source and destination IP addresses are 24 bit addresses and therefore, the controller needs to transmit 65536 different test packets for verification. However, due to the generated bug, each Flow Entry
is not able to match correctly 256 flows out of 65536 flows. At second 0, the controller starts verification of the Flow Entries. In Fig. 8, traffic generated due the verification of one Flow Entry is shown.

Figure 8 shows that the verification time (time to find matching errors) using the packet reception method is 69 seconds (Fig. 8A) and using the binary-search method is 171 seconds (Fig. 8B). Figure 8A shows that downstream and upstream traffic due to the packet-reception method are about 16.2 Mb/s. Figure 8B shows that there is about 16 Mb/s downstream traffic and about 5 Mb/s upstream traffic using the binary-search method. In this method, downstream traffic is present because the controller sends test packets in the downstream direction in the form of packet-out messages to verify the Flow Entries of switches. The TCP session in switches then sends the acknowledgments of the sent traffic in the upstream direction. The upstream traffic in the binary-search method (Fig. 8) includes this acknowledgment traffic. Additionally, it includes the traffic sent by the switches in sending counters of Flow Entries. However, the total traffic in the downstream and upstream direction using the packet-reception method is only 0.2 Mb/s more than the traffic in the downstream direction using the binary-search method (i.e., 16 Mb/s). It means that there is only 0.2 Mb/s additional traffic generated by TCP for acknowledgments of test packets in the packet reception method. This traffic is very small compared to the acknowledgment traffic in the binary search method (5 Mb/s). This is because of the push ACK functionality [19] of TCP in which TCP sends data in the acknowledgments of packets.

Figure 8B also shows the iterations (BS1 and BS2) of the binary-search method. As the counter read interval of Open vSwitch in our implementation is 2 seconds, we see a downstream spike of 2 seconds after BS1, BS2 and so on.

5.2 VM-Induced Verification Experiment

In the VM-induced verification, the controller triggers the creation of a VM (light weight Linux container) and connects it with one of the switches in the network (Hamburg in Fig. 7). We use the packet-reception method for calculating the matching errors in the network. We compare the verification time when our approach (i.e., load balancing approach, Sect. 4.1) or the shortest path approach is used to the verification time when our approach (i.e., load balancing approach) is used to establish OpenFlow session paths between switches and VM.

We now explain the emulation methodology of the VM-induced verification experiment using the traffic captured on the controller and the VM link. Traffic is shown when the VM is created in the Hamburg switch of the emulated topology (Fig. 7) and verification uses data plane links for transmitting or receiving test packets. The emulation methodology of this experiment is same as the controller verification experiment (Fig. 8) in which switches establishes OpenFlow session paths with the controller from second −100 to −90 and at second −50, the controller adds Flow Entries in switches to forward data traffic. At second 0, the controller starts verification by copying Flow Entries to another table, creating a VM at the Hamburg switch, and establishing Flow Entries for making OpenFlow sessions between the VM and switches. We see small spike in Fig. 9 at second 0 due to this traffic.

In our emulation, the controller creates a VM using the RPC (Remote Procedural Call) commands. Figure 9B shows that the time to create a VM is about 19 seconds. After creating the VM, the VM establishes OpenFlow session paths with all the switches in the network and starts verification. For verification, we reserved 1 Mb/s bandwidth in each switch link. In our mechanism, the controller controls the rate of test packets according to the bandwidth available in each link of its OpenFlow session. Additionally, the bandwidth between VM and the Hamburg switch is kept as 4 Mb/s. Figure 9B shows that the total verification time of a Flow Entry is 396 seconds.

5.3 Validation on Multiple Topologies

In the validation of multiple topologies experiments, different topologies are used for verification. The topologies are: core topology (CT), Basic Reference Topology (BT), and Ring topology (RT). The difference between BT and CT is that BT contains more switches than CT. BT contains 28 switches and CT contains 16 switches. The difference of BT and RT is in the degree of meshedness. RT has a lower meshedness than BT. All the other details of the topologies can be found in [16]. In these experiments, the VM is connected with one of the switches in the network in different experiments and the verification time is evaluated for each placement of the VM. In a network, there may be only some switches that have a direct connection with the (data center hosting the) VM. However, for the completeness of the multiple topology experiments, we assume that each switch in the considered topology has this capability.

6. Results

In this section, we present the results gathered by performing all the experiments.

6.1 Controller-Induced Verification Experiment

Figure 10 depicts the verification time when the bandwidth between the controller and switches are varied. For transmitting test packets, the controller sets the rate of the test
packets according to the bandwidth available between the controller and switches for verification. In case of the binary-search method, the verification time is calculated by subtracting the time when the last counter read reply message is received by the controller with the time when the first counter read request message is sent by the controller. In case of the packet-reception method, the verification time is calculated by subtracting the time when the first packet-out containing a test packet is sent by the controller with the time when the last packet-in containing a test packet is received by the controller. All the results are taken 50 times and the average is shown in Fig. 10.

As expected, Fig. 10 shows that the verification time of the Flow Entries decreases with the increase in the bandwidth reserved for verification. In this experiment, five Flow Entries are verified in each switch for Flow-Match Header issues. Figure 10 also shows that the verification time in the binary-search method is longer than the verification time in the packet-reception method. This is because the binary-search method sends more number of test packets for verification through binary search iterations, leading to increase in the verification time. In addition, as the counter update interval is 2 seconds in our emulation, it further increases the verification time in the binary-search method. Furthermore, the binary search method performs the verification of the Flow Entries one by one. However, the packet-reception method can perform the verification of all the Flow Entries at the same time, leading to decrease in the verification time. For the packet reception method, the minimum value of the verification time is 1.2 seconds in our emulations.

Figure 11 shows that the upstream bandwidth usage in the packet-reception method is higher than the upstream bandwidth usage in the binary search method. This is because the packet-reception method sends back the matched test packets to the controller, leading to increase in the bandwidth requirements in the upstream direction. Furthermore, the binary search method drops all the test packets matched with a Flow Entry. However, we see in Fig. 11 that when the downstream traffic increases, the traffic in upstream direction due to the binary-search method also increases. These are due to the acknowledgments of test packets sent in the upstream direction by TCP.

Figure 10 and Fig. 11 show the results when there are only 5 wildcarded Flow Entries (wildcards in the last 8 bits of source and destination IP address) in each switch for verification. In this case, verification is completed in limited time. However, when the number of Flow Entries is increased in switches, the verification time will increase significantly. Fig. 12 shows the results when the number of Flow Entries is increased from 5 to 400. In this experiment, 5 Mb/s of bandwidth is available in between each switch and the controller for performing verification.

Figure 12 illustrates that the verification time becomes significantly long as the number of Flow Entries increases. This is because if more Flow Entries are present for verification, more test packets are required to be sent to the switches. As the bandwidth is limited, the controller needs to wait long time to send all the test packets, increasing the verification time.

We see that the binary-search method leads to long verification time as compared to the packet-reception method (Fig. 10 and Fig. 12). However, as the bandwidth requirements of the binary-search method in the upstream direction are less compared to those of the packet-reception method (Fig. 11), the binary-search method can be used when upstream bandwidth is a bottle-neck.

6.2 VM-Induced (In-Band) Verification Experiment

For in-band verification, we assume that different queues are installed in OpenFlow switches for control and data traffic. Configuration of these queues is described in [16]. As traffic flows using different queues do not interfere with each other [16], we do not consider data traffic in our experiments.

Figure 13 shows the results of the experiments when the bandwidth of each switch links is limited between 0.5 to 5.5 Mb/s (value is shown in the figure) for verification. The emulation methodology of these experiments is pro-
We compare the results of the load distributed/balancing approach (discussed in Sect. 4.1) with the shortest path approach (Fig. 13). In this experiment, the bandwidth is limited and therefore, if the same subset of links are used for verification paths of many switches (most probable case in the shortest path approach), it will increase the verification time (explained in Sect. 4). Fig. 13 shows that the verification time using the load distributed approach is shorter than the verification time using the shortest path approach (even though the path is shortest). This is because using the load balancing approach, the load is distributed among all the nodes in the network and the same subset of links are not used for many OpenFlow session paths. This leads to decrease in the waiting time of switches for performing verification. Additionally, it shows that if the bandwidth in each switch link for verification increases, verification can be performed in limited time.

6.3 Validation on Multiple Topologies

This experiment will evaluate how the verification time varies with the placement of a VM in a network for a range of different topologies, given that the bandwidth of each link between switches available for verification is restricted 1 Mb/s. Figure 14 shows the minimum value, lower quartile, median, upper quartile and the maximum value of the verification time when the VM is placed at the different locations. It shows that if the VM is not placed at the correct switch, the verification time could be as worse as the maximum value. Figure 14 also confirms that the verification time is minimal for all topologies when the VM is placed at the switch containing the maximum value of “Betweenness centrality”. Additionally, we see that the verification time has the order $CT < BT < RT$. This is because BT contains more switches than CT and hence, the VM needs to verify more switches, leading to an increase in the verification time. Additionally, as RT has a lower meshedness than BT, there can be many overlapping session paths links in RT than BT, which leads to an increase in the verification time in RT. Furthermore, as the bandwidth is limited, we see that the load distributed approach performs better than the shortest path approach.

7. Conclusion

In this article, we have proposed a mechanism for the verification of the Flow-Match Header part of Flow Entries in OpenFlow switches. The proposed mechanism might be executed within the OpenFlow controller(s), or within external devices or servers (e.g., on VMs), using the existing data network (in-band verification) or using an external control/verification network (out-of-band). The proposed mechanism was evaluated in extensive emulation experiments. The results illustrated that the verification time depends on the bandwidth available in the network for verification. If bandwidth is unlimited, verification can be achieved in a very short time interval. However, if bandwidth limitations exist, the verification time might increase significantly. Therefore, we proposed a load balancing approach to distribute the load induced by verification traffic among many links in the network. Our results indicate that the approach performs better compared to a load-agnostic shortest path strategy when limited bandwidth is available. Additionally, we evaluated the relationship between verification time and the placement of the verification functionality in the network (i.e., VM placement). Experiments validated that placing the verification functionality close to or at the node with maximal “betweenness centrality” is beneficial with respect to reducing the verification time of the entire process.
Acknowledgment

This research has received funding from the EU FP7 under agreement n° 619609 (UNIFY).

References


[12] ONF OpenFlow features and specifications: www.opennetworking.org


Sachin Sharma received a M.Tech. degree in computer applications in 2007 from IIT Delhi. From 2007 to 2010, he worked as a senior R & D Engineer in Tejas Networks, Bangalore. Since 2010, he has been working for a Ph.D. degree from Ghent University, Belgium. During his Ph.D., he has been involved in many European funded projects: SPARC, CityFlow, OFELIA, and UNIFY. He has more than 20 scientific publications in international conferences and journals.

Wouter Tavernier received a M.Sc. in Computer Science in 2002 from Ghent University. After a two-year period as business analyst at Accenture Belgium, he joined IBCN group of Ghent University in 2005 to research future Internet-topics. This research has lead to a Ph.D. degree in Computer Science Engineering from Ghent University in 2012. His current research focuses on SDN and Service Function Chaining in the context of national and European research projects such as UNIFY.

Sahel Sahhaf received a M.Sc. Degree in Information Technology from Sharif University of Technology (SUT), Tehran, Iran in 2010. Since then she joined the Internet Based Communication Networks and Services (IBCN) research group at Ghent University iMinds as a Ph.D. student. Her research focuses on novel network architectures for the next generation Internet (Greedy router architecture).

Didier Colle received his M.Sc. and Ph.D. degrees in electrotechnical engineering (option: communications) from Ghent University in 1997 and 2002, respectively, and became an associate professor at the same university in 2011. He is group leader of the Future Internet Department of iMinds. His research has published 300 articles in international journals and conference proceedings. He has been very active in FIRE projects, with a focus on OpenFlow and software defined networks.

Mario Pickavet is full professor at Ghent University, where he is teaching courses on discrete mathematics and network modeling. He is co-leading the research cluster on Network Modeling, Design and Evaluation. He has published about 300 international publications, both in journals (IEEE JSAC, IEEE Comm. Mag., Proceedings of the IEEE, ... ) and in conference proceedings. He is co-author of the book Network Recovery: Protection and Restoration of Optical, SONET-SDH, IP, and MPLS.
Piet Demeester is professor in the faculty of Engineering at Ghent University. He is head of the research group Internet Based Communication Networks and Services (IBCN, Ghent University) and leading the Internet Technologies Department of the strategic research center iMinds. He is co-author over 1000 publications in international journals or conference proceedings. He is Fellow of the IEEE.