A Simple Information Flow Security Model for Software-Defined Networks

Dmitry Ju. Chalyy, Evgeny S. Nikitin, Ekaterina Ju. Antoshina
P.G. Demidov Yaroslavl State University
Yaroslavl, Russia
chaly@uniyar.ac.ru, nik.zhenya@gmail.com, kantoshina@gmail.com

***Abstract*—The software-defined networks (SDN) is an emerging paradigm of the networking which became a enabler technology for many modern applications. Cloud computing is a prominent example in the list of such applications including policy-based access control, network virtualization and others. Software nature of the network management can provide flexibility and fast-paced innovations in the netwoking. The other side arise from a complex nature of software, thus, an increasing need for a means to assure its correctness and security. Despite of industrial introduction in cloud environements and other settings there is still a need for abstract models for SDN to tackle challenges in many directions: behavioural models, security models etc. [3].**

The security is a broad field of study even if we limit our attention to software-defined networks security. Furthermore, we bring the confidentiality into our focus. This property asserts that the secret data can not be inferred by an attacker or unintentionally. This is a critical property for multi-tenant SDN environements since the network management software must ensure that no confidential data of one tenant are leaked to other tenants in spite of using the same physical infrastructure. We define a notion of end-to-end security in context of softare-defined networks and propose a model which makes possible to reason about confidentiality and to check that confidential information flows does not interfere with non-confidential ones.

I. INTRODUCTION

The traditional approach to the networking assumes that the network is constructed using vendor-specific hardware which is tightly coupled with a proprietary software implementing distributed protocols. Protocols can provide various services including topology discovery, routing, access control, quality of service and other features. Network operators must install these devices, configure them and set necessary settings for every protocol they intend to use. This tight integration of forwarding and control functionality within proprietary devices restricts innovations and slows down introduction of new network services to modern networks. Bringing open standards and programmability to networks are key points of introduction of software-defined networks (SDN).

Software-defined networks has drawed a lot of attention in recent years and provide a rich set of concepts for centralized management of modern networks. The main principles of SDNs is to provide general principles of packet forwarding level of networks and to decouple control software from forwarding devices. This makes possible to bring innovations to networks without changing the underlying hardware just relying on well-defined standard collection of packet-processing functions which are formed the body of the OpenFlow standard [8]. The SDN controller platform provides the centralized management and orchestration of the whole network inspecting network packets and installing forwarding rules to OpenFlow switches under management.

However, open standards does not solve security problems which are the great challenge in todays networking. An overview of recent challenges for the network security in context of software-defined networks are highlighted in [2]. The centralized control of SDN can benefit in enforcing security strategies but the lack of the models makes this problem challenging [3]. When speaking of network security more precisely we can discuss three problems: integrity, availability and confidentiality. The integrity assumes no data is corrupt due to internal or external events or missconfiguration. Configuration errors can lead to network partioning or misbehaving. For example, this can arise when multiple users write conflicting forwarding rules to OpenFlow switches. This problem was in the focus of research in [10] where authors propose a model checking-based approach to solve such inconsistencies. The availability property means that data is available when is needed. At some extent this property is achieved by load balancing in SDN [12]. The confidentiality considers that secret data cannot be inferred by the attacker. This policy can be imposed using access control lists, encryption etc. The recent research in this direction is represented by works on access control list introduction to SDN [11]. We will focus in our work in confidentiality.

The confidentiality property can be seen in a broad sense but we focus our attention to the end-to-end confidentiality where we must ensure that confidential data is not inferred by the attacker or unintentionally. The confidentiality is hard to achieve even if the system under consideration is not exposed to attacks and we consider insecurities arised form unintentional bugs in implementation. The support for confidentiality checking must be integrated in every high-level language for programming networks. The confidentiality at some extent can be achieved when network resources are separated from each other in slices [5], however, slices are rather isolate part of the network than check confidentiality.

There is an extensive work on semantic foundations of networking programming languages which can provide a solid basis for reasoning about networks. One of the first attemps is Frenetic [6] language which provide abstractions for SDN programming and means for combining these abstractions in a meningful and consistent way. The NetKAT [7] project define such a semantic which can prove reachability in networks and address several security properties at once but the decision procedure for this formalism has PSPACE complexity. On the other hand the confidentiality property was investigated for programming languages [1] and implemented for model [14] and industry-level languages [13]. This approach is based on rigorous semantic rules which impose restrictions for information flows in programming languages. The main idea is not to isolate flows belonging to various slices but allow the network be organized in such a way that different security levels can be used for identifying flows. In our model all hosts can have security levels and proposed type system automatically decides whether the flow is secure.

II. Information Security Model for SDN

A. *Security models for programming languages*

The last two decades showed an increasing interest to security problems in programming languages since there are many critical applications in military, finance, web, medical and many other systems. Traditional programming languages does not have means to preserve confidentiality on a solid basis. Such a solid basis can be achieved using formal methods based on rigorous semantics. Language-based formalisms for enforcing information flow security are appealing tools for tackling this challenge.

The first look to the confidentiality is to enforce isolation mechanisms for different security levels. This can not be recognized as a satisfactory approach since programmers tend to open communication channels across isolation boundaries. Information-flow control allow to analyze such a fine-grained policies defining security semantics for programming languages. Once defined and incorporated to the programming language such a semantic can help to prevent insecure information flows. The key property of a secure information flow is noninterference which essentialy means that a variation of confidential variable does not cause a variation of public variable. In our context we define noninterference as a policy where confidential network flows does not produce any effect on non-confidential.

Static and dynamic language-based security analysis is approved in many areas: mobile programs [17], data bases [18], hardware design [15] and web programming [16]. In this work we try to apply similar approach to a broad domain of software-defined networks.

B. *Software-Defined Network Model*

The main idea behind software-defined network approach is separation of data and control planes of the network. The data plane, or forwarding plane, main responsibility is to do operations on packets arriving to inbound interface. The set of operations can be very rich but the basic ones are: forward to a specific switch interface, drop, update packet headers and many others. The control plane decides when such operations must take place. It must maintain a network state and provide necessary functionality such as calculate routes, support security and install forwarding rules to switches.

Thus, the central entity of a software-defined network is a managed switch which partition the network to multiple logical networks and provide a common programming interface. Such a switch represent a commodity hardware device which comply to the OpenFlow switch specification [9]. The OpenFlow switch is a main component of data plane of the network.

The controller is a principal component of the network control plane which orchestrates OpenFlow switches. Thus, the controller must be connected to a switch through a secure channel [8]. The controller collect information about network traffic, state of the links, interfaces and other events. Aggregating this information helps the controller to maintain network state. Based on this state the controller makes decisions and install forwarding rules to OpenFlow switches using OpenFlow protocol. The controller can implement various applications such as learning switch, a firewall, access control system and many others. The OpenFlow standard does not impose any restrictions to the controller or switch architecture, it rather specify the interface between a switch and a controller.

The OpenFlow switch contains a set of physical or logical *ports* which are the interfaces for passing packets between the switch and the network. According to OpenFlow specification [9] the OpenFlow switch consists of an *OpenFlow channel*, one or more *flow tables*, a *group table*. The OpenFlow channel is used for managing the switch by the controller and to pass relevant data about the traffic under management to the controller. Forwarding and processing packets is implemented by the means provided by flow tables. The controller can add, update and modify flow entries in the flow tables of the switch. Such an entry consists of *match fields*, *counters* and a set of *instructions* to apply to matching packets. The group table enables for a switch additional methods of forwarding by representing a set of ports as a single entity. Thus, group tables does not represent a fundamentally different abstraction and can be modelled via flow tables. We exclude group tables out of our consideration making our model as simple as possible.

Each arriving packet is matched to flow table entries starting from the first one. If the match is found then the instructions associated with this flow entry are executed. If the packet mismatched to each table entry the outcome depends on the table-miss flow entry. Such a packet can be passed to the controller, dropped or handed to next flow table. In our work we will assume that the packet is passed to the controller..

The standard [9] proposes a set of instructions which describe packet forwarding, modification or a group table processing. Switch designers are free in implementation decisions of flow entry instructions provided that the semantics of the instructions are preserved. Flow entries are removed from the switch either a request from a controller, flow entry expiry mechanism or the optional switch eviction mechanism.

Counters are the variables which contains statistical information about flow, for example, received bytes, packets, packet lookups, packets matches etc.

Let us consider a set of instructions which can be executed if a packet is matched to a flow table entry. The standard proposes instructions some of which are required to implement by switch designers and the rest are optional instructions. The action list is associated with each packet during packet processing pipeline. The actions are executed in order specified by the list and are applied immediately to the packet. We consider only the following actions:

- **Output.** This action specifies the port to which the associated packet will be forwarded.

- **Drop.** The packet can be discarded from the network using this action.
- **Set (optional).** The set action allows to modify packet header fields, such as IP and MAC addresses, various tags etc.
- **Delete.** This action deletes flow entries according to a match.

We limit us in considering only listed actions trying to capture the most relevant OpenFlow processing features and not to overwhelm the model.

Thus, we can limit our model to the following. Upon receiving incoming packet p the controller emits an ordered list of match fields each of which paired with an action. This list is installed to the switch. We assume that initialy the controller default miss-match action is to send the packet to the controller.

Summarizing, the controller implements specific network applications. There are many of them. For example, the controller can implement a simple hub application where it installs such forwarding rules to a switch where the incoming packet is flooded to all switch ports except ingress port. Other applications include learning switch, where controller learns what subnets are reachable from different switch ports and install forwarding rules in such a manner that incoming packet goes to a port from which it destination host is reachable, otherwise it is flooded. The controller can implement various security checking policies, for example, allowing to forward packet from authenticated hosts and dropping packets from other hosts.

C. End-to-end Security Model

We will consider a simple model for a software-defined network. We will assume that a network consists of endpoints or hosts which generate data traffic and a set of unified intermediate nodes forwarding traffic. These forwarding devices are OpenFlow switches implementing capabilities listed in previous section. There is a single node representing a controller application which is connected to all the switches of the network and manage switches.

Since the controller application gather all the information about network under management we can assume that the security level of each endpoint is known. The security level can be revealed using some kind of a protocol or can be defined ad-hoc. For the sake of simplicity we will assume that there are two kinds of endpoints: *high* security and *low* security hosts. Since a host is identified by the IP address we can think that the controller can map the space of IP addresses of the network under management into a security level. We will denote a security level of a host h as $h : low$ or $h : high$.

The network itself or its subnets aggregates hosts with different security levels. We will define security predicates $exists$ and $forall$ which will have security types for a set of hosts $\{h_1, \ldots, h_n\}$:

$$\frac{\{h_1 : low, \ldots, h_n : low\}}{\vdash forall(h_1, \ldots, h_n) : low} \tag{1}$$

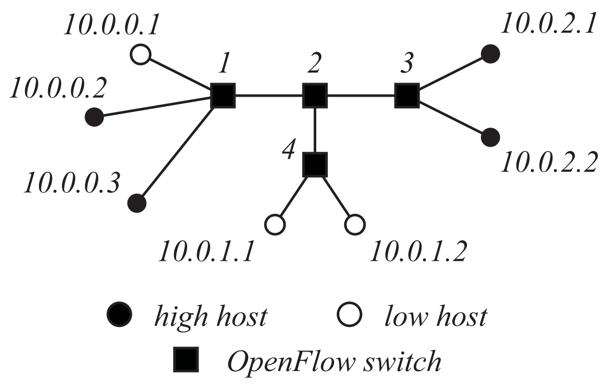

Fig. 1. Sample network with high and low security hosts

$$\frac{\{h_1 : high, \ldots, h_n : high\}}{\vdash forall(h_1, \ldots, h_n) : high\}} \tag{2}$$

$$\frac{\{h_1, \ldots, h_n : \exists h_i : high\}}{\vdash exists(h_1, \ldots, h_n) : high} \tag{3}$$

If the set of hosts is homogenous, so all the hosts have the same security levels, then predicate $forall$ can be typed and the type value is the same as the type of any host in the set. For the network depicted at the Fig. 1 the following holds:

$$forall(\{10.0.0.1, 10.0.1.2\}) : low,$$

$$forall(\{10.0.0.2, 10.0.2.1\}) : high$$

but

$$forall(\{10.0.1.2, 10.0.2.1\})$$

can not be typed. This predicate isolates high and low security hosts. On the other hand the $exists$ predicate is $high$ only if there exists a host in the set of hosts which type is $high$. For example, $exists(\{10.0.0.2, 10.0.1.1\}) : high$. This predicate can not be typed as low and our further discussion shows that we only need to capture a possibility to reach a high security host.

Each flow table entry contains a matching field. This field is a predicate which partions a set of all flows through the network. The standard [9] proposes that matching field is a conjunctive predicate with each conjunct can impose conditions on different parts of the packet header. Each flow has a source and a destination host. Thus, a matching field can be modelled as a pair $match = (match_{src}, match_{dst})$, where $match_{src}$ is a template for matching source host of the packet and $match_{dst}$ is a template for matching destination host. We define functions src and dst which map a matching field to the set of source and destination hosts respectively which evaluate the matching field to be true. For example, $src(10.0.0.X) = \{10.0.0.1, 10.0.0.2, 10.0.0.3\}$. If the controller gathers all the information about network hosts then the security type of considered predicates can be effectively computed.

The next part of the model is a *packet processing context*. When the Openflow switch can not match the packet to any flow table entry the switch can forward the packet to the controller. The controller can examine header fields of the packet specifically IP address/MAC or any other identifying header field and determine which host emitted the packet. The security type of this host imply the processing context of the packet. The processing context defined in a straighforward way:

$$\frac{\vdash forall(src(pkt)) : pc}{[pc] \vdash pkt} \tag{4}$$

So the packet processing context pc agrees with the security type of the source host which originally emitted the packet. We consider a packet processing context as a first action in the instruction list emitted by the controller.

If the controller decides to emit *Output(port)* action paired with a matching condition $match$, then it can be typed as following:

$$\frac{\vdash exists(dst(match)) : high \quad \vdash exists(portset) : high}{[high] \vdash match \times Output(port)} \tag{5}$$

$$\frac{\vdash forall(src(match)) : low}{[low] \vdash match \times Output(port)} \tag{6}$$

We denote as $portset$ a set of hosts reachable from port of the switch where the *Output* action will forward packets. The reachability here is assumed in a broad fashion, i.e. if there are links that form path from the port to a host then the host is in the set $portset$. The definition of the $portset$ imply a strict condition on reachability, because the existence of a such path does not guarantee that packets emitted from the port can effectively reach the host from the $portset$. Some of them, for example, may be dropped at an intermediate switch of the path. Here we try to capture a potential reachability. The set $portset$ can be effectively computed since the controller collects information about network graph.

The result of *Output(port)* typization depends heavily on matching condition. If this condition forwards traffic to high security hosts, then there must be a high security host reachable from the port. In this case the security context of *Output(port)* is high. If the source of the traffic is a low security host, then it can be forwarded anywhere and the security context of this action is low. The *Output(port)* action can not be typed if match condition specifies that traffic from high security hosts must be forwarded to low security host. In such a case $forall(src(match))$ can not be typed as low and $exists(dst(match))$ can not be typed as high implying that premises for both rules does not hold.

The *Drop* action can be typed as following:

$$\frac{\vdash forall(src(match)) : pc}{[pc] \vdash match \times Drop} \tag{7}$$

For the *Drop* action we strictly isolate flows of different security levels, that is, if the flow originates from a low security host then its packets can be dropped in a low context, otherwise the context is high. Such a typization prevents high security packet processing context to install drop rules to the switch such that the rules drop low security packets. Violation of such a behaviour can lead to a covert channel when low security hosts discover that high security host installs such a drop rule. Setting low type to the *Drop* rule ensures that low security packet processing context can install a drop rule which affects only low security flows. Non-interference property holds even if we allow a low security packet processing context to drop high security flows since no information about high security flows can not be inferred. Discarding such a behaviour guarantee integrity for high security hosts at some extent.

For the *Set(pattern)* action we propose the following rules:

$$\frac{\vdash forall(src(match)) : low \quad \vdash forall(src(pattern)) : low}{[low] \vdash match \times Set(pattern)} \tag{8}$$

$$\frac{\begin{array}{c}\vdash forall(src(match)) : high \\ \vdash forall(dst(match)) : high \\ \vdash forall(dst(pattern)) : high \\ \vdash forall(src(pattern)) : high\end{array}}{[high] \vdash match \times Set(pattern)} \tag{9}$$

The first rule guarantees that any low security flow can not become a high security flow by changing the source address of the packet. Imposing such a condition we achieve a certain level of integrity since a low packet can not start to be a high packet and become a reason for a controller to influence to other high security flows. The second rule assures that a high security flow stays a high security flow making sure that there is no information leak to the low security plane.

The security type of the *Delete* action can be inferred using the following rules:

$$\frac{\vdash forall(dst(match)) : high}{[high] \vdash match \times Delete} \tag{10}$$

$$\frac{\vdash forall(src(match)) : low}{[low] \vdash match \times Delete} \tag{11}$$

These two rules guarantee that the eviction of flows from the switch is done in the respective security context.

Actions are combined to the list starting with packet processing context virtual action using the next typing rule:

$$\frac{[pc] \vdash A \; [pc] \vdash B}{[pc] \vdash A; B} \tag{12}$$

Proposed rules constitues a security-type system which describes what security type must be assigned to a list of actions. This list of actions is formed by a controller in response to a packet incoming from the switch. The packet itself specifies the first action in the list called packet processing context. If the whole list is typable using proposed security-type system then the list ensures non-interference between flows of different security levels and some level of integrity which states that low security flows can not be a reason for dropping high security packets.

III. Example Application of the Model

We consider a learning switch application as an example. The switches in the network initially have no flow entries and forward incoming packets to the controller. The controller examines each packet and stores in the internal database the source address of the packet along with the port from where it is received. The port and packet headers are forwared to the controller as an OpenFlow *packet in* message. The next time the switch receives the packet destined to the address that was learned earlier, the controller can infer the port to which the packet must be forwarded. If the port can not be determined then the packet is flooded to all the switch ports.

Algorithm 1 Learning switch algorithm

```
1:  $pkt \leftarrow$ packet arrived to the controller
2:  $port \leftarrow$ from which port $pkt$ received
3:  **if** find(src($pkt$)) is null **then**
4:      push (src($pkt$), $port$)
5:  **end if**
6:  $fport \leftarrow$ find(dst($pkt$))
7:  **if** $fport$ is null **then**
8:      **for all** switch port $i$ other than $port$ **do**
9:          **emit** (src($pkt$),dst($pkt$))$\times$Output($i$)
10:     **end for**
11: **else**
12:     **emit** (src($pkt$), dst($pkt$))$\times$Output($fport$)
13:     **emit** (dst($pkt$), src($pkt$))$\times$Output($port$)
14: **end if**
```

The simple algorithm for the learning switch is shown as Algorithm 1. The input data for the algorithm is an incoming packet pkt and the port $port$ from which it has been received. The controller maintains an internal database which can be implemented as a hash which supports the following operations:

- *push(address, port)*. The operation creates a mapping between the $address$ and the $port$ in the internal database.
- *find(address)*. This is a query to the database which returns port number associated with $address$ and $null$ if there is no such an association;

There is a **emit** operator in our language which appends the action to the list of instructions destined to the switch. The list is sent to the switch when the algorithm is stopped. Then we can analyze the list and find if it is secure or not.

The Algorithm 1 checks whether a mapping between a source address of pkt and $port$ exists and writes such an association if not in lines 3–5. Then we try to find if we have learned the port to which we can forward the packet pkt (line 6). If no such a port exists then we flood the packet to all ports except ingress port (lines 8–10). Otherwise, we emit forwarding rules which set up a duplex channel between source and destination hosts of the packet (lines 12–13). We assume that entries responsible for flooding packets are eventually will be evicted from switches and replaced by direct forwarding entries.

Recall a network from Fig. 1. Assume that the controller database is empty and there is no forwarding rules at switches, so each switch sends a *packet in* message to the controller upon a packet receipt. The security flaw arises even when the first packet travels from any high security host. For example, if the host 10.0.0.2 sends a packet pkt to the host 10.0.2.1, then the following list of rules will be emitted by the controller to the switch 1 according to the lines 8–10 of the Algorithm 1:

$$\begin{gathered}(10.0.0.2, 10.0.2.1) \times Output(1)\\ (10.0.0.2, 10.0.2.1) \times Output(3)\\ (10.0.0.2, 10.0.2.1) \times Output(4)\end{gathered}$$

The first instruction installs the rule which forwards all packets from high security host 10.0.0.2 to a low security host 10.0.0.1. Let us try to discover a security type of packet pkt processing.

First, by the rule 2 we can infer that

$$\frac{10.0.0.2 : high}{\vdash forall(\{10.0.0.2\}) : high}$$

Since $src(pkt) = \{10.0.0.2\}$ using rule 4 the following holds

$$\frac{\vdash forall(\{10.0.0.2\}) : high}{[high] \vdash pkt}$$

Next, we should discover the type of the action $(10.0.0.2, 10.0.2.1) \times Output(1)$. Let us denote as $match = (10.0.0.2, 10.0.2.1)$, $src(match) = \{10.0.0.2\}$, $dst(match) = \{10.0.2.1\}$ and the $portset = \{10.0.0.1\}$. Thus,

$$\vdash exists(src(match)) : high$$

but

$$\nvdash exists(portset) : high$$

$$\nvdash forall(\{10.0.0.2\}) : low$$

so the premises for the rule 5 does not hold.

Likewise,

$$\nvdash \forall(src(match)) : low$$

hence we can not infer the only premise for the rule 6. Thus, the considered action can not be typed, so the whole list can not be typed.

The Algorithm 2 proposes an enhanced version of the learning switch. This version is free from many security leaks but let us analyze it formally. The algorithms breaks into two parts. The first one is represented by lines 8–19 where packets from low sources are processed. If the output port can not be identified, then the packet is flooded to all ports of the switch (lines 9–11), otherwise forwarding rules are installed to the switch. This rules include one which redirects packet pkt to the destination host (line 13 and the other which either creates a channel with the opposite direction (line 15) or set the action to $Drop$ if the opposite forwarding rule forms a route from high host to low host (line 17). The second part of the algorithm process packets from high sources (lines 21–31). If the destination for such a high packet is a low host, then we drop the packet (line 22). If the controller does not find

Algorithm 2 Secure learning switch algorithm

```
 1: $pkt \leftarrow$ packet arrived to the controller
 2: $port \leftarrow$ from which port $pkt$ received
 3: if find(src($pkt$)) is null then
 4:     push(src($pkt$), $port$)
 5: end if
 6: $fport \leftarrow$find(dst($pkt$))
 7: if src($pkt$):$low$ then
 8:     if fport is null then
 9:         for all switch port $i$ other than $port$ do
10:             emit (src($pkt$), dst($pkt$))$\times$Output($i$)
11:         end for
12:     else
13:         (src($pkt$), dst($pkt$))$\times$Output($fport$)
14:         if (dst($pkt$):$low$) then
15:             emit (dst($pkt$), src($pkt$))$\times$Output($port$)
16:         else
17:             emit (dst($pkt$), src($pkt$))$\times$Drop
18:         end if
19:     end if
20: else
21:     if dst($pkt$):$low$ then
22:         emit (src($pkt$), dst($pkt$))$\times$Drop
23:     else
24:         if $fport$ is null then
25:             for all switch port $i$ other than $port$ and $exists(i)$ :
                $high$ do
26:                 emit (src($pkt$), dst($pkt$))$\times$Output($i$)
27:             end for
28:         else
29:             emit (src($pkt$), dst($pkt$))$\times$Output($fport$)
30:             emit (dst($pkt$), src($pkt$))$\times$Output($port$)
31:         end if
32:     end if
33: end if
```

the port to forward the packet, then the packet is flooded but only to high ports (lines 25–27), otherwise forwarding rules are installed (lines 29–30).

Let us show how security properties of the Algorithm 2 can be proved. If condition at the line 8 is true then the following holds for lines 8–19 by the rule 1:

$$\frac{\{src(pkt) : low\}}{\vdash forall(src(pkt)) : low}$$

And by the rule 4:

$$\frac{\vdash forall(src(pkt)) : low}{[low] \vdash pkt}$$

Assume $fport$ is null, then the packet must be flooded to all ports except $port$ (lines 9–11). So, the controller emits *packet out* messages which can be typed using inference rule 6:

$$\frac{\vdash forall(src(pkt)) : low}{[low] \vdash (src(pkt), dst(pkt)) \times Output(i)}$$

Applying rule 12:

$$\frac{[low] \vdash pkt \quad [low] \vdash (src(pkt), dst(pkt)) \times Output(i)}{[low] \vdash pkt; (src(pkt), dst(pkt)) \times Output(i)}$$

Thus, the whole list of emitted actions is typed and these actions are safe.

Assume $fport$ is not null, then controller emits action at the line 13 which is safety can be ensured using the same inference as in flooding case above. The second action of the list depends on security type of dst(pkt). If it is low, then the action at the line 15 is emitted. The security type of the action is the following:

$$match = (dst(pkt), src(pkt))$$

$$\frac{\{src(match) : low\}}{\vdash forall(src(match)) : low} \quad \text{(rule 1)}$$

$$\frac{forall(src(match)) : low}{\vdash [low] \vdash (src(pkt), dst(pkt)) \times Output(port)} \quad \text{(rule 6)}$$

Thus, the security type of all emitted actions agree, so the whole list can be typed as low. If dst(pkt) is high (line 17) then only the following can be inferred:

$$match = (dst(pkt), src(pkt))$$

$$\frac{\{src(match) : high\}}{\vdash forall(src(match)) : high} \quad \text{(rule 2)}$$

$$\frac{\vdash forall(src(match)) : high}{[high] \vdash match \times Drop} \quad \text{(rule 7)}$$

This means that the security type of the $Drop$ action from the line 17 does not agree with the security type of previous actions and packet processing context which are low. Thus, the $Drop$ action can not be considered safe. Indeed, low packets must not trigger dropping packets originated from high security hosts. If we carefully examine the code, we will see that such a drop is made at line 22 when the packet processing context is high. Hence, we can remove line 17 from our algorithm without harming learning switch functionality.

If the packet pkt is originated from a high security host then the Algorithm 2 proceeds to lines 21–31. The packet processing context is now $high$:

$$\frac{\{src(pkt) : high\}}{\vdash forall(src(pkt)) : high} \quad \text{(rule 2)}$$

$$\frac{\vdash forall(src(pkt)) : high}{[high] \vdash pkt} \quad \text{(rule 4)}$$

Then three possibilities can occur:

1) Either a $Drop$ action is emitted (line 22):

$$match = (src(pkt), dst(pkt))$$

$$\frac{\vdash forall(src(match)) : high}{[high] \vdash match \times Drop} \quad \text{(rule 7)}$$

2) Or packet is flooded using a list of $Output$ actions (lines 25–27):

$$\vdash exists(i) : high \quad \text{(condition in line 25)}$$

$$match = (src(pkt), dst(pkt))$$

since condition in line 21 does not hold

$$\frac{\{dst(match) : high\}}{\vdash exists(dst(match)) : high}$$

so using the rule 5 we can obtain

$$\frac{\vdash exists(dst(match)) : high \quad \vdash exists(i) : high}{[high] \vdash match \times Output(i)}$$

3) Or bidirectional forwarding is set (lines 29–30). Since both src(pkt):$high$ and dst(pkt):$high$ and it was learned that such packets are came from ports $port$ and $fport$ respectively, we can conclude that $exists(port) : high$ and $exists(fport) : high$. Using inference similar to the previous case we can obtain that both $Output$ actions are typed as $[high]$.

Thus, in all three cases all emitted actions are typed as $high$. This agrees with the packet processing context and we can conclude that the whole list of emitted actions must be typed as $[high]$. That is the list is safe.

We have considered all the cases and all lists of actions the controller can install to a switch. We have found a case where a packet from a low security flow can trigger dropping packet from a high security flow. This shows that the proposed approach can find very subtle security discrepancies. In the context of our application this can not be considered as a security flaw, but it can lead to security leaks in more general settings.

IV. Conclusion

Security of software-defined networks is challenging. There is a lack of formal models for making security analysis for software-defined networks and the paper proposes such an approach which is based on formal security-type system. This system ensures that the controller application does not violate security properties such as confidentiality and, at some extent, integrity. The security system can be implemented as a software module of the controller and check security properties online.

There is more to explore in this direction. There are both theoretic and practical challenges. It is interesting to explore soundness and completness of proposed type system. Another fascinating problem is to combine security-type systems for programming languages and the proposed one for achieving a solid theoretical basis for static security analysis which can prove properties of an SDN controller at the compilation stage.

Acknowledgment

The authors would like to appreciate the Russian Fund for Basic Research (RFBR) for partial support under project 14-01-31539.

The research was conducted at P.G. Demidov Yaroslavl State University using financial support of the Ministry of Education and Science of the Russian Federation under contract ID RFMEFI57414X0036.

References

[1] A. Sabelfeld, A.C. Myers, "Language-Based Information-Flow Security", *IEEE Journal on Selected Areas in Communications*, vol. 21, 2003, pp. 5–19.

[2] R. Smeliansky, "SDN for Network Security", *Proc. of Int. Conf. "Modern Networking Technologies (MoNeTec), Moscow, Russia*, 2014, pp. 155–159.

[3] M. Casado, N. Foster, A. Guha, "Abstractions for Software-Defined Networks", *Communications of the ACM*, vol. 57, No 10, 2014, pp. 86–95.

[4] S. Gutz, A. Story, C. Schlesinger, N. Foster, "Splendid Isolation: A Slice Abstraction for Software-Defined Networks", *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN) , Helsinki, Finland*, August, 2012, pp. 79–84.

[5] R. Sherwood et al. "Carving Research Slices Out of Your Production Networks with OpenFlow", *ACM SIGCOMM Computer Communication Review*, vol. 40, No 1, 2010, pp. 129–130.

[6] N. Foster, M.J. Freedman, Ch. Monasanto, J. Rexford, A. Story, D. Walker, "Frenetic: A Network Programming Language" *ACM Int. Conf. on Functional Programming*, Japan, 2011, pp. 279–291.

[7] C.J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, D. Walker *NetKAT: Semantic Foundations for Networks*, Proc. of ACM Symp. on Principles of Programming Languages, 2014, pp. 113–126.

[8] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, "OpenFlow: Enabling Innovation in Campus Networks", *SIGCOMM Computer Communications Review*, Vol. 38, No 2, 2008, pp. 69–74.

[9] "OpenFlow Switch Specification v. 1.4.0", Open Networking Foundation, 2013, 205 p. URL: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf, Last accessed: 05.03.2015.

[10] E. Al-Shaer, S. Al-Haj, "FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures", *In Proc. of SafeConfig*, 2010, pp. 37–44.

[11] M. Casado, M.J. Freedman, J. Pettit, J. Luo, N. McKeown, S. Shenker, "Ethane: Taking control of the enterprise", *Proc. of SIGCOMM, 2007.*

[12] Hong, C-Y, Kandula, S., Mahajan, R., Zhang, M., Gill, V., Nanduri, M. and Wattenhofer, R. Achieving high utilization with software-driven WAN. In Proceedings of SIGCOMM, 2013.

[13] Danfeng Zhang, Owen Arden, Jed Liu, K. Vikram, S. Chong, A. Myers, "Jif: Java+Information flow", http://www.cs.cornell.edu/jif/

[14] E.Ju. Antoshina, A.N. Barakova, E.S. Nikitin, D. Ju. Chalyy, "A translator with a security static analysis feature of an information flow for a simple programming language", *Automatic Control and Computer Sciences*, Vol. 48, No 7, 2014, pp. 589–593.

[15] D. Zhang, Y. Wang, G. Edward Suh, A.C. Myers, "A hardware design language for Timing-Sensitive Information-Flow Security", *Proc. of the 20th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS15)*, 2015.

[16] D. Hedin, A. Birgisson, L. Bello, A. Sabelfeld, "JSFlow: Tracking Information Flow in JavaScript and its APIs", *Proc. of the 29th Annual ACM Symposium on Applied Computing*, 2014, pp. 1663–1671.

[17] O. Arden, M.D. George, J. Liu, K. Vikram, A. Askarov, A.C. Myers, "Sharing Mobile Code Securely With Information Flow Control", *IEEE Symp. on Security and Privacy (SP)*, 2012, pp. 191–205.

[18] A. Cheung, S. Madden, A. Solar-Lezama, O. Arden, A.C. Myers, "Using Program Analysis to Improve Database Applications", *IEEE Data Eng. Bull.*, Vol. 37, No 1, 2014, pp. 48–59.