Моделирование и анализ информационных систем. Т. 22, №6 (2015), с. 735–749
Modeling and Analysis of Information Systems. Vol. 22, No 6 (2015), pp. 735–749

©Chaly D. Ju., Nikitin E. S., Antoshina E. Ju., Sokolov V. A., 2015

DOI: 10.18255/1818-1015-2015-6-735-749

UDC 519.987

End-to-end Information Flow Security Model for Software-Defined Networks

Chaly D. Ju.[1], Nikitin E. S., Antoshina E. Ju., Sokolov V. A.[2]

Received October 21, 2015

Software-defined networks (SDN) are a novel paradigm of networking which became an enabler technology for many modern applications such as network virtualization, policy-based access control and many others. Software can provide flexibility and fast-paced innovations in the networking; however, it has a complex nature. In this connection there is an increasing necessity of means for assuring its correctness and security. Abstract models for SDN can tackle these challenges. This paper addresses to confidentiality and some integrity properties of SDNs. These are critical properties for multi-tenant SDN environments, 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 software-defined networks and propose a semantic model where the reasoning is possible about confidentiality, and we can check that confidential information flows do not interfere with non-confidential ones. We show that the model can be extended in order to reason about networks with secure and insecure links which can arise, for example, in wireless environments.

The article is published in the authors' wording.

Keywords: SDN, security, formal models

For citation: Chaly D. Ju., Nikitin E. S., Antoshina E. Ju., Sokolov V. A., "End-to-end Information Flow Security Model for Software-Defined Networks", *Modeling and Analysis of Information Systems*, **22**:6 (2015), 735–749.

On the authors:
Chaly Dmitriy Jurevich, orcid.org/0000-0003-0553-7387, PhD,
P.G. Demidov Yaroslavl State University,
Sovetskaya str., 14, Yaroslavl, 150000, Russia, e-mail: dmitry.chaly@gmail.com

Nikitin Evgeniy Sergeevich, orcid.org/0000-0002-2341-9950, student,
P.G. Demidov Yaroslavl State University,
Sovetskaya str., 14, Yaroslavl, 150000, Russia, e-mail: nik.zhenya@gmail.com

Antoshina Ekaterina Jurevna, orcid.org/0009-0203-2514-7755, graduate student,
P.G. Demidov Yaroslavl State University,
Sovetskaya str., 14, Yaroslavl, 150000, Russia, e-mail: kantoshina@gmai.com

Sokolov Valeriy Anatolevich, orcid.org/0000-0003-1427-4937, doctor of science,
P.G. Demidov Yaroslavl State University,
Sovetskaya str., 14, Yaroslavl, 150000, Russia, e-mail: valery-sokolov@yandex.ru

Acknowledgments:
[1]This work was supported by the RFBR under the project 14-01-31539 mol-a.
[2]This work was supported by the Ministry of Education and Science of the Russian Federation under contract ID RFMEFI57414X0036.

Introduction

The traditional approach to the networking assumes that a network is constructed using vendor-specific hardware which is tightly coupled with a proprietary software which implements 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 and configure 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 have drawn a lot of attention in recent years and provide a rich set of concepts for centralized management of modern networks. The main aim of SDNs is to provide general principles of packet forwarding and to decouple control software from forwarding devices. This makes it possible to bring innovations to networks without changing the underlying hardware just relying on a well-defined standard collection of packet-processing functions that forms the body of the OpenFlow standard [9]. Software controller provides a centralized management and orchestration of the whole network inspecting network packets and installing forwarding rules to switches under management.

However, the standard does not solve security problems which are the great challenge in todays networking [12]. The centralized control of SDN can benefit in enforcing security strategies, however, the lack of models makes this problem challenging [4]. We can discuss the security of SDN in three dimensions: integrity, availability and confidentiality. The integrity assumes that no data is corrupt due to internal or external events or misconfiguration. This problem was in the focus of study in [1] where the authors propose a model checking-based approach to find configuration inconsistencies that can lead to network partitioning. The availability property means that data are available when needed. At some extent this property is achieved by load balancing in SDN [8].

The confidentiality considers that secret data cannot be inferred by an attacker or unintentionally. This policy can be imposed by using access control lists, encryption etc. One of the recent attempts that introduce access control lists to SDN is [5]. However, access control does not prevent leaks of confidential data through improperly configured or buggy software [11]. The confidentiality property can be seen in a broad sense, so we focuse on the end-to-end confidentiality. We assume that an attacker can observe non-confidential entities of the network and has a limited access to the network infrastructure. The confidentiality can be achieved at some extent when network resources are separated from each other in slices [7], however, slices isolate flows in the network, thus, are too restrictive. Software nature of network control in SDN is a cause for a try to apply security methods that are developed for programming languages [11].

There is an extensive work on semantic foundations of networking programming languages that can provide a solid basis for reasoning about networks. One of the first attempts was Frenetic language [6] that provided abstractions for SDN programming and means for combining these abstractions in a meaningful and consistent way. The NetKAT project [2] defined a semantics that can help to prove reachability in networks

(which is an integrity property) and address several security properties at once, however, the decision procedure for this formalism has PSPACE complexity. Focusing only on confidentiality may reduce complexity of verification. The confidentiality property was investigated for programming languages [11] and implemented for model [3] and industry-level languages [13]. This approach is based on rigorous semantic rules that impose restrictions on information flows in programming languages.

In this paper we propose a framework for checking confidentiality on-the-fly for modern SDNs that are conformed to the OpenFlow standard. We introduce a set of semantic rules that help us to verify that controller application does not allow non-confidential information flows. We assume that network consists of high and low security nodes and latter we extend our concept to a model of network that can contain secure and insecure links.

Consider a simple model of a software-defined network. Let us assume that the network consists of endpoints or hosts that generate data traffic and a set of unified intermediate nodes forwarding the traffic. These forwarding devices are OpenFlow switches that conform to standard [9]. There is a single node representing a controller application that manages all the switches by using secure channels. Thus, a network can be represented as a graph where the nodes are either hosts or switches, and the edges are links.

The OpenFlow switch contains a set of physical or logical *ports* which are interfaces for passing packets between the switch and the network. According to specification [10] the OpenFlow switch consists of an *OpenFlow channel*, one or more *flow tables*, and a *group table*. The OpenFlow channel is used for managing the switch and for passing relevant data about the traffic under management to the controller. Flow tables provide means for forwarding and processing packets. The controller can add, update or modify flow entries in flow tables. Such an entry consists of *match fields*, *counters* and a set of *instructions* to apply to matching packets. The group table enables additional methods of forwarding by representing a set of ports as a single entity. Thus, group tables do not represent a fundamentally different abstraction and can be modeled via flow tables. So, we exclude group tables out of consideration.

Each arriving packet is matched to flow table entries starting from the first one. If the match is found, instructions associated with this flow entry are executed. If the packet is 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 the next flow table. We will assume that the packet is passed to the controller.

The match field is a predicate which partitions the set of all flows passing through the network. Standard [10] proposes that matching field is a conjunctive predicate where each conjunct can impose conditions on various packet headers including Ethernet, IP, TCP, etc. Each flow has a source and a destination host. Thus, the matching field can be modeled as $m_{src} \wedge m_{dst}$, where m_{src} and m_{dst} are conjuncts for matching the source and destination hosts of the flow, respectively.

Counters are variables that contain statistical information about flows. Since counters have no direct impact on forwarding, exclude them out of consideration.

Let us consider instructions that can be executed if a packet is matched to a flow table entry. The standard proposes that instructions are lists of actions. Some of these actions are required to be implemented by switch designers and the rest are optional.

The actions are executed in the order specified by the list and are applied immediately to the packet. We consider only the following actions:

- *Output(port).* 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.* The optional 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 ourselves to considering only listed actions when trying to capture most relevant OpenFlow processing features and not to overwhelm the model.

The packet processing model is the following. Upon receiving an incoming packet p, the controller emits an ordered list of match fields each of which is paired with an action. This list is installed to the switch.

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

1. End-to-end Security Model for SDN

The controller application gathers all the information about the network under management. So, 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 assume that there are two security levels of endpoints: *high* and *low*. 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 set of security levels. Denote a security level of a host h as $h : low$ or $h : high$.

For further discussion we need means for reasoning about sets of hosts. The network itself or its subnets aggregates hosts with different security levels. Define security predicates *exists* and $forall$ that will give us a security type for a set of hosts $\{h_1, \dots, h_n\}$ as shown in Fig. 1.

If the set of hosts is homogeneous, i.e. all hosts have the same security level, the predicate $forall$ can be typed with the same security type as any host in the set. On the other hand, the *exists* predicate is *high* only if the set contains a *high* host. This predicate can not be typed as *low* and it will be seen later that we only need to check a possibility to reach a high security host.

One of the primary functions of the controller is routing that is essentially reasoning about reachability in networks. Model the network as a graph, so we are forced with

$$\frac{\{h_1 : low, \ldots, h_n : low\}}{\vdash forall(h_1, \ldots, h_n) : low} \quad (1) \qquad \frac{\{h_1 : high, \ldots, h_n : high\}}{\vdash forall(h_1, \ldots, h_n) : high} \quad (2)$$

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

Fig 1. Security types for sets of hosts

deducing reachable hosts from a given switch. Thus, we define a function $reachable(s, p)$ that evaluates a set of hosts reachable from a switch s if we first go to the port p. It is not easy to calculate reachability in real networks since the network can be dynamic because of mobility of hosts and installed forwarding rules. However, a superset of the set of reachable hosts can be computed using breadth-first search on a network graph. More accurate algorithms that take into account network policies can be found in [2].

The data plane of the network is represented by switches that use flow tables for implementation of network policies. Each flow table entry contains a matching field that is modeled as a predicate $match = m_{src} \wedge m_{dst}$. We define the functions src and dst that map a predicate to a set of source and destination hosts, respectively, such that the predicate is true.

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 model assumes that the packet is forwarded to the controller. The controller can examine headers of the packet and determine the host that emitted the packet. Security type of the host implies the packet processing context so we can analyze whether the controller generates a secure response to the packet or not.

A security-type system can help to reason about the security type of a single interaction between a switch and a controller. Figure 2 presents typing rules for instructions that can be installed to a switch s by the controller in response to a packet pkt. We can use the presented security-type system for inferring a type of the interaction. If the type can be inferred, the interaction is secure, otherwise it allows leaks of confidential data.

Let us consider a proposed set of typing rules in more detail. Rule 4 assigns a type for a packet processing context in such a way that the context pc agrees with the security type of the source host of the packet pkt. The packet processing context is a virtual action in the list formed by the controller.

For the *Drop* action (rule 5) we strictly isolate flows of different security levels, that is, the source host of the flow must correspond to the context of the action. Such a type setting prevents interference between packet processing contexts and actions of different security levels. Violation of this can lead to a covert channel when low hosts discover that a high host installs *Drop* action by observing occasional drops. Setting low type to the *Drop* action ensures that under the low security packet processing context a drop can occur only for 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 be inferred. However, we discard this and guarantee that integrity for high security flows can not be broken by low hosts.

$$\frac{\vdash forall(src(pkt)) : pc}{[pc] \vdash pkt} \quad (4) \qquad \frac{\vdash forall(src(match)) : pc}{[pc] \vdash match \times Drop} \quad (5)$$

$$\frac{\vdash exists(dst(match)) : high \quad \vdash exists(reachable(s, port)) : high}{[high] \vdash match \times Output(port)} \quad (6)$$

$$\frac{\vdash forall(src(match)) : low}{[low] \vdash match \times Output(port)} \quad (7)$$

$$\frac{\vdash forall(dst(match)) : high}{[high] \vdash match \times Delete} \quad (8) \qquad \frac{\vdash forall(src(match)) : low}{[low] \vdash match \times Delete} \quad (9)$$

$$\frac{\vdash forall(src(match)) : low \quad \vdash forall(src(pattern)) : low}{[low] \vdash match \times Set(pattern)} \quad (10)$$

$$\frac{\begin{array}{l}\vdash forall(src(match)) : high \\ \vdash forall(dst(pattern)) : high \\ \vdash forall(src(pattern)) : high\end{array}}{[high] \vdash match \times Set(pattern)} \quad (11) \qquad \frac{[pc] \vdash A \; [pc] \vdash B}{[pc] \vdash A; B} \quad (12)$$

Fig 2. Security-type system for SDN

The *Output(port)* action type depends heavily on the matching condition (rules 6–7). If the match forwards traffic to high security hosts, 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, it can be forwarded anywhere and the security context of this action is low. The *Output(port)* action can not be typed if the match condition specifies that traffic from high security hosts must be forwarded to a low security host. If this is the 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 do not hold.

Rules 8–9 for *Delete* action guarantee that the eviction of flows from the flow table of the switch is done in the respective security context. So, a low packet can not be a reason to remove high matches and vice versa.

Rule 10 guarantees that any low security flow can not become a high security flow by changing the source address of the packet. By imposing this condition we achieve a certain level of integrity since a low packet can not become a high packet that may later influence other high security flows. Rule 11 assures that a high security flow stays high providing no information leak to the low security plane. In both rules we denote as a *pattern* the data that have to be written to the packet header.

The controller can respond with several actions at once, thus we must have means for inferring a security type for a list of actions. This can be done using rule 12 that assigns a security type for a composition. Here, A and B can be either single actions or lists of actions.

The proposed rules constitute 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 specifies the first action in the list called a packet processing context. If the whole list can be typed using the proposed security-type system, the list ensures non-interference among flows of different security levels and fulfills some integrity properties.

This security-type system can be further extended to SDNs with insecure links. We can define an insecure link as a channel that can not be trusted since they are exposed to everyone like Wi-Fi medium or may be public channels shared by various tenants. This setting leads to a new confidentiality violation since high data traffic may be forwarded to an insecure link. It could be noted that any link can be secured using traffic encryption. We propose the following extension to our model. Let us assume that every link has a security level (*high* for secure links and *low* for insecure ones) and it is known to the controller. It is the same that we did for endpoints. Also we must provide means of reasoning about secure paths in the network.

Let us define a function $reachable_s(s, port)$ that calculates a set of hosts that are reachable via paths such that every link in the path is secure. Since the controller has the information about the network graph, it can be done using breadth-first search or taking into account current network policies [2].

Since a confidentiality flaw can occur when high traffic is forwarded to an insecure link, we must only refine rule 6 that is used for inferring the type for *Output* action considering high traffic. We propose the following change:

$$\frac{\vdash exists(dst(match)) : high \quad \vdash exists(reachable_s(s, port)) : high}{[high] \vdash match \times Output(port)} \tag{13}$$

Thus, we allow high traffic only to those switch ports that start with a secure link and have the possibility to reach the destination host using a secure path.

This shows that the proposed model can be used as a basis for reasoning about various aspects of confidentiality in software-defined networks.

2. An Example of the Model Application

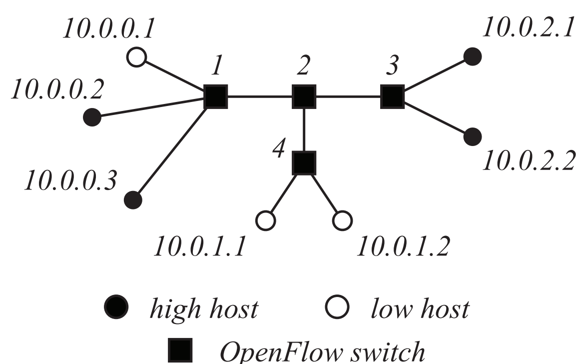

Fig 3. A sample network with high and low security hosts

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 was received. The port and packet headers are forwarded to the controller as an OpenFlow *packet in* message. Next time the switch receives the packet destined to the address that was seen earlier, the controller can infer the port to which the packet must be forwarded. If the port can not be determined, the packet is flooded to all the switch ports.

Algorithm 1 Learning switch algorithm

```
1: pkt ← packet arrived to the controller
2: port ← from which port pkt received
3: if find(src(pkt)) is null then
4:     push (src(pkt), port)
5: end if
6: fport ← 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))×Output(i)
10:    end for
11: else
12:    emit (src(pkt), dst(pkt))×Output(fport)
13:    emit (dst(pkt), src(pkt))×Output(port)
14: end if
```

A 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 an **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.

Algorithm 1 checks whether a mapping between a source address of *pkt* and *port* exists. If there is no such mapping, it writes it in lines 3–5. Thereafter, 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 will be eventually evicted from switches and replaced by direct forwarding entries.

Recall the network from Fig. 3. 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, the following list of rules will be emitted by the controller to the switch 1 according to lines 8–10 of Algorithm 1:

$$(10.0.0.2, 10.0.2.1) \times Output(1)$$
$$(10.0.0.2, 10.0.2.1) \times Output(3)$$
$$(10.0.0.2, 10.0.2.1) \times Output(4)$$

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 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 $reachable(s, port) = \{10.0.0.1\}$. Thus,

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

but

$$\nvdash exists(reachable(s, port)) : high$$

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

so the premises for rule 6 not hold.

Likewise,

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

hence we can not infer the only premise for rule 7. Thus, the considered action can not be typed, so the whole list can not be typed.

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, the packet is flooded to all ports of the switch (lines 9–11), otherwise forwarding rules are installed to the switch. These rules include the one which redirects packet *pkt* to the destination host (line 13 and another which either create a channel with the opposite direction (line 15) or sets 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 processes packets from high sources (lines 21–31). If the destination for such a high packet is a low host, we drop the packet (line 22). If the

Algorithm 2 Secure learning switch algorithm

```
1: pkt ← packet arrived to the controller
2: port ← from which port pkt received
3: if find(src(pkt)) is null then
4:    push(src(pkt), port)
5: end if
6: fport ←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))×Output(i)
11:      end for
12:   else
13:      (src(pkt), dst(pkt))×Output(fport)
14:      if (dst(pkt):low) then
15:         emit (dst(pkt), src(pkt))×Output(port)
16:      else
17:         emit (dst(pkt), src(pkt))×Drop
18:      end if
19:   end if
20: else
21:   if dst(pkt):low then
22:      emit (src(pkt), dst(pkt))×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))×Output(i)
27:         end for
28:      else
29:         emit (src(pkt), dst(pkt))×Output(fport)
30:         emit (dst(pkt), src(pkt))×Output(port)
31:      end if
32:   end if
33: end if
```

controller does not find the port to forward the packet, 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 Algorithm 2 can be proved. If the condition in line 8 is true, the following holds for lines 8–19 by rule 1:

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

And by rule 4:

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

Assume that $fport$ is null, 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 rule 7:

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

Applying rule 12, we have

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

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

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

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

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

$$\frac{forall(src(match)) : low}{\vdash [low] \vdash (src(pkt), dst(pkt)) \times Output(port)} \quad \text{(rule 7)}.$$

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), only the following can be inferred:

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

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

$$\frac{\vdash forall(src(match)) : high}{[high] \vdash match \times Drop} \quad \text{(rule 5)}.$$

This means that the security type of the *Drop* action from line 17 does not agree with the security type of previous actions and the packet processing context which are *low*. Thus, the *Drop* action can not be considered safe. Indeed, low packets must not trigger packet drops originated from high security hosts. If we carefully examine the code, we

will see that such a drop is made in line 22 when the packet processing context is high. Hence, we can remove line 17 from our algorithm without harming the learning switch functionality.

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

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

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

In this case three possibilities can occur:

1. A $Drop$ action is emitted (line 22):

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

$$\frac{\vdash forall(src(match)) : high}{[high] \vdash match \times Drop} \quad \text{(rule 5)}.$$

2. The packet is flooded by using the 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 rule 6 we can obtain

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

3. Bidirectional forwarding is set (lines 29–30). Since both src(pkt):$high$ and dst(pkt):$high$ are fulfilled and it was determined that such packets came from ports $port$ and $fport$, respectively, we can conclude that $exists(port) : high$ and $exists(fport) : high$. Using the same as shown earlier we can obtain that both $Output$ actions are typed as $[high]$.

Thus, in all three cases the 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 found a case where a packet from a low security flow can trigger packet drops 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.

3. Conclusion

Security is challenging in networking and must be further investigated for software-defined networks. There is a lack of formal models for making security analysis [4] and the paper proposes the approach that is based on a formal security-type system. This system ensures that the controller application does not violate security properties such as confidentiality and, at some extent, integrity. We have extended the proposed system so that can verify new confidentiality properties in case of insecure network links. The security system can be implemented as a software module of the controller and check whether network applications violate security properties.

There are both theoretic and practical challenges when considering SDN security. It is interesting to explore soundness and completeness of the proposed type system. Another fascinating problem is to introduce other security-type systems that have been recently developed for programming languages using the proposed approach for achieving a solid theoretical basis for static security analysis that can prove properties of an SDN controller at the compilation stage.

References

[1] E. Al-Shaer, S. Al-Haj, "FlowChecker: configuration analysis and verification of federated OpenFlow infrastructures", SafeConfig 2010 : 2nd ACM Workshop on Assurable and Usable Security Configuration (October 4, 2010, Chicago, IL, USA), 37–44.

[2] C. J. Anderson et al., "NetKAT: semantic foundations for networks", POPL 2014: 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (January 22–24, 2014, San Diego, USA), 113–126.

[3] E. Ju. Antoshina et al., "A translator with a security static analysis feature of an information flow for a simple programming language.", *Autom. Control and Comp. Sciences*, **48**:7 (2014), 589–593.

[4] M. Casado, N. Foster, A. Guha, "Abstractions for software-defined networks", *Communications of the ACM*, **57**:10 (2014), 86–95.

[5] M. Casado et al., "Ethane: taking control of the enterprise", ACM SIGCOMM 2007: Data Communications Festival (Augest 27–31, 2007, Kyoto, Japan).

[6] N. Foster et al., "Frenetic: a network programming language", The 16th ACM SIGPLAN International Conference on Functional Programming (September 19–21, 2011, Tokyo, Japan), 279–291.

[7] S. Gutz et al., "Splendid isolation: a slice abstraction for software-defined networks", ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN) (August 13, 2012, Helsinki, Finland), 2012, 79–84.

[8] C.-Y. Hong et al., "Achieving high utilization with software-driven WAN", ACM SIGCOMM 2013 (August 12 – 16, 2013, Hong Kong, China).

[9] N. McKeown et al., "OpenFlow: enabling innovation in campus networks", *ACM Comp. Comm. Review*, **38**:2 (2008), 69 – 74.

[10] Open Networking Foundation, "OpenFlow switch specification v. 1.4.0", *URL: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf*, Last accessed: 10.05.2015.

[11] A. Sabelfeld, A.C. Myers, "Language-based information-flow security", *IEEE Journal on Selected Areas in Communications*, **21** (2003), 5–19.

[12] R. Smeliansky, "SDN for network security", Modern Networking Technologies: SDN & NFV – The Next Generation of Computational Infrastructure (October 28–29, 2014, Moscow, Russia), 155–159.

[13] D. Zhang et al., "Jif: Java+ information flow", *URL: http://www.cs.cornell.edu/jif/*, Last accessed: 10.05.2015.

[14] D. Zhang et al., "A Hardware Design Language for Timing-Sensitive Information-Flow Security", ASPLOS 2015 : Architectural Support for Programming Languages and Operating Systems (Mar 14 – 18, 2015, Istanbul, Turkey).

[15] D. Hedin et al., "JSFlow: Tracking Information Flow in JavaScript and its APIs", The 29th Symposium On Applied Computing (March 24 – 28, 2014, Gyeongju, Korea), 1663–1671.

[16] O. Arden et al., "Sharing Mobile Code Securely With Information Flow Control", *IEEE Symp. on Security and Privacy (SP)*, 2012, 191–205.

[17] A. Cheung et al., "Using Program Analysis to Improve Database Applications", *IEEE Data Eng. Bull.*, **37**:1 (2014), 48–59.

DOI: 10.18255/1818-1015-2015-6-735-749

Модель безопасности информационных потоков для программно-конфигурируемых сетей

Чалый Д. Ю.[1], Никитин Е. С., Антошина Е. Ю., Соколов В. А.[2]

получена 21 октября 2015

Программно-конфигурируемые сети (ПКС, SDN, Software-defined Networks) являются новой парадигмой организации сетей, которая используется во многих современных приложениях, таких как виртуализация сети, управление доступом на основе политик безопасности и многих других. Программное обеспечение ПКС обеспечивает гибкость и быстрый темп инноваций в сети, однако оно имеет сложную природу, в связи с чем возникает необходимость в средствах обеспечения его корректности и безопасности. Абстрактные модели для ПКС могут решить эти задачи. Данная работа направлена на разработку моделей безопасного взаимодействия в ПКС, акцентируя внимание на таких свойствах безопасности, как конфиденциальность и, частично, целостность. Это критические свойства безопасности многопользовательских сетей, поскольку программное обеспечение, управляющее сетью, должно гарантировать, что конфиденциальные данные одного пользователя не будут переданы другим (нежелательным) пользователям. Мы определили понятие сквозной безопасности в контексте ПКС и предложили семантическую модель, позволяющую сделать обоснованный вывод о соблюдении конфиденциальности, и мы можем проверить, что конфиденциальные информационные потоки не смешиваются с не конфиденциальными. Мы показываем, что модель может быть расширена до обоснования соблюдения конфиденциальности в сетях с безопасными и небезопасными каналами связи, которые могут возникнуть, например, в беспроводных средах.

Статья представляет собой расширенную версию доклада на VI Международном семинаре "Program Semantics, Specification and Verification: Theory and Applications", Казань, 2015.

Статья публикуется в авторской редакции.

Ключевые слова: ПКС, безопасность, формальные модели

Для цитирования: Чалый Д. Ю., Никитин Е. С., Антошина Е. Ю., Соколов В. А., "Модель безопасности информационных потоков для программно-конфигурируемых сетей", *Моделирование и анализ информационных систем*, **22**:6 (2015), 735–749.

Об авторах:

Чалый Дмитрий Юрьевич, orcid.org/0000-0003-0553-7387, канд. физ.-мат. наук., доцент,
Ярославский государственный университет им. П.Г. Демидова,
ул. Советская, 14, г. Ярославль, 150000 Россия, e-mail: dmitry.chaly@gmail.com

Никитин Евгений Сергеевич, orcid.org/0000-0002-2341-9950 , студент,
Ярославский государственный университет им. П.Г. Демидова,
ул. Советская, 14, г. Ярославль, 150000 Россия, e-mail: nik.zhenya@gmail.com

Антошина Екатерина Юрьевна, orcid.org/0000-0003-1081-1758, аспирант,
Ярославский государственный университет им. П.Г. Демидова,
ул. Советская, 14, г. Ярославль, 150000 Россия, e-mail: kantoshina@gmai.com

Соколов Валерий Анатольевич, orcid.org/0000-0003-1427-4937, доктор физ.-мат. наук, профессор,
Ярославский государственный университет им. П.Г. Демидова,
ул. Советская, 14, г. Ярославль, 150000 Россия, e-mail: valery-sokolov@yandex.ru

Благодарности:

[1]Исследование выполнено при финансовой поддержке РФФИ в рамках научного проекта 14-01-31539 мол-а.

[2]Исследование выполнено при финансовой поддержке Минобрнауки в рамках контракта ID RFMEFI57414X0036.