Hindawi
Mathematical Problems in Engineering
Volume 2017, Article ID 8740217, 11 pages
https://doi.org/10.1155/2017/8740217

Research Article

Quantitative Analysis of the Security of Software-Defined Network Controller Using Threat/Effort Model

Zehui Wu and Qiang Wei

Zhengzhou Information Science and Technology Institute, Zhengzhou 450001, China

Correspondence should be addressed to Zehui Wu; wuzehui2016@sina.com

Received 22 June 2016; Revised 20 November 2016; Accepted 24 January 2017; Published 1 March 2017

Academic Editor: Jean-Christophe Ponsart

Copyright © 2017 Zehui Wu and Qiang Wei. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

SDN-based controller, which is responsible for the configuration and management of the network, is the core of Software-Defined Networks. Current methods, which focus on the secure mechanism, use qualitative analysis to estimate the security of controllers, leading to inaccurate results frequently. In this paper, we employ a quantitative approach to overcome the above shortage. Under the analysis of the controller threat model we give the formal model results of the *API*s, the protocol interfaces, and the data items of controller and further provide our *Threat/Effort* quantitative calculation model. With the help of *Threat/Effort* model, we are able to compare not only the security of different versions of the same kind controller but also different kinds of controllers and provide a basis for controller selection and secure development. We evaluated our approach in four widely used SDN-based controllers which are *POX*, *OpenDaylight*, *Floodlight*, and *Ryu*. The test, which shows the similarity outcomes with the traditional qualitative analysis, demonstrates that with our approach we are able to get the specific security values of different controllers and presents more accurate results.

1. Introduction

SDN (Software-Defined Network) has three advantages which are separation of control and forward, virtualization, and dynamic programmability, making it top priority for solving current problems in the network [1, 2]. New wave of research is set off in both academia and industry. In 2012, Google forwarded data between data centers using *OpenFlow* which is an open-source protocol [3] and deployed in WAN the network controller [4] which is developed together by *Nicira* and itself. In 2014, *Facebook* launched "Wedge," a SDN-based network switch to connect servers in the data center, in order to create high-speed data center network [5]. In March 2014, *Huawei* gave product solution with combination of SDN and cloud computing [6] and later in May 2015 released the first agile instrumentation solution in the world [7]. *Gartner* [8], an international institute, even ranked SDN technology as one of the top 10 key technologies in IT field over the next five years.

Currently, open-source and commercial controllers are generally based on modular design and virtualized realization, leading to difference in both the controller's basic architecture and the programming language for realizing virtualized controller. For example, *NOX* is implemented in *C++* and *POX* and *Ryu* are implemented in *Python*, while *Floodlight* and *OpenDaylight* are implemented in *Java*. It leads to big difference for different types of controllers in performance and security [9], such that, in the case of *Floodlight*, the controller's northbound *REST API* is not yet equipped with authentication for protecting resource so that the attacker can use this vulnerability to initiate unauthorized access to the controller [10, 11]. Although SDN will be configured by the way by professionals and even they may have a basic knowledge about security issues, an automatic analysis will be still interesting for them. Besides, their study should at least be completed with a survey of the threats related to the technology and that the choice cannot be only based on such criterion independent of the technology. The attacker can cause the denial of service in whole network by tampering with the controller's flow table and steal victim's information by modifying the destination of items in flow table to specified target [12].

Therefore, it becomes an important premise of SDN-based network to compare existing controllers in security

measurement and select those controllers that meet the demand of users. Current security analysis of controllers is mainly qualitative, such as analyzing whether there is authentication and encryption for communication between controllers and switches [13, 14] and whether access to northbound *REST API* of controllers is under access control mechanism [15, 16], as well as whether application execution environment of controllers is equipped with sandbox isolation mechanism [17]. However, these methods can only take qualitative analysis of security mechanisms that the controller adopts, unable to compare the controller's security under different security mechanisms. In addition, controllers and these security mechanisms are software realized by programming, so these methods cannot determine whether there is any vulnerability existing in the code implementing these security mechanisms.

Therefore, comparison of security between different controllers can be carried out using traditional security comparison method for software. For example, vulnerabilities detection methods for software can be adopted, such as source code auditing tools *Coverity* [18] and dynamic vulnerability detection tools *BitBlaze* [19]. By detecting number and threat level of vulnerabilities of different controllers, security of controllers can be compared. But this method has the following two shortages: (1) the same vulnerability can cause different effect in the hands of different attackers, so the threat level of vulnerability cannot be accurately assessed; (2) there is no reasonable quantitative relation between the number and the threat level of vulnerabilities. For example, qualitative and quantitative comparison cannot be conducted between 100 vulnerabilities of medium threat level and single vulnerability of high threat level. Therefore, we cannot accurately evaluate the security of software through the above methods depending on vulnerability detection.

With regard to the above problem, Howard [20] in *Microsoft* proposed *RASQ* (Relative Attack Surface Quotient) to evaluate the system's security. Here are the details: first of all, it determines which processes and services in *Windows* are often used for attack based on experience, such as open sockets, dynamic web pages, and user account management; after that Michael defines 20 attack vectors and attaches each element in vectors with experience-based weight; finally, the sum of its elements' weight for each attack vector is calculated, while the sum of all vectors' weight is the quantitative value of system security.

Because *RASQ* relies too much on the experience when selecting the attack vectors and assigning values for those vectors, as well as its limitation in evaluating the security of operating systems, Manadhata and Wing [21] improved *RASQ* and proposed quantitative measurement for attack surface. Compared with *RASQ*, this method no longer depends on the experience when selecting the attack vectors and assigning values for them, making the measurement more objective and more common. However, these methods evaluate the security of applications according to security mechanisms of the operating system and do not take quantitative measurement for security of the application itself. Besides, these methods do not evaluate the types of attack, thus making them lacking practicability and guidance. For example, the controllers are required to have strong ability against *DoS* (Denial of Service) attack in certain network environment, and the methods above could not complete security measurement.

In order to solve the above problems, we propose a quantitative analysis method of controllers' vulnerabilities based on *Threat/Effort* model (*TE* model for short) in this paper. Compared with *RASQ* and improved *RASQ*, *TE* model does not depends on the security mechanisms of controller's underlying operating system. Instead it implements the quantitative analysis of controller's vulnerabilities through analyzing the controller's code. *TE* model sets up formal model for entry and exit of objects under test, defines precondition and postcondition of controller's resource, conducts quantitative calculation of *TE* value for each resource according to the conditions, and finally makes weighted calculation to get quantitative value of controller's vulnerability. In addition, we introduce the attack index in this paper, and calculation of attack index of controller during quantitative evaluation could help to analyze the ability of defending *DoS* attack for different controllers.

We introduce the notion of attack surface and the attack surface threat model for SDN controller in the first part of this paper, which is the basis of *TE* model. The second part details the process of building *TE* model and describes composition and principle of the model. The third part describes how to conduct quantitative measurement of vulnerabilities of SDN controller according to *TE* model and gives specific steps and calculations. The fourth part presents the experiment using current mainstream SDN controllers and then gives analysis of the results. The fifth part concludes the whole paper and discusses the advantages and disadvantages of our method.

2. Threat Model of SDN Controller

We describe possible threats facing the controller based on its architecture, after which we build threat model for the controller.

2.1. Architecture of SDN Controller. Figure 1 shows the hierarchical architecture of SDN controller, which is divided into two layers as basic function layer and network infrastructure service layer [22]. In order for flexibility of adding interface protocols to support dynamic deployment of SDN, the basic function layer should firstly be equipped with function of protocol adaptation. Current protocol adaptation is mainly implemented to support *OpenFlow* and *OF-CONFIG* (*OpenFlow* Configuration and Management Protocol), of which the former defines the south interface of SDN-based network architecture and regulates the way for the controller to distribute the flow table to the *OpenFlow* switch, while the latter provides an open interface for remote configuration and control of *OpenFlow* switch. Besides, after the work of protocol adaptation, the controller needs to provide support of development of upper-layer applications. There are mainly four parts as module management, event mechanism, job logging, and resource database.

The network infrastructure service layer provides basic network function for upper-layer business logic. As a part of

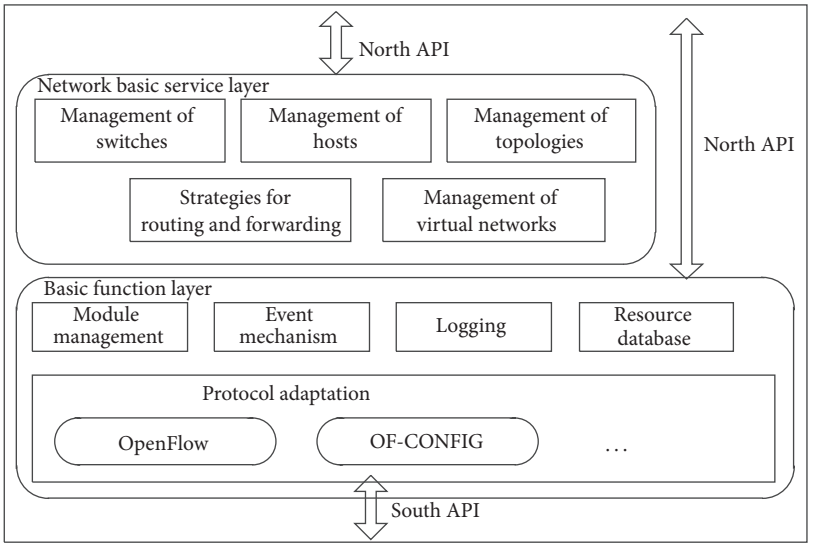


Figure 1: Layered hierarchical architecture of SDN controller.

practical controller, modules of this layer can realize a series of basic functions such as device management and status check by using interfaces from the basic function layer. This layer mainly contains five functional modules as switch management, host management, topology management, routing and forwarding strategy, and division of virtual machines. Besides, the controller provides programming interfaces of all layers for developers of upper-layer applications, thus creating a powerful universal platform for development and operation of upper-layer network applications.

Previous analysis shows that SDN controllers are faced with threats from the following three aspects.

(1) Protocol interface, including south protocols of *OpenFlow* and *OF-CONFIG*, which communicates with underlying controller through secure passage: if there is any vulnerability in the passage, the attacker can pretend to be the controller (or switch) to steal information such as the flow table from switches (or the controller). At the same time, north protocols are also included. However, standards for north protocols are not unified yet, so north protocols will not be studied in this paper.

(2) Internal *APIs*, which mainly refer to those *APIs* fulfilling specific functions and having calling relationship during the realization of internal modules of the controller: if a certain *API* has improper authority or a vulnerability during authentication, it can be exploited by the attacker to conduct malicious operations such as code embedding and information stealth.

(3) Data resources, including network topology, flow table, and configuration information stored in the database: if the attacker has access to such resources, he/she can temper network data or make it a part of the network detection for subsequent attacks.

2.2. Threat Model of Controller. With the above security analysis of the architecture of SDN controller, we know that security threats facing the controller comes from three aspects: internal *API*s, protocol interfaces, and data resources. Then we can describe the threat model of controller. The attacker exploits resource (protocol, *API*, and data resource) of the controller to get access to it. Therefore, the more resource the attacker gets, the more possible the attack can cause a big loss.

Figure 2 shows the threat model of controller. From this figure, we know that SDN controller is located in the heart of a SDN-based network, so the attacker can use *API*s, protocols, or data resources under control to attack the SDN controller. It should be pointed out that not all of the resources of the controller could be used by the attacker and not all impacts of controller resources are the same. So it is necessary to define which resources can be used by the attacker and how to measure the influence of different resources on the attack. If a kind of resource can be exploited by an attacker get access to the controller, it will be defined as useful, and this resource belongs to vulnerability set of the controller. In this paper, we use entry point and exit point to identify these resources. In addition, the influence of a kind resource on the attack is related to the damage this resource can cause to the controller. For example, exploiting *API*s writing topology structure can definitely cause more damage than exploiting *API*s reading the structure. Therefore, we use quantitative value of *TE* model as measurement for evaluating the influence of resources on the attack.

3. Formalizing Vulnerable Set of Controller

On the basis of discussion about threat model of SDN controller in last section, we elaborate formalized definition of *TE* model in this part, of which model of entry point and exit point is the base of selecting resources that are influential to the attack from all resources of the controller, and *TE* value, as well as attack index, is the base of quantitative measurement of controller's vulnerability.

3.1. Symbolized Expression of SDN Network. If C represents the set of SDN controllers, for a given controller c ($c \in C$),

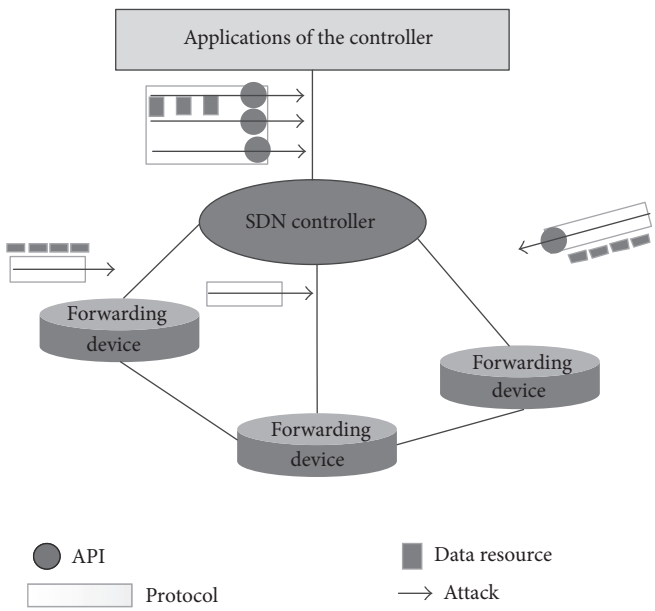


Figure 2: Illustration of vulnerability set of SDN controller.

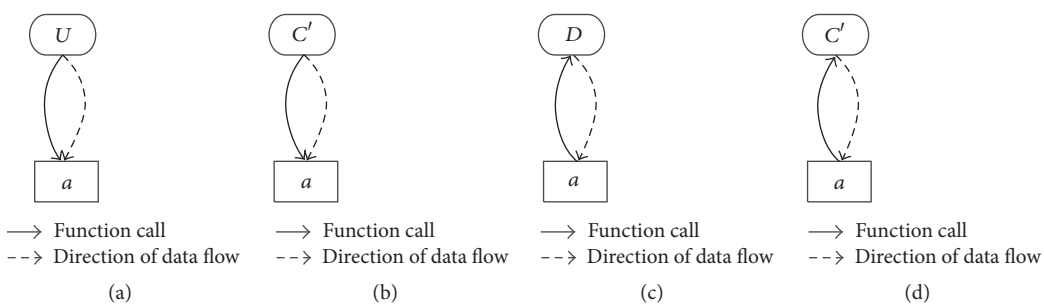


Figure 3: Relationship between direct entry point function and direction of data flow.

we define its network environment as N_c, $N_c = \langle U, D, T\rangle$, where U is the set of users, D is the set of data, and T is the subset of C except c; that is, $T = C/\{c\}$. It should be pointed out that, in a SDN network, elements in U may be hosts or switches, while elements of D may be information such as flow tables and topologies. T contains all controllers that connect to this controller. Then the attacker conducts the attack by controlling U.

3.2. Definition of Entry Point and Exit Point. We define *API*s responsible for receiving (directly or indirectly) network inputs in source code of the controller as the entry point for the system. For the *API* function a, there are two types of entry point as direct entry point and indirect entry point.

3.2.1. Direct Entry Point. For controller c, there are three cases that we call c which directly receives data from the network. (1) User U (in Figure 3(a)) or controller c (in Figure 3(b)) calls function a and inputs data item d ($d \in D$) into a. (2) Function a directly reads data from data set D (in Figure 3(c)). (3) a calls function of controller c' and gets data item d (in Figure 3(d)) from return value of the function called.

Definition 1 (direct entry point). We define the input function of the controller as a. When a meets at least one of the following three conditions, we call a as direct entry point of c. (1) User U contains output function a (in Figure 3(a)). (2) Controller c' ($c' \in T$) contains output function a (in Figures 3(b) and 3(d)). (3) Data set D contains output function a (in Figure 3(c)).

3.2.2. Indirect Entry Point. For controller c, there are two cases when we call c which indirectly receives data from network. (1) Controller c has function a_1 which *directly* receives data item d from SDN network and passes it to a as input of function a (in Figure 4(a)) or a receives return value d of a_1 as input (in Figure 4(b)). (2) Controller c has function a_2, which *indirectly* receives data item d from SDN network, and passes it to a as input of a (in Figure 4(c)) or a receives return value of a_2 as input (in Figure 4(d)). For example, a function responsible for authentication in the controller receives users' logging information but passes that information to other functions of authentication module for processing; then this function is an indirect entry point of the system.

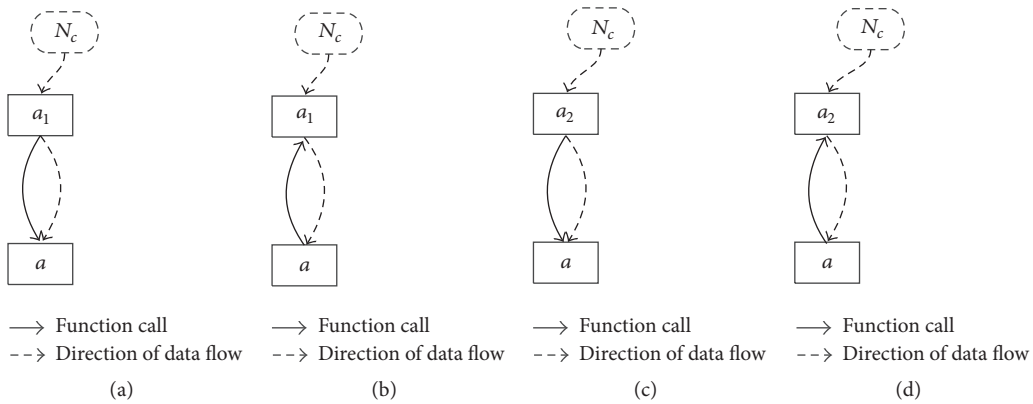


Figure 4: Relationship between indirect entry point function and direction of data flow.

For controller c and input function a, if a directly receives input data item d from network, and successive functions of a depend on the value d, then d is postcondition of function a, which is expressed as $d \in Res(a.post)$, of which symbol $Res(p)$ represents the resource set of function p.

Definition 2 (indirect entry point). We call internal function a as indirect entry point of controller c when a satisfies at least one of the following conditions. (1) There is a direct entry point a_1 with $a_1.post \Rightarrow a.pre$, and there is a data item d satisfying $d \in Res(a_1.post) \wedge d \in Res(a.pre)$ (in Figures 4(a) and 4(b)). (2) There is an indirect entry point a_2 with $a_2.post \Rightarrow a.pre$, and there is a data item d satisfying $d \in Res(a_2.post) \wedge d \in Res(a.pre)$ (in Figures 4(c) and 4(d)). The set of entry points of controller c is the union of direct entry points and indirect entry points.

Direct exit point and indirect entry point can be defined in the similar way as shown below.

Definition 3 (direct exit point). We call output function a as direct exit point of controller c when a satisfies at least one of following three conditions. (1) User U calls output function a. (2) Controller c' ($c' \in T$) calls output function a. (3) Data set D calls output function a.

Definition 4 (indirect exit point). We call intermediate function a indirect exit point of controller c when a satisfies at least one of following two conditions. (1) There is a direct exit point a_1 with $a.post \Rightarrow a_1.pre$, and there is a data item d satisfying $d \in Res(a.post) \wedge Res(a_1.pre)$. (2) There is an indirect entry point a_2 with $a.post \Rightarrow a_2.pre$, and there is a data item d satisfying $d \in Res(a.post) \wedge Res(a_2.pre)$. The set of exit points of controller c is the union of direct entry points and indirect entry points.

We mainly formally model the *API* functions of the controller in the above. And protocols and data resources can be defined based on preconditions and postconditions of the function.

Definition 5 (protocol interface). The protocol interface of controller c is p, which is precondition of direct entry point a (or direct exit point) of the controller, that is, $c \in Res(a.pre)$.

Definition 6 (data resource). We call data item d ($d \in D$) the data item of the controller when it satisfies at least one of two conditions. (1) There is a direct entry point a with $d \in Res(a.post)$. (2) There is a direct exit point a with $d \in Res(a.pre)$.

3.3. Defining TE Value of Vulnerable Set. In common case, the attacker uses a subset of resource set of the controller to complete the attack against the controller. For example, the attacker can use entry point, exit point, and protocol as well as data resource to send data to the controller or to steal data from the controller. Therefore, the vulnerability set of the controller is comprised of entry point set, exit point set, protocol set, and data resource set.

Definition 7. For given controller c and network environment N_c, vulnerability set of c can be expressed using the following tuple $\langle A^{N_c}, P^{N_c}, D^{N_c} \rangle$, of which A^{N_c} is the set of entry points and exit points, P^{N_c} is the set of protocol interfaces, and D^{N_c} is the set of data resource.

Therefore, we can compare this tuples of two controllers which are two versions of the same controller type or having similar functions, so as to get the difference in security.

To achieve quantitative measurement of the vulnerability set, we introduce the cost-effectiveness model (i.e., *TE* model) in economics to evaluate the vulnerability set of controller's resource. Cost-effectiveness ratio involves two aspects: one is the level of damage (namely, threat, short as T value, which is expressed as $r.th$) which the attacker can cause for the controller using that resource, and the other is the effort (namely, effort, short as E value, which is expressed as $r.ef$) the attacker has to pay for getting that resource. The higher the former and the lower the latter, the more the threat this resource will bring to the controller. We use total-order symbol $\succ$ to indicate the influence of resource in the

controller. For two resources r_1 and r_2, if the influence of the former is larger that of the latter, their relationship of cost-effectiveness can be expressed as $r_1 \succ r_2$.

Definition 8 (definition of *TE* value for the resource). For controller c and its two resources r_1 and r_2, we have $r_1 \succeq r_2$ when one of the three conditions is met:

(1) $r_1.th \geq r_2.th \wedge r_2.ef \geq r_1.ef$;

(2) $r_1.th = r_2.th \wedge r_2.ef \geq r_1.ef$;

(3) $r_1.th \geq r_2.th \wedge r_2.ef = r_1.ef$.

We have the following definition if we combine Definition 8 and the tuple $\langle A^{N_c}, P^{N_c}, D^{N_c}\rangle$ of the controller's vulnerability set.

Definition 9 (definition of *TE* value for the vulnerability set). For given controller c, there are two functions a_1 and a_2, two protocol interfaces p_1 and p_2, two data items d_1 and d_2, and an attribute $at \in \{th, ef\}$. We have $\langle a_1, p_1, d_1\rangle \succ \langle a_2, p_2, d_2\rangle$ when one of the following three conditions is met:

(1) $a_1.at \succ a_2.at \wedge p_1.at \succeq p_2.at \wedge d_1.at \succeq d_2.at$;

(2) $a_1.at \succeq a_2.at \wedge p_1.at \succ p_2.at \wedge d_1.at \succeq d_2.at$;

(3) $a_1.at \succeq a_2.at \wedge p_1.at \succeq p_2.at \wedge d_1.at \succ d_2.at$.

4. Quantitative Measurement of *TE* Model

In the last section, we have given formalized description of vulnerability set of the controller, which, however, could only help conduct qualitative description of two given controllers. For example, the vulnerability set of one controller is larger than the other. Thus we could not have quantitative measurement of the vulnerability set. In this section, we introduce calculation method of cost-effectiveness ratio based on *TE* model, with which we can achieve quantitative measurement of the vulnerability set.

4.1. TE Value and Attack Index. If the attacker obtained access to functions with high authority, he/she could bring more threat to the controller. Therefore, it is worthy of paying more to obtain resource with high authority from the attacker's point of view. Thus, use of cost-effectiveness ratio can describe the influence that the resource has on the vulnerability set. *TE* value of the controller is expressed as $\eta = r.th/r.ef$, of which η is called the cost-effectiveness ratio of the resource.

We define the mapping function $der_a : api \rightarrow \eta$ to complete mapping from the function to the cost-effectiveness ratio. Similarly, we can define $der_p : protocol \rightarrow \eta$ to complete mapping from the protocol to the cost-effectiveness ratio and define $der_d : data \rightarrow \eta$ to complete mapping from the data item to the cost-effectiveness ratio. When we conduct practical calculation of the cost-effectiveness ratio of the resource, it is necessary to assign values which are usually concrete to attributes. For example, calculation of the cost-effectiveness ratio of a certain function, we can assign a certain numerical value to that function with its priority and access authority, and the value assigned to functions with high authority should be larger than that assigned to functions with low authority. Actual assignment needs to be done according to object, environment, experience, and mastered knowledge, so the assignment is subjective to some extent. In this paper, we can fulfill the assignment by analysis of parameters' sensitivity in [20].

In the vulnerability set of the controller, T value of function a is its postcondition, because T value indicates the number of other functions that function a can call, that is, number of functions after a in the sequence of function calling. Therefore, T value can describe the contribution of function a to the vulnerability set, and the larger T value is, the more functions a can call and the larger the number of this kind of sequence is. Similarly, E value of function a shows the number of functions that can call a, that is, functions before a in the sequence of function calling that can call a. The more this kind of sequences, the smaller E value of function a. Therefore, the cost-effectiveness ratio of a can be determined by number of sequences that a resides. For functions a_1 and a_2, $der_a(a_1) > der_a(a_2)$ means that the number of sequences that a_1 occurs is larger than that of a_2. Similarly, if protocol interface p occurs in preconditions of function $a_{1,2,\ldots,n}$, its T value can be expressed as n. If $a_{1,2,\ldots,n}$ occurs in preconditions of p, the E value of p can also be expressed as n. Calculation of cost-effectiveness ratio for data item d can be done using similar method.

Since different types of attack require different vulnerability sets, the above definition for cost-effectiveness ratio leads to definition of attack index.

Definition 10 (definition of attack index). For given controller c with its attack surface as $\langle A^{N_c}, P^{N_c}, D^{N_c}\rangle$ and attack behavior k against it, the attack index of k against the controller is $k_c = \sum_x der_{t,t\in\{a,p,d\}}$, of which x indicates vulnerability sets that are used in calculating attack index for different types of attacks. For example, the attack index of *DoS* is $k_c(DoS) = \sum_{P^{N_c}} der_p + \sum_{D^{N_c}} der_d$.

It should be pointed out that the DoS attack discussed in this paper refers to the DoS attack by exploiting the target system vulnerabilities. Traditional DoS attacks based on traffic and DDoS attacks are beyond the scope of this paper.

4.2. Steps for Quantitative Measurement of TE Model. In this paper, we conduct quantitative measurement of from three dimensions: *API*, protocol, and data resource.

Definition 11. For given controller c with vulnerability set as $\langle A^{N_c}, P^{N_c}, D^{N_c}\rangle$, the quantitative value of its vulnerability set is $\langle \sum_{a\in A^{N_c}} der_a(a), \sum_{p\in P^{N_c}} der_p(p), \sum_{d\in D^{N_c}} der_d(d)\rangle$. The steps for calculation are as follows.

(1) For given controller c and its execution environment N_c, we first construct A^{N_c} which is the set of entry points and exit points of c and then build P^{N_c} which is the set of protocol interfaces, as well as D^{N_c} which is the set of data items.

(2) Calculate cost-effectiveness ratio. For each function $a \in A^{N_c}$, calculate $der_a(a)$; for each passage $p \in P^{N_c}$, calculate

$der_p(p)$; for each incredible data item $d \in D^{N_c}$, calculate $der_d(d)$.

(3) Calculate $\langle \sum_{a\in A^{N_c}} der_a(a), \sum_{p\in P^{N_c}} der_p(p), \sum_{d\in D^{N_c}} der_d(d)\rangle$, which is the quantitative value of the vulnerability set.

The method used in this paper is similar to risk assessment method in [9]. In the risk assessment method, $\sum_{e\in E} p(e)C(e)$ is calculated for event set E. In this expression, $C(e)$ is the execution sequence of event e, and $p(e)$ is the probability for executing e.

Events in the risk assessment model of events are similar to resources used in evaluating vulnerability set of controller in this paper. Take function m which is used in buffer overflow attacks as an example. The attack can succeed only if there is any buffer overflow vulnerability in m, so $p(m)$ indicates the probability of vulnerability existing in m. In case of passage c, $p(c)$ means the probability of vulnerability existing in sending (or receiving) data module in passage c. Accordingly, quantitative value for evaluating the attack surface should be expressed as follows:

$$\left\langle \sum_{a\in A^{N_c}} p(a)\, der_a(a), \sum_{p\in P^{N_c}} p(p)\, der_p(p), \sum_{d\in D^{N_c}} p(d)\, der_d(d) \right\rangle. \tag{1}$$

However, it is difficult to predict whether there is any vulnerability existing in certain function. Moreover, function that is not vulnerability at current time is possible to become vulnerable along with change of execution environment. Therefore, we set probabilities $p(a)$, $p(p)$, and $p(d)$ as 1 in this paper.

The specific calculation process is shown in Figure 5. First we get calling graph by analyzing source code of the controller, thus further obtaining entry point and exit point. At the same time, we compile the source code to execute the controller and monitor its execution, thus obtaining set of protocol interfaces and data items. Finally, measurement value of the vulnerability set is calculated.

5. Test and Evaluation

In this part, we test and evaluate the method proposed in this paper through feasibility test and comparison test. In the feasibility test, commercial SDN controller *Ryu* is used. The reason why we choose *Ryu* is that, on the one hand, it is more comprehensive in fields like expansibility, security, and stability when compared with other commercial controllers such as *ONOS* and *OpenDaylight*. On the other hand, *Ryu* is developed in Python, whose architecture and code composition facilitate the quantitative measurement in the method, thus reducing test complexity. In the comparison test, we firstly compare different versions of *Ryu* to analyze security difference between them. The following comparison is done between current mainstream controllers *Ryu*, *OpenDaylight*, *ONOS*, and *Floodlight*, as well as *POX* of early stage, and security difference between those controllers.

5.1. Feasibility Test. Figure 5 shows the process for quantitative measurement of vulnerability set of SDN controller. The feasibility test is carried out following this procedure.

5.1.1. Preprocessing of Source Code. The source code structure for controller *Ryu* contains modules like *app*, *base*, *cmd*, *contrib*, *controller*, *lib*, *ofproto*, *services*, *tests*, and *topology*. *app* module is responsible for the north interface of the controller, and *controller* module is responsible for the south interface, while *ofproto* module and *topology* module are responsible for analyzing *OpenFlow* protocol and managing entire network topology. When calculating *TE* value with the method proposed in this paper, we will use *Oftctl_rest*, *app.Rest_conf*, *app.Rest_router*, and *app.Rest_topology* components in *app* module, as well as *controller.dpset*, *controller.Ofp_handler*, and *controller.controller* components in *controller* module. In the stage for preprocessing source code, we need to extract the required components from the controller, as well as corresponding source code for components.

5.1.2. Generation of Calling Graph. For source code after preprocessing, we write a script *FTracer.py* for tracing functions so as to enumerate functions in source code and to generate function calling graph. Figure 6(left) shows the function calling graph of ryu-manager component, from which we can infer that there are three types of calling graphs:

(a) Sequence of function calls, which describes the calling relationship between functions, as is shown with solid lines in Figure 6.

(b) Sequential execution sequence of functions, which shows the sequential execution relationship between other functions called inside a certain function, as is shown with dotted lines in Figure 6.

(c) Sequence of loop. Functions might be called multiple times in different sequences, which would lead to loops in the graph. In this paper, we use the method of node copy to solve the problem of loop, as is shown with shadow nodes in Figure 6.

Take function *send_msg()* in Figure 6(right) as an example. This function occurs in 8 sequences, of which the function occurs in 8 sequences (A-B-C; A-B-D; A-B-E; A-B-G; A-B-H; A-B-I; A-B-J; A-B-K;), of which the number of sequences where it occurs as precondition is 7 (B-C; B-D; B-G; B-H; B-I; J-L; K-M), while the number of sequences where it occurs as postcondition is 3 (A-B; N-J; N-K). Following the above method, we generate calling graphs for all related 91 functions in *app.oftctl_rest*, *app.rest_conf*, *app.rest_router*, and *app.rest_topology* components of *app* module, as well as in *controller.dpset*, *controller.ofp_handler*, and *controller.controller* components of *controller* module.

5.1.3. Process Monitoring. Process monitoring is to get P^{N_c} and D^{N_c} of vulnerability set during dynamic execution of the controller. In our work, we instrument function *Controller.OpenFlowController::server_loop()* in *Ryu*'s source

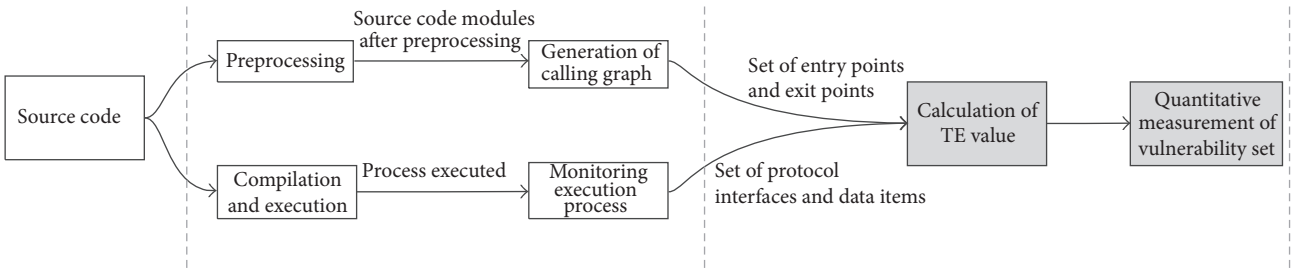


Figure 5: Flow graph of quantitative measurement for vulnerability set.

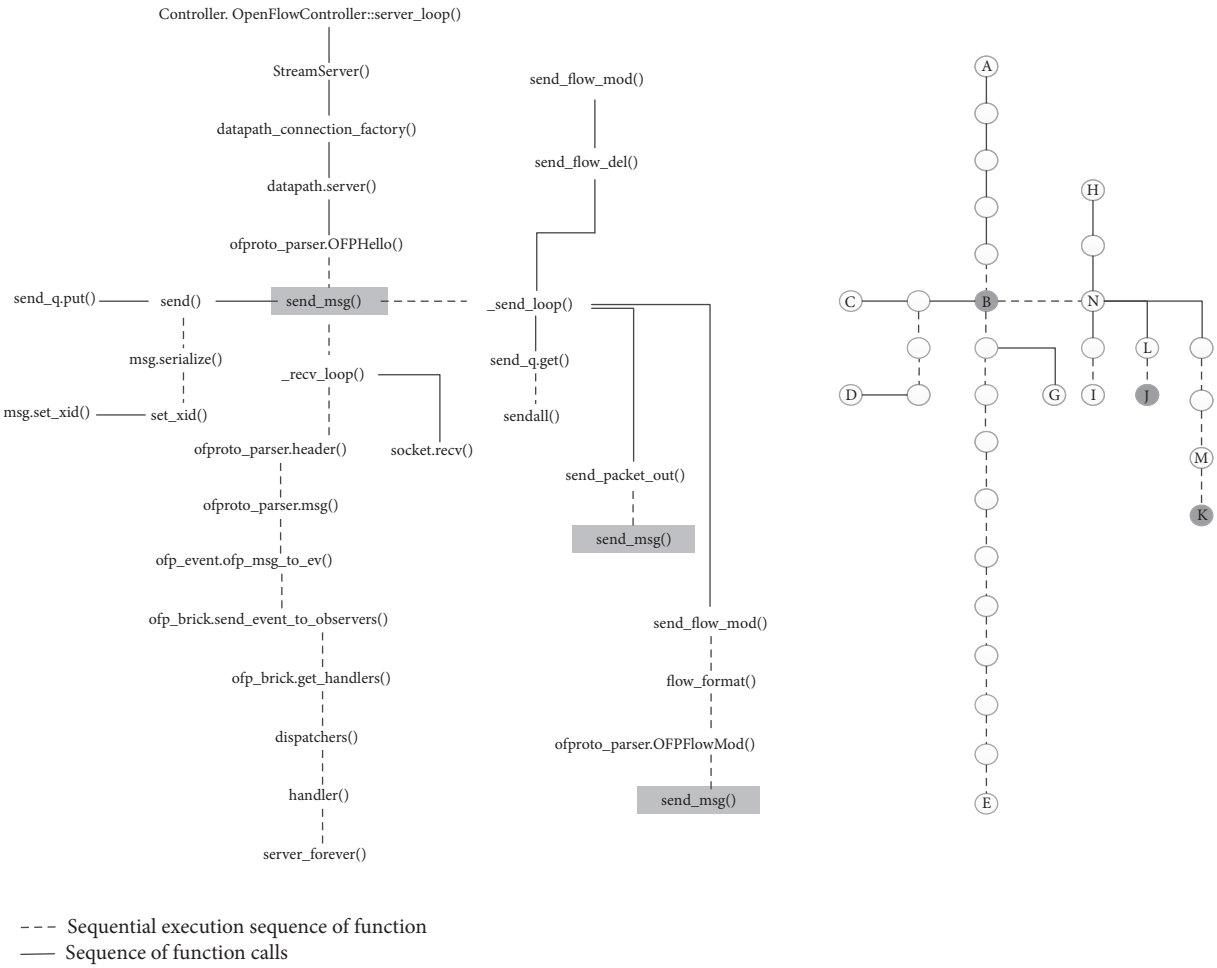


Figure 6: Function calling graph of ryu-manager component.

code, which is designed to record protocol interfaces and data items used for the controller to execute upper-layer applications and communicate with *OpenFlow* switches. In order to simulate execution of upper-layer applications and communication with *OpenFlow* switches, we use *mininet* which is simulation environment of SDN-based network to create a SDN-based network and use *Ryu VM* migration program to simulate the upper-layer application. Through process monitoring, we get information from communication port 6633 of *OpenFlow* v1.3, and handshake content between the controller and the switch from south interface of *Ryu*, and obtain data resources like event, network, and *flow table* object which are accessed by *Ryu* using *REST API* from south interface. During execution, *Ryu* will use objects to dynamically store network topology and event information, so data resources obtained during process monitoring in this article are expressed in form of the object of a class.

For the protocol port and data resource obtained from process monitoring, we need to use function calling graph to check for data in and out this port, which functions in

Table 1: Calculation results of TE values of entry points and exit points of the controller.

Name of components	Number of functions	DEP	DExP	IEP	IExP	*TE* value
app.ofctl_rest	3	2	0	1	0	3.15
app.rest_conf	4	1	0	3	0	3.07
app.rest_rou*TE*r	6	4	0	2	0	2.91
app.rest_topology	4	1	0	3	0	3.11
controller.dpset	19	7	7	12	9	1.77
controller.ofp_handler	31	23	13	8	6	1.69
controller.controller	24	9	11	15	9	1.76

Note. DEP: direct entry point; DExP: direct exit point; IEP: indirect entry point; IExP: indirect exit point.

Table 2: Results of *TE* value calculation for protocol interfaces of the controller.

Type of protocol parameters	Number of presequences	Number of postsequences	*TE* value
flow_mod	6	19	3.17
features_request	2	3	1.50
features_reply	1	1	1.00
tcp_msg	17	11	0.65
hello_msg	2	2	1.00

Table 3: Results of *TE* value calculation for data resource.

Type of protocol parameters	Number of presequences	Number of postsequences	*TE* value
Event	21	39	1.86
Network	19	12	0.63
Flow table	8	11	1.36

the function calling graph that data resource object is their preconditions or postcondition. The values of fragile set are determined by the number of front and rear sequences.

5.1.4. Calculation of TE Value

(i) Entry Points and Exit Points. The definitions of the level of damage and level of effort in Section 3 are abstract descriptions, which could be represented and calculated in several ways. And in our work, we use the number of presequences and the number of postsequences to represent the level of damage and the level of effort. Thus, in Tables 1, 2, and 3, we use the ratio of the number of presequences and the number of postsequences to indicate the value of *TE*. The above method can generate calling graphs of all components after preprocessing, with which we can obtain the number of presequences and postsequences of all functions. It should be pointed out that since component ryu-manager is responsible for launch, registration, and close of applications and other components of the controller, we only identify the indirect entry points and indirect exit points of ryu-manager, thus ensuring the code coverage as well as reducing the complexity of test. For all functions of entry/exit points, we first get the number of presequences and postsequences of each function in the calling graph according to the *TE* model, and then we can get the *TE* value according to the equation $\eta = r.th/r.ef$ (i.e., number of presequences divides number of postsequences). The sum of *TE* values of all functions in the module is the *TE* value of that module. Number of functions and number of entry/exit points generated for the above components, along with TE values calculated, are shown in Table 1.

(ii) Protocol Interfaces. Through process monitoring, we can get information of communication port 6633 of *OpenFlow* v1.3 from south interface of *Ryu*. We analyze position where data into and out of port 6633 resides in the calling graph, obtain the number of direct presequences and direct postsequences, and then get *TE* value based on *TE* model. It is important to note that the direct presequences and direct postsequences of protocol interface data and successive data resource are functions which use the data as preconditions or postconditions. It describes the number of functions of which the data occurs as direct preconditions and postconditions. Table 2 shows the result of *TE* value calculation for protocol interfaces.

(iii) Data Resources. During execution, the controller needs to finish works like calculating routing path and dynamic update of flow table according to global topology view, so we can get data resource of event, network, and flow table objects which are used by *Ryu* with *REST API*, and we get them from north interface through process monitoring. During execution, *Ryu* uses objects to dynamically store information like network topologies and events, so data resource obtained during process monitoring in this article is stored as objects of classes. Table 3 shows the result for calculating *TE* values of data resources.

5.1.5. Quantitative Calculation of Vulnerability Set. With the above results of calculating *TE* values of entry points and exit points, protocol interfaces, and data resources, we can get the final *TE* value of the vulnerability set of SDN controller as 28.63 according to the formula $\langle \sum_{a\in A^{N_c}} der_a(a), \sum_{p\in P^{N_c}} der_p(p), \sum_{d\in D^{N_c}} der_d(d)\rangle$.

5.2. Comparison Test. According to steps taken for feasibility test, we conduct the comparison test for current mainstream, open-source controllers such as *POX* (v1.0), *Ryu* (v1.0, v2.0, v3.0, v4.0), *OpenDaylight (ODL Lithium-SR4)*,

Table 4: Collection of comparison results of *TE* values for controllers.

TE value	POX	ODL	Floodlight	Ryu 1.0	Ryu 2.0	Ryu 3.0	Ryu 4.0
Entry point	15.29	12.27	14.09	16.08	8.54	8.01	8.14
Exit point	22.81	11.95	16.88	14.63	9.71	9.45	9.37
Protocol interface	9.66	17.64	14.34	17.15	6.95	7.32	7.51
Data resource	18.38	9.61	10.57	8.99	4.19	3.85	4.06
Vulnerability set	66.14	51.47	55.88	56.85	29.39	28.63	29.08

and *Floodlight* (v1.2). Since we cannot get source code for nonopen source controllers like *ONOS*, *ROSEMARY*, and *SE-Floodlight*, which means that we cannot obtain their *TE* values of vulnerability sets, we leave them not participating in quantitative comparison.

Table 4 shows collection of comparison result of *TE* values of controllers, of which the entry points contain direct entry points and indirect entry points. Rows of the table show values of each element of vulnerability sets of different controllers, while columns show values of each element of vulnerability sets of single controller. From Table 4, we know that *TE* values of vulnerability sets of *Ryu* are all relatively small, of which the value of *Ryu3.0* is the smallest, which also has lowest security threat among all 7 controllers under comparison. On the contrary, *POX* is the controller with largest *TE* value of vulnerability set. There are two mainly two reasons. On the one hand, *POX* is a version of early prototype *NOX* of *OpenFlow* protocol implemented in *Python*, and early version of *OpenFlow* protocol does not consider any security problems. On the other hand, *POX* controller is implemented without any authentication, encryption, or *API* role authorization. It is important to note that *ODL* and *Floodlight* which are commercial controllers achieving relatively wide adoption have relatively high *TE* values of vulnerability sets. Through analysis of their source code structure, we know that *ODL* is implemented based on *OSGi* framework which does not integrate any security mechanism, and *Floodlight* is implemented in *Java*. Although *Java* has built-in security features, there are problems of excessive authorization in the derived classes of framework classes, which leads to high *TE* value of vulnerability set. Subsequent implementation introduces formal security audit which can fix this problem.

According to Definition 10, calculation formula of attack index is $k_c = \sum_x der_{t,t\in\{a,p,d\}}$. Calculation of *DoS* attack index of the above controllers using $k_c(DoS) = \sum_{P^{N_c}} der_p + \sum_{D^{N_c}} der_d$ will get comparison results as is shown in Figure 7.

Figure 7 shows comparison result of *DoS* attack indexes for different controllers. It can be seen from this figure that *DoS* attack index of *Ryu* is the minimum, which is close to the results from Khondoker et al. [23] and Sandra, indicating that *Ryu* is a controller with high comprehensive security among numerous open-source SDN controllers. Besides, module *simple_switch_snort.py* in source code of *Ryu* introduces the function of intrusion detection from *Snort* into SDN-based network and forwards packets received from network to the *Snort* network for intrusion detection. When packets against rules of *Snort* occur, it will alarm. But quantitative analysis of this module cannot be conducted in the *TE* model of this paper. And the follow-up study will expand the *TE* model to include third-party security mechanism of the controller into quantitative analysis.

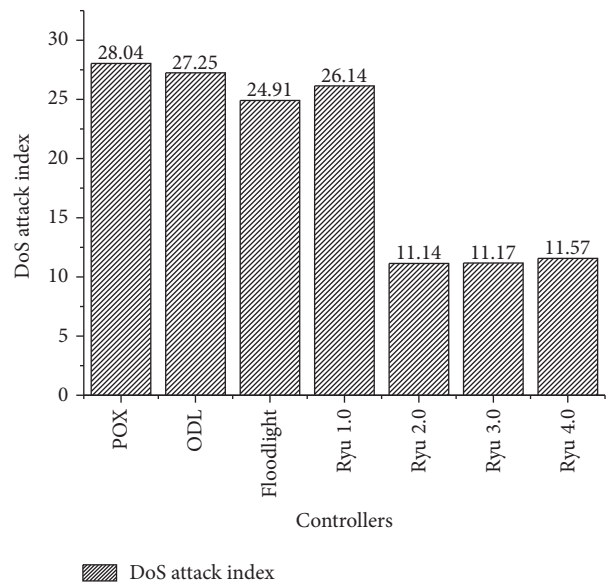


Figure 7: Comparison of *DoS* attack indexes against different controllers.

6. Conclusion

In this paper, we propose a kind of quantitative measurement of security for SDN controller. Compared with current qualitative evaluation methods, this method can get specific quantitative values, and its evaluation standard is more detailed and reliable. At the same time, the introduction of attack index can help conduct security assessment for different types of attacks. This method can be used as basis for the user to select secure SDN controller, as well as guidance for developers to improve security of the controller. But there are still some shortages: (1) east/west interfaces are not included in the model. Since the standards for east/west interfaces have not been unified yet, these interfaces are not modeled in this paper; (2) source code has not achieved full coverage testing yet. Test in this paper is mainly conducted on main process and main part of the source code, so test coverage needs further improvement. With improvement of the north interface of SDN controller, the successive work in

this paper can be making further improvement on the model and test coverage.

Competing Interests

The authors declare that they have no competing interests.

References

[1] Q.-Y. Zuo, M. Chen, G.-S. Zhao, C.-Y. Xing, G.-M. Zhang, and P.-C. Jiang, "Research on OpenFlow-based SDN technologies," *Journal of Software*, vol. 24, no. 5, pp. 1078–1097, 2013.

[2] I. Alsmadi and D. Xu, "Security of software defined networks: a survey," *Computers and Security*, vol. 53, pp. 79–106, 2015.

[3] Icebeen, "Google's using SDN to realize data centers interconnection[EB/OL]," 2014, http://www.educity.cn/net/1417699.html.

[4] Nicira: network virtualization—next revolution of the Internet, 2012–2016, http://www.199it.com/archives/33042.html.

[5] Z. Zou, Facebook launches its 'Wedge' switches for open data centers, June 2014, http://network.chinabyte.com/225/12992725.shtml.

[6] Z. Wu, HuaWei publishes 2014 serial SDN solutions, March 2014, http://www.wuzhongzx.com/zxzx/show.php?itemid=718892.

[7] Agile Network of HuaWei, HNC2015—HuaWei publishes the first SDN-based agile IOT solution in the world, May 2015, http://www.wtoutiao.com/p/e78rQ1.html.

[8] J. Skorupa, M. Fabbi, and A. K. Sharma, *Ending the Confusion About Software-Defined Networking: A Taxonomy*, Gartner—G00248592, 2013, http://www.gartner.com/id=2367616.

[9] Y. Jarraya, T. Madi, and M. Debbabi, "A survey and a layered taxonomy of software-defined networking," *IEEE Communications Surveys and Tutorials*, vol. 16, no. 4, pp. 1955–1980, 2014.

[10] M. Dover, A switch table vulnerability in the Open Floodlight SDN controller, March 2014, http://dovernetworks.com/wp-content/uploads/2014/03/OpenFloodlight-03052014.pdf.

[11] M. Dover, A denial of service attack against the Open Floodlight SDN controller, December 2013, http://dovernetworks.com/wp-content/uploads/2013/12/OpenFloodlight-12302013.pdf.

[12] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: new attacks and countermeasures," in *Proceedings of the Network and Distributed System Security Symposium*, Internet Society, San Diego, Calif, USA, February 2015.

[13] Open Networking Foundation, "*OpenFlow* Switch Specification," version 1.4.0 (Wire Protocol 0x05), October 2013.

[14] "CacheBleed: A Timing Attack on OpenSSL Constant Time RSA[EB/OL]," 2016, https://ssrg.nicta.com.au/projects/TS/cachebleed/cachebleed.pdf.

[15] P. Porras, S. Cheung, M. Fong et al., "Securing the software-defined network control layer," in *Proceedings of the International Conference on Network and Distributed System Security Symposium*, San Diego, Calif, USA, February 2015.

[16] S. Scott-Hayward and C. Kane, "OperationCheckpoint: SDN application control," in *Proceedings of International Conference on Network Protocols*, pp. 618–623, Research Triangle Park, NC, USA, October 2014.

[17] S. Shin, Y. Song, T. Lee et al., "Rosemary: a robust, secure, and high-performance network operating system," in *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS '14)*, pp. 78–89, Scottsdale, Ariz, USA, November 2014.

[18] Coverity, 2008–2016, http://www.coverity.com/.

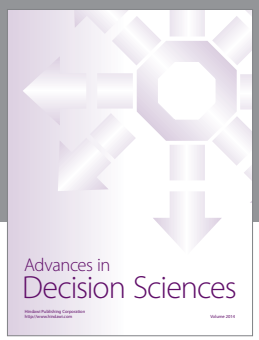
[19] D. Song, D. Brumley, H. Yin et al., "BitBlaze: a new approach to computer security via binary analysis," in *Information Systems Security*, vol. 5352 of *Lecture Notes in Computer Science*, pp. 1–25, Springer, Berlin, Germany, 2008.

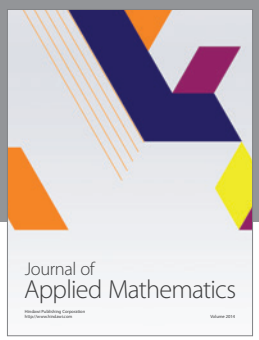
[20] M. Howard, "Fending off future attacks by reducing attack surface," 2003, https://msdn.microsoft.com/en-us/library/ms972812.aspx.

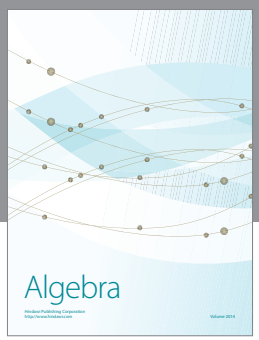
[21] P. K. Manadhata and J. M. Wing, "A formal model for a Sys*TE*m's attack surface," in *Moving Target Defense*, pp. 1–28, Springer, New York, NY, USA, 2011.

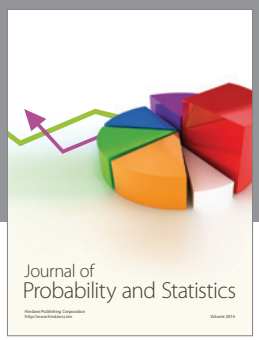
[22] M. M. Wang, J. W. Liu, J. Chen, J. Mao, and K. F. Mao, "Software defined networking: security model, threats and mechanism," *Journal of Software*, vol. 27, no. 4, pp. 969–992, 2016 (Chinese).

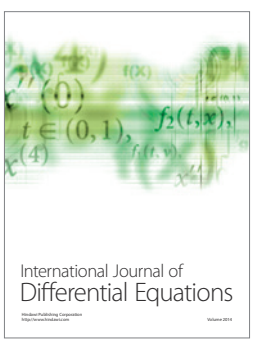
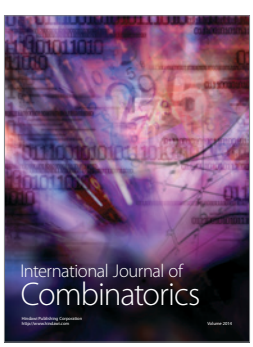
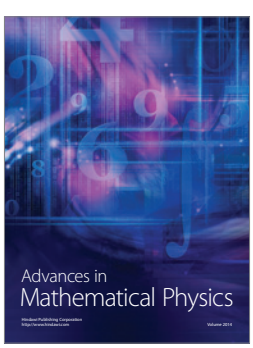
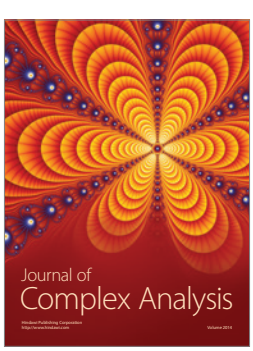
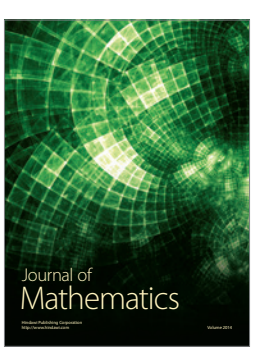
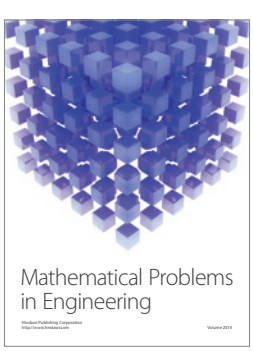
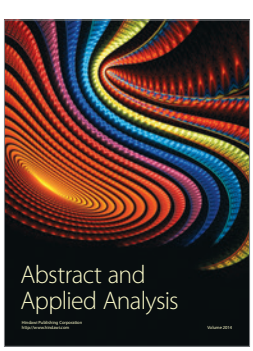
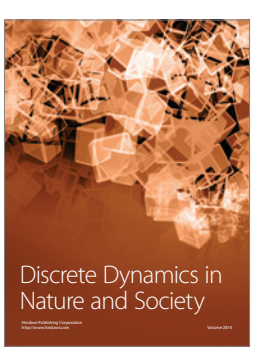
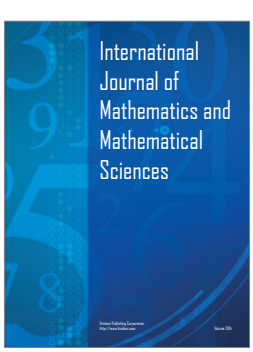
[23] R. Khondoker, A. Zaalouk, R. Marx, and K. Bayarou, "Feature-based comparison and selection of Software Defined Networking (SDN) controllers," in *Proceedings of the World Congress on Computer Applications and Information Systems (WCCAIS '14)*, January 2014.









Hindawi

Submit your manuscripts at
https://www.hindawi.com