Hindawi
Security and Communication Networks
Volume 2023, Article ID 9864245, 1 page
https://doi.org/10.1155/2023/9864245

Retraction

Retracted: Vulnerability Digging for Software-Defined Network Controller Using Event Flow Graph Analysis

Security and Communication Networks

Received 11 July 2023; Accepted 11 July 2023; Published 12 July 2023

This article has been retracted by Hindawi following an investigation undertaken by the publisher [1]. This investigation has uncovered evidence of one or more of the following indicators of systematic manipulation of the publication process:

(1) Discrepancies in scope

(2) Discrepancies in the description of the research reported

(3) Discrepancies between the availability of data and the research described

(4) Inappropriate citations

(5) Incoherent, meaningless and/or irrelevant content included in the article

(6) Peer-review manipulation

The presence of these indicators undermines our confidence in the integrity of the article's content and we cannot, therefore, vouch for its reliability. Please note that this notice is intended solely to alert readers that the content of this article is unreliable. We have not investigated whether authors were aware of or involved in the systematic manipulation of the publication process.

Wiley and Hindawi regrets that the usual quality checks did not identify these issues before publication and have since put additional measures in place to safeguard research integrity.

We wish to credit our own Research Integrity and Research Publishing teams and anonymous and named external researchers and research integrity experts for contributing to this investigation.

The corresponding author, as the representative of all authors, has been given the opportunity to register their agreement or disagreement to this retraction. We have kept a record of any response received.

References

[1] Z. Wu, W. Zhang, Y. Wang, and C. Yan, "Vulnerability Digging for Software-Defined Network Controller Using Event Flow Graph Analysis," *Security and Communication Networks*, vol. 2022, Article ID 9642517, 19 pages, 2022.

Hindawi
Security and Communication Networks
Volume 2022, Article ID 9642517, 19 pages
https://doi.org/10.1155/2022/9642517

Research Article

Vulnerability Digging for Software-Defined Network Controller Using Event Flow Graph Analysis

Zehui Wu, Wenbin Zhang, Yunchao Wang, and Chenyu Yan

State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, China

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

Received 11 May 2022; Revised 26 May 2022; Accepted 26 May 2022; Published 24 June 2022

Academic Editor: Mohammad Ayoub Khan

Software-defined network (SDN) controllers, the core of SDN network architecture, need to deal with network events of the whole network, which has huge program state space and complex logic dependency, with security issues related. Vulnerabilities in the SDN controller can paralyze the whole network. Existing controller testing methods are difficult to dig into the hidden logic vulnerability for their ignorance of the complex events interactions among controllers, apps, and data plane inputs. Different from file processing software, network software is driven by events, and the event flow can more accurately and comprehensively reflect the execution process. In this work, we propose an SDN controller vulnerability digging method based on event flow graph analysis. The proposed method consists of three main steps: first, we execute the instrumented controller in a normal environment and generate event flow graphs and then extract their features as reference. Second, we generate and execute test cases using the fuzzing method and dig the newly built event flow graphs with potential vulnerabilities. Finally, we manually examine and validate the potential vulnerabilities. To accurately discover abnormal subgraphs, we utilize graph feature extraction and analysis technologies, such as graph mining and clustering, to distinguish the normal graph and abnormal graph. We implement our method on the Ryu controller and compare it with other SDN testing methods, such as BEADS and Delta. The evaluation indicates that our method uncovered three new vulnerabilities that other methods failed to find.

1. Introduction

As a network architecture based on the idea of network virtualization, SDN (software defined networks) separates controlling and forwarding of the network and provides an application programming interface to network applications, which achieve diversified network functions and make network management more elastic and flexible [1, 2]. Due to these advantages, SDN has been deployed in many data centers and cloud computing environments and has a promising future [3, 4].

However, SDN also has security risks. The centralized control and programmability of the network extend the attack surface [5]. Especially, the controller with the global view of the network is the main target of the attackers aiming at paralyzing the whole network. The programming interface not only brings convenience for applications to access network resources and control network behavior but also provides a shortcut for malicious applications to damage the network. In addition, there are logical loopholes in the controller, resulting in flow rule conflict, packet forwarding error, and other hazards. Former research on SDN security analyzed the overall network architecture or the vulnerability of isolated parts and revealed the security threats existing in SDN, such as network element counterfeiting [6–8], data tampering [9–14], information disclosure [15–18], denial of service [19–23], authority promotion [24, 25], and so on, and putt forward various attack methods.

Global network monitoring is the real value of centralized control architecture network, which requires the controller to have the ability to supervise all network states. However, the data plane has high complexity. In different scenarios, the state of the data plane varies greatly in terms of structure and parameters and changes at any time. Theoretically, the network state that the controller needs to deal with is infinite, and correspondingly, the program state space of the controller is also infinite. This complexity brings many difficulties in testing the SDN controller.

Existing SDN security testing frameworks, such as Delta [26], BEADS [2], and ATTAIN [27], apply the ideas of traditional software vulnerability detecting methods such as fuzzing or symbolic execution. These methods improve the efficiency of SDN security testing. However, they do not consider the complex states when generating test cases and monitoring errors, thus not capable of exploring deep states and finding logic errors.

We use a runtime event flow graph to model the runtime state of the controller in granularity of events and propose a method for controller vulnerability detection based on event flow graph analysis. We monitor the event generating, distributing, and processing process at runtime by instrumenting the controller and construct an event flow graph accordingly. To build ground truth, the event flow graphs of the normal operations of the network are firstly analyzed to obtain the statistical and structural features. Then we build a variety of mutated test cases using fuzzing ideas. For each test case, we analyze the features of the event flow graph obtained during its execution and compare them with the features of normal execution to judge whether there are anomalies. If an abnormality is observed, we analyze the deep causes and judge if a serious loophole exists in the controller.

To accurately distinguish the difference between event flow graphs of normal and abnormal execution, we utilize graph analysis technologies such as frequent subgraph mining and graph clustering. To generate test cases able to trigger more event processing functions quickly, we design a data plane simulating method to interact with the controller under test with premeditated behaviors. In this paper, we use the open-source, Python-based Ryu SDN controller as a representative case study and identify three vulnerabilities. The results demonstrate that our method has good performance in program state coverage and vulnerability detection.

Our main contributions are as follows:

(1) We model vulnerability discovering of SDN controller as a graph comparing problem.

(2) We build an event flow graph to represent the execution state of the controller and distinguish the distinction between different graphs to find the potential vulnerabilities. Instead of simply sending crafted packets to the controller, we use a data plane simulating method to trigger more event processing functions.

(3) We implement our approach and compare the ability to discover vulnerability with existing methods. We find three new vulnerabilities.

2. Background

2.1. SDN Architecture. Open networking foundation [28] divides the SDN architecture into three layers: data plane, control plane, and application plane, as shown in Figure 1. The control plane communicates with the application plane and the data plane through the north channel and the south channel. At present, there is no standard communication protocol for the north channel, and the most widely used communication protocol for the south channel is the OpenFlow protocol [29].

The data plane is formed by the interconnection of forwarding devices such as switches. The switch used by SDN supports OpenFlow protocol, usually does not have self-learning ability, and is only responsible for data forwarding. Each switch is composed of the following components: packet receiving and sending ports, internal flow table, and communication interface with the controller. The flow rules are stored in the flow table, and the switch processes the received packets according to the flow rules.

The control plane is composed of one or more controllers and is the core of SDN architecture. The core services of the controller maintain the view of the whole network, such as network topology and information of hosts and switches. The network state of the data plane is obtained through the south interface. Through the application programming interface, the application plane can request the network state from the control plane to realize different network functions. In multicontroller SDN, different controllers conduct state synchronization and data exchange through the east-west interface.

The application plane is composed of application programs with different network functions, which is the typical embodiment of SDN programmability. The application plane communicates with the control plane through the north interface and uses the obtained network state information to realize the network functions such as routing, load balancing, firewall, QoS, and so on. The application converts the network management policy into flow rules and installs them to the data plane switches through the controller.

2.2. Event Flow. Popular SDN controllers are asynchronous event-driven architectures, which use events as the basic data structure to transfer information between controllers and applications [25]. Events are abstractions of data plane and control plane activities, such as receiving a new packet or changing the state of a data plane equipment. According to the objects described, events are divided into different types, such as host events, connection events, and events inside the control plane. As shown in Figure 1, an event is generated by the event generator and then distributed to the corresponding event listener for processing. For example, the OpenFlow message manager will capsulate the received data from the data plane as OpenFlow events and send them to applications processing this message. The application calls the API provided by the controller to register as a listener for a certain type of events to receive and process them. When processing events, the application can also generate new events and distribute them to other listeners. Therefore, an event flow graph is formed according to the relationship of event subscription and distribution between the controller core services and the applications.

The security state of the SDN control plane depends not only on the security state of a single controller or application component but also on whether the process of event interaction of all interdependent components is safe.

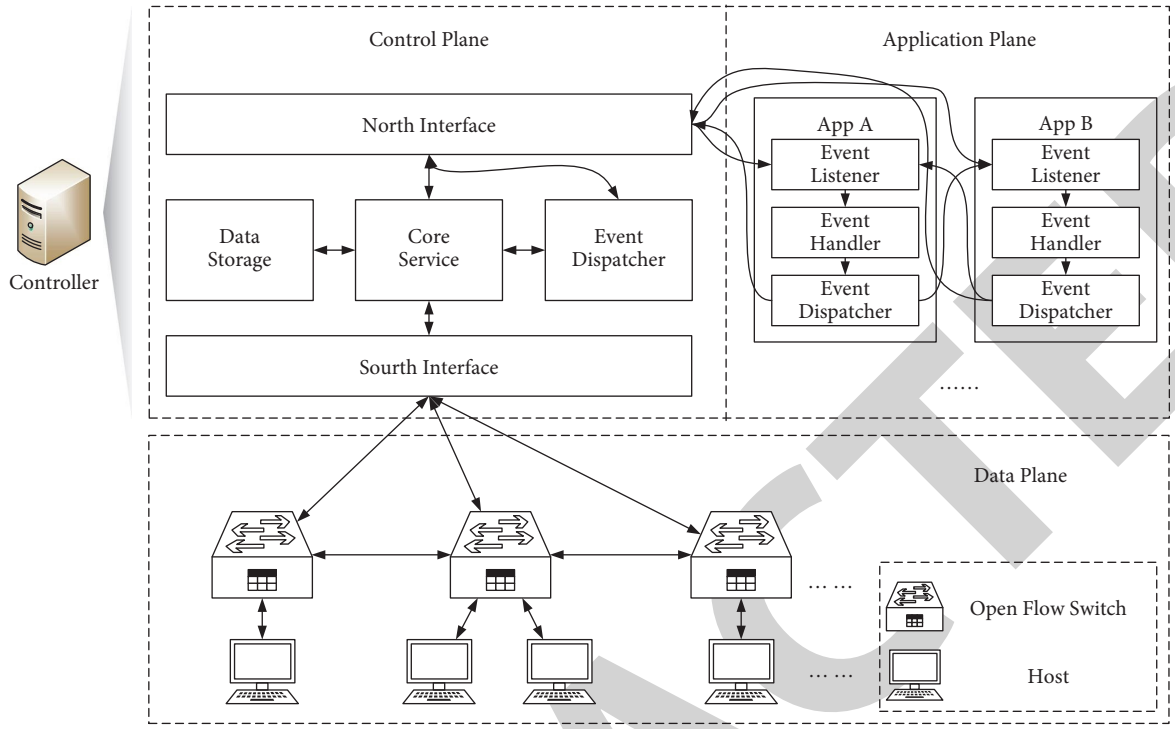

Figure 1: SDN architecture.

Applications may produce unpredictable behaviors and trigger unknown security vulnerabilities when working together, much more complex than running independently according to their own logic. Malicious data from the data plane or application plane may cause the event handler of the control plane to execute the wrong code path and cause conflicts between applications.

Most vulnerabilities triggered by complex events interaction are logic vulnerabilities. Detecting such vulnerabilities requires a local and global semantic understanding of events and their processing process [25]. The event flow graph can observe the dependencies between various components of the controller from a macro perspective, which is helpful to reveal the logic vulnerabilities.

3. Related Work

3.1. Automatic SDN Vulnerability Mining Methods. In order to comprehensively evaluate the security of SDN, some researchers have tried to automatically mine the vulnerabilities of SDN, proposed some detection methods, and implemented corresponding tools.

In 2016, [30] conducted security tests on SDN for the southbound interface and used existing tools such as ProxyFuzz to construct packets and inject to the network for vulnerability detection. It is a relatively primary test method, which is difficult to detect complex vulnerabilities.

In 2017, [31] designed a protocol-aware SDN test tool FlowFuzz and constructed an effective input corpus. They adopted guided or random input generation strategies and used code coverage as test feedback. However, the false positive caused by switch reconnection is high, and the crash cannot be reproduced, causing no security vulnerability detected. However, this work points out a direction for SDN automatic testing: the test process should be fully controllable, the controller execution path should be fully explored, and the feedback mechanism needs to be integrated.

In 2017, Lee et al. [32] implemented an SDN vulnerability identification framework, Delta. Delta consists of an agent manager module, host agent module, application agent module, channel agent module, and fuzzing module, which can reproduce the attack scenario and find new vulnerabilities. Delta models the controller workflow as a state transition graph and performs fuzzing by manipulating the control flow sequence for input randomization. However, the test strategy of only numerical fuzzy and control flow randomization is also hard to test the deeper controller logic.

In 2017, Jero et al. [2] proposed the SDN testing framework BEADS. Unlike most test cases generated by random fuzzing, which can only test the OpenFlow message parser, BEADS considers the semantics of data packet fields to test the deeper level of the message processing algorithm.

In 2017, Ujcich et al. [27] proposed ATTAIN, an attack injection framework for SDN. ATTAIN defines an attack model that combines system components and the attacker's ability to affect the control plane. It also defines an attack language to evaluate the implementation of SDN.

In 2012, Canini [32] proposed NICE, a method to explore the state space of the whole SDN network system using a model check. In order to solve the scalability problem caused by the huge system state, model checking is enhanced by symbolic execution.

In 2020, Ujcich et al. [25] proposed the SDN vulnerability detection tool EventScope, which can analyze the use

of SDN control plane events. If the application is missing to handle certain events, it will be used as a candidate vulnerability. However, EventScope only analyzes the static event flow graph of the controller, ignoring the complex dynamic event interaction when the controller is running.

In 2021, Jafarian et al. [33] proposed a multistage modular approach for detecting security anomalies in the SDN environment (SADM-SDNC). They first collect traffic features of the network and then use the C-support vector classification algorithm to evaluate the controller with regard to anomaly detection.

These methods ignore the complex states and internal activities of the running controller, thus not capable of exploring deep states and finding logic errors. In contrast, our work uses an event flow graph to model the control flow and data flow between core components of SDN controller and applications and deeply analyze controller behaviors with the acquired relationship between control and data.

3.2. Graph Analysis. A graph is a common data structure, which can represent the distribution state of nodes and the relationship between them. It is widely used in the real world. In recent years, with the development of big data, graph mining has become an important branch in the field of data mining, mainly including subgraph mining (frequent subgraph mining [34, 35], area molecular graph mining [36], significant subgraph mining [37], graph clustering, graph matching, etc). Frequent subgraph mining is the process of identifying frequent subgraphs whose frequency is not less than the specified threshold from a group of graphs (graph database) or a single large graph. It is widely used in graph classification, graph index, and graph clustering.

In the field of program analysis and security, a lot of work has been learned from the above methods in the field of graph theory, for example, code similarity detection based on graph matching [38], malicious program detection method based on subgraph matching [39], program fault location method based on graph mining [40], and so on.

4. Framework Overview

The method proposed in this paper draws lessons from the idea of graph mining and graph matching. Figure 2 depicts the workflow of our method, which has two main parts: normal event flow graph feature extraction and anomaly detection.

The first part extracts the features of the normal event flow graph. Firstly, we execute the instrumented controller in a normal environment to observe the event interaction process during execution and generate event flow graphs, building an event flow graph set. At this stage, the number distribution of event generation and processing is calculated as normal statistical features, and the normal structural features of the event flow graph set are extracted.

The second part is vulnerability detection. When performing exception detection, many test cases are generated using the fuzzing method. For each test case, the events processing its runtime information are also obtained, and the event flow graph is constructed. Next, the statistical and structural features are also extracted for anomaly detecting.

5. Feature Extraction of Event Flow Graph

The abnormalities of the event flow graph can be divided into two categories: statistical abnormalities and structural abnormalities. Statistical abnormality refers to the abnormal number and generation rate of some events during the operation of the controller, which can be caused by DoS or other attacks. Structural anomalies include abnormal graph patterns in both the static structure and dynamic growth of the event flow graph. To detect these abnormalities, we extract corresponding features of the event flow graph.

5.1. Analysis of Controller Event Mechanism. The runtime event flow graph is obtained when the instrumented controller is running. Before instrumenting, we need to analyze what event types are defined by the controller and how it generates, distributes, and handles events and then insert monitoring code at key locations.

Take the Ryu controller as an example. Ryu uses *set_ev_cls()* decorator, a Python feature, to allow an application to listen to an event. Applications generating events call *get_observers()* to get all listeners for the events and then call *send_event()* to send the events to them. The event production and consumption relationship between applications is set in the initialization phase of the controller. For example, the switch manager will generate an EventSwitchEnter event, indicating that a new switch has been added to the network, and send this event to the pending event queue of the topology manager, which has registered to listen for this event.

Formally, the event flow graph is a directed graph, represented as $G = (V, E)$, V is a collection of vertices of the following types: event generator, event, and event handler. Some nodes are both event generators and event handlers. E represents a collection of edges of two types: event generation and event processing.

Therefore, by inserting monitoring code before the event sending statement and event processing method, the execution of the controller can be tracked, and the dynamic runtime event flow diagram can be constructed. The local static event flow graph of the Ryu controller is shown in Figure 3, in which the purple circular node represents the event generation function of the application. The blue circular nodes represent events. The green circular node represents the event handling function of the application and will generate new events at the same time. The yellow rectangular node represents the application's event handler, which will not generate new events.

5.2. Constructing Event Flow Graph. The event information obtained during the operation of the instrumented controller is a triple sequence composed of "event handler, event, and time" or "event provider, event, and time," which needs to be converted into an event flow graph.

In order to support the subsequent structural feature extraction, we constructed two kinds of event flow graphs, namely static event flow graph and dynamic event flow graph. The static event flow graph includes two parts: the

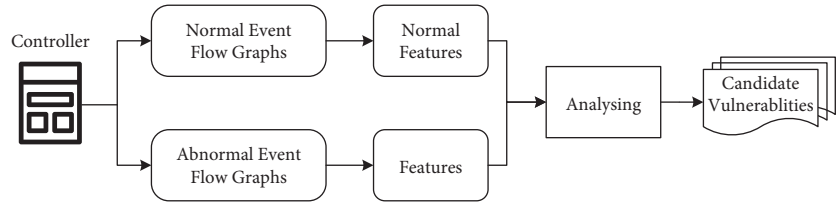

Figure 2: Overview of the method.

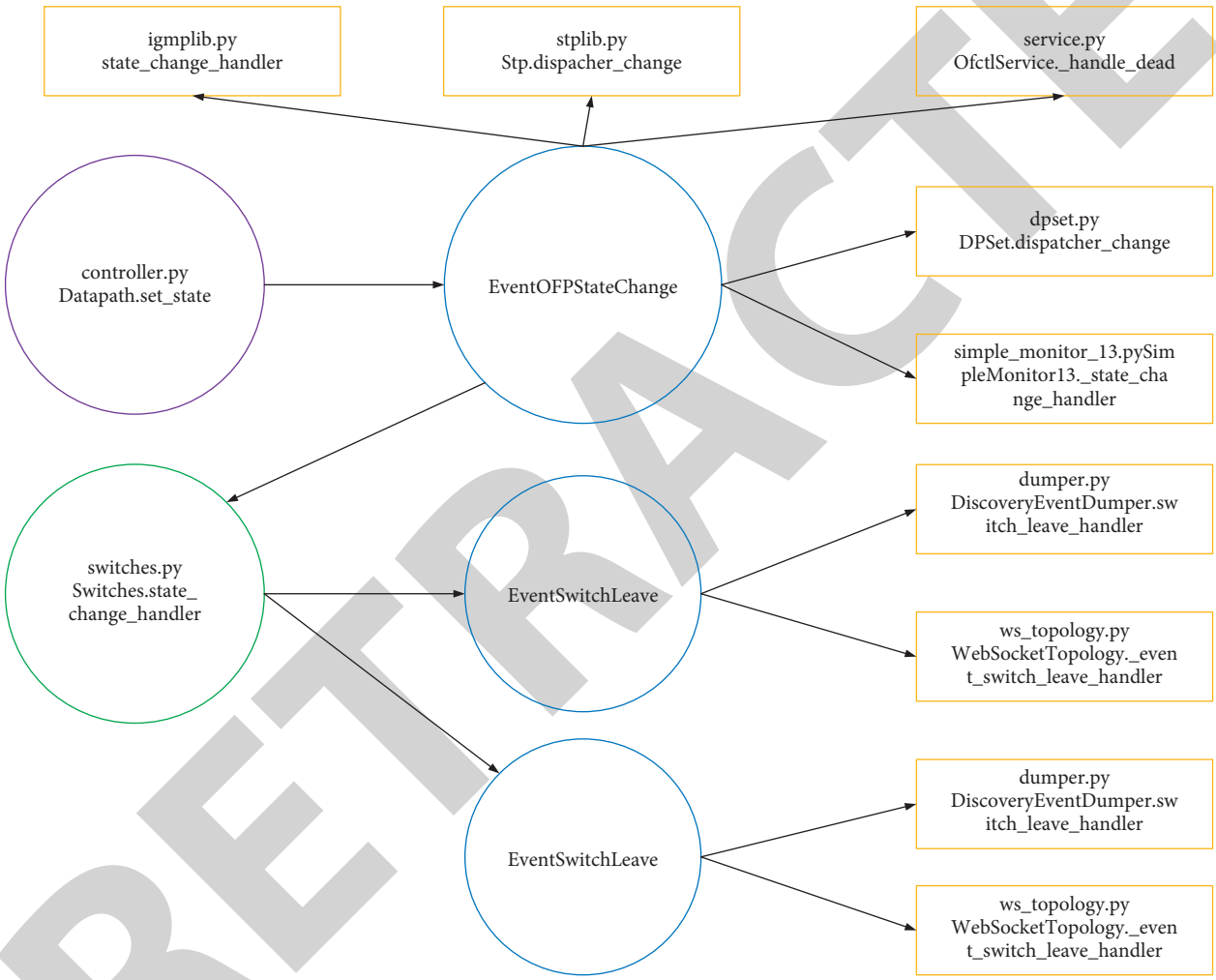

Figure 3: Partial event flow graph of Ryu.

overall event flow graph and the slice event flow graph. A dynamic event flow graph is a sequence of event flow graphs that changes according to time information. The constructing algorithm of the static event flow graph is outlined in Algorithm 1.

Here, G is the event flow graph to be constructed. And the frequency table FreMat is used to record the frequency information of each node as the weight to reflect the high-frequency execution path of the program.

Because the type of events is limited, if the weight information of edges is not considered, the constructed event flow graph is likely to be isomorphic with the global static event flow graph, and there is no distinction between different graphs. To solve this problem, the static event flow graph we constructed is a multigraph, where multiple edges exist with different time information between each pair of nodes. By slicing and extracting the event flow graph in a certain time period, we can obtain a subgraph representing the corresponding execution logic. As shown in Figure 4, Figure 4(a) is the constructed multiple event flow graph, and Figures 4(b)–4(d) are the event flow graph in time periods T1, T2, and T3, respectively. The size of the time period can be adjusted as needed.

After constructing the static event flow graph, the original event generation and processing sequence are retained to restore the context information when analyzing

```
Inputs: triple sequence S
Outputs: static event flow graph G, event frequency table FreMat
(1) G = {}, FreMat = {}
(2) for node in sequence:
(3)     if node ∈ G:
(4)         FreMat[node]++
(5)     if node ∉ G:
(6)         G.addnode(node),
(7)         FreMat.add(node)
(8)         if(node.EventHandler),
(9)             G.addedge(node.EventHandler, node.Event, node.Time)
(10)            if(node.EventProvider),
(11)                G.addedge(node. EventProvider, node.Event, node.Time)
```

Algorithm 1: Static event flow graph constructing.

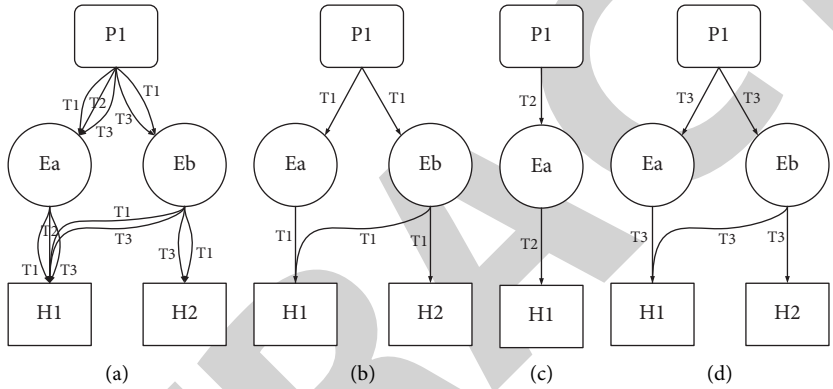

Figure 4: Static event flow graph.

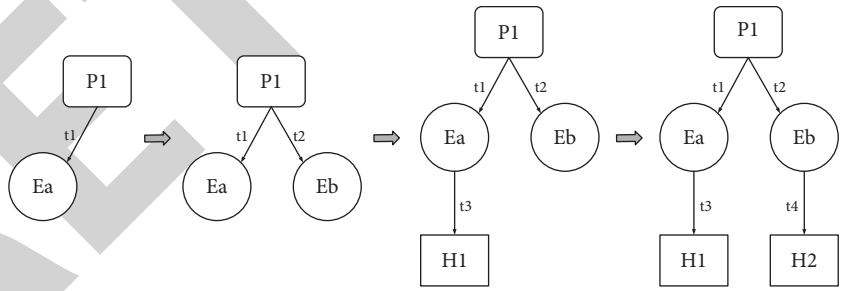

Figure 5: Evolution sequence of the dynamic event flow graph.

the abnormal event flow graph structure and to construct the dynamic event flow graph evolution sequence. The evolution sequence of the dynamic event flow graph is shown in Figure 5, which shows the growth of the event flow graph from time $t1$ to $t4$.

5.3. Statistical Feature Extraction of Event Flow Graph. In the process of network operation, different events are generated and processed with different frequencies. For example, EventOFPPacketIn triggered by the data plane is frequently generated and processed, while EventOFPFlowMod is generated and sent to the data plane under specific circumstances. In the normal operation of the network, the distribution of the proportion of various events is within a certain range. However, due to the short-term changes of low-frequency events, great differences in the number proportion exist in different event flow graphs, so it is improper to use the average proportion for anomaly judgment. Therefore, we calculate the change rate of each event as a supplement. If the generation rate of an event changes greatly within a time period, it indicates that the

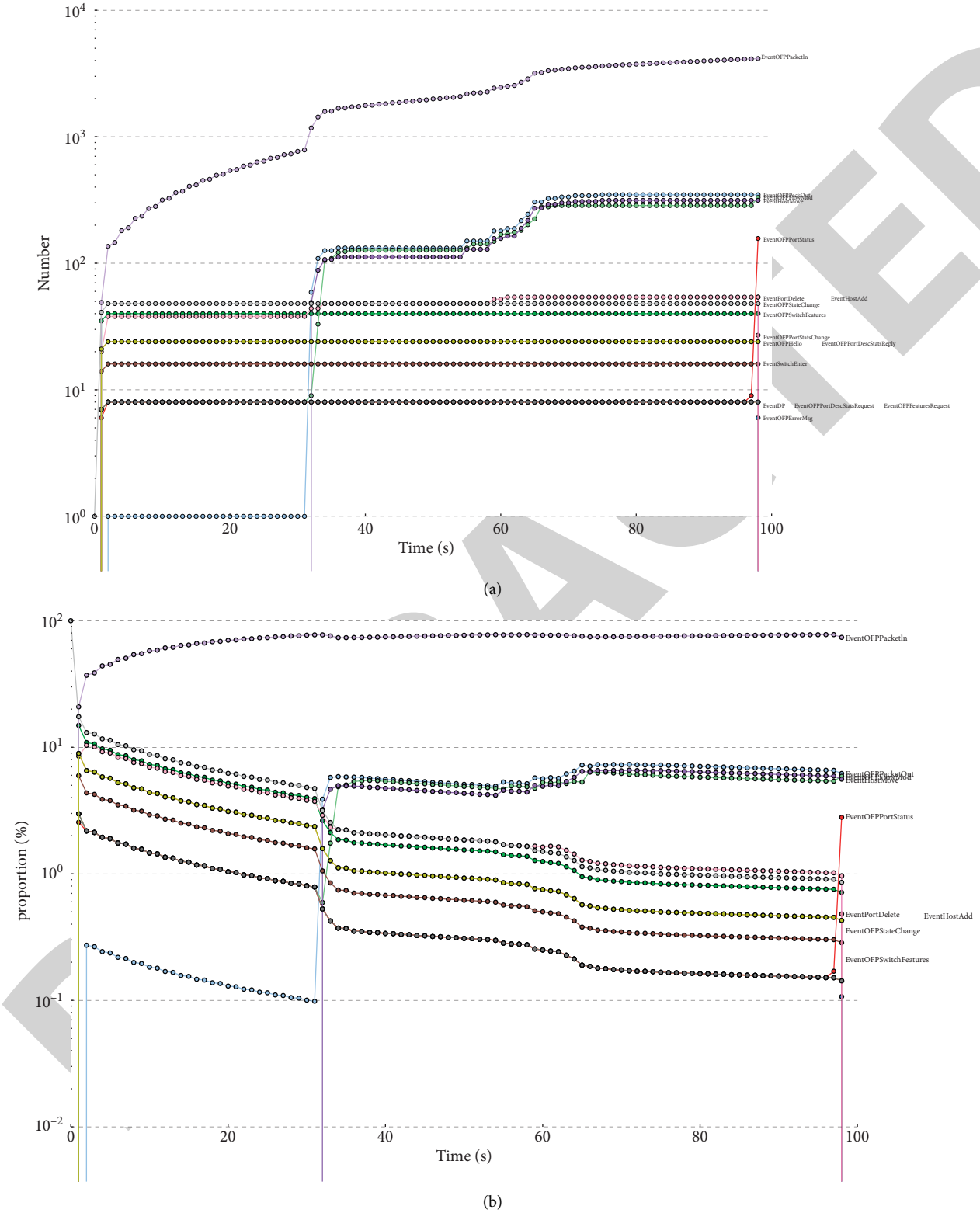

Figure 6: Changes in the proportion and number of events.

network has abnormal behavior. We collected 500 event flow graphs, and the changes in the average number and the changes in the average proportion with time are shown in Figures 6(a) and 6(b), respectively.

Figure 6 shows the trend chart of the number and proportion of events drawn at an interval of 1 second. Due to the fast generation speed of events, some events, such as EventOFPErrorMsg, reach the maximum value

Inputs: event flow graph set GS, minimum frequency α
Outputs: frequency subgraph set S
(1) Sort nodes and edges in GS by frequency
(2) Remove nodes and edges with low frequency
(3) Relabel the remained nodes and edges by frequency
(4) $S' \leftarrow$ all frequent 1-edge subgraph
(5) Sort subgraphs in S' in DFS dictionary order
(6) $S \leftarrow S'$
(7) for each edge $e \in S'$ do
(8) $s = e, s.\text{GS} = \{G \mid G \in \text{GS}, e \in G\}$
(9) Subgraph_Mining(GS, S, s)
(10) $GS \leftarrow GS - e$
(11) if $\mid GS \mid < \alpha$
(12) break

Algorithm 2: Topk_FrequentSubGraphMining(GS, α).

(1) if s ! = min(s)
(2) return
(3) $S \leftarrow S \cup \{s\}$
(4) Use the rightmost extension on each graph in GS to generate a set of extended subgraphs
(5) for each c, c is s' child do
(6) if sup $(c) \geq \alpha$
(7) $S \leftarrow c$
(8) SubGraph_Mining(GS, S, s)

Algorithm 3: Subgraph_Mining(GS, S, s).

Inputs: the root node $v \in G$, G is an event flow graph, d is the depth
Outputs: sg is a subgraph with root node is v and depth is d.
(1) visited = [], queue = []
(2) visited.add(v), queue.put(v), $d = d - 1$
(3) while d >0,
(4) $node$ = queue.get()
(5) $neighbors = \{ngb \mid ngb$ is neighbor of $node\}$
(6) for each ngb in $neighbors$:
(7) sg.add(ngb)
(8) queue.put(ngb)
(9) $d = d - 1$

Algorithm 4: BFS-SubG(v, G, d).

within 1 second, showing a vertical growth trend in the chart.

5.4. Structure Feature Extraction of Event Flow Graph. Structural features include static structural features and dynamic structural features. The static structure feature refers to the feature of the event flow graph without time information, while the dynamic feature considers the time information of events.

5.4.1. Static Structural Features

(1) Frequent Subgraph Mining. The purpose of frequent subgraph mining is to extract the conventional execution paths in most execution of the controller. The probability of error of these paths is relatively low.

Frequent subgraph mining is to find all subgraph patterns that meet the frequency threshold from the graph database based on the frequency threshold given by the user. The definition of frequency [41] is that given a graph database $\text{GS} = \{G_1, G_1, \ldots, G_n\}$ and a graph pattern g, the set of graphs supporting g is $\text{Cov}(g) = \{G_i \mid g \subseteq G_i, G_i \in \text{GS}\}$. The support degree of g is $|\text{Cov}(g)|$, expressed as $\sup(g)$. The frequency of g in graph database is $|\text{Cov}(g)|/\text{GS}|$, expressed as $\text{freq}(g)$.

We use gSpan [35] algorithm to mine frequent subgraphs. gSpan is a classical frequent subgraph mining algorithm based on depth-first search. Its basic idea is to normalize and expand the graph and transform the frequent

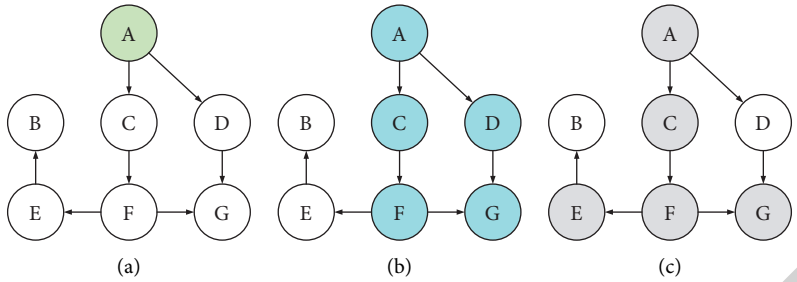

Figure 7: Schematic diagram of BFS-SubG algorithm.

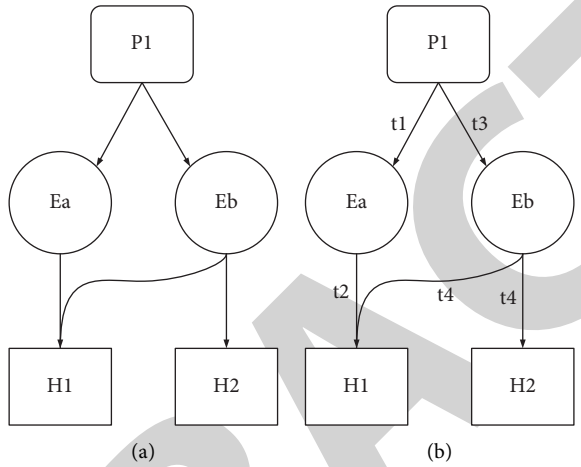

Figure 8: Comparison between static event flow graph and dynamic event flow graph.

subgraph mining problem into the corresponding normalized representation mining problem. The algorithm is outlined in Algorithm 2.

Called subprocedure subgraph_Mining (*GS*), see Algorithm 3.

Suppose N event flow graphs are generated, and the minimum support degree is defined as $\alpha = 0.8\,\mathrm{N}$. The mined frequent subgraphs are sorted according to the support degree, and the former k subgraphs are taken as the set of frequent subgraphs. The generated Top-k frequent subgraph set is expressed as $\mathrm{freq}(D) = \{g \mid \sup(g) > \alpha\}$ and $|\mathrm{freq}(D)| = k$. The selection of k is determined by the execution features of SDN controller.

(2) Constructing Feature Subgraph Set. Using the subgraph generation algorithm BFS-SubG based on breadth-first search, a subgraph with depth d can be extracted from an event flow graph G with any node as the root node. The algorithm is outlined in Algorithm 4.

The above algorithm can extract the subgraph with v as the root node and depth d in the event flow graph. For event flow graph G, all subgraphs obtained using the above algorithm are represented as $\mathrm{subgraphs}(G) = \{g \mid g \subseteq G, g.\,\mathrm{deepth} \in \{0, 1, \ldots, D\}\}$, where D is the maximum depth. Figures 7(b) and 7(c) are subgraphs of Figure 7(a), with node A as the root node and depth of 3 and 4, respectively.

After the subgraphs of an event flow graph are generated, its feature subgraphs can be obtained by subtracting the frequent subgraphs of GS. These feature subgraphs indicate the special paths of the execution triggered by the inputs.

5.4.2. Dynamic Structural Features

(1) Dynamic Event Flow Graph. If the edges do not contain time information, the topological order between adjacent nodes in the static event flow graph is fixed, so the structure of the graph is determined. If adding time information, the static event flow graph will have dynamic features, and there are more possibilities for the topological order of its different subgraphs. Anomalies may exist in these new structures.

The comparison between the static event flow graph and the dynamic event flow graph is shown in Figure 8. Figure 8(a) is a part of the static event flow graph. The events Ea and Eb on the left and right subtrees have no sequential relationship. Figure 8(b) is the dynamic event flow graph with time information, so the time sequence of Ea and Eb can be determined, so as to the sequence relationship of subsequent events. Assuming that there is a logical causal relationship between Ea and Ea on the impact of the network state, it is difficult to mine the abnormal structure of the logical relationship between events in the static event flow graph because the temporal relationship between them is

Input: event flow graph sequence G_1^T, minimum support degree σ, k is time step of subgraph sequence。
Output: all subgraph sequences S_1^k that satisfy Support$(S_1^k) \geq \sigma$
(1) result←∅
(2) f Edges is the set of frequent edges in G_1^T
(3) for each $e \in f$ Edges:
(4) result←result ∪ SG Seq Extension$(e, G_1^T, \sigma, k, f$ Edges$)$
(5) remove e from f Edges
(6) return result

Algorithm 5: FSGSeqMining.

Input: initial sequence of event flow graph subgraphs sequence S_1^k, event flow graph sequence G_1^T, minimum support degree σ, k is time step of subgraph sequence, f Edges is the set of frequent edges
Output: all subgraph sequences extended from S_1^k
(1) result←S_1^k, candidateSet←∅
(2) for each $e \in f$ Edges and node $u \in V_{S_1}^k$:
(3) if (e can be extended from u):
(4) ext = S_1^kextende
(5) if (ext is Null):
(6) candidateSet←candidateSet ∪ ext
(7) for each $c \in$ candidateSet:
(8) Solve the constraint satisfaction problem of c to G_1^T
(9) if(each variable has at least σ valid value):
(10) result←result ∪ SG Seq Extension$(e, G_1^T, \sigma, k, f$ Edges$)$
(11) return result

Algorithm 6: SGSeqExtension.

Table 1: Proportion range and average quantity curve parameters of some events.

Event	Range of proportion	Average number change curve parameters	
		Coefficient	Intercept
EventOFPFlowMod	(0.00, 5.40)	[0.0, 2.77, −0.65, 0.07, −0.0, 0.0, −0.0]	0.0
EventDP	(0.00, 2.17)	[0.0, 2.77, −0.65, 0.07, −0.0, 0.0, −0.0]	0.0
EventOFPPortDescStatsRequest	(0.00, 2.17)	[0.0, 2.77, −0.65, 0.07, −0.0, 0.0, −0.0]	0.0
EventOFPPortstatus	(0.00, 2.93)	[0.0, 7.78, −2.52, 0.35, −0.02, 0.0, −0.0]	−1.0
EventOFPPortStateChange	(0.00, 0.51)	[0.0, 0.34, −0.14, 0.02, −0.0, 0.0, 0.0]	0.0
EventOFPPacketIn	(0.00, 78.77)	[0.0, 28.81, −3.64, 0.423, −0.03, 0.0, −0.0]	−4.0
EventSwitchEnter	(0.00, 4.35)	[0.0, 5.65, −1.38, 0.16, −0.01, 0.0, −0.0]	0.0
EventOFPSwitchFeatures	(0.00, 10.87)	[0.0, 14.17, −3.49, 0.41, −0.02, 0.0, −0.0]	−1.0
EventHostAdd	(0.00, 10.33)	[0.0, 6.31, −1.52, 0.17, −0.01, 0.0, −0.0]	0.0
EventOFPStateChange	(0.91, 100.00)	[0.0, 18.89, −4.80, 0.57, −0.03, 0.0, −0.0]	1.0
EventPortDelete	(0.00, 1.02)	[0.0, 0.65, −0.23, 0.05, −0.0, 0.0, −0.0]	0.0
EventOFPFeaturesRequest	(0.00, 2.17)	[0.0, 2.77, −0.65, 0.07, −0.0, 0.0, −0.0]	0.0
EventOFPEchoRequest	(0.00, 0.72)	[0.0, 0.02, −0.03, 0.01, −0.0, 0.0, −0.0]	0.0
EventOFPEchoReply	(0.00, 0.24)	[0.0, 0.19, −0.08, 0.01, −0.0, 0.0, 0.0]	0.0
EventOFPHello	(0.00, 6.52)	[0.0, 8.42, −2.03, 0.23, −0.01, 0.0, −0.0]	−0.1

hidden. Considering the influence of time information, we use frequent evolution pattern mining to mine frequent subgraph sequences in the dynamic event flow graph.

(2) Frequent Evolution Pattern Mining. A frequent evolution pattern in dynamic graphs refers to a sequence of subgraphs that frequently appear in large graph sequences [42]. In the dynamic event flow graph, the frequent evolution pattern represents the specific substructure formed by some events with logical causality.

Given a dynamic event flow graph sequence $G_1^T = (G_1, G_2, \ldots, G_T)$ with length $T > 1$, the purpose is to mine the subgraph sequence $S_1^k = (S_1, S_2, \ldots, S_k)$ with length $k > 1$, and the subgraph sequence satisfies the following constraints:

(1) If there a sequence of graphs $g = (g_1, g_2, \ldots, g_k)$, $\forall 1 \leq i \leq k$, g_i is a subgraph of G_{j+1}, where $0 \leq j \leq T - k + 1$, and S_i is isomorphic with g; then it it is said the subgraph sequence S_1^k appears once in G_1^T

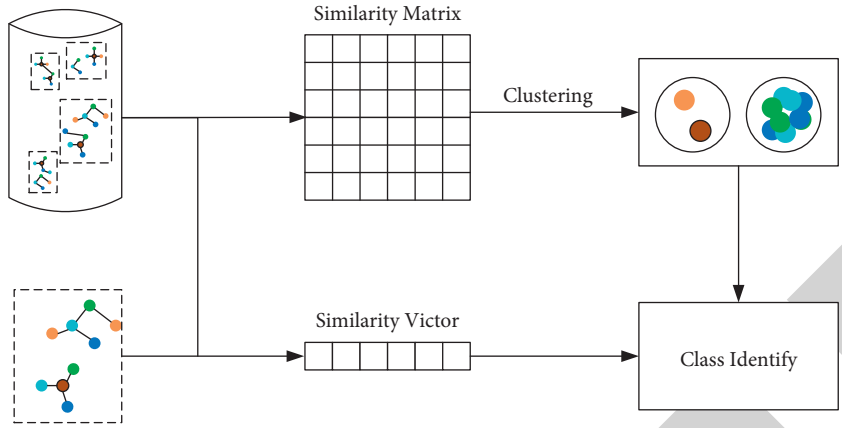

Figure 9: Judgment of abnormal static structure.

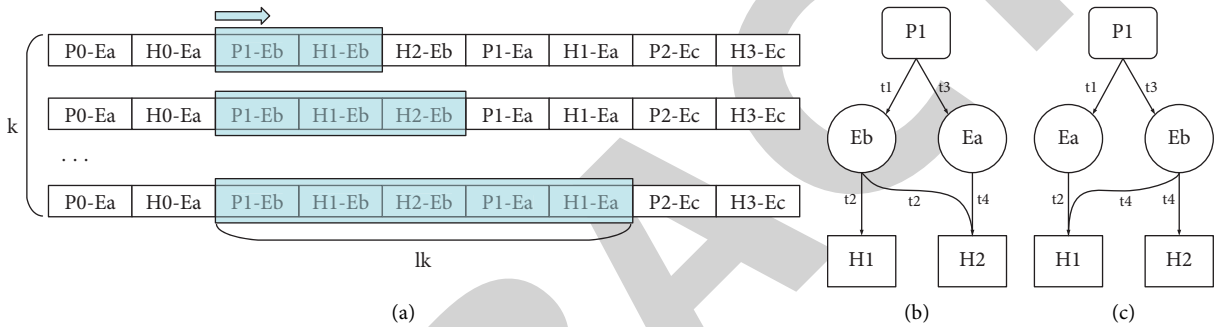

Figure 10: Judgment of abnormal dynamic structure.

(2) The times the subgraph sequence S_1^k appears in G_1^T is greater than a given support threshold σ, that is, Support $(S_1^k) \geq \sigma$

Based on pattern growth, a larger subgraph is obtained by expanding the smaller subgraph, and all subgraph sequences satisfying the minimum support are mined. We present the algorithm in Algorithm 5.

See Algorithm 6 for the subprocess *SGSeqExtension* called in line 4.

6. Anomaly Detection

6.1. Statistical Anomaly Detection. If there is a large deviation of the proportion or the generation rate of certain events compared with the normal conditions, we consider that the network operation is abnormal.

We use the data fitting method to calculate the range of the number proportion and the parameters of the number change curve of each type of event in the normal event flow graph. In order to ensure reliability, we collect 500 normal event flow graphs, calculate the proportion range of various events, and use a 6-order degree polynomial to fit the average quantity change curve. Table 1 lists the number proportion range and average number change curve parameters of some events.

For the event flow graph to be tested, we calculate the proportion of each event at a certain time and calculate the distance between its number and the normal average number change curve. If both are abnormal, we regard it as a candidate anomaly.

Table 2: Parameters for the test environment.

Target	Configuration
CPU	Intel(R) Core i7-8550U
Memory	4 GB
OS	Ubuntu 16.04
Python	3.5
Ryu	4.34

6.2. Anomaly Detection of Event Flow Graph Structure. For the event flow graph to be detected, we examine whether there is a structural anomaly from both static and dynamic aspects.

6.2.1. Abnormal Substructure of Static Event Flow Graph. In the feature extraction stage of a static event flow graph, a feature subgraph database composed of multiple event flow graph feature subgraphs is constructed. For the event flow graph to be detected, the similarity between its own feature subgraph and the feature subgraph of the normal event flow graph is calculated to judge whether the subgraph is abnormal. The process is shown in Figure 9.

```
Inputs: target file path: f_path network setup parameters:controllerOpts, switchOpts, hostOpts, linkOpts
Outputs: target network script: f_path
(1) f = open(f_path)
(2) f.write(imported files)
(3) f.write("net.addController(controllerOpts)")
(4) for switch_opt in switchOpts:
(5)     f.write("net.addSwitch(switch_opt)")
(6) for host_opt in hostOpts:
(7)     f.write("net.addHost(host_opt)")
(8) for link_opt in linkOpts:
(9)     f.write("net.addLink(host_opt)")
(10) f.write("net.build()")
(11) for switch_name, switch_cmd in switchOpts:
(12)     f.write("switch_name.cmd(switch_cmd)")
(13) for host_name, host_cmd in hostOpts:
(14)     f.write("host_name.cmd(host_cmd)")
(15) f.write("net.stop()")
(16) f.close()
```

Algorithm 7: Data plane network construction algorithm.

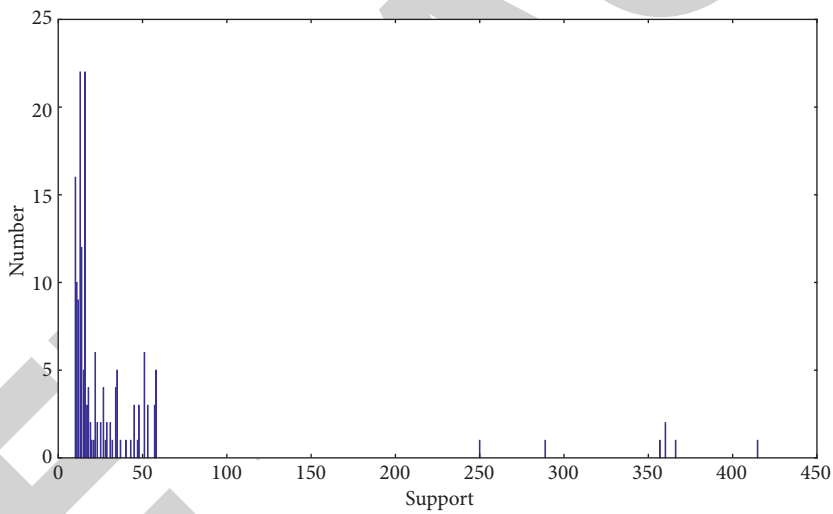

Figure 11: Frequency distribution of subgraphs.

Firstly, the similarity between the feature subgraphs of each pair of normal event flow graphs is calculated to generate a similarity matrix. Each column vector of the matrix represents the similarity between an event flow graph and other event flow graphs in the event flow graph set. Then we cluster these event flow graphs into several classes with different structures, which can represent most of the structural features of event flow graphs obtained under normal network conditions.

When detecting the event flow graph generated by a test case, we first generate its feature subgraph using the method described earlier. Then we calculate its similarity vector with the feature subgraphs of normal event flow graphs. Finally, the trained clustering model is used to calculate whether the feature vector belongs to a certain class. If it cannot be divided into any category, the event flow graph is considered to have an abnormal structure.

To calculate the similarity between graphs, we select the traditional similarity calculation method based on the maximum common subgraph (MCS). The distance function of graphs G_1 and G_2 is defined as follows:

$$d_{\mathrm{mcs}}(G_1, G_2) = 1 - \frac{\left|\mathrm{mcs}(G_1, G_2)\right|}{\max\left(\left|G_1\right|, \left|G_2\right|\right)}. \tag{1}$$

6.2.2. Abnormal Evolution Model of Dynamic Event Flow Graph. In the feature extraction stage of the dynamic event flow graph, frequent evolution pattern mining is carried out on the dynamic event flow graph data set with different support σ and step length k to obtain a set of subgraph sequences $S_d^\sigma = \{S_1^k \mid 2 \le k \le d, \mathrm{Support}(S_1^k) \ge \sigma\}$. These sequences represent the normal operation logic of the network. For the event flow graph to be detected, we check

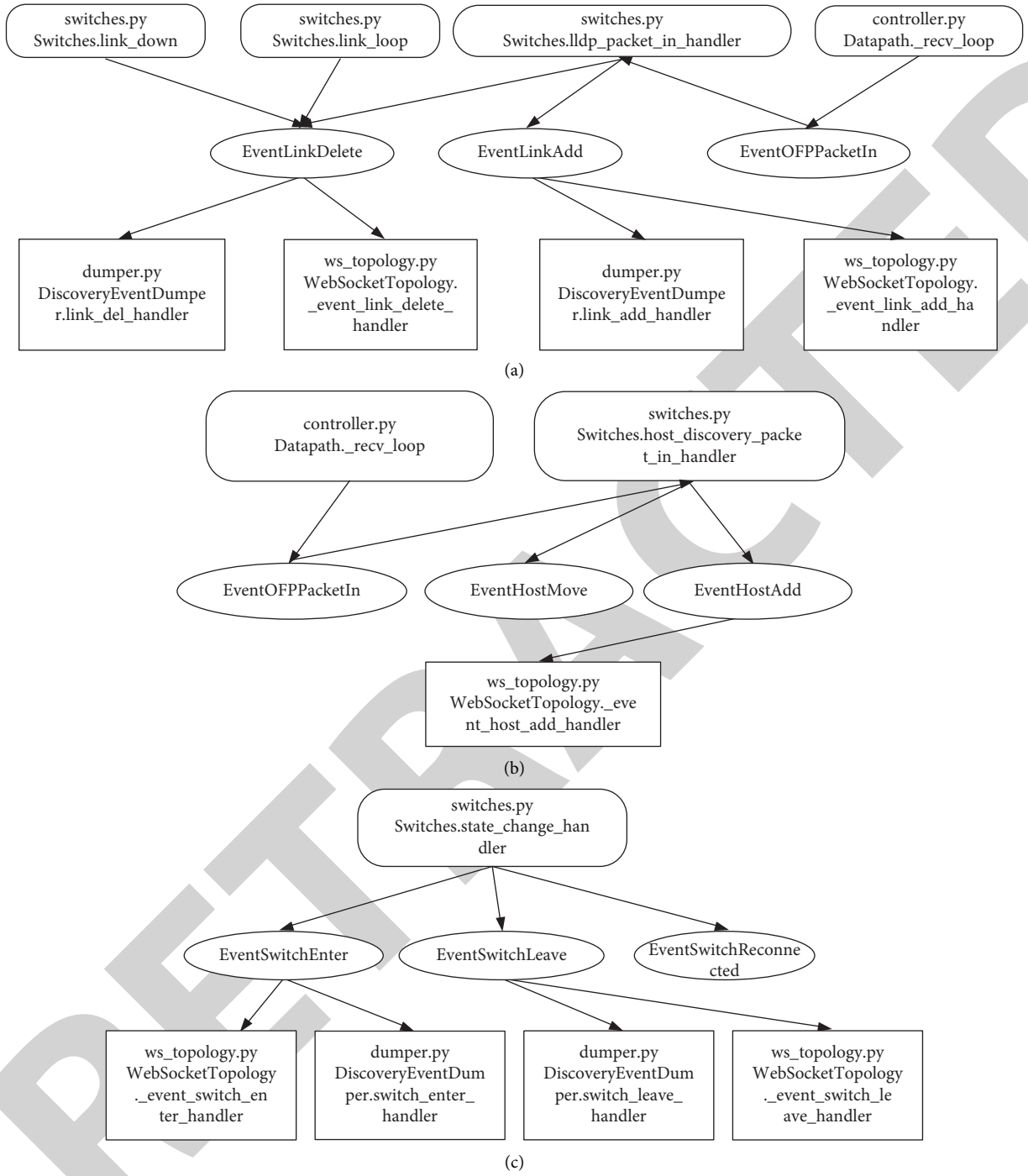

Figure 12: Some frequent subgraphs.

whether there is a subgraph structure contrary to the subgraph sequence of S_d^σ.

Assuming that the dynamic event flow graph to be detected is S_{test}, we use a sliding window with a length of l_1 to l_k (the number of nodes in the k-th event flow graph in S_1^k) to check whether there is any violation of the evolution sequence in the graph in S_1^k in its original event sequence The schematic graph is shown in Figure 10(a).

Assuming the event sequence in the first $k-1$ sliding windows is transformed into an event flow graph, which is consistent with the first $k-1$ event flow graph in S_1^k, the length of the k-th sliding window is l, whose transformed event flow graph is shown in Figure 10(b); the corresponding subgraph in S_1^k is 4.10(c). The two are inconsistent, so there is an abnormal event processing sequence.

6.2.3. Local Event Flow Graph Analysis. After mining the abnormal subgraph structure, further analysis is needed to verify the true harmfulness of the abnormal structure. The

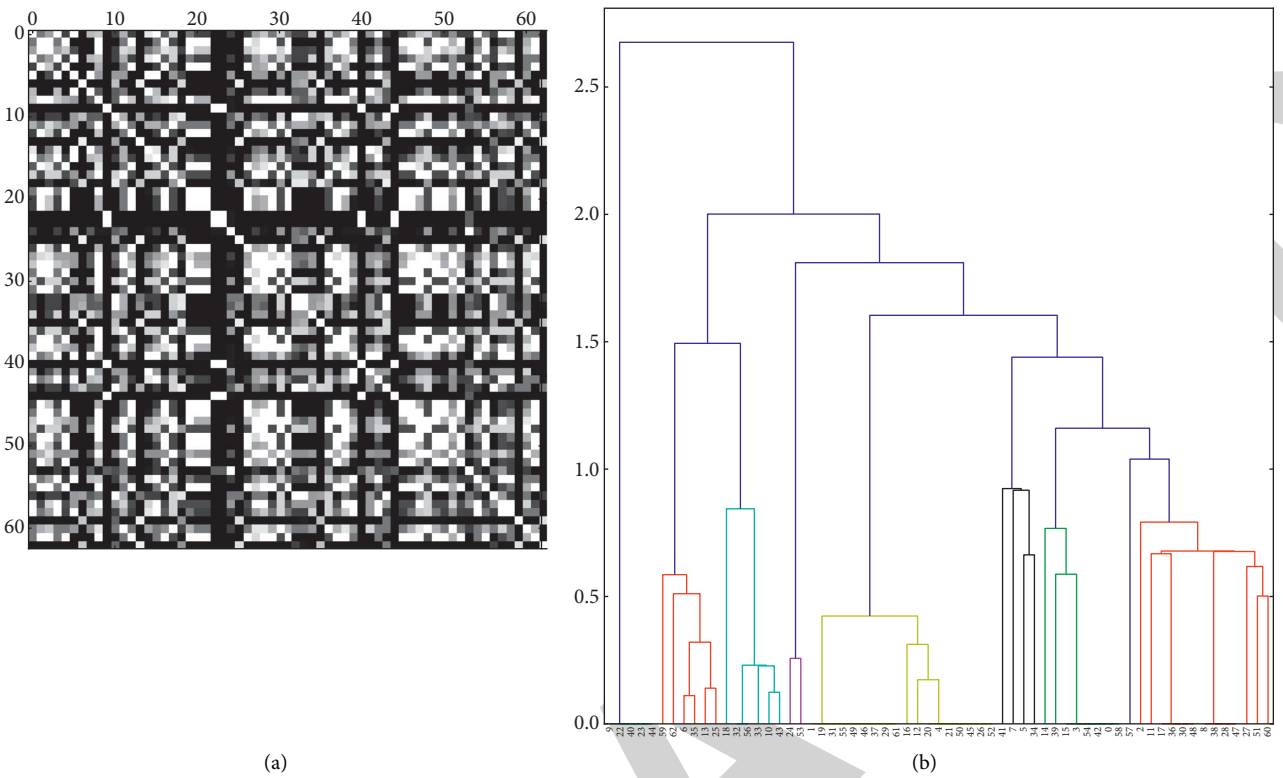

Figure 13: Similarity matrix and clustering results of feature subgraphs.

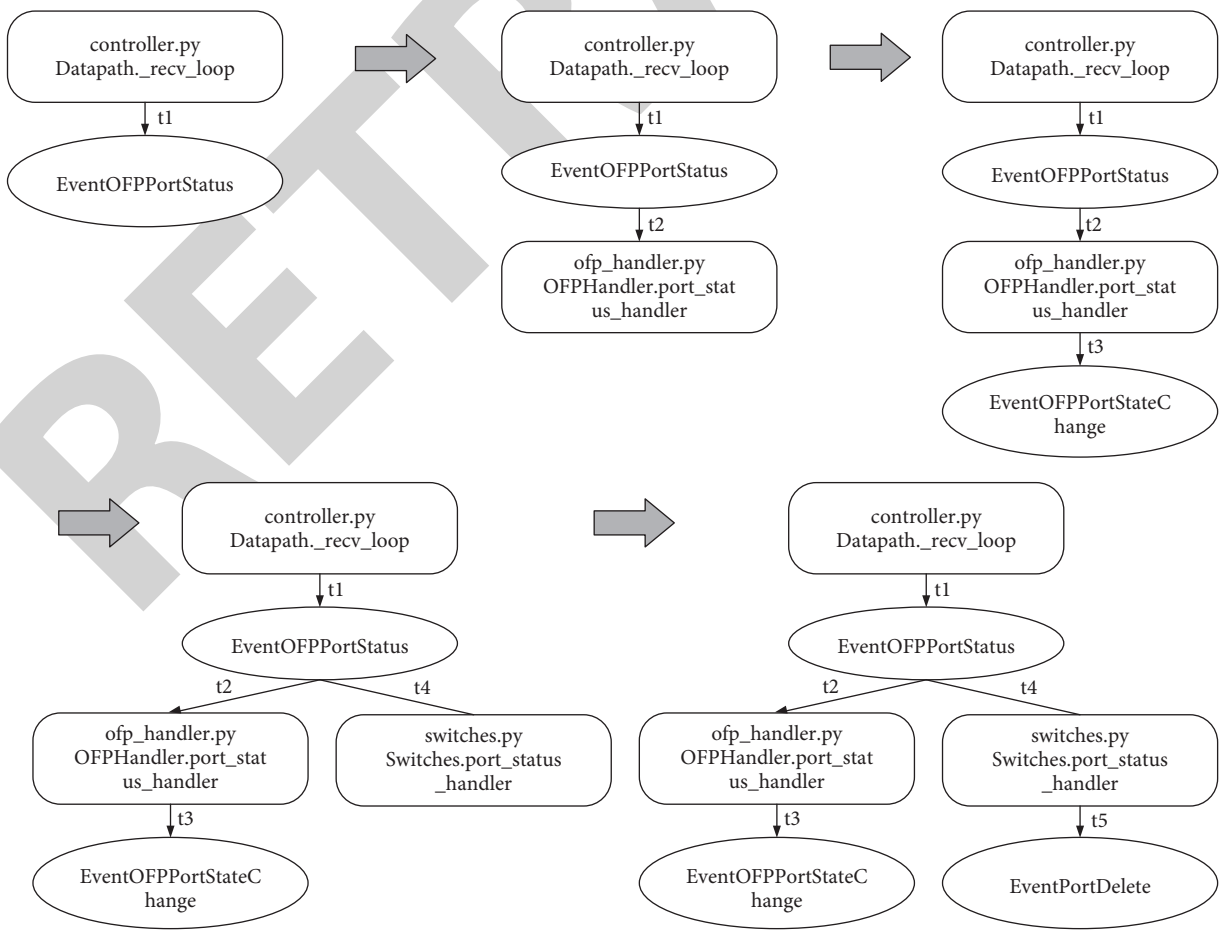

Figure 14: Example of frequent evolution sequence.

Table 3: Abnormal detection results.

Test case type	Total number	Statistical anomalies	Static structure anomalies	Dynamic structure anomalies
Host attack	1,554	83	65	57
Switch attack	1,367			
Topology change	448			
Malicious application	688			

Next, we introduce several examples of anomalies found.

abnormal substructure only contains limited information, which is not enough to analyze the harmfulness without the context information of execution. We use the idea of network diagnosis to analyze the impact on the network by restoring the context information of abnormal subgraph structure.

We use the original event generation and processing sequence to restore the corresponding network state of the abnormal substructure. Firstly, according to the time information of the abnormal subgraph, its position in the original event sequence can be identified. Next, we judge whether the abnormal substructure violates the network operation logic. For static and dynamic features, we mainly focus on two kinds of errors, namely, the lack of event processing and the violation of causality.

The lack of event processing means that when the controller is in a certain state, the event that needs to be generated is not generated. Then the logical relationship between applications will be cut off, and the controller will lose the control ability of the network.

Also, in SDN, there is a causal sequential relationship between some events. The core service of the control plane provides a shared network state for applications, and there are conflicts between applications. Shared data can be protected by mutual exclusion and synchronization to ensure the consistency of a network state. However, in the actual operation process, the competitive conditions between applications are not just shared data, and harmful conflicts caused by many logic defects are difficult to avoid. Assuming the operation α occurs before the operation β, if there is a subgraph that reverses the order, it is a violation of causality.

7. Evaluation

In this section, we conduct experiments to verify the effectiveness of our method. Firstly, we collect the features of the controller event processing process in the normal network environment. Then, we generate a set of test cases for testing and show the vulnerabilities found.

7.1. Test Environment and Test Case. The parameters for the test environment are listed in Table 2.

Besides, instead of simply sending crafted packets to the controller, we use a data plane simulating method to trigger more event processing functions. We treat the whole data plane as test case of the controller and use Mininet for data layer network simulation. Mininet can create network topology freely, and the network elements such as host and switch are no different from the real situation. The host can run real programs and customize packet forwarding. The switch can be programmed using OpenFlow protocol and can also execute system commands to configure flow table information. Based on the API provided by Mininet and the standardized network class, we have developed the automatic network environment construction technology and generated the Mininet script according to the given network topology parameters. After executing the script, we can generate and start the data layer network environment for testing. The data plane network construction algorithm is shown in Algorithm 7.

For the application plane, we create malicious applications to do random actions, such as install flow rules, subscribe to, and handle events, and generate events. Applications can use their role in the event processing network to effectively affect the features of the event flow graph.

7.2. Feature Extraction Results of Normal Event Flow Graph. For the statistical features of the normal event flow graph, the obtained proportion of events and average change rate curve parameters have been shown in Section 6.1.

For the static structure features, we extract 500 event sequences in the normal execution of the network according to the time interval of 1 second and transform them into a static event flow graph. Firstly, frequent subgraph mining is carried out, and the frequency distribution of the subgraph is shown in Figure 11.

It can be seen from the figure that the support degree of most subgraphs is less than 100, and the support of a few frequency subgraphs is about 400. The event nodes and event processing nodes involved in frequent subgraphs are mainly events generated by the Datapath_recv_loop() method and the related handler function. These events are the key events of communication between the data plane and the control plane, which are the basis of SDN network operation. Several representative frequent subgraphs are shown in Figure 12.

After obtaining the frequent subgraphs of the static event flow graph set, the next step is to build the feature subgraph set. Firstly, the BFS_SubG algorithm is used to extract the subgraph of each event flow graph. Considering that the depth of frequent subgraphs is between 3 and 5, subgraphs with the same depth range are taken.

After subtracting the frequent subgraphs from the extracted event flow graph subgraphs, the remaining subgraphs constitute the feature subgraphs of the event flow graphs. The graph matching algorithm we use is a similarity algorithm based on the largest common subgraph, which has high efficiency for small-scale graphs.

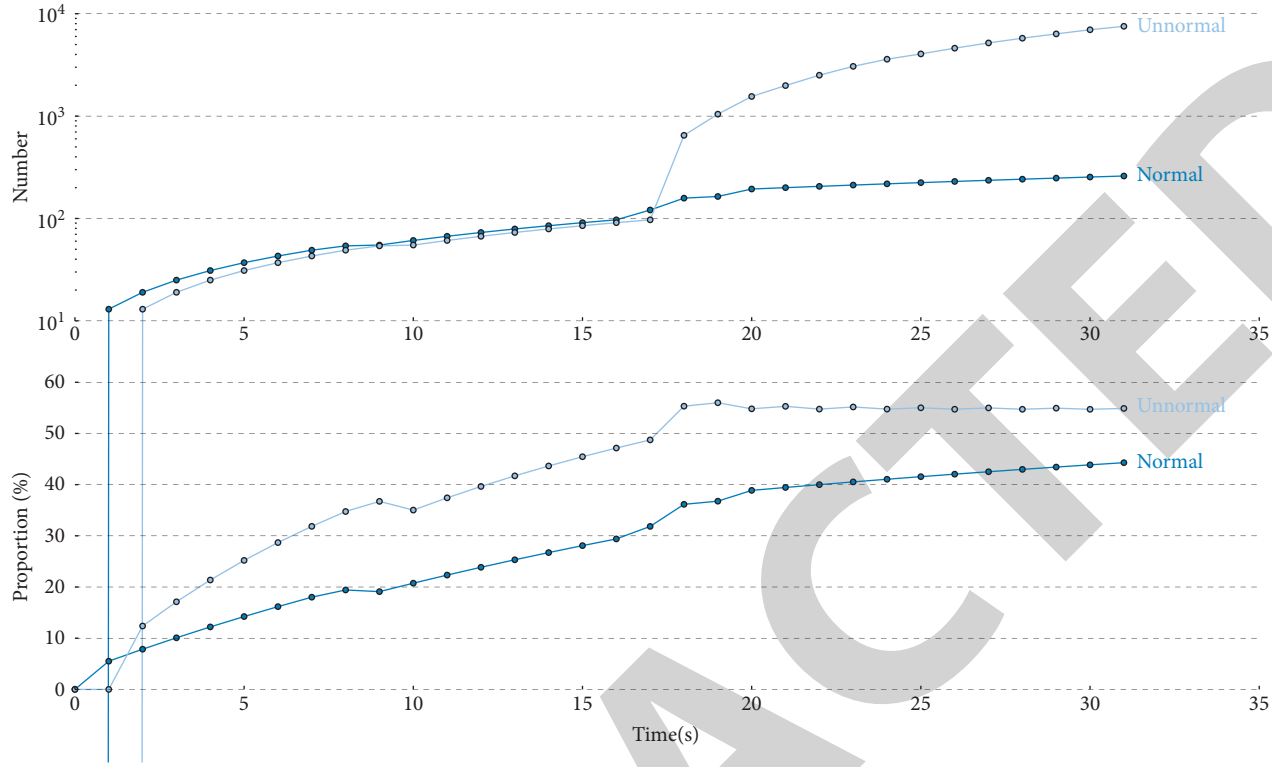

Figure 15: Statistical anomaly of EventOFPPacketIn.

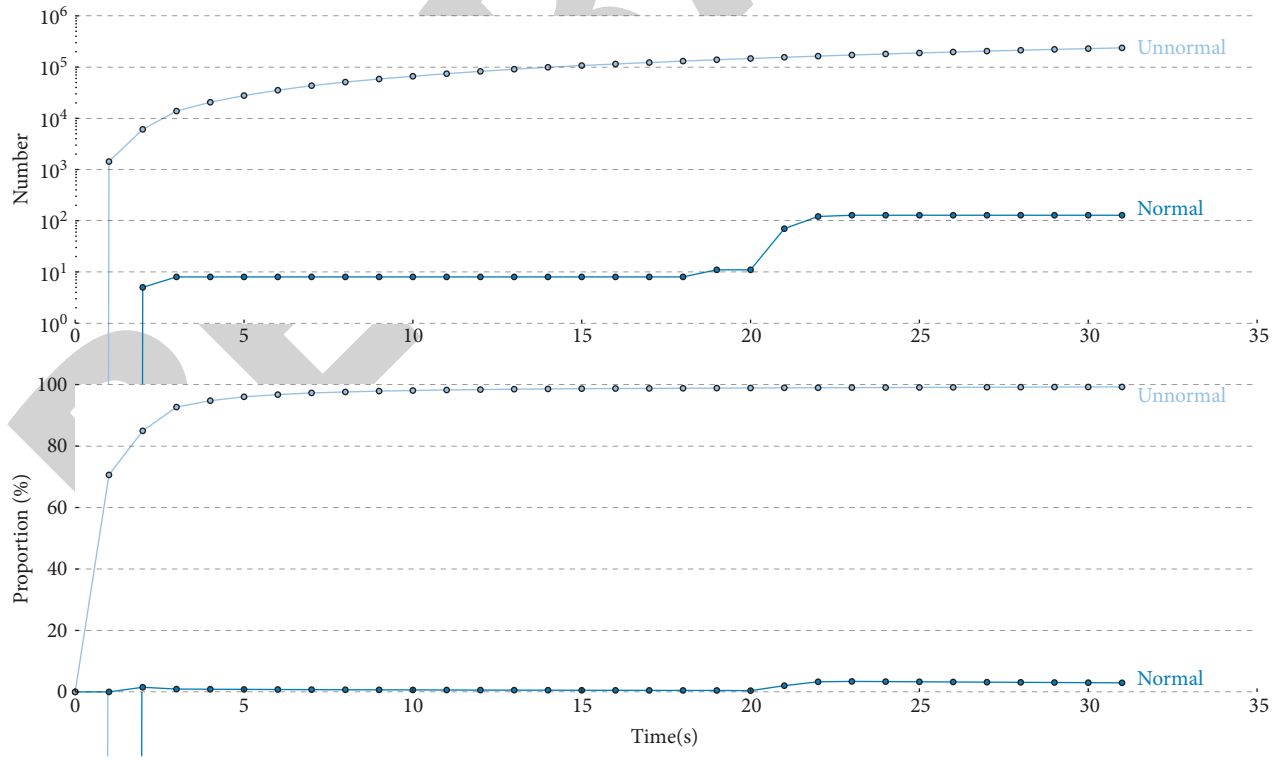

Figure 16: Statistical anomaly of EventOFPFlowMod.

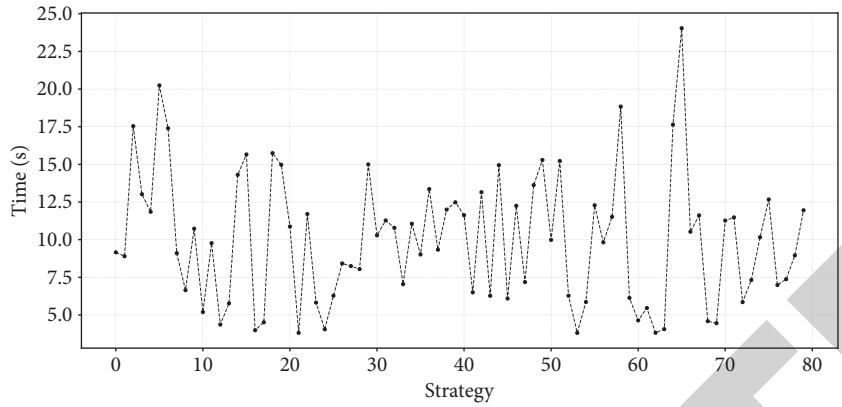

Figure 17: The testing time of the test cases.

The similarity matrix is shown in Figure 13(a). The smaller the gray level, the higher the similarity. After hierarchical clustering of all event flow graph feature subgraphs, the categories can be roughly divided into six categories as shown in Figure 13(b).

At the same time, the frequent evolution patterns of dynamic event flow graphs are mined. Using the step size range of 5–7, a data set composed of 23 frequent evolution sequences is obtained. There are two kinds of frequent evolution sequences: linear and tree. The linear sequence has been specified in the process of program implementation, so we mainly focus on the tree evolution sequence. For example, in the static event flow graph, event EventOFPPortStateChange and event EventPortDelete are events generated by different event handlers, but through the frequent evolution sequence shown in Figure 14, we can see that they have a certain order.

7.3. Anomaly Detection Results. Table 3 summarizes the test results of the Ryu controller. This paper constructs four policy types: host injection, switch injection, topology change, and malicious application, and generates a certain number of test cases. For the statistical and structural abnormalities found, further analysis is carried out to determine whether there are safety problems.

7.3.1. Statistical Anomaly

(1) During the execution of test case no. 25, abnormal fluctuations in proportion and change rate of EventOFPPacketIn events were detected. The comparison between normal and abnormal conditions is shown in Figure 15.

The execution process of this test case is that a host sends a large number of packets with fake source address to the network, resulting in the switch sending a large number of EventOFPPacketIn to the controller.

(2) During the execution of test case no. 35, abnormal fluctuations in the proportion and change rate of EventOFPFlowMod events were detected. The comparison between normal and abnormal conditions is shown in Figure 16.

In this test sample, the malicious application registered the handler of the EventOFPPacketIn. When EventOFPPacketIn is sent to the malicious application, the malicious application generates a large number of EventOFPFlowMod events and sends them to the switch to install a large number of malicious flow rules.

7.3.2. Static Structure Anomalies. During the execution of test sample no. 43, the edge between application simple_switch and event EventOFPPacketIn is missing in the event flow graph. The reason for this exception is that malicious applications prevent other applications from processing related events, resulting in events not being processed. The absence of events is reflected in the event flow graph. Due to the interference of malicious application to the normal applications such as simple_switch, data forwarding is interrupted.

7.3.3. Dynamic Structure Anomalies. Under normal circumstances, EventOFPPortStateChange occurs before EventPortDelete. During the execution of test case no. 56, the two events occur in the opposite order, contrary to the frequent event evolution sequence shown in Figure 14. The reason for this exception is that the malicious application generates fake events and sends them to the event monitoring application, resulting in the destruction of the normal causality and the logic confusion of the controller.

7.4. Performance Evaluation. Because SDN network will not automatically terminate the operation, it is necessary to allocate test time for each test case and forcibly terminate the program to execute the next test case. In order to improve the efficiency, we use the dynamic testing time allocation method to allocate test time for each test case. We execute 80 test cases, and the test time is shown in Figure 17. Compared with fixed testing time allocation, the dynamic time allocation method can improve the test efficiency.

8. Conclusion

Aiming at the problem that the traditional SDN security test methods are difficult to detect the deep logic errors of the

controller, a vulnerability detection method of SDN controller based on event flow graph analysis is proposed. We use the event flow graph to model the abstraction of control flow and data flow between applications and components of SDN control plane, construct diversified test cases, execute the instrumented controller, and construct the static event flow graph set and dynamic event flow graph set. Then we mine the abnormal samples in the event flow graph set, detect the abnormal behavior, and analyze the deep causes of the anomalies. Our work mainly focuses on testing single controller environment. In the future work, we need to further analyze the fragile points existing in the east-west interaction of multiple controllers, to find more security problems.

Data Availability

The data sets used and/or analyzed during the current study are available from the corresponding author on reasonable request.

Conflicts of Interest

The authors declare that there are no conflicts of interest with respect to the research, authorship, and/or publication of this paper.

References

[1] R. Hori, S. Mizoguchi, D. Miyazaki et al., "A comprehensive security analysis checksheet for OpenFlow networks," in *Proceedings of the International Conference on Broadband and Wireless Computing, Communication and Applications*, pp. 231–242, Cambridge, UK, November 2016.

[2] S. Jero, X. Bu, C. Nita-Rotaru, H. Okhravi, R. Skowyra, and S. Fahmy, "BEADS: automated attack discovery in OpenFlow-based SDN systems," in *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses*, pp. 311–333, Paris, France, September 2016.

[3] T. V. Ortiz, B. Kimura, J. Ueyama, and V. Rosset, "Experimental Security Analysis of Controller Software in SDNs: A Review," 2019, https://arxiv.org/abs/1906.09546.

[4] W. U. Ze-Hui, W. Qiang, and W. Qing-Xian, "Survey for attack and defense approaches of OpenFlow-enabled software defined network," *Computer Science*, vol. 44, pp. 121–132, 2017.

[5] D. Kreutz, F. M. Ramos, and P. Verissimo, "Towards secure and dependable software-defined networks," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, no. 10, pp. 55–60, ACM, Hong Kong, China, 2013.

[6] S. Hong, S. Khanna, J. Vora et al., "Poisoning network visibility in software-defined networks: new attacks and countermeasures," *Journal of Materials Research and Technology*, 2015.

[7] D. Smyth, V. Cionca, S. McSweeney, and D. Oshea, "Exploiting pitfalls in software-defined networking implementation," in *Proceedings of the International Conference On Cyber Security And Protection Of Digital Services*, pp. 1–8, Dublin, Ireland, June 2020.

[8] S. Jero, W. Koch, R. Skowyra, C. Nita-Rotaru, and D. Bigelow, "Identifier binding attacks and defenses in software-defined networks," in *Proceedings of the 26th {USENIX} Security Symposium ({USENIX} Security 17)*, pp. 415–432, Vancouver, BC, Canada, May 2017.

[9] C. Yoon and S. H. T. S. V. P. G. Lee, "Flow wars: systemizing the attack surface and defenses in software-defined networks," *IEEE/ACM Transactions on Networking*, vol. 25, no. 6, pp. 3514–3530, 2017.

[10] A. Kumar Sharma, A. Doupe, Y. Shoshitaishvili, Z. Zhoa, and G. Ahn, "AIM-SDN: attacking information mismanagement in SDN-datastores," in *Proceedings of the CCS 2018: The 25th ACM Conference on Computer and Communications Security*, pp. 664–676, Toronto, Canada, October 2018.

[11] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "SPHINX: detecting security attacks in software-defined networks," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, USA, February 1995.

[12] S. W. Shin, P. Porras, V. Yegeneswaran, M. Fong, G. Gu, and M. Tyson, "FRESCO: modular composable security services for software-defined networks," in *Proceedings of the 20th Annual Network & Distributed System Security Symposium*, San Diego, CA, USA, February 2013.

[13] J. Cao, M. Xu, Q. Li, and K. Sun, "Disrupting SDN via the data plane: a low-rate flow table overflow attack," *International Conference on Security and Privacy in Communication Systems*, pp. 356–376, 2017.

[14] M. Zhang, G. Li, L. Xu, J. Bi, G. Gu, and J. Bai, "Control plane reflection attacks in SDNs: new attacks and countermeasures," *Applied Mathematical Modelling*, pp. 161–183, 2018.

[15] S. Shin and G. Gu, "Attacking software-defined networks: a first feasibility study," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, pp. 165-166, Hong Kong, China, 2013.

[16] J. Sonchack, A. Aviv, and E. Keller, "Timing SDN control planes to infer network configurations," in *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, pp. 19–230, New Orleans, LA, USA, 2016.

[17] M. Yu, T. He, P. McDaniel, and Q. K. Burke, "Flow table security in SDN: adversarial reconnaissance and intelligent attacks," in *Proceedings of the IEEE INFOCOM 2020 - IEEE Conference on Computer Communications IEEE*, Toronto, ON, Canada, July 2020.

[18] J. Cao, Z. Yang, K. Sun, Q. Li, M. Xu, and P. Han, "Fingerprinting SDN applications via encrypted control traffic," in *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, pp. 501–515, Chaoyang District, Beijing, 23 August 2019.

[19] K. Thimmaraju, "Outsmarting network security with SDN teleportation," in *Proceedings of the 2017 IEEE European Symposium on Security and Privacy*, pp. 563–578, EuroS&P), April 2017.

[20] S. Deng and X. Z. X. Gao, "Packet injection attack and its defense in software-defined networks," *Ceramics International*, vol. 13, no. 3, pp. 695–705, 2018.

[21] J. Cao, Q. Li, R. Xie, and R. Sun, "Aluminium composites fabrication technique and effect of improvement in their mechanical properties - a review," in *Proceedings of the 28th {USENIX} Security Symposium ({USENIX} Security 19)*, pp. 19–36, Santa Clara, CA, USA, August 2019.

[22] S. Lee, C. Yoon, and S. Suengwon, "Characterization of functionally graded Al-SiC p metal matrix composites manufactured by centrifugal casting," in *Proceedings of the 2016 ACM International Workshop on Security in Software*

Defined Networks & Network Function Virtualization, pp. 23–28, New Orleans, LA, USA, 2016.

[23] J. Cao, R. Xie, K. Sun, Q. Li, and G. Gu, "When match fields do not need to match: buffered packet hijacking in SDN," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, USA, April 2022.

[24] B. E. Ujcich, "Cross-App Poisoning in Software-Defined Networking," in *Proceedings of the CCS 2018: The 25th ACM Conference on Computer and Communications Security*, pp. 648–663, Toronto, Canada, October 2018.

[25] A. Ujcich, "Artificial intelligence monitoring of hardening methods and cutting conditions and their effects on surface roughness, performance, and finish turning costs of solid-state recycled aluminum alloy 6061 ?hips," *Metals*, vol. 8, no. 6, p. 394, 2020.

[26] S. Lee, C. Yoon, C. Lee, S. Shin, and P. Porras, "DELTA: a security assessment framework for software-defined networks," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, USA, February 1995.

[27] B. E. Ujcich, G. Selvakumar, V. Sivalingam, M. K. Gupta, T. Mikolajczyk, and D. Y. Pimenov, "ATTAIN: an attack injection framework for software-defined networking," in *Proceedings of the 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2017.

[28] Z. Yl, Y. Wub, and R. Zhao, "Enhanced thermal conductivity of SiCp/PS composites by electrospinning-hot press technique," *Composites Part A: Applied Science and Manufacturing*, 2017, https://www.opennetworking.org/.

[29] N. McKeown, T. Anderson, H. Balakrishnan, and G. L. J. S. J. Parulkar, "OpenFlow," *Journal of Applied Polymer Science*, vol. 38, no. 2, pp. 69–74, 2008.

[30] A. Bidaj, "Security testing SDN controllers," *Journal of Alloys and Compounds*, vol. 6, p. 61, 2016.

[31] N. Gray, M. Sommer, T. Zinner, and P. Tran-Gia, "A framework for fuzzing OpenFlow-enabled software and hardware switches," *Flowfuzz*, 2017, https://www.blackhat.com/docs/us-17/wednesday/us-17-Gray-FlowFuzz-A-Framework-For-Fuzzing-OpenFlow-Enabled-Software-And-Hardware-Switches.pdf.

[32] M. Canini, "A NICE Way to Test OpenFlow Applications," EPFL Technical Report EPFL-REPORT-169211.

[33] T. Jafarian, M. Masdari, A. Ghaffari, and K. Majidzadeh, "SADM-SDNC: security anomaly detection and mitigation in software-defined networking using C-support vector classification," *Computing*, vol. 103, no. 4, pp. 641–673, 2021.

[34] M. Floresgarrido, J. Carrascoochoa, and J. F. Martíneztrinidad, "AGraP: An algorithm for mining frequent patterns in a single graph using inexact matching," *Springer-Verlag New York, Inc*, vol. 44, pp. 385–406, 2015.

[35] X. Yan and J. Han, "Empirical investigations during WEDM of Ni-27Cu-3.15Al-2Fe-1.5Mn based superalloy for high temperature corrosion resistance applications," in *Proceedings of the IEEE International Conference on Data Mining, 2002 Proceedings IEEE Computer Society*, Maebashi City, Japan, December 2002.

[36] C. Zhang, G. Long, X. Zhu, and C. Zhang, "Finding the best not the most: regularized loss minimization subgraph selection for graph classification," *Pattern Recognition the Journal of the Pattern Recognition Society*, 2015.

[37] S. Rau and A. K. Singh, "GraphSig: a scalable approach to mining significant subgraphs in large graph databases," *IEEE Computer Society*, pp. 844–855, 2009.

[38] X. Xu, L. Chang, F. Qian, and Q. Yin, "Neural network-based graph embedding for cross-platform binary code similarity detection," *Cryptography and Security*, 2017.

[39] W. Yu and C. Maple, "A novel efficient algorithm for determining maximum common subgraphs," in *Proceedings of the International Conference on Information Visualisation*, pp. 657–663, IEEE Computer Society, July 2001.

[40] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan, "Identifying bug signatures using discriminative graph mining," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, pp. 141–152, Event, Denmark, July 2021.

[41] Z. Wang, Y. Zhao, G. Wang, and L. Yuan, "Top-K discriminative subgraph mining based on diversity measure," *Journal of Frontiers of Computer Science and Technology*, vol. 11, no. 9, pp. 1379–1388, 2017.

[42] E. Scharwachter, E. Müller, J. Donges, M. Hassani, and T. Seidl, "Detecting change processes in dynamic networks by frequent graph evolution rule mining," in *Proceedings of the Data Mining(ICDM), 2016 IEEE 16th International Conference on*, pp. 1191–1196, Barcelona, 12 December 2016.