ProvGuard: Detecting SDN Control Policy Manipulation via Contextual Semantics of Provenance Graphs

Ziwen Liu*, Jian Mao*‡§✉, Jun Zeng†, Jiawei Li*†, Qixiao Lin*,
Jiahao Liu†, Jianwei Zhuge¶‖✉ and Zhenkai Liang†
* Beihang University, {liuziwen, maojian, daweix, linqx529}@buaa.edu.cn
† National University of Singapore, {junzeng, jiahao99, liangzk}@comp.nus.edu.sg
‡ Tianmushan Laboratory § Hangzhou Innovation Institute, Beihang University
¶ Tsinghua University, zhugejw@tsinghua.edu.cn ‖ Zhongguancun Laboratory

Abstract*—Software-Defined Networking (SDN) improves network flexibility by decoupling control functions (control plane) from forwarding devices (data plane). However, the logically centralized control plane is vulnerable to Control Policy Manipulation (CPM), which introduces incorrect policies by manipulating the controller's network view. Current methods for anomaly detection and configuration verification have limitations in detecting CPM attacks because they focus solely on the data plane. Certain covert CPM attacks are indistinguishable from normal behavior without analyzing the causality of the controller's decisions. In this paper, we propose PROVGUARD, a provenance graph-based detection framework that identifies CPM attacks by monitoring controller activities. PROVGUARD leverages static analysis to identify data-plane-related controller operations and guide controller instrumentation, constructing a provenance graph from captured control plane activities. PROVGUARD reduces redundancies and extracts paths in the provenance graph as contexts to capture concise and long-term features. Suspicious behaviors are flagged by identifying paths that cause prediction errors beyond the normal range, based on a sequence-to-sequence prediction model. We implemented a prototype of PROVGUARD on the Floodlight controller. Our approach successfully identified all four typical CPM attacks that previous methods could not fully address and provided valuable insights for investigating attack behaviors.

I. Introduction

Software-Defined Networking (SDN) separates network control from individual forwarding devices into a logically centralized control plane [9]. The control plane (CP), consisting of one or more controllers, manages network devices and provides global network state information to applications. An SDN controller determines and issues *control policies* that direct the actions of forwarding devices in the data plane (DP), making it essential to the network's functionality and security. Consequently, the controller becomes a primary target for attacks aiming to disrupt the network. *Control Policy Manipulation* (CPM) is a type of SDN attack that alters or deactivates forwarding rules and security policies by exploiting vulnerabilities in the controller's processing logic [11], [31]. Since controllers maintain network state knowledge by extracting information from DP messages, compromised DP devices can influence control decisions by injecting malicious payloads or subtly changing their states. Malicious manipulation of the controller's cognition and decision-making threatens network correctness and security, enabling various attacks, such as traffic eavesdropping, man-in-the-middle attacks [11], network device hijacking [13], bypassing access control [28], and black hole routing [2], [11], [12].

Prior efforts to mitigate CPM attacks include anomaly detection [8], [11], [23], [24] and network configuration verification [1], [14], [22], [25]. Anomaly detection methods defend against policy manipulation by identifying illegal network state changes according to predefined rules, including attack-specific rules [11], [24] and generic invariant rules (*e.g.,* valid network identifier bindings and bidirectionality of links) [8]. Network configuration verification methods detect control decision errors and conflicts by verifying network-wide invariants (*e.g.,* path reachability, loop-free routing) and the consistency of forwarding rules and security policies.

However, these rule-based anomaly detection methods and network configuration verification tools have limitations in detecting CPM attacks. Specifically, they rely solely on metadata extracted from control-data plane messages to characterize and verify network states, making it difficult to distinguish covert CPM attacks from normal behavior. Malicious devices can exploit controller logic defects to manipulate policies through seemingly legitimate state changes, such as bypassing access control rules by orchestrating compliant host migrations [28]. Furthermore, rule-based methods require prior knowledge of attack behaviors, limiting their ability to detect manipulation attacks that exploit unknown control logic vulnerabilities. Additionally, adversaries can deceive controllers into issuing incorrect or malicious instructions without causing policy-level conflicts. For instance, link fabrication [11] can poison the

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.241244
www.ndss-symposium.org

controller's view of network topology, facilitating man-in-the-middle attacks. Traditional network configuration verification methods fail to detect such attacks since they do not produce policy conflicts or violate network-wide invariants.

Key observation. Malicious data-plane devices manipulate control decisions by fabricating or orchestrating the messages they send to the controller. The controller's operations for processing these data-plane packets implicitly contain contextual semantics that influence control decisions. For example, the controller generates forwarding rules based on host IP addresses, which are influenced by IP change events triggered by messages from the data plane. The contexts in which IP change events are handled by controllers reflect how data-plane devices affect IP address knowledge and control decisions. Therefore, such implicit contextual information derived from the handling of data-plane events in the controller allows for a deeper understanding of how control decisions are influenced and potentially compromised. Analyzing the contextual semantics from control plane logs (especially controller activities) can serve as a basis for detecting anomalous changes in control policies caused by CPM attacks.

Our approach. In this paper, we propose a detection framework, **PROVGUARD**, designed to identify CPM attacks originating from the data plane using the *provenance graph* of controller activities. Our approach models controller handling operations through static program analysis, instruments the controller, and collects network activities related to data-plane message processing at runtime. We represent execution traces of network behaviors as a provenance graph and leverage the causality within this graph to extract contexts related to packet processing and control policy generation. Based on the semantics of these causal contexts, our approach detects CPM attacks by identifying deviant contexts without requiring prior knowledge of specific attack patterns.

There are two main challenges in extracting behavioral contexts from the provenance graph of the SDN control plane: (a) the graph contains the entire history of controller activity, making it difficult to capture and correlate critical contexts of long-term behaviors, and (b) the collected controller operations (*i.e.,* control-plane logs) contain significant redundant information that can obscure the semantics of attack-related operations. To accurately characterize the influence of external inputs on control policies and filter out unrelated behaviors, we identify controller operations triggered by DP packets through static program analysis and collect relevant controller activities. We then quantify the importance of edges in the provenance graph during log processing to further reduce redundancy. Instead of directly using graph structures, we extract paths as behavioral contexts to capture relationships between operations associated with long-term behaviors and detect anomalies by identifying deviant contexts that cause larger prediction errors. Additionally, our method provides relevant execution traces of suspicious contexts to aid in anomaly investigation.

We evaluate the effectiveness of **PROVGUARD** in identifying four typical CPM attacks in a simulated network managed by a Floodlight controller. Experimental results demonstrate that it successfully recognizes deviant contexts for all attack cases and provides related execution traces to aid manual investigation without requiring expert knowledge. By locating suspicious contexts in the provenance graph, our method reduces the manual audit workload to 6.02% of edges. In terms of overhead, controller instrumentation increases the average round-trip time by 1.8% to 24% across networks of varying diameters, with a storage overhead of 1.3 GB/hr for logs.

Contribution. In summary, we make the following contributions in this paper.

- We design a context-aware, graph-based anomaly detection framework for SDN, which transforms controller activity logs into a provenance graph and identifies anomalies based on contextual deviations.
- We propose a path extraction method to capture the context of long-term CPM attacks. To extract concise contexts, we leverage static program analysis to identify relevant controller operations and introduce a redundancy reduction method that filters out noise based on operation importance.
- We prototype and evaluate our approach. Experimental results demonstrate that our method outperforms existing techniques in detecting control policy manipulation attacks without requiring expert knowledge. Furthermore, it facilitates anomaly investigation by reducing manual workload and highlighting attack-related controller activities.

Paper Organization. The rest of this paper is organized as follows. Section II discusses the background and motivation of our work. Section III describes the threat model, the data provenance graph model introduced in our solution, and the overview of our approach. Section IV presents our CPM detecting approach, **PROVGUARD**. We implement and evaluate our approach in Section VI. Section VII discusses the related work, and Section VIII concludes the paper.

II. BACKGROUND AND MOTIVATION

A. Control Policy Generation

The SDN controller manages data-plane resources through communication channels based on the OpenFlow protocol. Each forwarding device maintains a flow table that dictates actions on incoming packets according to flow rules (*i.e.,* flow entries). If there is no matching rule, the packet will be encapsulated in a `PACKET_IN` message and sent to the controller. By parsing this message, the controller obtains information such as switch port, network identifier, and source or destination IP address to calculate control policies. Forwarding decisions are sent back to data-plane devices via `PACKET_OUT` or `FLOW_MOD` messages to instruct how to process and forward the packets.

In addition, some controller services actively probe the network state, and their knowledge of network state also influences control decisions. For instance, the controller acquires knowledge of links between forwarding devices through the link discovery service. Specifically, the controller periodically sends *Link Layer Discovery Protocol* (LLDP) packets to switches. When a switch receives an LLDP packet, it broadcasts the packet into the network. The next-hop switch then reports this LLDP packet to the controller. In this way, the controller can determine connections between switches and calculate forwarding rules based on the network topology.

B. Control Policy Manipulation

Since control plane (CP) decisions impact the correctness and security of the entire network, CPM attacks have received increasing attention in SDN security research [11], [15], [31]. Adversaries can exploit communication channels between network entities and the CP to inject fake or fabricated messages, altering the controller's network view and leading to incorrect control decisions. In this section, we introduce two typical CPM attacks - *access control bypass via host migration* and *link fabrication* - and discuss the limitations of prior approaches.

1) Access Control Bypass via Host Migration: The Access Control List (ACL) application in the SDN controller calculates and issues access control rules to switches. However, due to a logical vulnerability in the ACL application of the Floodlight[1] controller, it does not update control policies when host locations are migrated. As a result, the intended access control rules may not be effectively enforced in the data plane. Exploiting this vulnerability, attackers can bypass ACL rules by migrating hosts. As illustrated in Fig. 1, an attacker controls two hosts (h1 and h3), with an ACL rule that blocks communication between 10.0.0.1 (h1) and 10.0.0.2 (h2). The attack proceeds as follows. Step ①: The attacker sends a packet from h1 to h3. Since switch s1 has no matching forwarding rule, the packet is encapsulated in a `PACKET_IN` message and sent to the controller. Step ②: The controller obtains h1's IP address and triggers an IP update event. The ACL application processes this event with the `IPV4Changed` handler. Step ③: The controller sends a `FLOW_MOD` message to s1, installing an ACL rule that denies traffic between h1 and h2. Step ④: The attacker migrates h1 to port p1 of switch s3, denoted as h1', which triggers a host migration event. However, since the ACL application's `deviceMoved` handler disregards the migration event, the access control policy in the DP remains not updated. Steps ⑤ and ⑥: When h1' sends packets to 10.0.0.2, the controller issues a forwarding rule invalidating the ACL rule.

Limitation of rule-based solutions. Existing rule-based solutions fail to identify this anomaly for various reasons. Detection methods designed for specific attacks [11], [13], [24] rely on expert knowledge, limiting their ability to detect unknown vulnerabilities. Typical rule-based detection frameworks [8], [23] collect and verify metadata of network flows to identify anomalies. In this case, rule-based checkers will modify existing invariants about the malicious host and update network identifier bindings (*e.g.,* MAC-IP-Switch-Port) for the migrating host based on potentially misleading events. Also, some detection approaches assume the controller is trusted

[1]http://www.projectfloodlight.org/

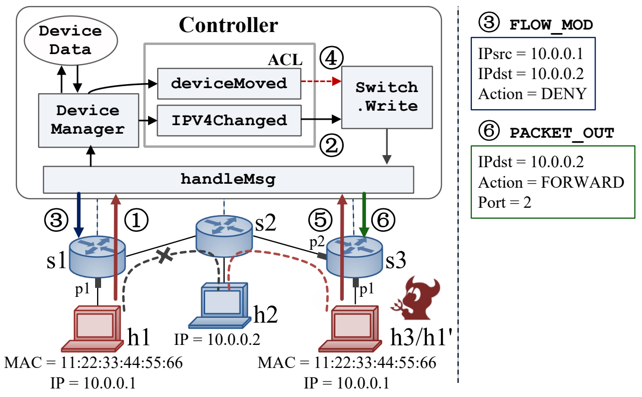


Fig. 1. Exploiting access control bypass vulnerability via host migration. The controller activities shown in this figure are a simplified illustration.

TABLE I
THE LEGEND IN FIG. 1 AND FIG.2.

DP	Description	CP	Description
——	network link	→	controller operation
- - - - - -	control-data plane communication	⇢	missed operation
→ → →	packet flow	▭	handler
- - - -	insecure network flow	⬭	data instance
--✕--	denied network flow	▭	event
		▱	attribute

(*e.g.,* Sphinx [8]), and will allow controller-generated flow rules (⑥), even if they violate the access control policy.

2) Link Fabrication: An adversary can deceive the controller into believing that a fabricated link exists between switches by modifying, forging, or relaying valid LLDP packets [2], [11]. As shown in Fig. 2, a malicious host h2 connects to switches s1 and s3. The attack proceeds as follows. Steps ① and ②: The controller receives a `PACKET_IN` message from h2 and updates its device data. Step ③: h2 relays LLDP packets between s1 and s3. Step ④: Each switch appears as the "next hop" of the other, which the controller interprets as a legitimate link.

Limitation of network verification-based approaches. Generally, network verification methods aim to detect misconfigurations and ensure flow rule consistency. However, maintaining flow policy consistency alone cannot prevent CPM attacks that do not introduce policy conflicts or violations. Link fabrication poisons the controller's network view and misleads it into generating conflict-free but insecure flow rules. Without analyzing controller activities, detection methods cannot identify incorrect network state knowledge.

C. Motivation

Existing anomaly detection and configuration verification methods struggle to identify CPM attacks because they primarily verify metadata in control-data plane messages based on predefined knowledge, overlooking the underlying *causality* of these attacks — specifically, how the controller's decisions

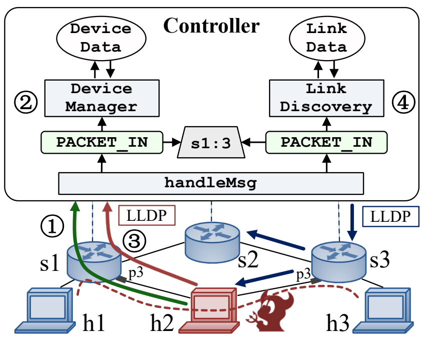


Fig. 2. An example of link fabrication attack. The controller activities shown in this figure are a simplified illustration.

are influenced by abnormal states. As shown in Fig. 1 and 2, controller operations provide direct insights into device behavior, network states, and their impact on control decisions. For instance, when a malicious host forges a link, two contradictory operations — host data update and link data update (as shown in Fig. 2) — signal suspicious activity, since a device cannot function as both a host and a switch simultaneously. As such, capturing the causality of controller operations is crucial for detecting CPM attacks.

To detect CPM attacks, particularly long-term attacks, the first step is to extract concise and effective contexts of control operations over an extended period. A straightforward approach is to use operation sequences segmented by finite time or event windows to capture contextual information. However, this strategy provides only a partial view of the attack and may overlook key operations that lead to the abnormal state, especially if the attack spans multiple windows. We observe that provenance graphs can describe the causal dependencies between various entities and operations, offering closely related contextual information that aids in anomaly detection. In system security, provenance graph-based behavior representation methods reconstruct system logs into a graph and extract semantics from substructures (*e.g.,* nodes, paths, or subgraphs) for behavior clustering and anomaly detection [10], [17], [32], [33]. Inspired by these methods, we propose using provenance graphs of SDN controller activities to identify CPM attacks. Specifically, we construct a provenance graph that captures controller activities related to network states and control decisions, allowing us to detect malicious behaviors by identifying deviant contexts within the graph. While incorporating provenance graphs into SDN security is promising, several challenges must be addressed to effectively extract and represent behavioral contexts. In the following section, we discuss these challenges and our goals for addressing them.

D. Challenges

In this section, we analyze the primary challenges that our work confronts. As a motivating example, we illustrate the provenance graph of an access control bypass attack in Fig. 3.

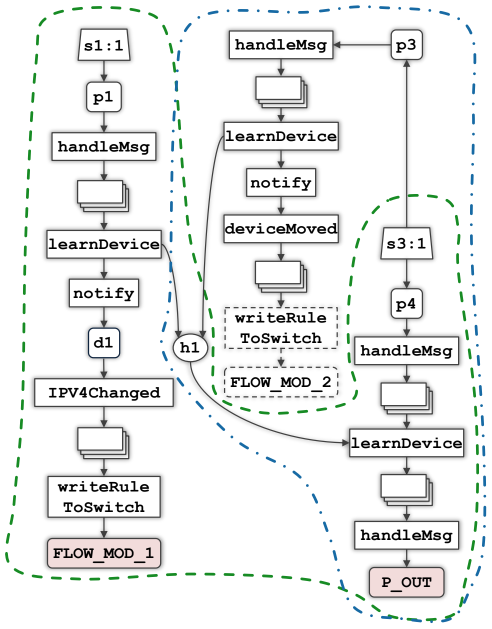


Fig. 3. Provenance graph of the access control bypass attack in Fig. 1. The corresponding relations between types of shapes and entities are trapezoid - attribute, rectangle - function, rounded rectangle - event, and oval - data instance. `p1` is the `PACKE_IN` message ①. `FLOW_MOD_1` is the message ③. `p3` is a `PACKE_IN` message triggering the host migration event ④. `p4` is the `PACKE_IN` message ⑤. `P_OUT` is the message ⑥.

Long-Term Feature Extraction. The provenance graph capturing the entire processing history can grow large and complex over time, making it challenging to identify CPM behaviors, especially long-term behavioral features. SDN controller provenance graphs exhibit two key characteristics: unclear behavioral boundaries and neighborhood similarity. The lack of clear boundaries between behaviors in controller activities complicates the correlation of different stages in a long-term attack. Consequently, querying the graph within a limited time window may fail to capture a complete picture of such attacks. For example, in the attack shown in Fig. 1, attackers may migrate host `h1` to a new port long after updating its IP address to evade detection. With a limited time window, analysts might only obtain the subgraph circled by the blue dot-dash line in Fig. 3, which appears to depict a normal host migration. Additionally, high similarity among neighboring nodes complicates the semantic representation of nodes using fixed-hop neighbor features, leading to confusion between different entities. For example, different instances of the same data object may become nearly indistinguishable due to highly similar adjacent contexts (functions that read or write them).

Moreover, initiating provenance graph searches from inappropriate starting points can lead to incomplete behavior extraction. For example, PicoSDN [27] finds the root causes of anomalies by tracing common ancestors from several initial search points (pieces of evidence). It can diagnose similar bypass attacks in the SDN controller ONOS[2] by backtracking

[2]https://wiki.onosproject.org/

TABLE II
NODES IN EXECUTION UNIT GRAPHS.

Node	Description
$function$	`ClassName.FunctionName`
$event$	`EventType.EventHash`
$data$	`ClassName.VarName.VarHash.Key`
$thread$	`ThreadID`
$attribute$	`IPAddress/MACAddress/SwitchPort`

TABLE III
EDGES IN EXECUTION UNIT GRAPHS.

Edge	Description
$call$	Caller ($function$) $\rightarrow$ Callee ($function$)
$dispatch$	Event dispatcher ($function$) $\rightarrow event$
$receiveBy$	$event \rightarrow$ Event listener ($function$)
$write$	$function \rightarrow data$
$readBy$	$data \rightarrow function$
$hasAttribute$	$event \rightarrow attribute$
exe	$thread \rightarrow function$

the operations associated with three messages: `FLOW_MOD_1`, which installs an ACL rule; `FLOW_MOD_2`, generated by the mobility application to delete the existing ACL rule; and `P_OUT`, which forwards a packet out. However, since there is no rule deletion mechanism in Floodlight's control logic, the absence of `FLOW_MOD_2` as an initial point prevents PicoSDN from discovering the `deviceMoved` event, leaving only the subgraph circled by the green dashed line in Fig. 3. For unhandled operations, analysts cannot define search starting points. This incomplete view of behavior resembles normal forwarding rule generation. In brief, we need to ***extract substructures that provide sufficient long-term behavioral contexts*** ($\mathbf{C_1}$).

Redundant Information. A large portion of routine operations in the controller's activity log can overshadow the semantics of attack-related operations. Existing diagnostic tools do not consider the relative importance of operations when describing causality, resulting in redundancy within search results. For example, the overlapping rectangles in Fig. 3 represent repeated operations in different processing pipelines. These repetitive operations contribute minimally to semantics and can even obscure distinctions between normal and abnormal behaviors. Therefore, ***redundancy reduction*** ($\mathbf{C_2}$) is essential.

III. OVERVIEW

This section outlines our assumptions about the attack scenario (III-A), introduces the relevant SDN controller knowledge essential for our design and our provenance model (III-B), and provides an overview of our approach (III-C).

A. Threat Model

In an SDN-based network, hosts, switches, and applications may disrupt the controller's forwarding or security policy calculation via malicious inputs. Among these, DP devices (hosts and switches) are the most challenging to supervise and authenticate, acting as the primary source of CPM attacks. We assume that DP devices may be compromised and can influence control decisions by injecting packets or changing their states. We also assume that most DP devices are benign and seldom trigger CP errors. Additionally, core services and applications in the CP are considered benign, though they may contain logical vulnerabilities. Administrative access to CP through command lines and Web APIs is assumed to be benign as well.

B. Data Provenance Graph

Our approach aims to infer the functional impact of data-plane messages on the SDN controller by examining the semantics of control-plane operations related to message processing. Several properties of the control plane are crucial for designing our solution:

- **Event-Driven Control.** Mainstream SDN controllers, such as Floodlight, ONOS, OpenDayLight[3], and POX[4], employ an *event-driven* architecture where concurrent modules (controller core services and applications) communicate through event dispatching and listening. For example, when a new host is detected, the host management module dispatches a host event (`DEVICE_ADDED`) to registered listeners (*e.g.,* handlers in ACL or firewall). Our provenance model characterizes event dispatching and listening to describe interactions between controller modules.
- **Execution Partitioning.** Some controller modules use event-handling loops to process messages, which can lead to incorrect associations of operations across loops executed by the same thread. *Execution partitioning* is an effective technique to mitigate this dependency explosion. By tracking the start of each event-handling loop, we can partition operations that process different messages, preventing unrelated transactions executed by the same thread from being incorrectly associated.

Through execution partitioning, collected logs are divided into *execution units*. We use the provenance graph model introduced in this section to describe relationships among data, entities, and operations within and across execution units. Assume we obtain N execution units by execution partitioning. A provenance graph $\mathcal{G}$ of SDN CP behavior comprises execution unit graphs and inter-unit edges:

$$\mathcal{G} = (\{\mathcal{U}_1, \mathcal{U}_2, ..., \mathcal{U}_N\}, \mathcal{E}_{inter}) = (\mathcal{V}, \mathcal{E}_{intra}, \mathcal{E}_{inter}) \quad (1)$$

$\mathcal{U}_i(i = 1, ..., N)$ denotes the subgraph of the i^{th} execution unit u_i. $\mathcal{E}_{inter}$ represents relations between nodes across execution units. The first equation represents that the provenance graph consists of execution unit subgraphs connected by inter-unit edges. $\mathcal{V}$ contains all nodes in $\mathcal{G}$. Intra-unit edges in $\mathcal{E}_{intra}$ represent actions/relations between nodes within each execution unit. The second equation indicates that the provenance

[3]https://docs.opendaylight.org/
[4]https://github.com/noxrepo/pox

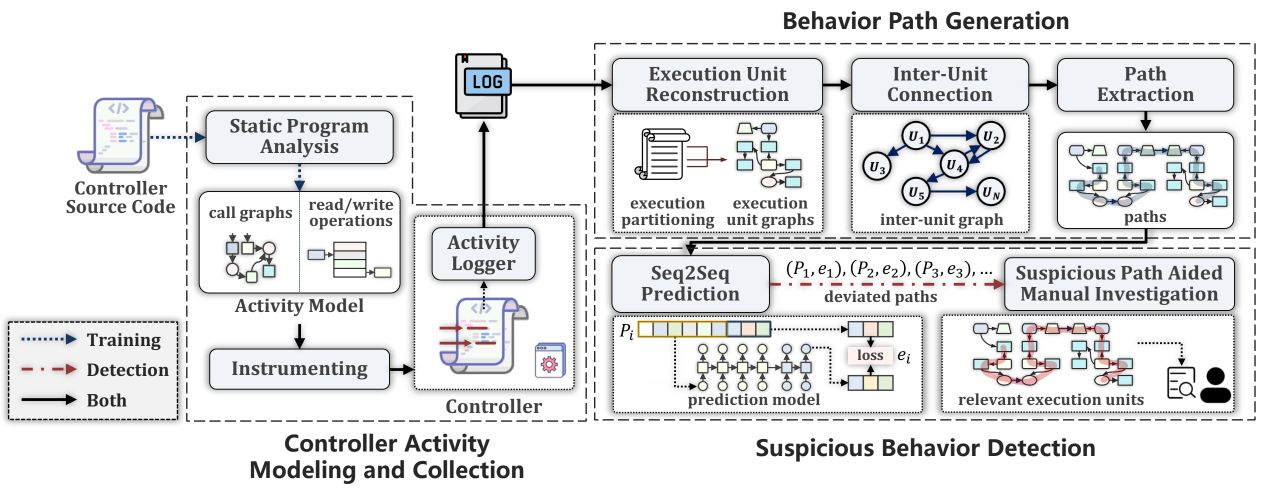


Fig. 4. The architecture of **ProvGuard**.

graph comprises the set of nodes and the sets of intra- and inter-unit edges.

Nodes in $\mathcal{V}$ fall into five types, summarized in Table II. (1) The $function$ nodes represent functions in the controller's source code, denoted by class and function name. (2) The $event$ nodes represent messages between controller modules, denoted by type and hash code. (3) The *data* nodes represent data instance variables maintained by controller modules to store network states. A *data* node is recorded by the class and variable name, along with its unique identifier (hash code). For data objects such as lists and mapping tables, the index or key value of read/write operations is also recorded for fine-grained description. (4) The *thread* nodes represent controller threads, identified by thread ID. To prevent unintended inter-unit associations, threads in different execution units are distinguished. (5) The *attribute* nodes represent event details, such as the MAC address, IP address, or attachment point associated with a DEVICE_ADDED event.

Intra-unit edges in $\mathcal{E}_{intra}$ are classified into seven types, detailed in Table III. (1) The *call* edges represent call relations, pointing from a caller to a callee. (2) The *dispatch* edges represent dispatching actions, pointing from a dispatcher to the dispatched event. (3) The $receiveBy$ edges represent listening actions, pointing from an event to its listener. (4) The $write$ edges represent data modification actions, pointing from a writing function to the modified data instance. (5) The $readBy$ edges represent reading actions, pointing from the accessed data instance to the reading function. (6) The $hasAttribute$ edges associate events with attributes. For example, PACKET_IN $\xrightarrow{hasAttribute}$ switch1:port1 specifies that a PACKET_IN message is received from port1 of switch1. (7) The exe edges represent thread execution actions, pointing from a thread to the function it executed. Inter-unit edges in $\mathcal{E}_{inter}$, denoted as $inter$, associate subgraphs to describe long-term behaviors. Section IV-B1 explains how we add $inter$ edges to the provenance graph. For clarity, a notation summary is available in Appendix A.

C. Approach Overview

The key idea behind **ProvGuard** is to analyze the causality of how incoming messages influence control decisions in the SDN control-plane provenance graph, enabling the detection of control policy manipulation attacks. **ProvGuard** extracts paths from the provenance graph as causally related contexts and identifies CPM attacks based on contextual semantics. As shown in Fig. 4, **ProvGuard** operates in three phases: controller activity modeling and collection, behavior path generation, and suspicious behavior detection.

Controller Activity Modeling & Collection. ProvGuard extracts the control logic and dataflow of the controller via static analysis, identifying activities related to DP message processing (*DP-related* activities) to build a controller activity model. According to the DP-related operations in the model, we instrument the controller and run the *Activity Logger* as a controller module to collect logs.

Behavior Path Generation. ProvGuard partitions the controller activity logs and reconstructs them into execution unit graphs interconnected by inter-unit edges through *Execution Unit Reconstruction* and *Inter-Unit Connection*. *Path Extraction* extracts behavior contexts by searching paths in the provenance graph.

Suspicious Behavior Detection. We apply a sequence-to-sequence (Seq2Seq) prediction model, trained on a dataset of normal behaviors, to measure contextual deviations and detect anomalies. In *Suspicious Path Aided Manual Investigation* phase, **ProvGuard** presents analysts with suspicious paths and illustrates relevant execution unit graphs for investigation.

IV. Design

A. Controller Activity Modeling & Collection

Our approach differs from previous forensic methods [29] by focusing on capturing the impact of data-plane messages on control policies to detect CPM attacks. To this end, we

Algorithm 1: DP-related Call Graph Extraction and Variable Operation Recognition

```
Input: Function set $\mathcal{F} = \{f_1, ..., f_k\}$, function call set
       $\mathcal{C} = \{(f_1, f_2, l_1), ..., (f_i, f_j, l_n)\}$, initial function
       $f_{init}$.
Output: DP-related function call set $\mathcal{C}_{out} \subseteq \mathcal{C}$ and variable
        operation set
        $\mathcal{O} = \{(f_1, v_1, o_1, l_1), ..., (f_p, v_q, o_r, l_s)\}$.
1  $worklist \leftarrow [f_{init}]$, $visited \leftarrow \{\}$, $\mathcal{F}_{sum} \leftarrow \{\}$;
2  $\mathcal{C}_{out} \leftarrow \{\}$, $\mathcal{O} \leftarrow \{\}$;
3  while $worklist \neq \emptyset$ do
4      $f \leftarrow getLast(worklist)$;
5      $doSkip \leftarrow False$;
6      for $(f, c, l) \in getCallees(f, \mathcal{C})$ do
7          if $c \notin F_{sum}.keys$ then
8              $doSkip \leftarrow True$;
9              $worklist$ append $c$;
10         end
11     end
12     continue if $doSkip$;
13     $\mathcal{C}_{out}$ appendAll $getCallees(f, \mathcal{C})$;
14     $dataFlow \leftarrow \{\}$;
15     for $s \in getStmts(f)$ do
16         $dataFlow \leftarrow$
             $updateDataFlow(s, dataFlow, \mathcal{F}_{sum})$;
17         $\mathcal{O}$ append $getVarOperation(s, dataFlow, \mathcal{F}_{sum})$;
18     end
19     for $r \in getReadVarFunction(dataFlow, \mathcal{F})$ do
20         $worklist$ append $r$ if $r \notin visited$;
21     end
22     $worklist$ remove $f$, $visited$ add $f$;
23     $\mathcal{F}_{sum}[f] \leftarrow dataFlow$;
24 end
```

use the data-plane message handler in the SDN controller as the entry point for static analysis to avoid recording irrelevant operations. Briefly, we model controller activities as DP-related function calls and variable read/write operations. Based on the identified DP-related operations, we inject collectors into the controller to capture activity data, which is then logged by the *Activity Logger*.

1) Static Program Analysis: To filter out irrelevant information and mitigate dependency explosion, we extract operations related to DP device behaviors through static analysis of the controller source code. Algorithm 1 presents the logic for generating the DP-related call graph and identifying variable operations. The listener function for DP messages serves as the initial function f_{init} (entry point) for generating the call graph. Starting from f_{init}, the algorithm traverses forward to record operations impacted by DP messages. We use all call relations $\mathcal{C} = \{(f_1, f_2, l_1), ..., (f_i, f_j, l_n)\}$ in the controller source code as an input, where (f_i, f_j, l_n) represents that function f_i calls function f_j at line l_n of f_i's source code. For each function f in $worklist$, we identify f's callees in $\mathcal{C}$ and append these call relations to the output $\mathcal{C}_{out}$ (Line 13).

To fully analyze the effect of DP packets on controller behavior, we account for variable read/write operations during packet handling. The algorithm generates a *dataflow summary* for each function to identify variable read/write operations. For each statement in function f, the algorithm updates f's $dataFlow$ (Line 16) and records the variables read/written by f (Line 17). DP-related variable operations are represented as $\mathcal{O} = \{(f_1, v_1, o_1, l_1), ..., (f_p, v_q, o_r, l_s)\}$, where $o_1, ..., o_r \in \{write, read\}$ and (f_p, v_q, o_r, l_s) represents the function f_p reading/writing (as specified by o_r) variable v_q at line l_s. As shown in Line 19, $getReadVarFunction()$ identifies functions that read variables written by f based on f's $dataFlow$. These reading functions are added to $worklist$ for further analysis (Line 20). Since the dataflow summarization in Algorithm 1 only covers the functions defined in the controller source code, we manually summarize the dataflow of library functions, such as `get()`, `add()`, and `append()` defined in library classes. To reduce this workload, we select a specific set of library functions that might be called during data-plane message processing. Specifically, we assume that any data accessed by a function could be tainted, and conduct pre-analysis to identify potentially invoked library functions.

2) Execution Unit Starting Points: To partition collected logs into *execution units*, we identify possible unit starting points by finding functions without callers in $\mathcal{C}_{out}$ generated by Algorithm 1. Collectors are then inserted at these points to mark the start of execution units and record the beginning of handling loops.

3) Network Activity Logging: Based on the model generated by Algorithm 1, we insert collectors into the controller source code to capture activity information, with the *Activity Logger* recording these details at runtime. Each log entry includes various runtime details depending on the type of operation, such as a timestamp with nanosecond precision, thread ID, operation type (*i.e.,* edge type), the executing function name, callee function name, variable name and memory address (*e.g.,* virtual memory address for Java-based controllers), concrete field of the variable, as well as event attributes (*e.g.,* ID, IP address, and switch port) related to objects (*e.g.,* switch, link and host) involved in the event.

B. Behavior Path Generation

1) Behavior Graph Reconstruction: In *Execution Unit Reconstruction*, operations in logs are grouped according to threads, and these thread-centric operations are partitioned into execution units at recorded starting points (mentioned in Section IV-A2). For each execution unit, log entries are converted into a single *execution unit graph* based on the provenance model described in Section III-B. For example, as shown in Fig. 5, we obtain three execution unit graphs, each depicting only partial operations for simplification. Then, *Inter-Unit Connection* connects execution unit graphs with *inter-unit* edges ($inter$) to construct the entire provenance graph. Specifically, execution units sharing the same attributes ($attribute$ nodes) or data instances ($data$ nodes) are associated via $inter$ edges. To alleviate dependency explosions, each data instance's write operation connects only to the next write and any intervening reads. For example, for a `host` data instance, read/write operations in execution units follow a sequence like $[(u_1, write), (u_2, read), (u_3, read), (u_4, write), (u_5, read)]$.

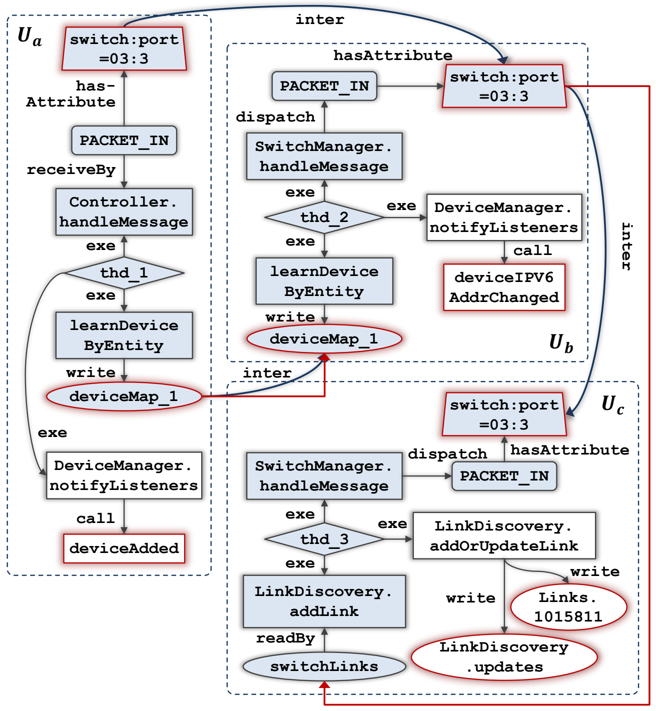


Fig. 5. Suspicious behavior graph of link fabrication. Diamonds represent threads, and other shapes have the same meanings as in Fig. 3. The anomalous path identified is highlighted in blue, while additional context, shown in white, includes relevant actions from execution unit graphs containing the suspicious path. Note that the *write*, *hasAttribute*, and *exe* edges are treated as bidirectional during path searching.

Inter-unit edges link $\texttt{host}_{u_1}$ to $\texttt{host}_{u_2}$, $\texttt{host}_{u_3}$, and $\texttt{host}_{u_4}$. After the next write in u_4, access on host (in u_5) is disconnected from u_1. For each attribute, inter-unit edges only connect nodes in earlier units to those in later ones. As illustrated in Fig. 5, after *Inter-Unit Connection*, the attribute nodes switch:port=03:3 and the data nodes deviceMap_1 in $\mathcal{U}_a$ and $\mathcal{U}_b$ are connected, and so are the attribute nodes in $\mathcal{U}_b$ and $\mathcal{U}_c$.

2) Redundancy Reduction: Repetitive operations and routines in the provenance graph do not contribute meaningful behavioral semantics. To reduce noise from these operations (**C$_2$**), we assess the importance of edges before *Path Extraction* and ignore trivial edges during path searches.

Maximum Common Subgraph Identification. To quantify importance, we first define the execution unit *pattern* p to identify units with the same operational flow. Using a function $s(\cdot)$, subgraphs are mapped to patterns by converting graph elements into structured text. For example, a data node deviceMap.85617803.3 is generalized as deviceMap<*> to match other instances in deviceMap. Also, $s(\cdot)$ merges repetitive edges within execution unit graphs. The mapping of an execution unit graph $\mathcal{U}_a (a = 1, ..., N)$ to its pattern p_i $(i < N)$ is represented as $p_i = s(\mathcal{U}_a)$. The set of patterns $\mathcal{P}$ results from applying $s(\cdot)$ to all execution unit graphs.

Based on $\mathcal{P}$, we compute the *Maximum Common Subgraph* (MCS) across execution unit graphs matching the same pattern, that indicates routines in each operational flow. We perform redundancy reduction within the scope of MCSs to protect other important operations from being removed. The insight is that edges outside an MCS reflect differences between units with the same operational flow, thus providing valuable contextual information. For example, a data instance counter, frequently written by the same function in execution unit graphs mapped to a packet handling pattern p_{pkt}, offers negligible behavioral semantics. The counter writing operation might be filtered out since it is included in the MCS of p_{pkt}. On the contrary, updates to different device<*> instances, being outside MCSs, are retained to capture behaviors tied to distinct devices.

For each pattern $p_i \in P$, its MCS S_{p_i} is the intersection of all execution unit graphs $\{\mathcal{U}_1^{p_i}, ..., \mathcal{U}_k^{p_i}\}$ mapped to p_i, represented as $\mathcal{S}_{p_i} = \bigcap_{j=1}^{k} \mathcal{U}_j^{p_i}$. To expedite MCS computation, we follow these steps: (i) for each pattern graph p_i, identify the set of edges $\{e_u\}$ matching each pattern edge, *i.e.,* $s(e_u) = e_p$ $(e_u \in \mathcal{U}^{p_i}, e_p \in p_i)$, (ii) prune pattern edges with more than one corresponding edge, and (iii) replace edges in the remaining pattern graph with their corresponding edges, *i.e.,* $e_p \rightarrow e_u$.

Intra-unit Redundancy Reduction. To measure the importance of operations, we use frequency as a metric, since operations that appear more frequently across different operational flows tend to offer less semantic differentiation between patterns. We apply Inverse Document Frequency (IDF) to weight edges in the provenance graph, making edge importance inversely proportional to its frequency. Specifically, an edge $e \in \mathcal{U}_j^{p_i}$ is filtered out if $w_e = log(\frac{N_E}{N_e}) < \delta_e, e \in \mathcal{S}_{p_i}$, where N_E is the total number of edges in execution unit graphs $(\mathcal{E}_{intra})$, N_e is the count of occurrences of edge e, and δ_e is a predefined importance threshold. To avoid removing critical edges, edge e is filtered only if it is within the MCS $\mathcal{S}_{p_i}$ (*i.e.,* $e \in \mathcal{S}_{p_i}$). Edges with importance w_e lower than the threshold δ_e are deemed redundant and excluded from further processing in the *Path Extraction* module.

Inter-unit Redundancy Reduction. For inter-unit edges, we evaluate their importance based on the importance of the units they connect. The importance of a unit u^{p_i} mapped to pattern p_i is defined as $w_{u_i} = log(\frac{N_u}{N_{p_i}})$, where N_u is the total count of execution units, and N_{p_i} is the number of execution unit graphs mapped to pattern p_i. To emphasize pivotal execution units, importance weights are propagated along inter-unit edges. During each propagation, a new importance weight for unit u_i is calculated as follows:

$$w_{u_i}^{new} = w_{u_i} + \frac{1}{N_{U_i^{adj}}} \sum_{u \in U_i^{adj}} w_u \tag{2}$$

where U_i^{adj} is the set of units adjacent to u_i, and $N_{U_i^{adj}}$ is the size of this adjacency set. An inter-unit edge $e_{ab} \in \mathcal{E}_{inter}$, connecting u_a to u_b, is ignored if the importance weight of either connected unit is below a preset threshold δ_u. This filtering process preserves inter-unit edges that link less frequent, and therefore more semantically distinct, units in long-term behavior patterns.

3) Path Extraction: To address the challenge $\mathbf{C_1}$, **ProvGuard** extracts paths in the provenance graph as contexts for anomaly detection instead of relying on graph semantics. By focusing on paths that span multiple execution units, this approach enables the detection of contextual anomalies in multi-phase, long-term attacks.

To reduce search costs, our method constructs fixed-length paths within each execution unit graph through a depth-first search and associates these intra-unit paths via inter-unit edges. Specifically, intra-unit paths originate from either $data$ or $attribute$ nodes, with the requirement that each starting node connects to at least one inter-unit edge. We choose $attribute$ and $data$ nodes as starting points for the following reasons. (1) Attributes serve as identifiers for network devices, allowing activities related to devices with the same label to be associated. For example, prior SDN anomaly detection methods identify devices using a unique identifier like a host's attachment point (switch port) to monitor device state [11], [27]. (2) Data instances relate to the controller's network state view. By connecting read and write operations of data instances, our method describes dataflows between units, capturing the causal relationships between network state changes and controller decisions. During intra-unit path extraction, the $write$, $hasAttribute$, and exe edges are treated as bidirectional, and a visited set is maintained to prevent revisiting nodes.

To characterize long-term behaviors, we associate a fixed number of intra-unit paths using inter-unit edges, forming a path that spans multiple execution unit graphs. Two subpaths are head-tail connected whenever an $inter$ edge points from a node in one subpath to a node in the other. In other words, we use $inter$ edges to associate inter-unit contexts, rather than including them as part of the path. For instance, in Fig. 5, subpaths form a complete path through solid red lines. Finally, a path is represented as a sequence of triples $\{(v_1, e_{12}, v_2), ..., (v_i, e_{ij}, v_j), ...\}$, where e_{ij} denotes the edge from v_i to v_j.

C. Suspicious Behavior Detection

We identify contextual discrepancies by evaluating the predictability of operations in paths, because anomalous events are harder to predict for models trained on normal behavior. We use an LSTM-based Seq2Seq model with the Luong attention mechanism to capture the semantics of long-term behaviors. LSTM is preferred over other Seq2Seq techniques because it can retain long-term dependencies, enabling us to capture potential relationships between operations in long behavioral sequences. The LSTM layers in the encoder incorporate the input sequence into a context vector, and during decoding, the model generates a subsequent sequence based on the context. For further details on the model, refer to the cited works [18], [26].

To input paths into the prediction model, we represent each triple in a path as an *action*. Suppose a path containing m edges spans n execution unit graphs. We convert the path into a sequence of length m: $\{x_1^1, x_2^1, ..., x_{m-1}^n, x_m^n\}$. Each x_a^b represents an action consisting of two nodes and an edge in a triple (*e.g.,* $x_1^1 = v_1\_e_{12}\_v_2$), where $a \in \{1, ..., m\}$ indexes the action within the path, and $b \in \{1, ..., n\}$ indexes the execution unit graph to which the action belongs. The prediction model takes $\{x_1^1, x_2^1, ..., x_{l-1}^{n-1}, x_l^{n-1}\}$ as input to predict the subsequent actions $\{x_{l+1}^n, ...x_m^n\}$. In this way, the behavioral semantics of subpaths in the first $n-1$ units provide the context to predict the behavior of subpath in the n^{th} unit.

To handle unseen data instances or attributes in new logs, we preprocess $data$ and $attribute$ nodes by combining the data type (*e.g.,* `device`, `link`) or the attribute type (*e.g.,* `MAC`, `IP`, `SWITCH_PORT`) with their sequence order within the same type. For example, in the sequence $\{f_1$`_write_link`$_A$, `link`$_A$`_readBy_`f_2, f_2`_write_link`$_B\}$, `link`$_A$ and `link`$_B$ are preprocessed as `link_1` and `link_2`, respectively.

We train the model on normal behavior sequences to minimize prediction error. This model uses cross-entropy to measure the deviation between predicted and target sequences. Mathematically, for an input sequence $\{x_1^1, x_2^1, ..., x_{l-1}^{n-1}, x_l^{n-1}\}$, the target value $\mathbf{r} = (\mathbf{r}_{l+1}, ..., \mathbf{r}_m)$ is a sequence of one-hot encoded vectors, representing the target action sequence $\{x_{l+1}^n, ..., x_m^n\}$. The predicted result $\mathbf{g} = (\mathbf{g}_{l+1}, ..., \mathbf{g}_m)$ is a vector sequence of $\{\hat{x}_{l+1}^n, ..., \hat{x}_m^n\}$, representing the predicted actions. The prediction error (cross-entropy) between $\mathbf{g}$ and $\mathbf{r}$ is calculated as follows:

$$\varepsilon_{predict} = \frac{1}{m-l} \sum_{j=l+1}^{m} \frac{1}{d} \sum_{i=1}^{d} (P(r_j^i) log P(g_j^i)) \tag{3}$$

where $P(\cdot_j^i)$ represents the probability of the j^{th} predicted action $\mathbf{g}_j = (g_j^1, ..., g_j^d)$ or target action $\mathbf{r}_j = (r_j^1, ..., r_j^d)$ on the i^{th} dimension, and d is the dimension of the one-hot encoded word vectors. Finally, in *Suspicious Path Aided Manual Investigation*, we select paths with a prediction error $\varepsilon_{predict}$ that falls outside the normal range and provide analysts with the relevant execution unit graphs of these suspicious paths to facilitate investigation.

V. Implementation

We prototyped our approach on the Java-based OpenFlow controller, Floodlight version 1.2, simulated a virtual network using Mininet[5], and utilized Cbench[6] to generate OpenFlow messages for performance analysis.

Static Program Analysis & Instrumentation. We implemented *Static Program Analysis* with the Soot framework for the Java-based controller. We set the `processOFMessage` method in the `OFChannelHandler` class, which listens to DP messages, as the entry point. Before executing Algorithm 1, we conducted a coarse-grained identification of read/write operations to avoid extensive manual effort in summarizing the dataflow of library methods. We identified 558 library methods, for which we wrote dataflow summaries.

[5]http://mininet.org

[6]https://github.com/andi-bigswitch/oflops

For execution partitioning, we identified 46 methods with no call sites through static analysis, treating these as the start of execution units. The analysis module ultimately output 828 DP-related operations for 430 methods, and we *automatically* instrumented the controller based on this output. Further details can be found in Appendix B.

Data Collection. We collected normal network activity logs in the simulated network to establish a dataset of benign behavior paths for model training. Two networks were simulated in Mininet: one with three hosts and another with five. SDN logs were collected under two conditions: with and without configured access control rules. We simulated representative host behaviors managed by the SDN controller, such as communication, host addition/removal, IP updates, and host migration. For more specific behaviors, we suggest using test cases provided by the SDN controller designer to capture expected contexts. To assess our approach's anomaly detection capabilities, we executed four typical CPM attacks — network identifier hijacking, link fabrication, access control rule bypass, and switch ID spoofing — on the simulated network, generating logs that include anomalous behaviors.

Behavior Path Generation. To filter out redundant edges, we set importance thresholds of $\delta_e = 0.4$ and $\delta_u = 0.7$. All importance scores of edges and units were normalized. During path search, each path spanned three execution units ($n = 3$), and each intra-unit subpath was limited to a length of five. Section VI-C discusses the reason for selecting these thresholds and path-length parameters. Given these settings, we extracted 487,939 paths from 27,614 normal behavior log entries for model training, and 19,395 *unique* paths from 27,492 log entries of potentially malicious behavior for model evaluation (*unique* paths are obtained by preprocessing and excluding duplicates).

Prediction Model Training. The embedding layer of our prediction model converts each one-hot encoded action into a 256-dimensional vector. The LSTM layer then incorporates each input sequence into a 1024-dimensional context vector. The prediction model was trained on 32,000 randomly selected normal paths, reaching convergence after 10 iterations.

Experimental Setup. The instrumented Floodlight controller, with the *Activity Logger* as an add-on application, was hosted on a 64-bit Ubuntu 20.04 virtual machine configured with 4 Intel Core i9-13900k 5.8 GHz CPUs and 16 GB of memory. Model training was conducted on macOS Ventura, using an 8-core Apple M1 CPU, an 8-core GPU, and 8 GB of memory with the Tensorflow 2.0 framework.

VI. Evaluation

In this section, we evaluate the prototype of **ProvGuard** in terms of its anomaly detection capability, the effect of contextual semantics, path search performance, workload reduction performance, and logging overhead to answer the following research questions (RQs):

- Can our approach extract paths of long-term behavior and distinguish suspicious contexts? How does it compare to other defense mechanisms in detecting CPM attacks?
- How does the contextual semantic information from the provenance graph contribute to anomaly detection?
- How effective is the redundancy reduction in filtering out noise and decreasing the number of extracted paths?
- To what extent can *Suspicious Path Aided Manual Investigation* reduce the workload of manual log auditing?
- Is the additional overhead from controller instrumentation and activity logging acceptable?

A. Attack Detection

To evaluate **ProvGuard**'s capability to detect CPM attacks, we examine four typical attack cases: network identifier hijacking, link fabrication, access control rule bypass, and switch ID spoofing. These cases include attacks launched from compromised switches (switch ID spoofing) and hosts (other cases), with tactics involving state manipulation (access control bypass) or packet forgery (other cases). We illustrate how the identified suspicious paths facilitate anomaly investigation and conceptually compare our approach with current state-of-the-art defense schemes. Additionally, we analyze anomaly detection performance by replaying the four typical attack cases, along with two variations, under different settings.

1) Case Study: For each case, we select the suspicious path with the highest prediction error to illustrate how **ProvGuard** assists in attack investigation. Due to space constraints, detailed analyses of the network identifier hijacking and switch ID spoofing cases are provided in Appendix C.

Link Fabrication Attack. An adversary can poison the controller's view of network topology by relaying or forging LLDP packets, misleading it into recognizing fabricated links. Details on implementing this attack are provided in Section II-B. As shown in Fig. 5, the suspicious path reveals that packets from a single switch port (`switch:port=03:3`) triggered both host information updates (processed by `deviceAdded` or `deviceIPV6AddrChanged`) and switch link modifications (handled by `addOrUpdateLink`). A similar path was observed in another set of execution unit graphs, where the switch port was `01:3` instead of `03:3`. According to this observation, analysts can infer that a malicious host is forging a fake link between ports `01:3` and `03:3`. Notably, although this attack did not generate observable DP anomalies, **ProvGuard** identified malicious behavior through CP actions, demonstrating its capability to capture CP behavioral conflicts critical for CPM detection.

Data Plane Access Control Bypass. An adversary can exploit a vulnerability in Floodlight's ACL application to illegally circumvent access control policies by triggering a host migration event. Details on this attack are provided in Section II-B. As illustrated in Fig. 6, the device `deviceMap_1`, initially attached to `switch:port=01:1`, changed its IPv4 address, triggering a `DEVICE` event. This `DEVICE` event, involving IP `10.0.0.1`, was handled by `ACL.deviceIPV4AddrChanged` and resulted in a `FLOW_MOD` message being sent to the switch. Later, `deviceMap_1` migrated to `switch:port=03:1`, activat-

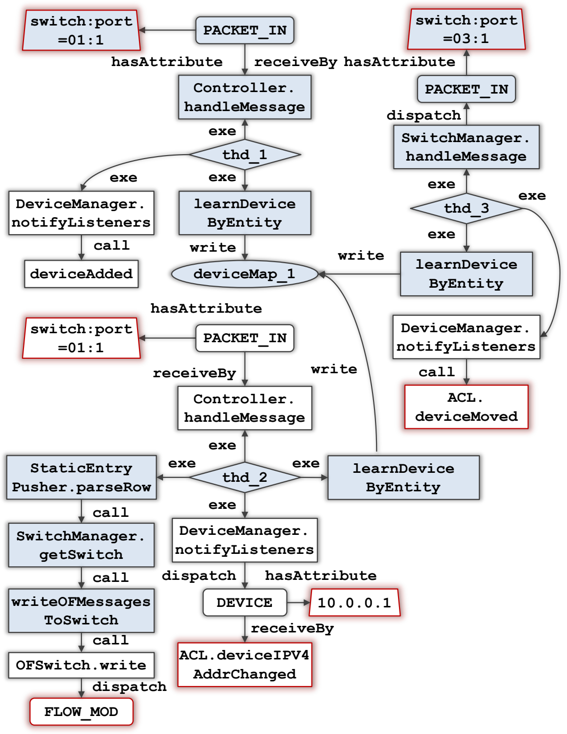


Fig. 6. Suspicious behavior graph of DP access control bypass. For simplicity, the same attribute/data nodes in different units are merged.

TABLE IV
CPM ATTACK DETECTION CAPABILITY COMPARISON.

	Network Identifier Hijacking	Link Fabrication	Access Control Bypass	Switch ID Spoofing
SPHINX [8]	●	●		
Veriflow [14]			●	
PacketChecker [7]	●			
TopoGuard [11]	●	●		
SPV [3]		●		
FlowChecker [1]			●	
ProvGuard	●	●	●	●

ing the `ACL.deviceMoved` method to handle the device migration event. However, `ACL.deviceMoved` failed to update the rules properly, as no new `FLOW_MOD` message was issued for `deviceMap_1`. Therefore, our method can help analysts to discover improper controller handling.

2) Comparison with Existing Defense Schemes: To illustrate the advantages of our approach, we conceptually compare **PROVGUARD** with existing defense schemes. As summarized in Table IV, rule-based efforts [3], [7], and [11] are tailored for specific attacks, limiting their generalizability. Flow rule verification approaches [1] and [14] are constrained, as they cannot detect abnormal flow rules without policy conflicts, such as in cases of forged bidirectional links. Attacks like network identifier hijacking add further complexity: when the flow rule derived from the victim host's information expires, the controller may generates a new forwarding rule based on the attacker host's information. In this case, rule conflict is not a necessary outcome, leading us to believe that flow rule verification is insufficient for addressing network identifier hijacking. Additionally, rule-based anomaly detection methods, *e.g.,* Sphinx [8], struggle with attacks leveraging unknown vulnerabilities in the control plane, such as access control bypassing. Also, switch ID spoofing attacks, which cause frequent removal of victim switches, interfere with the controller's flow rule calculations for packets traversing the attacking and victim switches. Consequently, defense mechanisms relying solely on metadata from flow rules and packets fall short in countering attacks like switch ID spoofing.

TABLE V
PERFORMANCE OF **PROVGUARD**

	Precision (%)			**Rate**[2]	**avg FP**[3]	
	3-host	5-host	diff. AP[1]		paths	units
Network Identifier Hijacking (IP)	100	94.85	100	30/30	0.28	0.17
Network Identifier Hijacking (MAC)	81.79	88.46	90.27	22/30	2.36	0.93
Bidirectional Link Fabrication	97.80	99.92	99.44	30/30	1.74	0.39
Unidirectional Link Fabrication	100	90.13	77.38	27/30	2.33	0.42
Access Control Bypass	100	95.55	99.95	30/30	3.59	0.73
Switch ID Spoofing	97.46	85.71	85.87	27/30	5.31	1.46

[1] We varied the attacker's host location (attachment point) within a 5-host network.
[2] Detection rate for each attack case.
[3] Average false positives per 100,000 log entries.

3) Detection Performance: We assessed **PROVGUARD**'s effectiveness against CPM attacks by repeating the four main attack scenarios, along with two variants, ten times each on 3-host and 5-host networks. To test the variants, we focused on two types of network identifier hijacking: IP address and MAC address hijacking, as well as two forms of LLDP packet spoofing attacks (forged bidirectional and unidirectional links). Due to the lack of ground truth, we defined correct detection as any path involving either attacking or victim devices.

The experimental results in Table V reveal that **PROVGUARD** effectively detects most CPM attacks. Occasionally, the model flags rare but normal behaviors as suspicious, with false positive paths ranging from 0.28 to 5.31 per 100,000 log entries, and less than two false-positive execution units requiring manual inspection. However, MAC hijacking attacks present a unique challenge, as the controller relies on MAC addresses as primary identifiers to track hosts. These attacks generate frequent host migration events that closely resemble normal host migrations. Despite this, the controller's provenance graph enables identification of suspicious patterns, like an abnormal fan-in of data nodes, which can be used to detect these frequent data updates. Therefore, although behavioral semantic difference is subtle, such anomalies related to MAC hijacking remain detectable. Filtering these frequent data update operations also reduces search space during path analysis. Similarly, switch ID hijacking, which prompts frequent switch removal events, can be effectively identified by filtering frequent operations or analyzing semantic deviations.

ANS$_1$: **ProvGuard** effectively captures long-term behavior features and outperforms existing detection approaches in identifying CPM attacks. It supports anomaly detection and investigation with minimal reliance on domain-specific knowledge or predefined rules.

B. The Effect of Contextual Semantics

We indirectly verified the effect of contextual semantics by demonstrating that isolated actions in paths lack sufficient differentiation to distinguish anomalies. Specifically, we selected key actions from suspicious paths and relevant execution unit graphs discussed in Section VI-A, such as calling `deviceAdded` (in Fig. 5 and 6), `deviceMoved` (in Fig. 6), `writeOFMessageToSwitch` (in Fig. 6), and `addOrUpdateLink` (in Fig. 5). We then analyzed the distribution of prediction errors in normal paths that included these actions. We also analyzed common actions, such as calling `learnDeviceByEntity` and updating `deviceMap`, to compare prediction error of paths containing these non-critical actions. As shown in Fig. 7, the violin plot illustrates the distribution of prediction errors for paths containing specific actions. Most paths show slight prediction errors, regardless of whether they include critical actions, indicating that individual actions alone do not significantly affect error distribution. Only different device updates (**writeOtherDevice** in Fig. 7) within the same context, such as two write operations on different device data instances within a single path, cause notable differences in error distribution. This difference arises because **writeOtherDevice** actions incorporate contextual information directly into individual actions via preprocessing, which indicates the impact of context.

ANS$_2$: It is not the presence of individual actions that leads to higher prediction errors, but the contextual discrepancies between actions. Thus, contextual semantics are crucial for effective anomaly detection.

C. Path Extraction Performance

This section evaluates the effectiveness of our redundancy reduction approach and explains the rationale behind our path feature selection.

Intra/inter-unit Edge Threshold. The *Path Extraction* applies redundancy filtering in execution unit graphs by excluding edges with importance weights below the intra-unit threshold δ_e. Fixing the inter-unit edge importance threshold δ_u at 0.7, we tested the impact of varying δ_e on the number of extracted paths. For this test, we collected activity logs from networks of varying sizes, each with a different number of active hosts. As shown in Fig. 8a, the number of paths decreases significantly when $\delta_e \geq 0.4$. Since the number of collected log entries varies by network scale, Fig. 8b demonstrates the ratio of paths to log entries (*i.e.,* the number of extracted paths per log entry). When $\delta_e \geq 0.4$, this ratio remains consistent despite

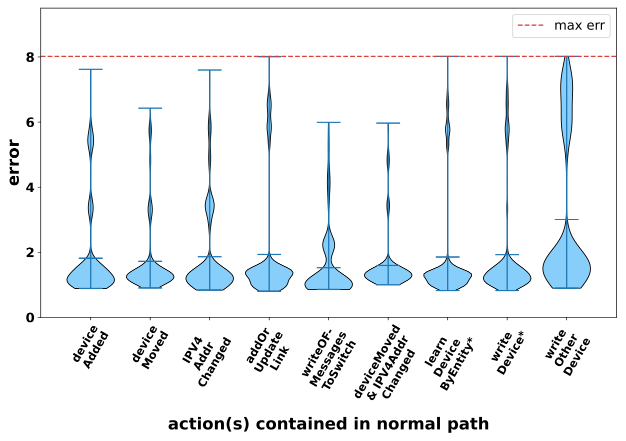


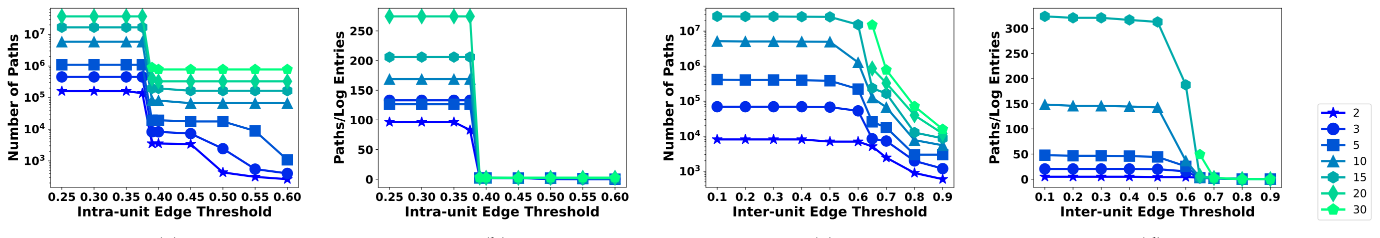
Fig. 7. Distribution of prediction errors for normal paths containing different actions. ∗-marked actions are non-critical, while unmarked actions are critical.

network size differences. This result indicates that $\delta_e \geq 0.4$ effectively reduces redundant intra-unit edges, achieving a steady extraction ratio regardless of network size. We also varied the inter-unit edge threshold δ_u while maintaining $\delta_e = 0.4$. Fig. 8c and 8d show that the number of paths decreases substantially when $\delta_u \geq 0.7$, and the ratio of paths to log entries remains stable regardless of network scale. This stability occurs because redundant operations are effectively filtered out, leaving only critical operations to form paths.

Path Extraction. Based on above analysis, we chose importance thresholds as $\delta_e = 0.4$ and $\delta_u = 0.7$. We further validated the path extraction performance with these settings under various network scales, as shown in Fig. 9. The redundancy filtering achieves an order-of-magnitude reduction in the total extracted paths per log entry and effectively decreases the computational costs of path search and contextual semantic analysis. Also, the number of unique paths remained nearly constant across different network scales and log volumes. This indicates that behavior graphs exhibit repeating patterns and that excluding duplicate paths during preprocessing effectively reduces the paths requiring further analysis. Additionally, we evaluated the impact of components in *Path Extraction* on anomaly detection performance in Appendix D.

ANS$_3$: Our approach effectively filters out mundane operations, significantly reducing semantic noise in logs.

Path Length and Span. We evaluated the diameter of execution unit graphs across various network sizes to determine optimal path-length parameters, as shown in Fig. 10. On average, the diameter remains close to 5 hops as network scale increases, with a maximum of 6. Based on these findings, we set the intra-unit subpath length to 5 hops to ensure thorough searches within each unit. For the overall path span, *i.e.,* the number of intra-unit subpaths, we accounted for the complexity of long-term behaviors. Considering common attack patterns and the model training workload, each behavior path spans three execution unit graphs. In this setup, the



(a) (b) (c) (d)

Fig. 8. Effect of intra/inter-unit edge threshold (δ_e/δ_u) on the number of extracted paths and average paths per log entry. As shown in the legend, different markers represent different network scales (number of active hosts). During the evaluation, all hosts in the network communicated with each other. Some data points are omitted to avoid excessive path counts. In Fig. 8a and 8b, the 30-host case is ignored for $\delta_e < 0.39$. In Fig. 8c and 8d, the 20-host and 30-host cases are excluded for $\delta_u < 0.65$.

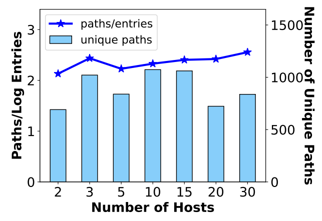


Fig. 9. The count of unique paths under different network scales ($\delta_e = 0.4$, $\delta_u = 0.7$).

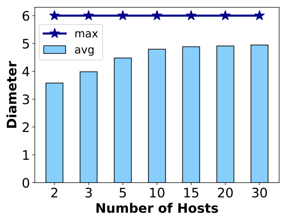


Fig. 10. Diameters of execution unit graphs under different network scales.

importance score of each execution unit is propagated twice along inter-unit edges, as specified in Eq (2).

D. Efficacy in Attack Investigation

ProvGuard provides analysts with execution unit graphs containing suspicious paths, narrowing the focus for manual review by localizing anomalies to specific CP activities. To assess the investigation workload, we simulated a network of three hosts and three switches, collecting logs of the attacks described in Section VI-A. Execution unit graphs containing paths with prediction errors exceeding the highest observed in normal behavior are flagged for manual review. For the four attack logs, the number of edges, execution units, and manual review workloads are shown in Fig. 11. Our approach reduces the manual analysis burden considerably. Of 1,378 execution unit graphs, only 51 require manual verification (3.70% of the total), and only 6.02% of edges require review. For instance, in the link fabrication attack log, only 18 out of 405 execution unit graphs need manual inspection. Further analysis reveals that these 18 subgraphs represent just eight execution unit patterns, reducing the actual workload even further.

ANS$_4$: Our approach significantly reduces the workload of manual investigation.

E. Overhead of Activity Logger

We evaluate the impact of *Activity Logger* on SDN network performance by examining latency and storage overheads.

Latency. Controller's response latency to DP messages impacts SDN network performance. Our approach collects network activity by instrumenting controller modules, which inevitably introduces additional latency. To quantify this, we used Cbench to generate traffic requiring the controller to calculate and install new forwarding rules. We measured the round-trip time (RTT) for each traffic packet across varying topologies. Fig. 12 compares response latencies before and after instrumentation. Activity collection caused RTT extensions averaging between 1.8% and 24% over the uninstrumented controller. Fig. 13 compares packet processing times for several controller applications, with the maximum average additional processing time being 78.4 μs.

Storage. The *Activity Logger* functions by continually logging streams of CP activity information, adding to CP infrastructure storage requirements. We evaluated storage overhead by simulating a network with 10 switches and 100 hosts, generating 1000 new flows per second [27], [29]. Under this workload, the *Activity Logger* produced an average of 1.3 GB/hr of audit log data, higher than ForenGuard's 0.93 GB/hr [29], another Floodlight monitoring system.

ANS$_5$: The latency and storage overheads of controller activity collection are within acceptable limits.

VII. Related Work

Anomaly Detection. Traditional approaches for defending against specific SDN attacks often rely on predefined rules, such as event condition checking, port state verification [11], network identifier binding checking [7], packet integrity checking [6], and packet latency checking [3], [24]. However, these rule-based methods are limited to known attack patterns and are ineffective for identifying unknown threats. Unlike these methods, our approach detects anomalies without preconfigured rules or prior knowledge of specific threats. Another common technique involves setting network invariants for real-time anomaly detection [8], [23]. Typically, Sphinx [8] dynamically constructs abstract flow graphs and detects security threats by verifying graph increments over time. However, without visibility into controller activities, these efforts fail

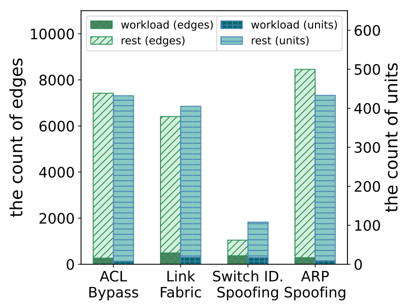


Fig. 11. Investigation workload for four attack logs.

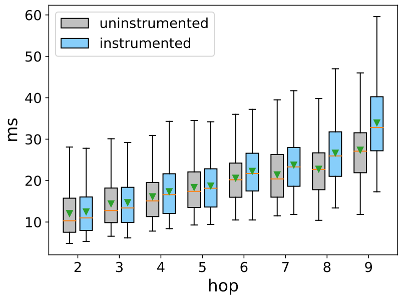


Fig. 12. Round-trip times under different hops before and after instrumentation.

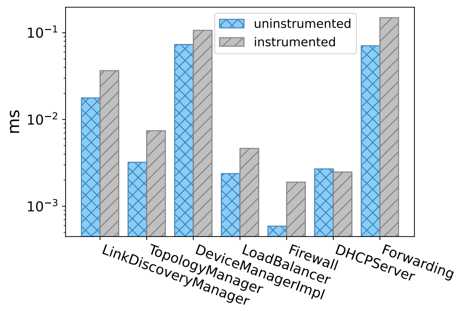


Fig. 13. Average latency of packet handling APIs.

to identify CPM attacks that exploit vulnerabilities in control logic through seemingly legitimate state changes.

Policy Verification. Policy verification techniques detect anomalies by verifying control policy consistency in flow table configurations [5], [14], [20]–[22], [25]. For example, VeriFlow [14] performs dynamic checking for policy violations based on network-wide invariants, such as destination reachability, security policies, and loop-free routing. Despite their advantages, these approaches are limited to assessing the legality of DP flow rules and cannot detect cases where the control plane's view of network state has been corrupted in ways that do not trigger policy conflicts.

Provenance for SDN. Provenance analysis in SDN aims to model network activities and investigate the root causes of anomalies [4], [16], [27], [29], [30]. Tools like ForenGuard [29] and PicoSDN [27] track the root causes of anomalies by collecting runtime activities and tracing back through execution graphs. However, as discussed in Section II-D, these tools face challenges in global graph-based anomaly detection due to unclear behavioral boundaries and semantic noise.

Graph-Based Anomaly Detection. Graph-based anomaly detection technology is widely applied in fields such as system security, digital content security, finance, and pharmacology. By leveraging the expressiveness of graphs, these techniques reframe anomaly detection as a task of identifying graph discrepancies [19]. Abnormal nodes, edges, paths, or subgraphs can reveal suspicious entities, relationships, contexts, and behaviors [10], [17], [33]. Building on these techniques, we introduce the first provenance graph-based anomaly detection framework specifically aimed at CPM attacks in SDN.

VIII. Conclusion

In this paper, we introduce **ProvGuard**, a provenance graph-based framework for detecting control policy manipulation attacks in SDN environments. Our approach models controller operations through static program analysis and instruments the controller to log network activities related to data plane message processing at runtime. This method represents execution traces of network behaviors as a provenance graph, where contextual deviations in paths signal potential anomalies. Our evaluation results demonstrate that **ProvGuard** can effectively detect a broader range of CPM attacks than prior approaches, with minimal dependence on expert knowledge, and provide execution graphs that facilitate efficient investigation.

Acknowledgment

We thanks the anonymous reviewers and our shepherd for their valuable feedback and insightful suggestions. This work is supported by the National Natural Science Foundation of China (No. 62172027, No. U24B20117), the National Research Foundation, Singapore, through the National Cybersecurity R&D Lab at the National University of Singapore under its National Cybersecurity R&D Programme (Award No. NCR25-NCL P3-0001), and the Zhejiang Provincial Natural Science Foundation of China (No. LZ23F020013). Any opinions, findings, conclusions, and recommendations presented in this material are those of the author(s) and do not reflect the views of the National Research Foundation, Singapore.

References

[1] E. Al-Shaer and S. Al-Haj, "FlowChecker: configuration analysis and verification of federated openflow infrastructures," in *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*. Chicago Illinois USA: ACM, Oct. 2010, pp. 37–44.

[2] T. Alharbi, M. Portmann, and F. Pakzad, "The (in)security of Topology Discovery in Software Defined Networks," in *2015 IEEE 40th Conference on Local Computer Networks (LCN)*, 2015, pp. 502–505.

[3] A. Alimohammadifar, S. Majumdar, T. Madi, Y. Jarraya, M. Pourzandi, L. Wang, and M. Debbabi, "Stealthy Probing-Based Verification (SPV): An Active Approach to Defending Software Defined Networks Against Topology Poisoning Attacks," in *Computer Security*, J. Lopez, J. Zhou, and M. Soriano, Eds. Cham: Springer International Publishing, 2018, vol. 11099, pp. 463–484, series Title: Lecture Notes in Computer Science.

[4] R. Birkner, D. Drachsler-Cohen, L. Vanbever, and M. Vechev, "Net2Text: Query-Guided Summarization of Network Forwarding Behaviors," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018, pp. 609–623.

[5] A. Chowdhary and D. Huang, "Object Oriented Policy Conflict Checking Framework in Cloud Networks (OOPC)," *IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING*, vol. 19, no. 5, 2022.

[6] S. Deng, W. Dai, X. Qing, and X. Gao, "Vulnerabilities in SDN Topology Discovery Mechanism: Novel Attacks and Countermeasures," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–11, 2024.

[7] S. Deng, X. Gao, Z. Lu, and X. Gao, "Packet Injection Attack and Its Defense in Software-Defined Networks," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 3, pp. 695–705, Mar. 2018.

[8] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "SPHINX: Detecting Security Attacks in Software-Defined Networks," in *Proceedings 2015 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2015.

[9] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, "A clean slate 4D approach to network control and management," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 5, pp. 41–54, Oct. 2005.

[10] X. Han, X. Yu, T. Pasquier, D. Li, J. Rhee, J. Mickens, M. Seltzer, and H. Chen, "SIGL: Securing software installations through deep graph learning," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2345–2362.

[11] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures," in *Proceedings 2015 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2015.

[12] J. Hua, Z. Zhou, and S. Zhong, "Flow Misleading: Worm-Hole Attack in Software-Defined Networking via Building In-Band Covert Channel," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1029–1043, 2021.

[13] S. Jero, W. Koch, R. Skowyra, H. Okhravi, C. Nita-Rotaru, and D. Bigelow, "Identifier Binding Attacks and Defenses in Software-Defined Networks," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 415–432.

[14] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying Network-Wide invariants in real time," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, Apr. 2013, pp. 15–27.

[15] S. Lee, C. Yoon, C. Lee, S. Shin, V. Yegneswaran, and P. Porras, "DELTA: A Security Assessment Framework for Software-Defined Networks," in *Proceedings 2017 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2017.

[16] X. Li, Y. Yu, K. Bu, Y. Chen, J. Yang, and R. Quan, "Thinking inside the Box: Differential Fault Localization for SDN Control Plane," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2019, pp. 353–359.

[17] F. Liu, Y. Wen, D. Zhang, X. Jiang, X. Xing, and D. Meng, "Log2vec: A Heterogeneous Graph Embedding Based Approach for Detecting Cyber Threats within Enterprise," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. London United Kingdom: ACM, Nov. 2019, pp. 1777–1794.

[18] T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, L. Màrquez, C. Callison-Burch, and J. Su, Eds. Lisbon, Portugal: Association for Computational Linguistics, Sep. 2015, pp. 1412–1421.

[19] X. Ma, J. Wu, S. Xue, J. Yang, C. Zhou, Q. Z. Sheng, H. Xiong, and L. Akoglu, "A comprehensive survey on graph anomaly detection with deep learning," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1, 2021.

[20] H. Pan, Z. Li, P. Zhang, P. Cui, K. Salamatian, and G. Xie, "Misconfiguration-Free Compositional SDN for Cloud Networks," *IEEE Trans. Dependable and Secure Comput.*, pp. 1–17, 2022.

[21] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for OpenFlow networks," in *Proceedings of the first workshop on Hot topics in software defined networks*. Helsinki Finland: ACM, Aug. 2012, pp. 121–126.

[22] P. Porras, S. Cheung, M. Fong, K. Skinner, and V. Yegneswaran, "Securing the Software Defined Network Control Layer," in *Proceedings 2015 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2015.

[23] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "AVANT-GUARD: scalable and vigilant switch flow management in software-defined networks," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*. Berlin, Germany: ACM Press, 2013, pp. 413–424.

[24] R. Skowyra, L. Xu, G. Gu, V. Dedhia, T. Hobson, H. Okhravi, and J. Landry, "Effective Topology Tampering Attacks and Defenses in Software-Defined Networks," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Luxembourg City: IEEE, Jun. 2018, pp. 374–385.

[25] S. Son, S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Model checking invariant security properties in OpenFlow," in *2013 IEEE International Conference on Communications (ICC)*. Budapest, Hungary: IEEE, Jun. 2013, pp. 1974–1979.

[26] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'14. Cambridge, MA, USA: MIT Press, 2014, p. 3104–3112.

[27] B. E. Ujcich, S. Jero, R. Skowyra, A. Bates, W. H. Sanders, and H. Okhravi, "Causal Analysis for Software-Defined Networking Attacks," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2021, pp. 3183–3200.

[28] B. E. Ujcich, S. Jero, R. Skowyra, S. R. Gomez, A. Bates, W. H. Sanders, and H. Okhravi, "Automated Discovery of Cross-Plane Event-Based Vulnerabilities in Software-Defined Networking," in *Proceedings 2020 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2020.

[29] H. Wang, G. Yang, P. Chinprutthiwong, L. Xu, Y. Zhang, and G. Gu, "Towards Fine-grained Network Security Forensics and Diagnosis in the SDN Era," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. Toronto Canada: ACM, Oct. 2018, pp. 3–16.

[30] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo, "Automated Bug Removal for Software-Defined Networks," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 719–733.

[31] F. Xiao, J. Zhang, J. Huang, G. Gu, D. Wu, and P. Liu, "Unexpected Data Dependency Creation and Chaining: A New Attack to SDN," in *2020 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2020, pp. 1512–1526.

[32] J. Zeng, Z. L. Chua, Y. Chen, K. Ji, Z. Liang, and J. Mao, "WATSON: Abstracting Behaviors from Audit Logs via Aggregation of Contextual Semantics," in *Proceedings 2021 Network and Distributed System Security Symposium*. Virtual: Internet Society, 2021.

[33] J. Zengy, X. Wang, J. Liu, Y. Chen, Z. Liang, T.-S. Chua, and Z. L. Chua, "Shadewatcher: Recommendation-guided cyber threat analysis using system audit records," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 489–506.

Appendix A
Notation Summary

The notations used throughout are summarized in Table VI.

Appendix B
Code Instrumentation

Fine-grained Operation Record. To mitigate dependency explosion, operations on different data instances like tables, queues, and lists are differentiated using indices, keys, or hashes. We manually marked fine-grained data fields for 61 read/write functions to distinguish data instances. For example, a hook records the hash of key `a` in `hashMap.put(a,b)` based on our specified data fields.

Beginning of Execution Unit. Through manual verification, various methods, such as REST request handlers (*e.g.*, `removeAllRules` in the `ACL` module), were identified as beginning of execution unit. Singleton methods like `timeoutLinks` in the `LinkDiscovery` module, which periodically deletes expired links, also mark the start of an execution unit. The selected entry point, `processOFMessage`, is also a beginning of unit.

Instrumenting Point. Table VII lists the number of instrumenting points for several controller modules.

TABLE VI
NOTATION SUMMARY.

Sign	Description
N	Number of execution units
$\mathcal{G}$	Provenance graph
u_i	i^{th} execution unit $(i = 1, ..., N)$
$\mathcal{U}_i$	Graph of u_i $(i = 1, ..., N)$
$\mathcal{V}$	Set of nodes in $\mathcal{G}$
$\mathcal{E}_{intra}$	Set of intra-unit edges in $\mathcal{G}$
$\mathcal{E}_{inter}$	Set of inter-unit edges in $\mathcal{G}$
$\mathcal{F}$	Set of functions in controller source code
f_{init}	Initial function of static analysis
$\mathcal{C}$	Set of call relations in controller source code
$\mathcal{C}_{out}$	Set of DP-related call relations
$\mathcal{O}$	Set of DP-related variable operations
p_i	Pattern of execution units $(i < N)$
$s(\cdot)$	Function mapping subgraphs to their patterns
$\mathcal{P}$	Set of patterns
δ_e	Intra-unit edge importance threshold
δ_u	Inter-unit edge importance threshold
$\mathcal{U}_j^{p_i}$	j^{th} unit graph mapped to p_i, where $s(\mathcal{U}_j) = p_i$
$\mathcal{S}_{p_i}$	MCS of unit graphs mapped to p_i, where $\mathcal{S}_{p_i} = \bigcap_{j=1}^{k} \mathcal{U}_j^{p_i}$
m	Number of edges in extracted path
n	Number of intra-unit subpaths in extracted path
v	Node in $\mathcal{V}$
e	Edge in $\mathcal{E}_{intra}$
x_a^b	Action in a path where $a \in \{1, ..., m\}$ indexes action in path, and $b \in \{1, ..., n\}$ indexes execution unit graphs to which the action belongs
$\mathbf{g}$	Predicted vector sequence
$\mathbf{r}$	Target vector sequence
d	Dimension of one-hot encoded word vectors
$P(\cdot_j^i)$	Probability of j^{th} action on i^{th} dimension $(i = 1, ..., d)$
$\mathbf{g}_j$	j^{th} predicted vector in $\mathbf{g}$, $\mathbf{g}_j = (g_j^1, ..., g_j^d)$
$\mathbf{r}_j$	j^{th} target vector in $\mathbf{r}$, $\mathbf{r}_j = (r_j^1, ..., r_j^d)$
$\varepsilon_{predict}$	Prediction error

TABLE VII
INSTRUMENTING POINTS FOR MODULES IN FLOODLIGHT.

Module	# of instrumenting points
Forwarding	22
DeviceManager	93
LinkDiscovery	89
AccessControlList	35
LearningSwitch	11
Topology	25
Firewall	10

APPENDIX C
ATTACK CASE STUDY

Network Identifier Hijacking Attack. A malicious host can broadcast spoofed ARP packets to bind its MAC address to a victim's IP address, causing network identifier hijacking. This attack alters device data instances in the controller and triggers IP update events. As illustrated in Fig. 14, two threads (`thd_1` and `thd_2`) triggered modifications on the device data instance `deviceMap_1`. The `DeviceManager.notifyListeners` method dispatched two `DEVICE` events containing different attributes, indicating a change in the device's IP address (from `IP_1` to `IP_2`). Further investigation of relevant execution unit graphs reveals that the malicious host on `switch:port=01:1` deceived the controller into incorrectly binding its MAC address with the victim's IP address (`IP_2`).

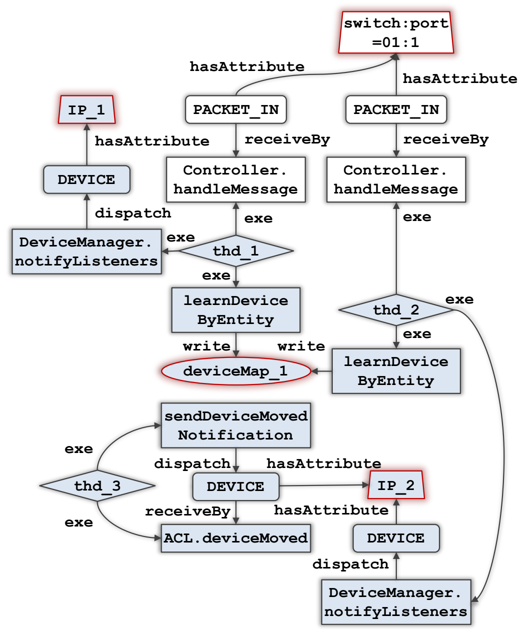


Fig. 14. Suspicious behavior graph of network identifier hijacking.

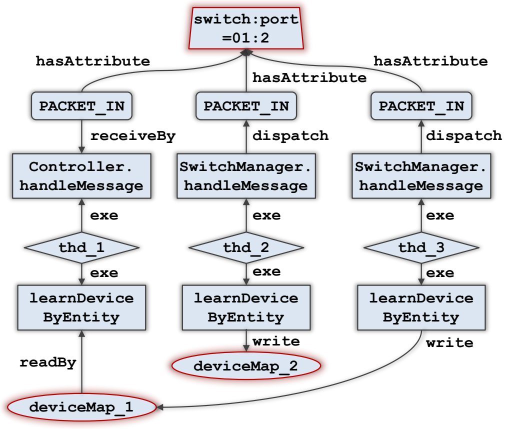


Fig. 15. Suspicious behavior graph of switch ID spoofing.

Switch ID Spoofing. Malicious forwarding devices or man-in-the-middle (MITM) attackers may alter the switch identifier during the handshake phase of control-data plane communication. The Floodlight controller identifies connected switches by their data plane ID, which attackers can forge to hijack legitimate switches. In the suspicious context shown in Fig. 15, two hosts on the same port, `switch:port=01:2`, triggered data updates for different host instances (differentiated by MAC addresses), since switch ID spoofing caused the controller to misinterpret different switch ports as identical.

APPENDIX D
PATH EXTRACTION COMPONENT EVALUATION

Our method uses *Intra-unit Redundancy Reduction* and *Inter-unit Redundancy Reduction* to remove unimportant edges before path extraction. As demonstrated in Section VI-A, the remaining paths, after redundancy reduction, still retain semantic deviations useful for detecting anomalies. To further analyze the impact of each component, we selected a benchmark set of logs from which **ProvGuard** can detect all attacks. By adjusting the importance thresholds δ_e and δ_u, we found that higher thresholds filtered out some critical behaviors, leading to detection failure for certain attacks, as noted in Table VIII. Additionally, we protect critical nodes by restricting redundancy reduction to the Maximum Common Subgraph (MCS) of execution unit graphs that match the same pattern. To assess the effect of this scheme, we set $\delta_e = 0.4$ and $\delta_u = 0.7$ while excluding MCS restriction. As a result, our approach failed to detect MAC hijacking attacks due to the omission of critical update operations.

TABLE VIII
The Effect of Components in Path Extraction

	default	$\delta_e = 0.7$	$\delta_u = 0.9$	no MCS
Host Location Hijacking (IP)	✓	✗	✗	✓
Host Location Hijacking (MAC)	✓	✗	✗	✗
Bidirectional Link Fabrication	✓	✗	✓	✓
Unidirectional Link Fabrication	✓	✗	✓	✓
Access Control Bypass	✓	✗	✗	✓
Switch ID Spoofing	✓	✓	✓	✓