Towards Fine-grained Network Security Forensics and Diagnosis in the SDN Era

Haopei Wang
Texas A&M University
haopei@tamu.edu

Guangliang Yang
Texas A&M University
ygl@tamu.edu

Phakpoom Chinprutthiwong
Texas A&M University
cpx0rpc@tamu.edu

Lei Xu
Texas A&M University
xray2012@tamu.edu

Yangyong Zhang
Texas A&M University
yangyong@tamu.edu

Guofei Gu
Texas A&M University
guofei@cse.tamu.edu

ABSTRACT

Diagnosing network security issues in traditional networks is difficult. It is even more frustrating in the emerging Software Defined Networks. The data/control plane decoupling of the SDN framework makes the traditional network troubleshooting tools unsuitable for pinpointing the root cause in the control plane. In this paper, we propose ForenGuard, which provides flow-level forensics and diagnosis functions in SDN networks. Unlike traditional forensics tools that only involve either network level or host level, ForenGuard monitors and records the runtime activities and their causal dependencies involving both the SDN control plane and data plane. Starting with a forwarding problem (e.g., disconnection) which could be caused by a security issue, ForenGuard can backtrack the previous activities in both the control and data plane through causal relationships and pinpoint the root cause of the problem. ForenGuard also provides a user-friendly interface that allows users to specify the detection point and diagnose complicated network problems. We implement a prototype system of ForenGuard on top of the Floodlight controller and use it to diagnose several real control plane attacks. We show that ForenGuard can quickly display causal relationships of activities and help to narrow down the range of suspicious activities that could be the root causes. Our performance evaluation shows that ForenGuard will add minor runtime overhead to the SDN control plane and can scale well in various network workloads.

KEYWORDS

Software Defined Networking, Security, Forensics, Diagnosis

ACM Reference Format:
Haopei Wang, Guangliang Yang, Phakpoom Chinprutthiwong, Lei Xu, Yangyong Zhang, and Guofei Gu. 2018. Towards Fine-grained Network Security Forensics and Diagnosis in the SDN Era. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3243734.3243749

1 INTRODUCTION

Network security diagnosis is important and useful since it can help the network administrator find a wide range of errors that may cause severe damages [29]. However, the emerging Software-Defined Networking (SDN) technique makes network security diagnosis much harder, because it decouples the control plane from the data plane and the logically centralized control plane is complicated and prone to security vulnerabilities [28, 44]. For example, when you observe a disconnection problem happen in a network running tens of SDN applications in the control plane, it is difficult to diagnose which application is exploited and how it makes the incorrect flow control decisions. Furthermore, since many existing SDN controllers are reactive and event-driven, the culprit events behind the misbehaving control plane are even much harder to be pinpointed. Fundamentally, there is a big gap in the SDN era, from observing the faulty forwarding behaviors in the data plane to finding out the root causes of the security problem in the SDN control plane.

In this work, we plan to bridge this gap by providing digital forensics that investigates the activities of the SDN framework and makes use of the recorded activities for networking security problems diagnosis. Previous research has worked on either network-level or host-level forensics. In the context of SDN, however, existing approaches cannot be directly used for our problem. This is because the networking security problems in SDN networks involve both the control plane and data plane, which makes individual either network-level or host-level forensics not effective; instead we need a systematic integration of both. In particular, in SDN networks, we observe forwarding problems from the data plane, but the culprits behind are typically in the control plane. That motivates us to monitor/record the fine-grained activities in the SDN framework and build causal dependency graphs among them. With careful diagnosis, the users can backtrack through dependency graphs and pinpoint the root cause of the security problems. To achieve this, we face the following challenges:

- *What kinds of activities in the SDN framework are required for the diagnosis purpose?* We aim to construct a model of concise set of activity types that can represent the execution of the SDN framework and aid the diagnosis. Since activity recording incurs overhead, the size of the set should be minimal.
- *How to build the causal relationship between different activities?* Simply dynamically taint-tracking all the control and

data flows in the control plane introduces huge overhead, while we aim to design a relatively lightweight solution.

- *How to efficiently and automatically query and locate the suspicious activities from the large forensics data?* There is an urgent need of a tool that helps users to diagnose issues, or even automatically locate the corresponding suspicious activities.

To address the first challenge, we model the states and transitions of the SDN data plane and the execution of the control plane. Using the model, the forensics results can concisely reason how each forwarding behavior occurs and provide easy-to-read information for diagnosis. To address the second challenge, we design a hybrid analysis approach that combines static analysis and dynamic profiling to track the information flows in the SDN framework. More specifically, we statically preprocess the controller/apps and then use runtime logging data to reconstruct *event-oriented execution traces* of the control plane and the *state transition graphs* of the data plane. To address the third challenge, we design a functional module that takes the description of the forwarding problem as input and automatically responds with the relevant suspicious activities as a reference for users. Besides this module, we also provide a command line tool that allows users to declaratively query for customized and detailed logged information.

We design a new system, ForenGuard, which provides fine-grained forensics and diagnosis functions in the SDN networks. The forensics function of ForenGuard involves both the SDN control plane and data plane. By monitoring and recording fine-grained activities in the SDN framework, we build dependency graphs based on their causal relationships. Our key insight is that the causal relationship can help users to backtrack the system activities and understand how each activity happens (e.g., which previous event triggers which module to generate which flow rule into the data plane, which causes a forwarding problem). The diagnosis function supports both fast querying for network forwarding issues and querying for detailed activities in the SDN framework. ForenGuard will respond user queries with the dependent graphs of activities that are relevant to the problem and help the users track back to the root cause of the forwarding problem.

We implement a prototype system of ForenGuard on top of the popular Floodlight [3] controller.[1] We show several use cases of ForenGuard that can quickly pinpoint the root causes which make use of different software vulnerabilities to launch attacks. Our evaluation results show that our system can provide fine-grained diagnosis for many types of networking problems and only introduce minor runtime overhead.

In summary, we make the following contributions:

- We propose a novel forensics scheme which dynamically logs the activities of both the SDN control plane and data plane, and builds event-oriented execution traces and state transition graphs for diagnosing network forwarding problems.
- We propose a user-friendly diagnosis tool which provides an inference-based approach to query the logged elements that have dependency relationships with the queried ones.

[1]Our technique is generic and extensible, and could be applicable to other mainstream controllers as well.

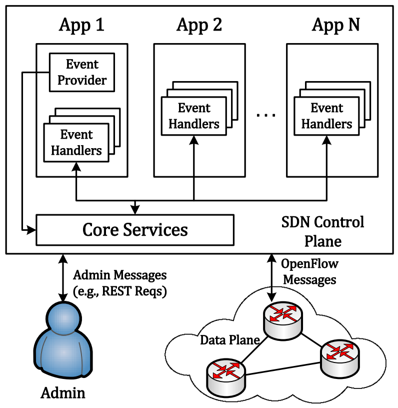


Figure 1: The Abstraction Model of the SDN Framework

- We implement a prototype system, ForenGuard, which helps network operators trace back past activities of both the control plane and data plane and pinpoint the root causes of network security problems. Our evaluation shows that ForenGuard is useful for diagnosing common SDN networking security problems with minor runtime overhead. We plan to open source ForenGuard to stimulate community effort and further research.

We construct our paper as follows. Section 2 specifies the research problems and motivates our solution. Section 3 shows a model of the control plane and data plane activities for forensics. Section 4 describes the detailed system design of ForenGuard. Section 5 provides detailed implementation of ForenGuard, case studies and evaluation results. Section 6 describes related work. Section 7 discusses limitations and future work. Finally, Section 8 concludes the paper.

2 BACKGROUND AND EXAMPLE

In this section, we first explain necessary background, the abstract model of the SDN framework in this paper and the threat model. Next, we use a running example which is a simple SDN controller application to explain research problems of diagnosing forwarding problems in SDN networks and motivate ForenGuard.

2.1 Abstract Model of SDN framework

We first define an abstract model of the SDN framework for forensics and diagnosis purposes. In this paper, our model includes only important elements which are the most useful ones for diagnosing networking problems that are caused by the misbehaving control plane. As shown in Figure 1, SDN decouples the network control plane from the data plane. The data plane consists of forwarding devices (i.e., SDN-enabled switches). Each switch contains large numbers of packet-forwarding rules, and each packet-forwarding rule is a tuple of pattern, action and priority. At a certain time, the state of the data plane is the value of all the packet-forwarding rules at all switches. The communication (i.e., OpenFlow [9] messages) between the control plane and the data plane may indicate the changes of the data plane state. For example, `FlowMod` message will

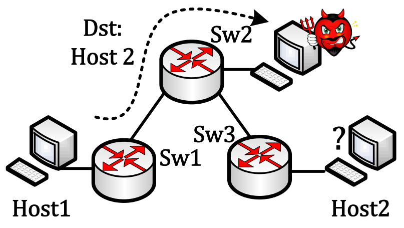


Figure 2: Attacking the LearningSwitch Application

install/delete/modify a rule. And it will trigger a `FlowRemoved` message to the control plane when a rule has expired or been removed.

About the control plane, we leverage the abstract modeling of the SDN control plane in ConGuard [44] and propose a similar model that can represent most of existing mainstream SDN controllers (e.g., POX [10], Floodlight [3], OpenDaylight [8]). In this model, the SDN control plane embraces an event-driven system. Multiple concurrent modules (also known as applications, we use the two words interchangeably in this paper) communicate via **events**. There is a *Core Services* module that works as the "event broker". It receives messages from the data plane (via OpenFlow messages) or the network administrator (via REST APIs) and dispatches the events (e.g., `PacketIn` event, `FlowRemoved` event). Other applications in the control plane subscribe the needed events from the *Core Services*. Each application has several event handler functions to process the events and make forwarding decisions. Some applications may dispatch their own event types, publish to the *Core Services* and allow other applications to subscribe. For example, in Floodlight [3] controller, the LinkDiscovery application will discover every link in the data plane and dispatch `LinkUp` and `LinkDown` events. Other applications like the TopologyManager module can receive the `LinkUp/Down` events and change the topology they have learned. In this paper, we focus on the event handler functions of every application because they represent the major logic that makes forwarding decisions.

2.2 Threat Model and Assumptions

Similar to existing research in digital/network forensic [22, 29, 30, 40], we trust the networking OS (i.e., the SDN controller) and our monitoring system (as an application in the SDN controller) and treat them as a trusted computing base (TCB). We assume no rootkit and also assume all applications running in the SDN control plane are initially benign but could be mis-configured or buggy/vulnerable. The bugs/vulnerabilities inside the applications written in Java in mainstream controllers typically do not cause buffer overflow or executable code injection. Instead, they might be exploited to crash the app [44] or mislead network forwarding decisions [15, 19, 44]. For example, TopoGuard project [19] discussed an issue in the topology discovery application which can be exploited to poison the topology learned by the controller and make wrong routing decisions. In this paper, these security issues of the SDN apps in the control plane that can be exploited and lead to network forwarding problems in the data plane are our targeted security problems. In our threat model, we assume an attacker can take control of host machines or compromised switches in the network and try to attack the SDN control plane by invoking/injecting certain network events, as shown in [15, 19, 44].

To make a practical forensics and diagnosis system, we assume the following additional assumptions: First, we assume the attacker takes action after ForenGuard is deployed. Second, even though the attacker can mislead the SDN control plane to make faulty forwarding decisions, she cannot fake or modify the runtime recording logs or disrupt the logging process, which could be achieved by using append-only secure log systems such as [11, 45]. Third, although ForenGuard injects some profiling instrumentation into the controller apps, it will not affect their original decision-making logic.

The goal of the diagnosis is to pinpoint the root cause of the caused forwarding problems, i.e., the violation of forwarding-related invariants. We consider three forwarding-related invariants: connectivity (routing between pairs of hosts), isolation (user-specified routing limitations), and virtualization (virtual network enforced flow handling policies). Finally, we focus on flow-level diagnosis (instead of packet-level diagnosis).

2.3 Running Example

```
 1| public class LearningSwitch {
 2| // Stores the learned state for each switch
 3| protected Map<IOFSwitch, Map<MacVlanPair, OFPort>>
 4|   macVlanToSwitchPortMap;
 5| private Command processPacketIn(sw, pkt) {
 6|   OFPort inPort = pkt.get(MatchField.IN_PORT));
 7|   MacAddress srcMac = pkt.get(MatchField.ETH_SRC);
 8|   MacAddress dstMac = pkt.get(MatchField.ETH_DST);
 9|   VlanVid vlan = pkt.get(MatchField.VLAN_VID);
10|
11|   // Learn the port for this source MAC/VLAN
12|   this.macVlanToSwitchPortMap.get(sw).put
13|   (new MacVlanPair(srcMac, vlan), inPort);
14|
15|   // Try to get the port for the dest MAC/VLAN
16|   OFPort outPort = macVlanToSwitchPortMap.
17|   get(sw).get(new MacVlanPair(dstMac, vlan));
18|
19|   if (outPort == null) {
20|     // Dest MAC/VLAN not learned, flood it
21|     this.writePacketOut(sw, pkt, OFPort.FLOOD);
22|   } else {
23|     // Dest MAC/VLAN learned, forward
24|     this.pushPacket(sw, pkt, outPort);
25|
26|     // Install flow entry matching this packet
27|     this.writeFlowMod(sw, OFFlowModCommand.ADD,
28|     OFBufferId.NO_BUFFER, pkt, outPort);
29|     //match 4-tuple: {src/dst MAC, VLAN and input port}
30|   }
31|   return Command.CONTINUE;
32| }}
```

Listing 1: Example Controller Application

Listing 1 (abstracted from a real-world SDN controller application [4]) shows a simple but vulnerable application that may be exploited by malicious end-hosts to launch the **host location hijacking attack**. The application implements a learning switch which uses the previous learned MAC/VLAN to port mapping (underlined variable) to install forwarding rules. When the application receives a `PacketIn` message (which means the first packet of a new flow), if the destination MAC/VLAN has been learned before from the switch (Line 23 - 30), the application will install a flow rule to forward this flow to the port in the pair with the MAC/VLAN, otherwise flood the packet (Line 20 - 21).

The above learning-based algorithm is vulnerable since the "learned" information could be spoofed that will mislead the future forwarding decision. Illustrated in Figure 2, an attacker can spoof the MAC address of Host 2 and make a connection to Host 1. The operation will make every switch in the network learn that the MAC of Host 2 matches the attacker's host. Later, when the real Host 2 makes a connection to Host 1, the traffic from Host 1 will be forwarded to the attacker. As a result, Host 2 does not have network connection to Host 1. However, it is hard for Host 2 to pinpoint the root cause. That is because she does not have enough information about what happened in the control plane and data plane in the past. Host 2 desires a tool that receives her trouble ticket and pinpoints the root cause of the forwarding problem.

2.4 Problem Statement

Traditional diagnosis tools can only locate the issues at either the network level (e.g., Anteater [29]) or host level (e.g., Forenscope [13]), and are not capable of integrating the two levels. Several troubleshooting and verification tools in the context of SDN have been proposed in recent years. They provide functions of static or dynamic network-wide invariant verification [23–25], model checking [12], packet history analysis [17], record and replay [43] and delta debugging [35]. However, these tools fall short because of limited expressiveness (invariant expression), scalability (exponential explosion), non-determinism (trace replay) or coarse granularity (network flow/flow rule level) issues.

Unlike existing approaches, we leverage the concept of forensics which records system activities in runtime and makes use of them for diagnosis. Suppose we have enough information about what happened in the SDN framework, for the above running example, our concrete diagnosing steps can be like follows:

Step 1. We first analyze the forwarding rules in the data plane to find out the set of rules that result in the forwarding problem. We identify them as "suspicious" forwarding rules. In the running example, the rules that forward the traffic whose MAC belongs to Host 2 to the attacker are suspicious rules.

Step 2. Based on the suspicious rules, we can list all OpenFlow messages that install/modify these rules.

Step 3. By recording the execution traces of the SDN applications, we can trace the relevant control plane activities which generate the messages.

Step 4. By analyzing the causal relationship among different activities in the execution trace that generate the messages, we finally find out that the wrong forwarding decision is made by two previous data plane activities. One is the new flow event from Host 1. The other is the new flow event (using spoofed source MAC) from the attacker. Obviously, the spoofed packet from the attacker is the root cause of the problem.

In summary, our idea is to record detailed activities in both the control and data plane and build the causal relationship between them. Nevertheless, realizing the forensics and diagnosis in SDN networks requires tackling three challenging problems:

- *First, how to decide useful activities that are necessary for the diagnosing purpose?*
- *Second, how to build the causal relationship among different activities?*
- *How to efficiently query/locate the suspicious activities from the big data?*

Besides, our system has the following design goals:

Fine Granularity: We aim to provide fine-grained details for the execution traces (e.g., every main step that makes the forwarding decision) and root causes of forwarding problems (e.g., which message/event/packet/piece of code is the root cause).

Minor Overhead: Forensics systems will introduce unavoidable overhead. To analyze the runtime behaviors of the SDN framework, unlike existing information flow analysis approaches (e.g., dynamic taint-tracking), we aim to design a relatively lightweight solution.

Easy-to-Query: Our tool aims to support both directly querying for network forwarding issues and querying for detail activities in both the control plane and the data plane, and provide user-friendly query interfaces.

3 MODELING OF THE SDN ACTIVITIES

In this section, we explain ForenGuard's modeling of activities in both the SDN control plane and the data plane.

Data Plane Activities: The purpose of recording the states of the data plane is to understand the forwarding behaviors at any time. First, we give a definition of the data plane state:

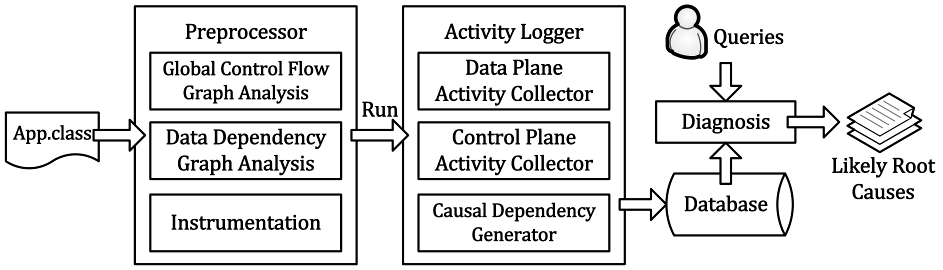
Definition 1: At time t, the state of the data plane (denoted as s_t) is the value of the set of all flow entries at all switches at time t.

$$\begin{aligned} s_t &= \{r_1, r_2, ...r_n\}|_{time=t} \\ r_i &= (switchID, entryID, (match, action, priority)) \end{aligned} \tag{1}$$

Definition 2: A transition (denoted as a_i) of the data plane is one OpenFlow message that is triggered by or will trigger the change of the state.

For instance, the `FlowMod` message sent from the control plane will install/modify/remove a flow rule in one switch. And `FlowRemoved` message sent from the data plane means a flow rule has been expired/removed. These two messages are types of transitions. We use $\rightarrow$ to describe the transition of data plane state. So if an activity a_i triggers that the state of the data plane transits from s_x to s_y, then: $s_x \xrightarrow{a_t} s_y$.

The state of the data plane can clearly show the forwarding behavior at that time. And the transitions can explain the reason of the state changes. In our diagnosis steps, we first search for the corresponding data plane state that starts to have the faulty forwarding behavior and then find the activity which causes the transition to that state. For instance, in our running example, when Host 2 observes that there is no network connection between Host 1 and Host 2, we start to search the state that tells us how the data plane forwards the traffic of Host 2 (either source or destination address is Host 2). We can quickly find that in some state, there is a forwarding path that matches Host 2's traffic but is between Host 1 and another location (not Host 2). Then by searching the transitions and corresponding activities, we find that there are several `FlowMod` messages that make the faulty forwarding path. After we find the faulty data plane states and corresponding activities, our next step is to move to the control plane and understand why and how the control plane makes such forwarding decisions.



Preprocessor
Global Control Flow Graph Analysis
Data Dependency Graph Analysis
Instrumentation
App.class
Run
Activity Logger
Data Plane Activity Collector
Control Plane Activity Collector
Causal Dependency Generator
Queries
Diagnosis
Database
Likely Root Causes

Figure 3: System Design of ForenGuard

Operation	Definition
Init(f, A, td)	Start the function f of app A in thread td
End(f, A, td)	Terminate the function f of A in thread td
Read(v, td)	Read variable v in thread td
Write(v, td)	Write variable v in thread td
Dispatch(e, td)	Dispatch event e in thread td
Receive(e, td)	Receive event e in thread td
Run(A)	$run()$ function of a singleton task in app A
Send(sw, msg, td)	Send message msg to switch sw in thread td

Table 1: Control Plane Operations

Control Plane Activities: We aim to record the execution of the control plane to understand how each application receives and dispatches events, and makes forwarding decisions during runtime. We model the execution of the controller as a sequence of operations to functions, state variables and events.

The operations in Table 1 list the activities that we think can explain the major decision-making logic of the control plane. We can divide the operations into three categories: function operations, variable operations and communication operations. The initiation and the termination of a function instance show the dynamic call graphs. Specifically, the ***Run*** operation means that some applications may have a singleton task that maintains a life-cycle of a run() function. This function itself could trigger some events or modify the value of state variables. For example, some singleton tasks will periodically clear the values of some state variables (e.g., clear the list of hosts information). For variable operations, the read and write operations of state variables help to understand the information flows in runtime. We define the state variables as the global variables in every application (e.g., the MAC/VLAN to port mapping table in the running example).[2] The other three operations are communication operations. The ***Send*** operation means this function generates new OpenFlow message to the data plane, which may trigger the state transition in the data plane.

The purpose of logging the execution of the control plane is to help pinpoint the root cause of some suspicious messages. When we figure out the suspicious messages that trigger the data plane state to have forwarding problems, we can observe the steps how the control plane generates the messages. When diagnosing the forwarding problem, the logged execution can explain which application, which operations and which events/variables affect the decisions made by the control plane. In the running example, when Host 2 reports the connection problem and we already find the suspicious OpenFlow messages, we can observe that the function `processPacketIn` receives some new flow events, checks the value of some fields in the MAC/VLAN to port mapping and generates the suspicious messages. So the new flow event that triggers the function to generate the faulty flow rule is the direct cause, and the runtime value of the mapping table is the indirect cause of the problem. Then we keep searching previous operations that write the certain filed of the mapping table. At last, we find another event which shows a new flow causes such MAC/VLAN to port pair to the mapping table, which is the root cause of the reported problem.

[2]In our implementation of ForenGuard that works on Java-based controllers, the state variables are the instance variables of the main class of each application.

4 SYSTEM DESIGN

In this paper, we propose a fine-grained forensics and diagnosis system, named ForenGuard, which can help network administrators to pinpoint security issues in software defined networks. The key idea behind is that ForenGuard makes the trade-off between SDN controller performance and the cost of monitoring sensitive operations. To this end, ForenGuard is designed as three-fold. First, ForenGuard applied static program analysis to identify the minimal set of variables and operations whose changes may be associated with future security issues. For convenience, we refer to these variables and operations as state variables and operations (according to our model of the control plane in Section 3). To monitor these variables and operations in the run-time with minimal overhead, ForenGuard instrumented the code of the target controller. To monitor the information flow in the run-time, we also design a novel lightweight flow tracking approach, which is also implemented in the instrumentation. Second, ForenGuard deploys and runs the newly instrumented SDN controller. By analyzing the controller log in real time, the network activities are constructed based on causal relationship. Finally, once administrators find a routing problem, ForenGuard can help figure out the root reason of the problem using an easy-to-query interface.

4.1 System Architecture

ForenGuard works on top of the SDN control plane and does not disrupt the normal operation of other controller applications. As showed in Figure 3, our system consists of three modules: 1) **Preprocessor**, which conducts static analysis to extract the concise set of activities for the recording purpose and further instruments SDN controller to monitor the sensitive operations and apply our lightweight information flow tracking approach; 2) **Activity Logger**, which runs the instrumented controller and dynamically reconstructs the causal relationships from the collected activity logs; 3) **Diagnosis**, which provides an easy-to-use diagnosis interface and can help pinpoint the root reason of a security problem. In the following of this section, we describe the design details of each module and corresponding techniques.

4.2 Preprocessor

The goals of the Preprocessor module are three-fold: using static analysis to extract activities, generating data dependency graphs and instrumenting the controller. The Preprocessor module statically analyzes the source code of an SDN controller.[3] As explained in Section 3, to reason about how each forwarding decision has been made from the control plane, we need to record the important operations and the information flows (e.g., which flow rule is triggered by which data plane events.). However, dynamic analysis (e.g., taint analysis) to track the information flows will inevitably add huge runtime overhead, which is unacceptable in the SDN control plane, while static analysis is not precise. Instead, we aim to achieve a trade-off between the overhead and precision. ForenGuard statically identifies the state variables, analyzes the data flows and instruments the read/write operations of the variables. Then, these state variables and operations are further recorded to build the information flows. For example, in the running example in Section 2, ForenGuard is able to analyze the information flows from the data sources (e.g., the `PacketIn` event and/or one filed of the MAC/VLAN to port map) to the generated messages. Next we will detail how ForenGuard conducts static analysis and instrumentation.

Static Analysis: The Preprocessor module consists of two sub-modules: *global control flow graph analysis* and *data dependency graph analysis*. Given an SDN controller application, ForenGuard runs the sub-module *global control flow graph analysis* to first convert its source code into an intermediate representative language (bytecode) and transform to a global control flow graph (CFG). Then, ForenGuard identifies the important operations according to the controller model by searching CFG and the paths to the operations. In the meantime, ForenGuard also identifies the state variables and searches all read/write operations of the variables. Here, we define the state variables as the class instance variables of the application. The insight behind is that, except the inputs (events) from south or north bound interfaces, instance variables are normally used to store the states of the application and make forwarding decisions. For example, in the motivating example, all previously learned information is saved in the MAC/VLAN to port map data structure. And every output flow rule is generated based on both the input events and the runtime values of the MAC/VLAN to port map data structure. Specially, we do not count variables that are used for logging (log system of the controller itself, not ForenGuard) or debugging, which are useless for our purpose.

Next, ForenGuard constructs the data dependency graph by applying the backward data flow tracking technique on the state variables identified in the previous analysis. To support the above

[3]We assume the SDN controller and third-party applications should be open source to the network administrator and operators.

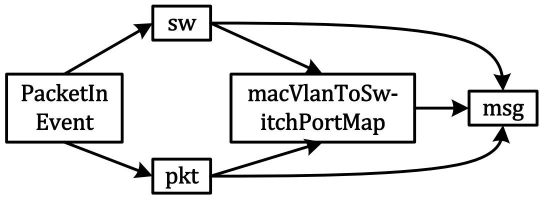


Figure 4: Data Dependency Graph of the Running Example

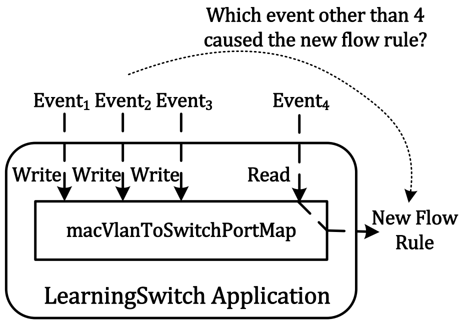


Figure 5: Challenge of Coarse-granularity

analysis, several challenges are addressed. First, different from regular programs, an SDN application does not have entry points, since the main function is missing. To apply data flow tracking as normal, entry points must be explicitly defined. To this end, our SDN model in Section 2 is leveraged, which provides sufficient hints. The major part of each application is multiple event-driven handler functions. The event handler functions are registered in the *Core Services* to subscribe the corresponding events. Therefore, we set the handler functions as the entry points for the data dependency analysis.

Second, to adapt data flow tracking on a SDN controller, we define sources and sinks as follows. The data sources we use are the parameters of the handler functions including the events and corresponding metadata (e.g., in-port of a new flow) and the state variables from read operations. The data sinks are state variables from write operations and generated flow rules (e.g., Line 27 of the running example).

ForenGuard performs context-sensitive, field-sensitive data flow analysis on controllers to build the data dependency graph (DDG). Figure 4 shows the data dependency graph (DDG) of the running example. The data of the MAC/VLAN to port map could be from the input parameters (*sw* and *pkt*) which are extracted from the *PacketIn* event. The generated flow rule *msg* (if that branch is triggered) is affected by the input parameters and the map. At runtime, ForenGuard will generate more concrete and precise information flows based on the logs of read/write operations of the state variables.

Technical Challenges: We discuses two technical challenges about the static analysis: inaccuracy and coarse-granularity. The inaccuracy of static analysis is well-known since it just explores all possible data flow paths but cannot track if one certain path is actually triggered in runtime. Another challenge is that static analysis can only provide coarse-grained data flow tracking results.

That is because each state variable may contain many fields, and it is hard to track which field every event actually accesses. In our running example, the MAC/VLAN to port mapping data structure contains multiple key-value pairs/entries/entries. Illustrated in Figure 5, suppose we already know Event 4 reads the variable *macVlanToSwitchPortMap* and then processes a new flow rule which causes the forwarding problem, however, it is still not clear which entry of the variable Event 4 reads, and which previous event adds/modifies this entry. To address them, we instrument the source code of the controller and applications to profile the detailed field read/write operations of each state variable.

Instrumentation: Based on the static analysis results, another sub-module *instrumentation* starts to instrument the controller applications at the bytecode level. The target of the instrumentation is to profile important operations of the control plane at runtime. The instrumented code will record the source code context (e.g., class name, line number, thread ID) as the metadata with the involved heap memory information (the virtual memory address in JVM) of the operation. Specially, for variable read/write operations, we do not record the runtime values of the variables for two reasons: First, recording the runtime values of the variables is too costly. Second, our purpose is to track the information flows, which has no need to track the concrete variable values. For example, we aim to track an information flow starting from a data plane event e_1 changing the value of $a.x$ (whose virtual memory address is m_1). Further, another information flow reads this memory location and finally generates a message msg_1 which installs a new flow rule . Then we can build the the causal relationship from e_1 to msg_1.

4.3 Activity Logger

After the Preprocessor module, we deploy the instrumented controller in an SDN network. The Activity Logger module works as a controller component and dynamically collects activities from both the control plane and the data plane and further builds the causal dependency relationships. The activities are handled by the three sub-modules: 1) *Data Plane Activity Collector* collects the runtime data plane activities; 2) *Control Plane Activity Collector* collects the runtime control plane runtime activities; 3) *Causal Dependency Generator* builds the causal dependency relationships between the collected activities and saves them into a database.

Data Plane Activity Collector: Section 3 defines the activities of the data plane. The Activity Logger module first keeps tracking all OpenFlow messages between the control plane and the data plane. Since we consider switches could be compromised in our threat model, the *Data Plane Activity Collector* sub-module does not directly monitor the states of the data plane switches through some administering channels (e.g., *ovs-ofctl*, *ovs-dpctl*). Instead, to flexibly track the states and any transitions of the data plane, the *Data Plane Activity Collector* sub-module makes use of the OpenFlow messages to speculate the states of the data plane switches. In the OpenFlow protocol, any changes in the data plane forwarding tables (install, modify, delete, expire) should be enforced by or inform the control plane via OpenFlow messages. Therefore, by tracking and analyzing all OpenFlow messages, it is already able to understand the state and changes of the data plane forwarding tables. In our tracking solution, the *Data Plane Activity Collector* sub-module always maintains a data structure that stores the current state of the data plane forwarding tables. Whenever it observes the OpenFlow message which shows a change of data plane forwarding table, the module will generate the new state of the table based on the meaning of that OpenFlow messages. For example, a *FlowRemoved* messages will indicate that a flow entry in one forwarding table has expired. Thus, the sub-module can delete the flow entry from its own data structure and log the change. In the future diagnosis phase, if the stored data plane state does not match the actual data plane forwarding behaviors, then there could be attacks from the compromised switches.

Control Plane Activity Collector: The control plane activities that we aim to collect are shown and explained in Section 3. The previous Preprocessor module already instruments the source code with the logic of recording these control plane activities. Thus in runtime, the instrumented statements will forward the log information to the *Control Plane Activity Collector* sub-module.

Causal Dependency Generator: The *Causal Dependency Generator* sub-module collects and processes the activities received from the *Data Plane Activity Collector* and *Control Plane Activity Collector* sub-modules. It reconstructs *event-oriented execution traces* of the control plane and the *state transition graphs* of the data plane, and then combines them together. *State transition graphs* include the data plane forwarding states and state transitions. *Event-oriented execution traces* include the function-level call graphs (function operations and communication operations) and information flows (variable operations) of the control plane. Figure 6 shows an example of these two types of data structures. In this figure, S_x denotes data plane forwarding states, e_x denotes events, f_x denotes function calls and $a.x$ and $b.y$ denote variables. Using these graphs, we can reason the causal relationship between activities.

Algorithm 1: Function Call Graph Reconstruction

```
Input: $S$ = list of function calls in [(thread ID: $T$, function
        name: $M$), ... ]
Input: $G$ = adjacency list representing the global control flow
        graph {node:[adjacency nodes], ...}
Output: $L$ = list of function calls representing dynamic
          call-graphs {thread:[[function calls],...], ...}
$stack[:] \leftarrow \emptyset$  # Initiate the stack as empty only at the first run
  of the algorithm
$L[:][-1] \leftarrow 0$;
foreach $S_i$ in $S$ do
    while $stack[S_i.T] \neq \emptyset$ do
        $R \leftarrow stack[S_i.T]$.top();
        if there is a path from $R$ to $S_i.M$ in $G$ then
            break
        $stack[S_i.T]$.pop();
    if $stack[S_i.T] \neq \emptyset$ then
        $L[T_i][-1]$.append($S_i$)
    else
        $L[T_i]$.append(new List($S_i$))
    $stack[T_i]$.push($S_i.M$)
```

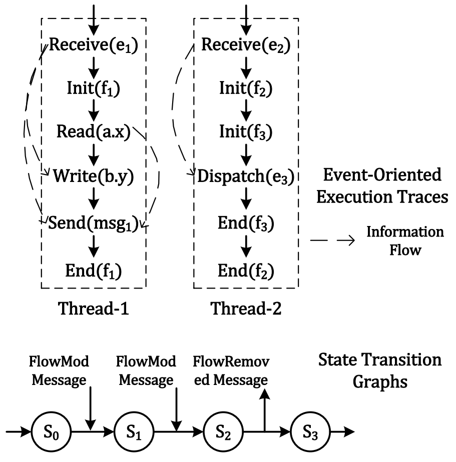


Figure 6: Execution Traces of the Control Plane and State Transition Graphs of the Data Plane

We design an algorithm (shown in Algorithm 1) to reconstruct the dynamic function-level call graphs. The output of the algorithm is a list of execution traces. Each execution trace is a sequence of function operations which represents the entire execution from the start of an event handler function to the end of the handler function. We build the data dependency relationships of different variables in each application, in the **Activity Logger** module, based on the recorded read/write operations of the fields of the variables. For example, suppose we have the result that event e has data flow relationship with the state variable $v.a$. When we dynamically log there is a write operation to $v.a$ with its object ID in the heap memory, and this execution trace is triggered by an event e_1, we can build the information flow from e_1 to $v.a$. In our running example, for every generated OpenFlow message, we can find the data sources which cause the messages. When diagnosing some suspicious messages, we can directly find the data sources of the messages, which could be the root causes. The *Causal Dependency Generator* sub-module maintains a list of all runtime objects which are fields of the state variables and the current data sources. After each operation, the *Causal Dependency Generator* sub-module may update the data sources of some objects. For example, a write operation will clear the previous data sources for the object and may build new data sources for this object.

4.4 Diagnosis

We design a command line tool for the users to query for recorded activities in the SDN framework. The usage of the tool is shown as the following:

$$Usage: \ Diagnosis \ [options]$$

The user can set up different options to satisfy their different query requirements. The option:

$$--query = trace|message|event|function|variable$$

tells the tool what to retrieve from the database and what to output. For all queries, our tool supports to set up a time filter:

$$--after = yyyyMMddHHmmss$$

$$--before = yyyyMMddHHmmss|now$$

By using the above two options, we can query for activities within a given time period. Our tool supports both fast querying for forward issues and querying for detailed activities. In the following we will explain how to use our tool to fast query for forwarding problems and how to query detailed activities.

Motivated from networking diagnosis tools, ForenGuard supports automatically querying for network forwarding problems including reachability, isolation, routing loop and way-point routing. Our tool provides an option:

$$--problem = routingloop|routingpath|waypoint$$

The argument *routingloop* is to detect routing loops and will output corresponding activities. The argument *routingpath* is to output the activities which are related to a certain network flow. To use this argument, the user should also specify the matching conditions for this network flow. For example, the user can use $--srcip$ and $--dstip$ to specify a flow between two ip addresses. Our tool currently supports to use the 5-tuple packet header to specify a network flow. This argument can verify both the conditions of reachability and isolation. The argument *waypoint* is to query for forwarding rules of certain traffic going through certain specific way_point. To use this argument, the user should specify both the network flow and the $--dpid$ of the way_point switch.

Users can also query for detailed activities through our tool. As shown previously, by using the $--query$ option, the users specify what kinds of activities they want to query. The user can use the argument *trace* for the corresponding execution traces, *message* for communication OpenFlow messages, *event* for event trigger and dispatch activities, *function* for function call activities and *variable* for variable access activities. The user can also set up several filters to specify what kinds of activities are needed. For example, to query for the execution traces that are relevant to a network flow whose source IP is 10.0.0.2 and destination port is 80, we can write:

$$--srcip = 10.0.0.2 \quad --dstport = 80$$

For messages, we can specify the application name and message types (PacketIn, FlowRemoved and etc.). Our tool is independent of controller types, programming language and hardware specifics.

Many network problems are caused by application crashes in the SDN control plane [37]. Unlike other types of root causes, the application crash does not directly output any harmful flow rules to the data plane. To diagnose this kind of problem, by showing the execution traces of the control plane, we can locate the crash point in the program first (e.g., in which function) and then list relevant activities in the execution trace. For example, many application crashes are caused by data races at instance variables [44]. From the execution traces, we can list the recent read/write operations of variables and check if there is data race happened.

4.5 Flexibility of Tuning Stored Activities

According to the modeling of the SDN activities in Section 3, by default our forensics function records all types of activities into

	Data Plane	Control Plane			
Activities	States	Functions	Variables	Events	OF Messages
Tunable	×	✓	✓	×	×
Data Size	28.6%	26.5%	11.5%	20.2%	13.2%

Table 2: Options to Tune the Recorded Activities

the database. To provide better flexibility, before deploying our system, we allow the users to tune their required types of activities to database storage (instead of all types) to reduce some storage overhead.

The options to tune the activities to be stored are shown in Table 2. To build the causal relationship of different activities, some types of activities are essential. For example, ForenGuard provides flow-level forensics and diagnosis. Thus the data plane states and the state transitions (i.e., OpenFlow messages) are necessary. To build the causal relationship between different modules/apps in the controller, the event dispatching and receiving information is also necessary. Other than these, other types of activities are tunable to be stored or not, because they are only used in the intermediate stages of building the information flows. According to the recorded data of several diagnosis cases shown in Section 5, we provide the rough percentage of data size of each type of activity in Table 2.

5 EVALUATION

In this section, we present the implementation details and the evaluation results of ForenGuard.

5.1 Implementation

Controller Module	# of Edges in the call-graph	# of State Variables	Instrumented Lines of Code
Forwarding	32	5	197
Hub	23	0	25
LearningSwitch	18	1	173
Topology	148	8	192
MacTracker	12	1	16
Firewall	27	1	145
LinkDiscovery	96	14	498

Table 3: Static Analysis and Instrumentation Results of Part of Controller Applications

We implement a prototype system of ForenGuard on top of the Floodlight [3] controller (Java language) version 1.0. ForenGuard extends the Soot [27] framework which provides the global control flow analysis, data dependency analysis and instrumentation function on the intermediate representation Jimple code of the controller. We separately analyze each module/application in Floodlight controller and set the event handler functions as the entry points for analysis. Our data dependency analysis is built on top of the flow-insensitive, context-sensitive and field-sensitive analysis using Soot Pointer Analysis Research Kit (SPARK).

Instrumentation: We do not instrument any statement which only accesses variables that are used for collecting system logs , debugging or providing interfaces. For read/write operations of state variables, we add instrumentation to log every read and write statement that accesses static and instance field variables on the heap memory. We observe that the SDN controller leverages heterogeneous storages for network state using complicated data types (e.g., the HashMap in the running example). For some methods of these kind of data types (e.g., HashMap.put()), the Jimple code would miss the read/write operations. This is because the analysis will not go through the HashMap.put() function and only consider this is a read operation (but actually a write operation). Therefore, we maintain a static mapping of those methods and their read/write operations for a set of commonly used data types. For example, we consider ArrayList.add() as a write operation. Besides, we log the memory access operation in a fine-grained field level (e.g., each entry of the hash map).

Event Dispatching: There are two types of event dispatching schemes in FloodLight controller, which are queue-based and observer-based. Queue-based schemes are mostly used for the Core Services to dispatch data plane events (e.g., `PacketIn Event`). Observer-based schemes are mostly used for inter-application event dispatch. For queue-based schemes, we log the write/read the global queue as ***Dispatch*** and ***Receive*** operations. For observer-based schemes, we log the statements of dispatching the events as the ***Dispatch*** operations and the invocations of handler functions as the ***Receive*** operations.

System Environment: We select MongoDB [6] as our database to store the activities and their causal relations. We use Mininet [5] to emulate the SDN data plane topologies. For the performance evaluation, we use Cbench [1] as a benchmark tool to generate OpenFlow messages. The setting of our host machines is dual-core Intel Core2 3GHz CPU running 64-bit Ubuntu Linux. We select some controller modules and show the static analysis and instrumentation results in Table 3.

5.2 Effectiveness Evaluation

Running Example: We first illustrate the forensics of the running example (mentioned earlier as Listing 1 in Section 2) and how ForenGuard helps diagnose the networking forwarding problem. If we observe that one host lost its network connectivity, we can use the `routing_path()` function to diagnose the issue. By calling the `routing_path()` function, ForenGuard can automatically find out the suspicious activities that cause the network problem. We visualize the activities that are recorded in the database (left side) and the result output by ForenGuard (right side) in Figure 7. To make the graph concise, we omit the timestamps and thread information of each activity and use numbers (instead of the actual names) to denote activity details (e.g., using f1,2,3... to show function calls). We can observe that, the two installed flow rules are the direct reason that causes the forwarding issue. Behind the two installed flow rules, there are four `PacketIn` events (Event1-4 in the figure) that are the potential root causes. By further checking the detailed information of these four events, we can reason where and why the events come from. Event 1 and 3 are triggered by the packet from the attacker to Host1 at Sw1 and Sw2. Event 4 and 2 are triggered by the response packet from Host1 to the attacker at Sw2 and Sw1. Therefore, we find Event 1 and 3 are the root causes of the issue and we can also locate the attacker. The figure shows that ForenGuard can significantly reduce the human effort to diagnose network forwarding problems.

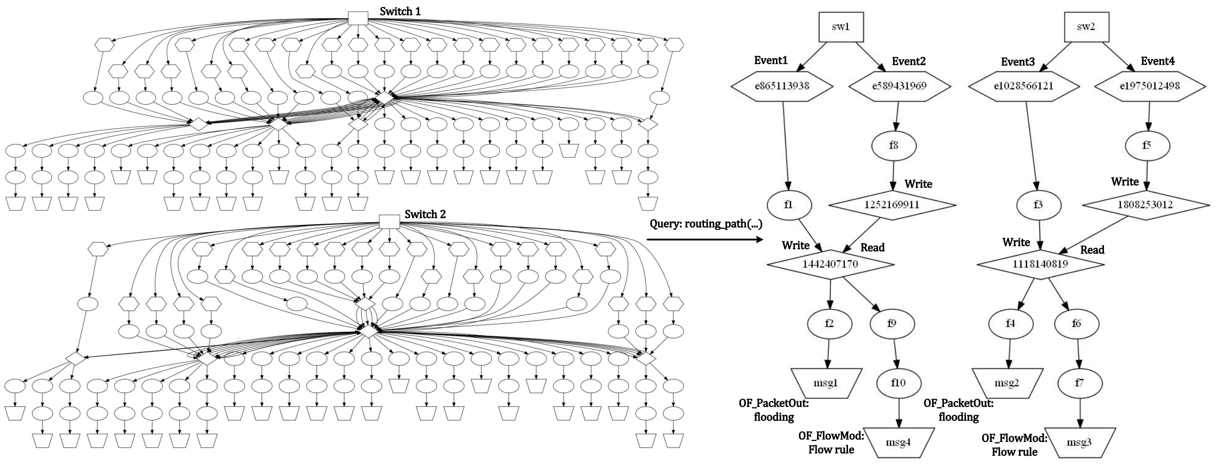


Figure 7: Simplified Dependency Graph of Execution Traces of the Running Example. Box denotes switches, Hexagon denotes events, Circle denotes function calls, Diamond denotes variable fields, Trapezium denotes OpenFlow messages.

Attack Code	Root Causes	Problem	# of Most Relevant Data Plane Activities	# of Most Relevant Control Plane Activities	# of Involved Applications
A1	Loss of LLDP Packets [35]	Routing Loop	6	18	5
A2	Race Condition [44]	Application Crash	3	9	2
A3	Link Fabrication [19]	Packet Loss	2	16	5
A4	Switch Table Flooding [28]	Disconnection	1	flooding	1
A5	Switch ID. Spoofing [28]	Disconnection	1	3	1
A6	Malformed Control Message [28]	Disconnection	1	3	1
A7	Control Message Manipulation [28]	Disconnection	1	3	1
A8	PacketIn Flooding [41]	Application Crash	flooding	flooding	6
A9	Host Location Hijacking [19]	Disconnection	2	14	1
A10	LoadBalancer Misconfiguration	Load Unbalanced	3	14	1
A11	Firewall Misconfiguration	Routing Loop	2	10	1

Table 4: Diagnosis Cases

Extended Evaluation: We reproduce 11 attack cases that cause network forwarding problems and use ForenGuard to diagnose the root causes. Most these attacks are reported from previous research [19, 28, 35, 41, 44]. Table 4 summarizes the cases and the observed problems from the data plane. Among these attacks, A3, A8 and A9 can be generated by an attacker from a compromised host. Attacks A1, A2, A4, A5 A6 and A7 are initiated from the data plane switches or man-in-the-middle attackers who can manipulate the control messages between the control plane and the data plane. Attacks A10 and A11 are from the north bound configuration of the controller through the REST interface. All the above attacks generate thousands of data plane activities and tens of thousands of control plane activities totally. To demonstrate how ForenGuard is helpful to diagnose the root causes, we also show the most relevant control and data plane activities that can identify the attacks after using ForenGuard to narrow down the recorded activities. The numbers of control/data plane activities show the most relevant activities after narrowing down from a large dataset of logs. Many attacks involve more than one application (e.g., A1), which means individually checking every application is hard to diagnose the root cause of these attacks. However, ForenGuard is able to find out the involved applications quickly and help to diagnose the problems.

By leveraging the simplified dependency graphs (e.g., the example in Figure 7) generated by ForenGuard, the network administrator can further pinpoint the root causes of each network forwarding problem. In the following, we show how an administrator can benefit from ForenGuard and pinpoint the root causes of two problems from Table 4 step by step.

Pinpoint the Problem in A3: Similar to the Host Location Hijacking attack in the running example, a malicious attacker can also launch Link Fabrication attack by poisoning state variables in some applications. Host 10.0.0.2 reports a packet loss problem to ForenGuard and the output results are shown in Figure 8. We omit some redundant activities and the the detailed information of most activities in the execution trace but remain the description of the important activities. From the results, we can first observe the flow rule that directly causes the packet loss problem. This flow rule is triggered by a `PacketIn` event and affected by a pre-generated routing decision. Then we can keep reasoning who makes this routing decision. The routing decision is triggered by a linkUpdate event, and this event is caused by a `PacketIn` event at Sw1 from port 3, which is the root cause of this packet loss issue. By further checking the details of this `PacketIn` event, we can see that this event is triggered by a faked LLDP packet from Port 3 of Sw1, which is where the attacker locates.

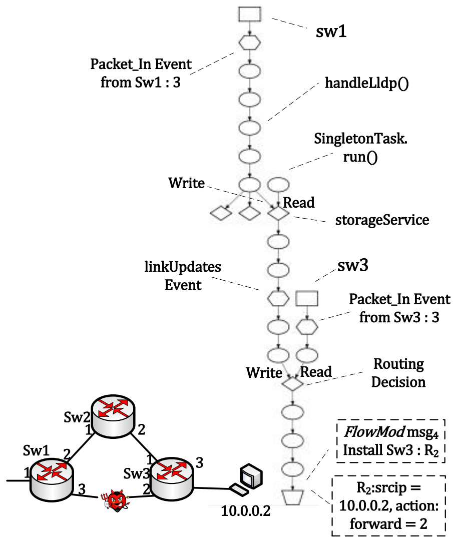

sw1
Packet_In Event from Sw1 : 3
handleLldp()
SingletonTask. run()
Write
Read
storageService
linkUpdates Event
sw3
Packet_In Event from Sw3 : 3
Write
Read
Routing Decision
FlowMod msg4 Install Sw3 : R2
R2:srcip = 10.0.0.2, action: forward = 2
Sw2
Sw1
Sw3
10.0.0.2

Figure 8: Diagnosing a Packet Loss Problem Using ForenGuard

Pinpoint the Problem in A11: There is a Firewall application in which users can configure firewall rules (e.g., block a black list of IPs). When the user observes a network disconnection from the data plane, he can report this problem to the network administrator by using the function `routing_path()`. The detailed output from ForenGuard are shown in Figure 10. The diagnosis process of ForenGuard is as follows: It will first search for forwarding graphs for the flows of the user and find the flow rules that drop the packets from this user. Then it keeps searching for the control plane execution traces that generate those messages. ForenGuard can quickly locate the Firewall application and observe the flow rule which drops the packets triggered by a new flow event and one entry of the variable *rules* which is configured from the REST API before.

5.3 Overhead and Scalability

ForenGuard instruments logging code into the controller and will add unavoidable overhead to the SDN control plane. To quantify the added overhead, we measure two performance metrics of the SDN controller with and without ForenGuard. One is the throughput overhead and the other is latency overhead, i.e., how much our system will affect the message processing throughput and latency of SDN controllers. To evaluate the throughput overhead, we use the Cbench tool to generate a large amount of new flow events and evaluate the maximum processing rate in the control plane. To evaluate the delay overhead, we make use of two frequent OpenFlow messages, `PacketIn` message and `StatsReq/Res` message. The `PacketIn` message is triggered by a new flow or a flow entry matching and sent from the data plane. The `StatsReq/Res` message is used for the control plane to query for flow stats from the data plane.

To measure the delay of processing `PacketIn` messages, we use a machine with two network cards to keep sending network packets through one network card to the network. The other network card of this machine is connected to the controller port of the switch and will receive the corresponding `PacketIn` messages. To measure the delay of processing `StatsRes` messages, we use the same machine to keep sending stats query messages to the data plane and measure the delay between the `StatsReq` and `StatsRes` messages.

Figure 9 shows the overhead evaluation results. Figure 9 (a) shows the throughput results with and without using ForenGuard. We can observe that ForenGuard decreases the throughput of the SDN control from 751.2 to 660.1 messages per second, i.e., about 12.1% decrease. Figure 9 (b) and (c) show the delay overhead when using ForenGuard. For `PacketIn` messages, the average processing time with and without ForenGuard is 0.886ms and 0.719ms, which means about 23.4% overhead. Similarly, for `StatsReq/Res` messages, the average processing time with and without ForenGuard is 1.12ms and 0.928ms, which means about 20.4% overhead. We think the overhead increased by ForenGuard is reasonably acceptable, especially compared with dynamic taint-tracking approaches which *normally suffer a slowdown of 2-10 times* [46].

The scalability results are important since network operators should decide how much computing and storage resources are needed to support ForenGuard. We measure the scalability of data generating rate in our system. The data generating rate measures how much data will be generated by our system and stored into the database. To measure the data generating rate, we use Mininet to emulate several network topologies (from a small size to a 10-switch topology). Every end host in the data plane will generate 10 new flow events (`PacketIn` messages) per second to the control plane. We keep running the system for around one hour per topology. Shown in Figure 11, the rate of logged data increases linearly with the size of the data plane. The workload of with about 1,000 new flows per second (the 10-switch topology) is comparable to the workload of typical enterprise networks [33]. For this workload, ForenGuard will averagely generate about 0.93GB data per hour into the database.

6 RELATED WORK

Digital Forensics: Digital forensics is a well studied research topic. In the past decade, research of network-level forensics focuses more on handling the large amount of data (storing, indexing and retrieval) in large-scale, complex networks. TimeMachine [30] records raw network packets and builds the index for the headers of the likely-interesting packets. Anteater [29] monitors the data plane state and uses formal analysis to check if the state violates specified invariants. Teryl et al. proposed a storage system [39] to efficiently build the index of payload information of network packets. VAST [40] is a platform that uses the actor model to capture different levels of network activities and provides a declarative language for query. Network provenance [49] is also a relevant research topic in recent years. The basic principle of ForenGuard is similar to network provenance, which is to track causality and capture diagnostic data at runtime that can be queried later. Unlike existing tools [14, 42] which mostly target declarative languages or require at least some manual annotations from software developers, ForenGuard can directly work on the general-purpose programming language (e.g., Java). On host-level forensics, Forenscope [13] proposes a framework that can investigate the state of a running operating system

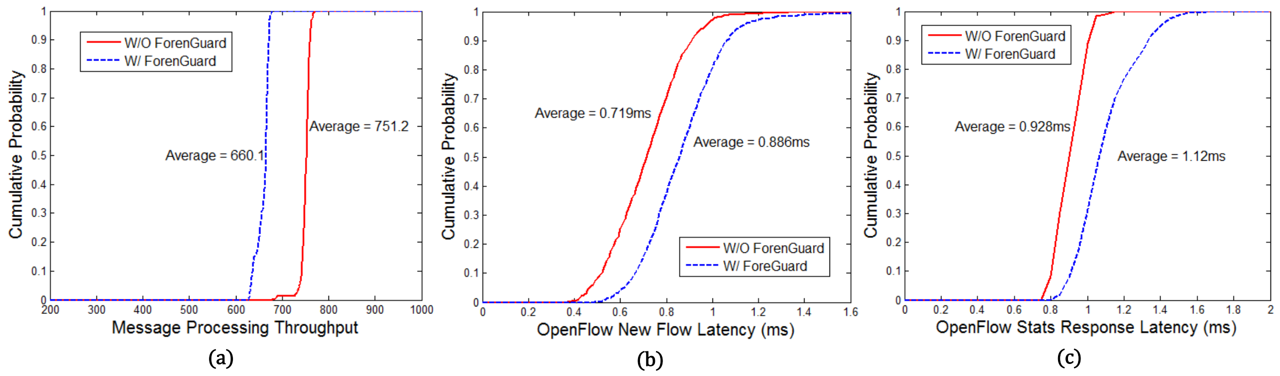


Figure 9: Overhead of ForenGuard

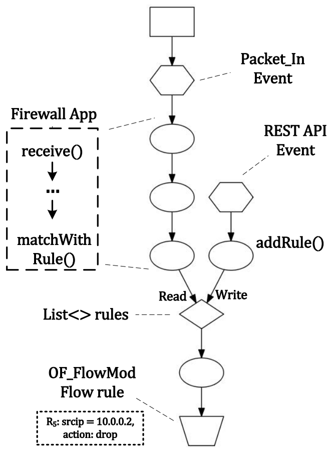


Figure 10: Diagnosing a Disconnection Problem Using ForenGuard

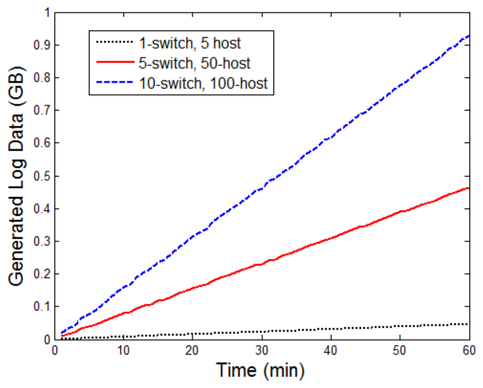


Figure 11: Log Data Generating Rate

without using taint or causing blurriness. BackTracker [26] records the files and processes in the operating system and builds them in a dependency graph for intrusion detection. Different from all above work, ForenGuard focuses on a unique context of SDN which decouples control and data planes and also requires both network and host level tracking.

SDN Troubleshooting: Peyman et al. [24] used packet header space analysis to statically check network specifications and configurations. Veriflow [25] and NetPlumber [23] verify network invariants dynamically when flow rules update. These verification approaches highly rely on the predefined invariant policies, but the lack of expressiveness can only help with known violations. OFRewind [43] can record and replay the communication messages between SDN control plane and data plane. STS [35] improves the delta-debugging algorithm that can generate a minimal sequence of inputs that can trigger a controller bug. However, the delta-debugging algorithm does not scale well with the network size and STS can only provide coarser-grained culprits. The most relevant work to our paper is NetSight [17] and path query [31]. NetSight [17] monitors packet history to analyze the data plane behaviors and troubleshoot the network. Path query [31] provides a query language for path-based traffic monitoring. Compared with NetSight, we directly record the activities of the control and data planes for troubleshooting. Also, unlike path query which provides the monitoring of network performance issues, our tool provides the monitoring and diagnosis of network forwarding/security issues.

SDN Security: SDN security gradually becomes a trending research topic in both academia and industry. Most existing work falls into two themes. The first theme makes use of the logically centralized control plane to implement security logic (e.g., monitoring and measurement [47, 48], access control [18, 32], firewall and IDS [20], DDoS detection [2, 16], security services composition [36]). The second theme focuses on the security challenges that are introduced by SDN itself. AVANT-GUARD [38] and FloodGuard [41] target on the denial-of-service threat to the control plane. FortNOX [34] proposes a security enforcement kernel to controllers. TopoGuard [19] detects and mitigates the topology poisoning attack caused by spoofed network packets from the attackers. Rosemary [37] enhances the resilience of the control plane by using a sandbox-based approach to prevent faulty applications from crashing the entire control plane. DELTA [28] introduces a fuzzing-based penetration testing framework for different controllers. ConGuard [44] detects harmful race conditions that could be exploited to launch attacks in the SDN controllers. Compared with SecureBinder [21] which targets a new attack which fools the network infrastructure devices (e.g., DHCP server), our tool targets the attacks that fool the SDN controller applications.

7 DISCUSSION AND FUTURE WORK

ForenGuard takes the first and significant step towards a network security forensics and diagnosis system in the SDN context. However, ForenGuard is still preliminary and has several limitations for future research work to improve, which we will discuss below.

Limitation on Threat Model. In this work, we do not assume malicious SDN apps in the first place because currently apps are well vetted before deployment due to their extreme importance to the operation of the entire networks. We also note that existing Java-based SDN apps leave less or no room for buffer overflow and code injection attacks. In the worst cases, even if an exploited malicious SDN app may directly attack ForenGuard, this could easily expose their existence; or they could intentionally generate fake executing logs to mislead the forensics function of ForenGuard, for which we think there are still anomalies that could be detected from code or behavior level. Nevertheless, we note that vetting/detecting malicious apps is a separate/orthogonal topic different from the forensic/diagnosis research targeted by this paper. Our future work will look into those issues.

Extension to Other Controllers and Distributed Controllers. ForenGuard leverages some generic principles used by all these controllers (e.g., how they dispatch and receive events), as well as some heuristics of the Java language (e.g., reasoning about reference data types like Set, List, Array and their methods according to Java 7). Therefore, we believe our technique is relatively generic and extensible to other mainstream Java-based controllers (e.g., OpenDaylight, ONOS) as well. However, we admit that it requires more efforts to implement our proposed approach to other non-Java controllers.

ForenGuard could also be extended to support different types of distributed controller models. For example, in the ONOS [7] model of distributed controllers, ForenGuard can work on each individual core/controller in the forensics stage, and then perform the diagnosis through the merged forensic data. This is one of our future work.

Accuracy of the Static Analysis. Our current implementation of ForenGuard relies on existing static analysis techniques in Soot. The techniques are known to be not 100% accurate. For example, the static data flow tracking is not flow-sensitive. However, we think the issue of the static analysis itself is beyond the scope of this study. ForenGuard is focusing more on what to forensic and how to diagnose security problems. However, our tool could also benefit from any future research in the area of improving static analysis.

Room for Optimization. Though ForenGuard provides the customizability to tune the recorded activity types, there is still room for the optimization of the storage. For example, ForenGuard could benefit from previous work (e.g., VAST [40]) which proposed several compression schemes for forensic data. In our future work, we will investigate more optimization schemes and study the proper design for our case.

8 CONCLUSION

In this paper, we propose ForenGuard, a first-in-its-kind SDN forensics and diagnosis tool that integrates both control and data planes, as well as both network and host level forensics and diagnosis. ForenGuard dynamically records fine-grained activities, builds them as *event-oriented execution traces* of the control plane and *state transition graphs* of the data plane, and provides diagnosis functions for users to locate the suspicious activities and pinpoint the root causes of the forwarding problems. The evaluation results show that ForenGuard is useful in SDN networks and only adds acceptable runtime overhead to the SDN control plane.

ACKNOWLEDGMENTS

This material is based upon work supported in part by the National Science Foundation (NSF) under Grant no. 1617985, 1642129, 1700544, and 1740791. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

REFERENCES

[1] Cbench Controller Benchmarker. https://github.com/andi-bigswitch/oflops/tree/master/cbench.
[2] DefenseFlow: SDN Applications and DDoS Attack Defense. http://www. radware.com/Products/DefenseFlow/.
[3] Floodlight Controller. http://www.projectfloodlight.org/floodlight/.
[4] LearningSwitch Application. https://github.com/floodlight/floodlight/blob/master/src/main/java/net/floodlightcontroller/learningswitch/LearningSwitch.java.
[5] Mininet: Rapid Prototyping for Software Defined Networks. http://yuba. stanford.edu/foswiki/bin/view/OpenFlow/.
[6] MongoDB. https://www.mongodb.com/.
[7] ONOS Controller Platform. https://onosproject.org/.
[8] OpenDayLight controller. https://www.opendaylight.org/.
[9] OpenFlow: Innovate Your Network. http://www.openflow.org.
[10] POX Controller. http://openflow.stanford.edu/ display/ONL/POX+Wiki.
[11] S. Crosby A. and D. S. Wallach. 2009. Efficient Data Structures for Tamper-evident Logging. In *Proceedings of the 18th Conference on USENIX Security Symposium (Usenix Security)*.
[12] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. 2012. A NICE Way to Test OpenFlow Applications. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[13] E. Chan, S. Venkataraman, F. David, A. Chaugule, and R. Campbell. 2010. Forenscope: a framework for live forensics. In *Proceedings of the 2010 Annual Computer Security Applications Conference (ACSAC)*.
[14] A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo. 2017. One Primitive to Diagnose Them All: Architectural Support for Internet Diagnostics. In *Proceedings of the Twelfth EuroSys Conference 2017 (EuroSys)*.
[15] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann. 2015. SPHINX: Detecting Security Attacks in Software-Defined Networks. In *Proceedings of the 22th Annual Network and Distributed System Security Symposium (NDSS)*.
[16] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey. 2015. Bohatei: Flexible and Elastic DDoS Defense. In *Proceedings of The 26th USENIX Security Symposium (Usenix Security)*.
[17] N. Handigol, B. Heller, V. Jeyakumar, D. MaziÃĺres, and N. McKeow. 2014. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[18] S. Hong, R. Baykov, L. Xu, S. Nadimpalli, and G. Gu. 2016. Towards SDN-Defined Programmable BYOD (Bring Your Own Device) Security. In *Proceedings of the 22th Annual Network and Distributed System Security Symposium (NDSS)*.
[19] S. Hong, L. Xu, H. Wang, and G. Gu. 2015. Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures. In *Proceedings of the 22th Annual Network and Distributed System Security Symposium (NDSS)*.
[20] H. Hu, W. Han, G. Ahn, and Z. Zhao. 2014. FlowGuard: Building Robust Firewalls for Software-defined Networks. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN)*.
[21] S. Jero, W. Koch, R. Skowyra, H. Okhravi, C. Nita-Rotaru, and D. Bigelow. 2017. Identifier Binding Attacks and Defenses in Software-Defined Networks. In *Proceeding of the 24th USENIX Security Symposium (USENIX Security)*.

[22] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, and W. Lee. 2017. Rain: Refinable Attack Investigation with On-demand Inter-Process Information Flow Tracking. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS).*

[23] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. 2013. Real Time Network Policy Checking using Header Space Analysis. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI).*

[24] P. Kazemian, G. Varghese, and N. McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI).*

[25] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. 2013. Veriflow: Verifying Network-Wide Invariants in Real Time. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI).*

[26] S. T. King and P. M. chen. 2003. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP).*

[27] P. Lam, E. Bodden, O. Lhotak, and L. Hendren. The soot framework for java program analysis: a retrospective. In *CETUS 2011.*

[28] S. Lee, C. Yoon, C. Lee, S. Shin, V. Yegneswaran, and P. Porras. 2017. DELTA: A Security Assessment Framework for Software-Defined Networks. In *Proceedings of The 2017 Network and Distributed System Security Symposium (NDSS).*

[29] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. 2011. Debugging the Data Plane with Anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM).*

[30] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider. 2008. Enriching Network Security Analysis with Time Travel. In *Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM).*

[31] S. Narayana, M. T. Arashloo, J. Rexford, and D. Walker. 2016. Compiling Path Queries. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI).*

[32] A. Nayak, A. Reimers, N. Feamster, and R. Clark. 2009. Resonance: Dynamic Access Control for Enterprise Networks. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking.*

[33] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney. 2005. A First Look at Modern Enterprise Traffic. In *Proceedings of the 2005 Internet Measurement Conference (IMC).*

[34] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu. 2012. A Security Enforcement Kernel for OpenFlow Networks. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN).*

[35] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H.B. Acharya, K. Zarifis, and S. Shenker. 2011. Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences. In *Proceedings of ACM SIGCOMM Computer Communication Review.*

[36] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson. 2013. FRESCO: Modular Composable Security Services for Software-Defined Networks. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS).*

[37] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang. 2014. Rosemary: A Robust, Secure, and High-Performance Network Operating System. In *Proceedings of the 21th ACM Conference on Computer and Communications Security (CCS).*

[38] S. Shin, V. Yegneswaran, P. Porras, and G. Gu. 2013. AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-Defined Networks. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS).*

[39] T. Taylor, S. E. Coull, F. Monrose, and J. McHugh. 2012. Toward Efficient Querying of Compressed Network Payloads. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC).*

[40] M. Vallentin, V. Paxson, and R. Sommer. 2016. VAST: A Unified Platform for Interactive Network Forensics. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI).*

[41] Haopei Wang, Lei Xu, and Guofei Gu. 2015. FloodGuard: A DoS Attack Prevention Extension in Software-Defined Networks. In *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).*

[42] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo. 2017. Automated Bug Removal for Software-Defined Networks. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI).*

[43] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldman. 2011. OFRewind: Enabling Record and Replay Troubleshooting for Networks. In *Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC).*

[44] L. Xu, J. Huang, S. Hong, J. Zhang, and G. Gu. 2017. Attacking the Brain: Races in the SDN Control Plane. In *Proceedings of The 26th USENIX Security Symposium (Usenix Security).*

[45] A. Yavuz, P. Ning, and M. Reiter. 2012. Efficient, compromise resilient and append-only cryptographic schemes for secure audit logging. *Financial Cryptography and Data Security* (2012).

[46] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. 2007. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS).*

[47] C. Yu, C. Lumezanu, V. Singh, Y. Zhang, G. Jiang, and H. V. Madhyastha. 2013. FlowSense: Monitoring Network Utilization with Zero Measurement Cost. In *Proceedings of the 14th International Conference on Passive and Active Measurement (PAM).*

[48] M. Yu, L. Jose, and R. Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI).*

[49] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. 2011. Secure Network Provenance. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP).*