Causal Analysis for Software-Defined Networking Attacks

Benjamin E. Ujcich
Georgetown University

Samuel Jero
MIT Lincoln Laboratory

Richard Skowyra
MIT Lincoln Laboratory

Adam Bates
University of Illinois at Urbana-Champaign

William H. Sanders
Carnegie Mellon University

Hamed Okhravi
MIT Lincoln Laboratory

Abstract

Software-defined networking (SDN) has emerged as a flexible network architecture for central and programmatic control. Although SDN can improve network security oversight and policy enforcement, ensuring the security of SDN from sophisticated attacks is an ongoing challenge for practitioners. Existing network forensics tools attempt to identify and track such attacks, but holistic causal reasoning across control and data planes remains challenging.

We present PICOSDN, a provenance-informed causal observer for SDN attack analysis. PICOSDN leverages fine-grained data and execution partitioning techniques, as well as a unified control and data plane model, to allow practitioners to efficiently determine root causes of attacks and to make informed decisions on mitigating them. We implement PICOSDN on the popular ONOS SDN controller. Our evaluation across several attack case studies shows that PICOSDN is practical for the identification, analysis, and mitigation of SDN attacks.

1 Introduction

Over the past decade, the software-defined networking (SDN) architecture has proliferated as a result of its flexibility and programmability. The SDN architecture decouples the decision-making of the *control plane* from the traffic being forwarded in the *data plane*, while logically centralizing the decision-making into a *controller* whose functionality can be extended through *network applications* (or *apps*).

SDN has been touted as an enhancement to network security services, given that its centralized design allows for complete oversight into network activities. However, the programmable nature of SDN creates new security challenges and threat vectors. In particular, the control plane's state and functionality can be maliciously influenced by data input originating from the data plane and apps. These *cross-plane* [13,24,41,49,53,62] and *cross-app* [8,52] attacks have significant security repercussions for the network's behavior, such as bypassing access control policies or redirecting data plane traffic. An adversary only needs to attack data plane hosts or apps, and does not have to compromise the controller.

In software-defined networks, as in traditional networks, security products such as firewalls and intrusion detection systems (e.g., Snort, Zeek/Bro, Splunk) must be deployed to continuously monitor potential security incidents. When these tools signal a security alert, the network operator must investigate the incident to diagnose the attack, establish possible root causes, and determine an appropriate response. This investigation stage is particularly essential when considering that security monitoring tools are notoriously prone to issuing false alarms [16]; however, in the case of SDN, the control plane and its novel attack vectors may also be implicated when incidents occur. To this end, recent network causality and provenance analysis tools have been proposed to aid in SDN forensics [15,52,55,61]. However, we argue that such tools have limitations in terms of providing the precise and holistic causal reasoning that is needed by investigators.

First, the control plane's causality (or provenance) model has a significant effect on the precision with which a practitioner can identify root causes. If the control plane's data structures are too coarse-grained or if the control plane uses long-running processes, this can lead to *dependency explosion* problems in which too many objects share the same provenance. That reduces the ability to identify precise causes.

Second, the control plane's decisions cause the data plane's configuration to change; the effects of the data plane's configuration on packets sent to the controller cause subsequent control plane actions. When such tools examine the control plane alone, the indirect causes of control plane actions that result from data plane packets will lead to an *incomplete*

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited. This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

dependency problem that ignores the data plane topology.

Third, a practitioner will want to know not only the root causes for an action but also the extent to which such root causes impacted other network activities. For instance, if a spoofed packet is found to be the attack vector for an attack, then the practitioner will want to investigate what else that spoofed packet influenced to understand whether other attacks and undesirable behavior have also occurred.

Overview We present PICOSDN, a tool for SDN attack analysis that mitigates the aforementioned dependency explosion and incomplete dependency challenges. PICOSDN allows practitioners to effectively and precisely identify root causes of attacks. Given evidence from an attack (*e.g.,* violations of intended network policies), PICOSDN determines common root causes in order to identify the extent to which those causes have affected other network activities.

PICOSDN's approach uses *data provenance*, a data plane model, and a set of techniques to track and analyze network history. PICOSDN records provenance graphically to allow for efficient queries over past state. Although similar network forensics tools have also used graphical structures [52, 55, 60], these tools' provenance models suffer from dependency explosion or incomplete dependency problems. To account for those challenges, PICOSDN performs fine-grained partitioning of control plane data objects and leverages app event listeners to further partition data and process execution, respectively. PICOSDN also incorporates the data plane's topology such that indirect control plane activities caused by data plane packets are correctly encoded, which mitigates incomplete dependencies. Finally, PICOSDN's toolkit reports the impacts of suspected root causes, identifies how network identifiers (*i.e.,* host identities) evolve over time, and summarizes how the network's configuration came to be.

We have implemented PICOSDN within the popular ONOS SDN controller [5]. Many telecommunications providers, such as Comcast, use ONOS or one of its proprietary derivatives. We evaluated PICOSDN by executing and analyzing recent SDN attack scenarios found in the literature and in the Common Vulnerabilities and Exposures (CVE) database. PICOSDN precisely identifies the root causes of such attacks, and we show how PICOSDN's provenance model provides better understanding than existing network tools do. Our implementation imposes an average overhead latency increase of between 7 and 21 ms for new forwarding rules, demonstrating PICOSDN's practicality in realistic settings.

Summary of Contributions Our main contributions are:

1. An approach to the **dependency explosion problem for SDN attack provenance** that utilizes event listeners as units of execution.
2. An approach to the **incomplete dependency problem for SDN attack provenance** that incorporates a data plane model and tracking of network identifiers.
3. The **design and implementation** of PICOSDN on ONOS to evaluate SDN attacks and to demonstrate PICOSDN's causal analysis benefits.
4. The **performance and security evaluations** of PICOSDN on recent SDN attacks.

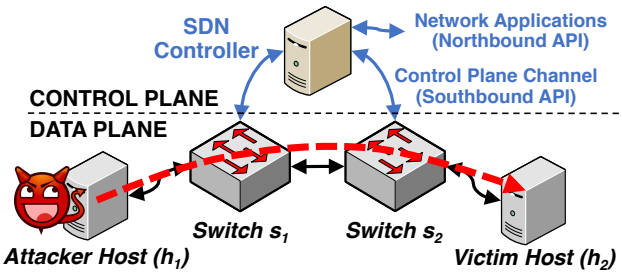


Figure 1: Topology of the CVE-2018-12691 attack scenario described in § 2.1. The red path represents the attacker's desired data plane communication from h_1 to h_2.

2 Background and Motivation

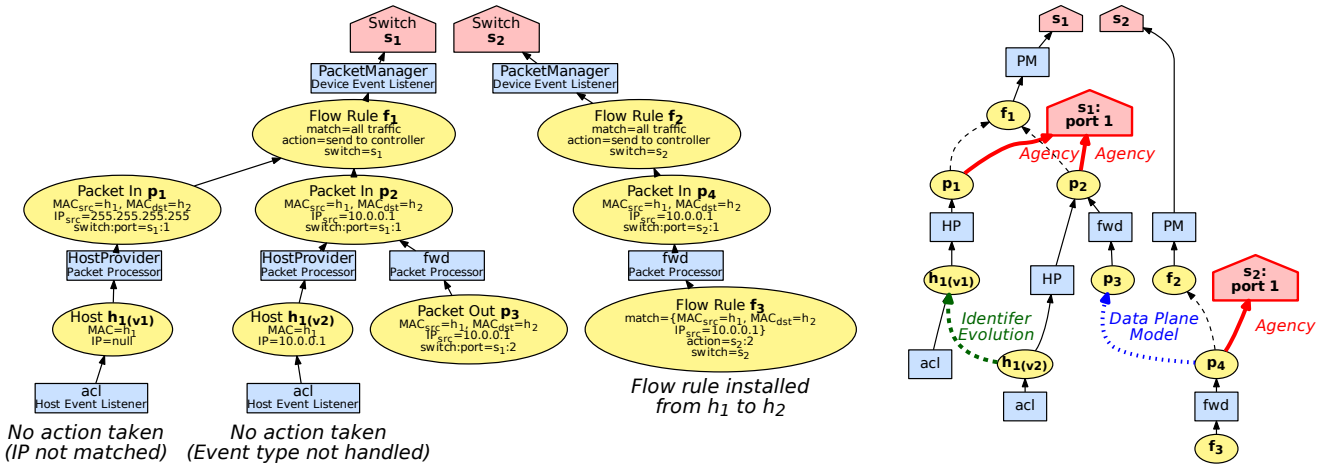
Many real-world SDN attacks leverage data plane dependencies and long-running state corruption tactics to achieve their goals. SDN controllers are susceptible to attacks from data plane hosts that poison the controller's network state view and cause incorrect decisions [13, 24, 41, 49, 53]. We consider a motivating attack to illustrate the limitations that a practitioner encounters when using existing network forensics tools.

2.1 Motivating Attack Example

Scenario Consider the control plane attack CVE-2018-12691 [53] in ONOS. It enables an attacker to use spoofed packets to circumvent firewall rules. This class of cross-plane attack leverages spoofed data plane input to fool the controller into maliciously changing the data plane forwarding. Complete prevention of such attacks is generally challenging, as spoofed information from data plane hosts is a notorious network security problem in SDN [13, 24, 28]. Such attacks can also be one part of a multi-stage attack in which the attacker's goal is to defeat the data plane access control policy and move laterally across data plane hosts to gain additional access [18].

Suppose that the attack is carried out on a network topology as shown in Figure 1. Assume that the controller runs a data plane access control application and a reactive[1] forwarding application. The attack works as follows. A malicious data plane host, h_1, wants to connect to a victim host, h_2, but the data plane access control policy is configured to deny traffic

[1] Although we discuss a reactive SDN configuration here as an example, PICOSDN's design generalizes to proactive SDN configurations, too. We refer the reader to § 8 for further discussion.



(a) Relevant provenance for the CVE-2018-12691 attack based on techniques from FORENGUARD [55]. The activities from switches s_1 and s_2 appear to be independent of each other, masking the derivation of a root cause of s_2's flow rule f_3 from host h_1's activities on switch s_1.

(b) Relevant provenance generated by PICOSDN for the same scenario as (a). This includes a data plane model, network identifiers, and precise responsibility (agency).

Figure 2: Provenance of the CVE-2018-12691 attack. Ellipses represent SDN control plane objects, rectangles represent SDN processes, and pentagons represent the SDN components responsible for each process or object (*i.e.,* the agency). The text of the labels in (b) are abbreviations from the text of the labels found in (a).

from h_1 to h_2 based on its IP address. The malicious host h_1 emits into the data plane a spoofed ICMP packet, p_1, with an invalid IP address. The controller creates a data structure, *the host representation object*, for h_1 with a valid MAC address but no IP address. The data plane access control application, acl, checks to see if it needs to insert new flow rules based on the data plane access control policy. As the controller does not associate h_1 with an IP address, no flow rules are installed.

Some time later, h_1 sends to h_2 a packet, p_2, with a valid source IP address. ONOS updates the host object for h_1 with h_1's actual IP address. Unfortunately, at this point, a bug stops the data plane access control application from handling events in which a host object is updated. Thus, the update never triggers the application to install flow deny rules that prevent h_1 from sending traffic to h_2. The result is that the reactive forwarding application forwards the packet out (p_3).

Environment In a typical enterprise environment, a variety of system- and network-layer monitoring tools are usually deployed [1, 17, 23, 45]). These services are largely reactive in nature, triggering threat alerts when a suspicious event occurs. After an alert is raised, it is then the responsibility of a network practitioner or security analyst to manually investigate the alert, determine its veracity, and determine an appropriate incident reponse. Threat investigation routines are carried out through the use of a variety of log analysis software, often referred to as *Security Indicator & Event Management* (*SIEM*) systems, (e.g., Splunk). Timely investigation of these alerts is critical, as failing to respond promptly can increase the attackers' dwell time and, therefore, the damage inflicted.

Investigation Some time later, a network practitioner is alerted to a suspicious event within the network—the *intrusion detection system* (*IDS*) has detected a large data transmission from from host h_1 to a known malicious domain. Unbeknownst to the practitioner, this flow represents an exfiltration of sensitive data from host h_2 to the open network via h_1, violating the intended data plane access control policy. As the practitioner begins to investigate the alert, they notice that a new flow rule was recently added between h_1 and h_2, but it isn't clear *how* or *why* this network reconfiguration occurred.

To understand the context of this change to the control plane, the practitioner attempts to perform causal analysis using a *provenance graph* over the control plane's past state, which is depicted in Figure 2a. As the practitioner now knows that a flow rule from h_1 and h_2 seems to have coincided with the security incident, they use this as an initial piece of *evidence*: a flow rule (f_3) was installed that allowed traffic from h_1 to h_2 on switch s_2. The practitioner then issues a query and identifies a set of possible root causes related to the lineage of that flow rule.

2.2 Existing Tool Limitations

However, the practitioner runs into several challenges when using existing tools to generate a graph such as the one in Figure 2a. Although linking h_1's packets to s_1's default flow rule (*i.e.,* f_1) does capture past causality, the practitioner is easily overwhelmed when *all* packets over all time from any of s_1's ports are also linked to that default flow rule. The practitioner also finds that switches s_1 and s_2 as principal agents become

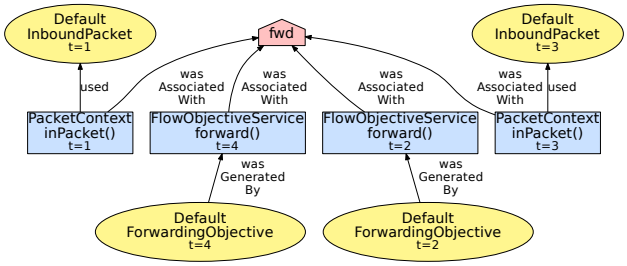


Figure 3: API-based provenance, based on techniques from ProvSDN [52], produces dependency explosion. When an app's event listener (fwd) is modeled as one long-running process, all API calls are considered as possible dependencies. For instance, the API call at time $t = 4$ may incorrectly appear to be dependent on all API calls from $t = [1,3]$.

too coarse-grained to enable pinpointing of attribution. Since existing tools do not account for the data plane as a causal influence, the result in Figure 2a is a set of two disconnected subgraphs. That disconnection prevents the practitioner from performing a meaningful backward trace. Finally, backward tracing alone would not provide the practitioner with details about the attack's other effects. We generalize those challenges and consider them in depth below.

Limitation (L1): Dependency explosion Provenance modeling suffers from the *dependency explosion problem* in which long-running processes or widely-used data structures within a system can create false dependencies. For instance, ProvSDN [52] uses an API-centric model. Figure 3 shows the provenance generated from two different calls to fwd's event handler, which results in four API calls in total. It is not obvious that an API call to forward() was initiated by one (and only one) API call to inPacket(). As a result, the API-centric model would create many false dependencies because an API call would be falsely dependent on all previous API calls.

ForenGuard's event-centric model uses execution partitioning, but if we apply it as shown in Figure 2a, we see that a controller that installs default flow rules (*i.e.,* f_1) will cause all unmatched packets (*i.e.,* p_1 and p_2) to become dependent on it. As a result, ForenGuard's modeling approach can suffer from data partitioning challenges when too many unrelated effects of a root cause must also be analyzed.

Limitation (L2): Coarse-grained responsibility and false attribution A similar challenge exists in the assignment of responsibility (or agency) in the data plane. In Figure 2a, the agency traces back to a switch, either s_1 or s_2. Although this correctly implies that one of the root causes of the attack is s_1 or s_2, it is not a particularly useful insight because all other activities have one of these root causes, too. Instead, should the responsibility be assigned to a notion of a host? Given that network identifiers (*e.g.,* MAC addresses) are easily spoofable, assigning agency to hosts would not solve the problem either; malicious hosts would simply induce false dependencies in the provenance graph.

Limitation (L3): Incomplete dependencies In contrast to false dependencies, *incomplete dependencies* occur when the provenance model does not capture enough information to link causally related activities. For SDN attacks, that occurs when the data plane's effects on the control plane are not captured by an implicit *data plane model*. In our attack scenario in § 2.1, the reactive forwarding application reacts to activities from switch s_1 before forwarding the packet (*i.e.,* p_3) out to other ports. On the other end of one of s_1's ports, switch s_2 receives that incoming packet (*i.e.,* p_4) and further processes it. Figure 2a's disconnected subgraphs appear to show that switch s_1's history of events is independent of switch s_2's history of events. Thus, if a practitioner were starting their investigation from a flow rule on switch s_2, they would not be able to see that the root cause occurred because of earlier events related to switch s_1 and the malicious host h_1's spoofed packets. ProvSDN and ForenGuard do not account for this kind of data plane model and would thus suffer from incomplete dependencies. Other tools [11, 57, 59, 61] model the implicit data plane, but are applicable only in the declarative networking paradigm. Most of the popular SDN controllers such as Floodlight, ONOS, and OpenDaylight, in contrast, use an operating-system-like imperative paradigm.

Limitation (L4): Interpretation and analysis Even if the dependency-related challenges previously described were mitigated, it can still be challenge to interpret provenance graphs. For instance, if the practitioner in our attack scenario from § 2.1 wanted to understand how network identifier bindings (*e.g.,* the network's bindings between a host's MAC address and its location in the data plane) changed over time, the provenance graph in Figure 2a would not support that; it does not directly link the host objects because their generation were not causally related.

ProvSDN and ForenGuard use *backward tracing* to start with a piece of evidence and find its information flow ancestors or set of root causes, respectively. However, if the practitioner wanted to know the other effects of the spoofed packet generated by h_1, that analysis would require *forward tracing* techniques that start at a cause and find its progeny to determine what other data and processes were affected. As neither ProvSDN nor ForenGuard performs forward tracing, the practitioner would not be able to discover other relevant unexpected artifacts of the attack, such as acl's failure to generate flow deny rules.

The practitioner ultimately wants to answer network connectivity questions of the form "Which packet(s) caused which flow rule(s) to be (or not to be) installed?" However, the SDN controller's event-based architecture can be itself complex [53]. Although the complexity must be recorded to

maintain the necessary dependencies, most of the complexity can be abstracted away to answer a practitioner's query. Thus, abstracted *summarization* is necessary for practitioners to understand attacks easily and quickly.

2.3 Our Approach

Motivated by the attack presented in § 2.1 and the previous tools' limitations noted in § 2.2, we highlight how PICOSDN would mitigate the issues. PICOSDN uses a provenance model that accounts for data and execution partitioning with precise agency, while also incorporating the implicit data plane effects on the control plane (§ 3). PICOSDN also provides techniques to aid in analysis (§ 5).

Applying PICOSDN produces the graph shown in Figure 2b. Rather than rely solely on the default flow rule f_1 as a cause, the practitioner can see that packets p_1 and p_2 originate at a host on switch s_1's port 1 (**L1**). That also allows the practitioner to precisely identify agency at the switch port (rather than switch) level (**L2**). The previously independent activities from each switch are linked by the data plane model that connects p_4 with p_3 (**L3**), which allows the practitioner to backtrace from s_2 to s_1 (**L4**). Finally, the practitioner can see how host h_1's network identifier information evolved over time (**L4**) and can summarize the past network state (**L4**).

3 PICOSDN Provenance Model

In order to reason about past activities and perform causal analysis, we first define a *provenance model* that formally specifies the relevant data, processes, and principal identities involved in such data's generation and use.[2] Our unified approach accounts for app, control, and data plane activities, which allows us to reason holistically about SDN attacks.

3.1 Definitions

A *provenance graph*, denoted by $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, is a directed acyclic graph (DAG) that represents the lineages of objects comprising the shared SDN control plane state. Informally stated, the graph shows all of the relevant processes and principal identities (*i.e.,* agents) that were involved in the use or generation of such control plane objects. We use the graph to analyze past activities to determine root causes (*i.e.,* backward tracing) and use those root causes to determine other relevant control plane activities (*i.e.,* forward tracing).

Each node $v \in \mathcal{V}$ belongs to one of three high-level classes: Entity, Activity, and Agent. Each high-level node class is explained with its respective subclasses in Table 1. We detail the design choices and semantics of these nodes in § 3.2. A node may also contain a dictionary of key–value pairs.

[2] Our model is loosely based on the W3C PROV data model [44].

Table 1: Nodes in the PICOSDN provenance graph model.

Node class	Node meaning and node subclasses
Entity	A data object within the SDN control plane state, used or generated through API service calls or event listeners *Subclasses:* Host, Packet (*subsubclasses*: PacketIn, PacketOut), FlowRule, Objective, Intent, Device, Port, Table, Meter, Group, Topology, Statistic
Activity	An event listener or a packet processor used by an SDN app or controller *Subclasses:* EventListener, PacketProcessor
Agent	An SDN app, an SDN controller core service, a switch port, or a switch (*i.e.,* device) *Subclasses:* App, CoreService, SwitchPort, Switch

Each edge (or relation) $e \in \mathcal{E}$ belongs to one of the classes listed in Table 2; rows that are indented show relations that have more precise subclasses and meanings from their superclass. Relations form the connections among the control plane objects, the network activities involved in their generation and use, and principal identities within the SDN components.

A *backward trace path*, denoted by $t_b = \langle v_0 \to e_0 \to \cdots \to e_i \to v_j \rangle, e_0 \ldots e_i \in \mathcal{E}_{\text{class}\neq\text{wasRevisionOf}}, v_0 \ldots v_j \in \mathcal{V}$, is a path of alternating nodes and edges that begins at a node of interest v_0 and ends at an ancestry node v_j. An ancestry node is a predecessor of a node of interest. Given that $\mathcal{G}$ is a DAG, nodes $v_1, \ldots, v_{j-1}$ are also ancestry nodes. A backward trace does not include any wasRevisionOf edges because such edges represent non-causal relations.

A *revision trace path*, denoted by $t_r = \langle v_0 \to e_0 \to \cdots \to e_i \to v_j \rangle, e_0 \ldots e_i \in \mathcal{E}_{\text{class}=\text{wasRevisionOf}}, v_0 \ldots v_j \in \mathcal{V}$, is a path of edges that begin at a node of interest v_0 and show the revisions of that node's object starting from an earlier revision node v_j. These revisions are non-causal and are used to identify changes to objects over time.

3.2 Model design choices

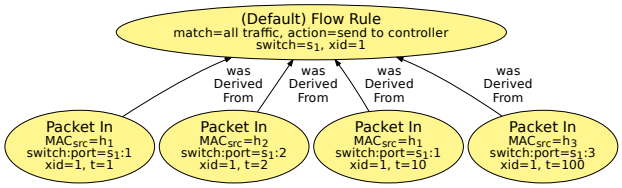
Given the aforementioned definitions, we now discuss the design decisions we made in PICOSDN's provenance model. We show how these decisions were influenced by the limitations found in previous work and how these decisions help us solve the challenges outlined in § 2.2.

Data and execution partitioning We achieve data partitioning with Entity objects by partitioning the data objects specified in the controller's API. For instance, the ONOS controller's host core service provides the API call getHosts(), which returns a set of Host objects. Thus, a natural way to partition data is to identify each Host object as a data partition. The Entity subclasses are generalizable to common SDN control plane state objects as found in the representative ONOS, OpenDaylight, and Floodlight SDN controllers.

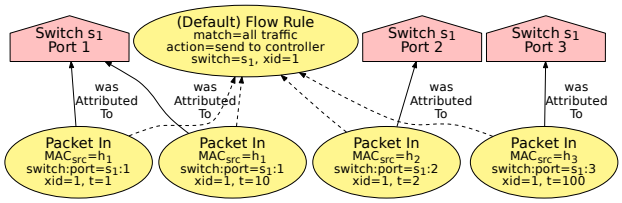
Default flow rules can generate dependency explosions because any incoming packet that does not match other flow

Table 2: Edges (relations) in the PICOSDN provenance graph model.

Valid edge (relation) class	Relation meaning
Entity wasGeneratedBy *Activity*	Creation of an SDN control plane state object
Activity used *Entity*	Use of an SDN control plane state object
EventListener used *Entity*	An event listener's use of the SDN control plane state object
PacketProcessor used *Packet*	A packet processor's use of a data plane packet
Entity wasInvalidatedBy *Activity*	Deletion of a data object within the SDN control plane state
Entity wasDerivedFrom *Entity*	Causal derivation of one SDN control plane state object to another object
PacketIn wasDerivedFrom *FlowRule*	Causal derivation of an incoming packet based on a previously-installed flow rule (*e.g.,* default flow rule)
PacketIn wasDerivedFrom *PacketOut*	Causal derivation of an incoming packet from one switch based on the outgoing packet of another switch
Entity wasRevisionOf *Entity*	Non-causal revision (*i.e.,* new version) of an SDN control plane state object
Activity wasAssociatedWith *Agent*	Agency or attribution of an SDN control plane event
Packet wasAttributedTo *SwitchPort*	Agency or attribution of a data plane packet with the respective switch port on which the packet was received



(a) Data dependency explosion using default flow rules (used in PROVSDN [52] and FORENGUARD [55]). All packets from switch s_1 that do not match any other flow rules become causally dependent on the default flow rule, which leads to dependency explosion.

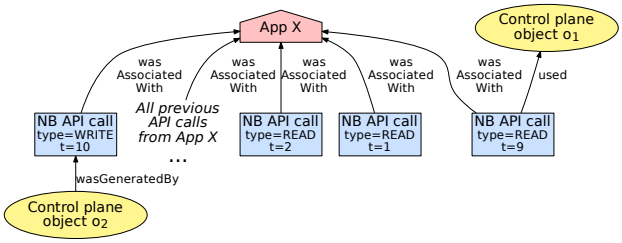


(b) Data partitioning using packets and switch port agents (used in PICOSDN). All packets per switch port are logically grouped together.

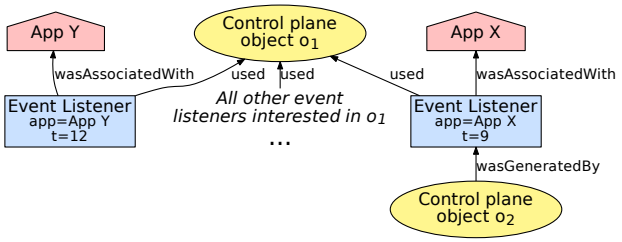
Figure 4: Data partitioning models for flow rules. Ellipses represent Entity nodes, and pentagons represent Agent nodes.

rules is sent to the controller for processing. All previously unseen packets become causally dependent on a generalized default flow rule, as shown in Figure 4a. To mitigate that problem, our model links any such packets to the respective edge ports that generated the packets, as shown in Figure 4b.

We achieve execution partitioning with Activity objects by partitioning each execution of recurring event listeners and packet processors into separate activities. Figure 5 shows the differences between API-based modeling and event-based modeling. With event-based modeling, we can more clearly show which Entity objects were used, generated, or invalided by a given Activity and mitigate the dependency explosion.



(a) API-based modeling (used in PROVSDN [52]). If one is tracing o_2's provenance via the API write at time $t = 10$, it will not be clear that *only* the API read of o_1 at $t = 9$ is causally associated with o_2. The other API reads at $t = 1$ and $t = 2$ represent false dependencies.

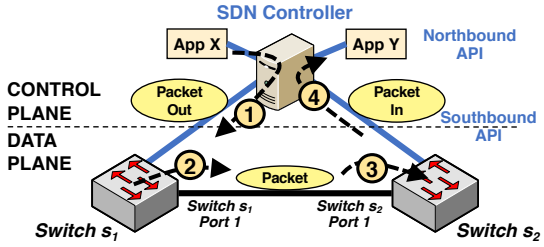


(b) Event-based modeling (used in PICOSDN). If one is tracing o_2's provenance via the event listener, it will be clear that o_2 is causally associated with o_1 through App X's event listener.

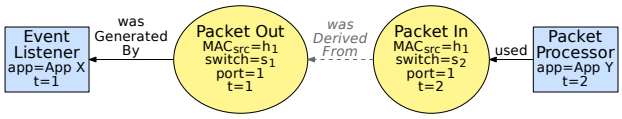
Figure 5: Comparison of execution partitioning models. Ellipses represent Entity nodes, rectangles represent Activity nodes, and pentagons represent Agent nodes.

Event listening SDN controllers dispatch events to event listeners. In ONOS, for example, the host service dispatches a HostEvent event (with the corresponding Host object) to any HostEvent listener. We model an event's data object as an Entity node that was used by EventListener nodes, with each event listener invocation represented as its own node.

Data plane model Figure 6 shows a diagram of data plane activities between two switches, s_1 and s_2. Figure 6a shows



(a) Control plane → data plane → control plane activity.



(b) Resulting control plane provenance graph. The dashed edge represents the provenance if we include a data plane model. Without the edge (and the data plane model), the PacketIn from s_2 would not appear to be causally dependent on PacketOut from s_1; that represents an incomplete dependency.

Figure 6: Data plane model. **1:** App X instructs the controller to emit a data plane packet from switch s_1. **2:** Switch s_1 emits the data plane packet on its link towards switch s_2. **3:** Switch s_2 receives the incoming data plane packet and sends it to the controller. **4:** App Y processes the data plane packet.

the temporal order of a control plane activity (*i.e.,* generation of an outgoing data plane packet), followed by a data plane activity (*i.e.,* transmission of a data plane packet), followed by another control plane activity (*i.e.,* processing of an incoming data plane packet). As shown in Figure 6b, a provenance model without the implicit causality of the data plane shows two separate subgraphs, which makes it impossible to perform a causally meaningful backward trace.

To mitigate that problem, we use a *data plane model* that includes the network's topology and related happens-before relationships among activities. Our provenance model includes a data-plane-based causal derivation in the relation *PacketIn* wasDerivedFrom *PacketOut* to represent the causality.

Network identifiers Control plane objects generated from data plane hosts pose a unique attribution challenge. Data plane hosts can spoof their principal identities, or *network identifiers*, relatively easily in SDN [28] as a result of network protocols (*e.g.,* the Address Resolution Protocol) that do not provide authentication and SDN controller programs that naïvely trust such information [53]. Ideally, each data plane host would have its own principal identity, but that is impossible if hosts can spoof their network identifiers.

To mitigate that problem, our provenance model offers two features: *edge ports as principal identities* and *network identifier revisions*. To enable those abilities, we model each edge port[3] as a principal identity, or Agent node; Figure 4b shows an example. As we assume in our threat model (described in detail in § 4) that switches are trusted, we can trust that the data plane traffic originating in a particular switch port is actually originating in that port. Whether or not a host claiming to have a particular identifier (*e.g.,* MAC address) on that port is legitimately located on that port cannot be verified from the data plane alone. To account for that, we model identifier changes by using the non-causal relation wasRevisionOf. It allows for a succinct trace of identifier changes over time.

[3] As opposed to an internal port that links a switch with another switch.

4 PICOSDN Threat Model

We assume that the SDN controller is trusted but that its services and functionality may be subverted by apps or by data plane input, which is similar to the threat model found in related work [52, 55]. Attackers will try to influence the control plane via *cross-app* poisoning attacks [52] or via *cross-plane* poisoning attacks [13,24,41,49]. As a result, we assume that all relevant attacks will make use of the SDN controller's API service calls, event dispatches, or both.

We further assume that switches and apps maintain their own principal identities and cannot spoof their identifiers, and indeed we can enforce that policy using a public-key infrastructure (PKI) [47]. However, we assume that data plane hosts *can* spoof their network identifiers (*e.g.,* MAC address).

5 PICOSDN Design

Based on the provenance model described in § 3, we now present the design of **p**rovenance-**i**nformed **c**ausal **o**bservation for **s**oftware-**d**efined **n**etworking, or PICOSDN. PICOSDN provides fine-grained data and execution partitioning to aid in the identification of SDN attack causes. PICOSDN's analysis capabilities allow a practitioner to identify evidence of malicious behavior, to pinpoint common causes, and to identify the extent to which other malicious activities have occurred.

Figure 7 shows an overview of the PICOSDN architecture. PICOSDN has two phases: a *runtime phase* (§ 5.1) that collects relevant provenance information during execution, and an *investigation phase* (§ 5.2) that analyzes the provenance.

PICOSDN is designed with the following goals in mind:

G1 *Precise Dependencies.* PICOSDN should reduce the units of execution to remove false execution dependencies that arise from long-running processes in the SDN control plane. PICOSDN should also reduce the unit size of data to remove false data dependencies.

G2 *Unified Network Model.* PICOSDN should leverage control *and* data plane activities, and thereby mitigate the incomplete dependency problem.

G3 *Iterative Analysis.* PICOSDN should perform backward and forward tracing to enable causal analysis of SDN attacks. It should efficiently summarize network activities and network identifier evolution.

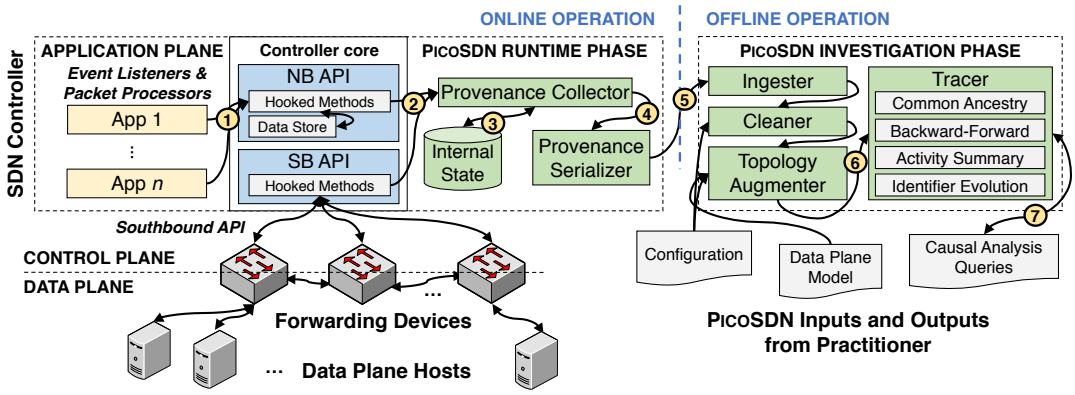


Figure 7: PICOSDN architecture overview with example workflow. **1:** An app makes an API call. **2:** PICOSDN's API hooks register the API call. **3:** The provenance collector checks its internal state and makes changes based on the API call. **4:** The provenance serializer generates the relevant graph. **5:** The ingester, cleaner, and topology augmenter prepare the graph. **6:** The tracer receives the graph. **7:** The tracer answers causal analysis queries based on the graph.

G4 *Activity Completeness.* PICOSDN should observe and record any apps, controller, or data plane activity relevant to network activities to ensure that it serves as a control plane reference monitor.

5.1 Runtime Phase

During the network's execution, PICOSDN's runtime phase records control plane activities in its *collector* and transforms them into a lightweight graph by using its *serializer*.

Collector The provenance collector consists of three components: *wrappers* around event dispatches and packet processors, *hooks* on API calls, and an *internal state* tracker.

We have instrumented wrappers around the SDN controller's event dispatcher and packet processor. The provenance collector uses these wrappers to maintain knowledge about which event listener or packet processor is currently handling the dispatch or processing, respectively; this achieves goal **G1**.

We have instrumented hooks on each of the SDN controller's API calls; this achieves goal **G4**. For a single-threaded controller, the reconstruction of the sequence of events, packets, and API calls is straightforward. However, in modern multi-threaded controllers, we also need a concurrency model to correctly link such calls to the right events. For event dispatching, we assume the following concurrency model: a particular event, ε_1, is processed *sequentially* by each interested event listener (*i.e.,* ε_1 is processed by listener l_1, then by l_2); different events, ε_1 and ε_2, may be processed *concurrently* (*i.e.,* ε_1 is processed by listener l_1 followed by l_2, while concurrently ε_2 is processed by listener l_3 followed by l_4). That is the model used by ONOS[4], among other SDN controllers. It allows PICOSDN's provenance collector to use hooks to correctly determine whether a particular API call should link the use or generation of control plane objects to the event listener (or packet processor) in execution at that time. Hooking the API calls and linking them with the event and packet wrappers in this way not only permits a transparent interposition over all app and data plane interactions with the control plane, but also avoids the limitations of prior work [55] that requires app instrumentation.

The provenance collector includes an internal state tracker that maintains knowledge of current events and control plane objects to detect when such objects change. The internal state is necessary to keep track of ephemeral objects' uniqueness that would not necessarily be captured by raw logging alone. (See § 8 for a discussion about internal state storage costs and external provenance storage costs.)

Serializer Once the provenance collector has determined the correct provenance based on context, the provenance serializer writes out a lightweight serialized graph of nodes and edges.

5.2 Investigation Phase

At some later point in time, PICOSDN's investigation phase uses the lightweight serialized graph as a basis for analysis. The *ingester* de-serializes the graph, the *cleaner* removes unnecessary provenance, and the *topology augmenter* incorporates the data plane model. The *tracer* answers practitioner queries. Each component is designed to be modular.

5.2.1 Ingester, Cleaner, and Topology Augmenter

The ingestor reads in the serialized graph. As most nodes contain additional details, the graph ingestor de-serializes the

[4]ONOS maintains several event dispatch queues based on the event type, and each queue is implemented in a separate thread. Given that listeners process a particular event sequentially, ONOS's event dispatcher sets a hard time limit for each event listener to avoid indefinite halting.

Algorithm 1 Data Plane Model

Input: graph $\mathcal{G}$, data plane topology states $\mathcal{D}_{set}$, time window τ_w, headers fields to match on H
Output: graph with data plane model $\mathcal{G}$
Initialize: $(\mathcal{V},\mathcal{E}) \leftarrow \mathcal{G}$
1: **for each** $\mathcal{D} \in \mathcal{D}_{set}$ **do**
2: $(\mathcal{N},\tau_{start},\tau_{end}) \leftarrow \mathcal{D}$ ▷ Data plane topology graph $\mathcal{N}$, epoch start τ_{start}, epoch end τ_{end}
3: $(\mathcal{N}_{switches},\mathcal{N}_{links}) \leftarrow \mathcal{N}$
4: **for each** $p_{in} \in \mathcal{V}_{\text{class=PacketIn}}$ **do** ▷ Packet p_{in}
5: **if** $\tau_{start} < p_{in}.\text{ts} < \tau_{end}$ **then** ▷ Timestamp p_{in}.ts
6: **for each** $p_{out} \in \mathcal{V}_{\text{class=PacketOut}}$ **do**
7: **if** $(p_{out}.\text{switch}, p_{in}.\text{switch}) \in \mathcal{N}_{links}$ **then**
8: **if** $p_{out}.\text{H} = p_{in}.\text{H}$ **then**
9: **if** $p_{out}.\text{ts} < p_{in}.\text{ts}$ and $p_{in}.\text{ts} - p_{out}.ts \leq \tau_w$ **then**
10: $\mathcal{V} \leftarrow \mathcal{V} \cup \{(p_{in}, p_{out})\}$
11: $\mathcal{G} \leftarrow (\mathcal{V},\mathcal{E})$
12: **return** $\mathcal{G}$

Algorithm 2 Common Ancestry Trace

Input: graph $\mathcal{G}$, evidence set N
Output: agent set Ag, activity set Ac, and entity set En
Initialize: $(\mathcal{V},\mathcal{E}) \leftarrow \mathcal{G}, Ag \leftarrow \emptyset, Ac \leftarrow \emptyset, En \leftarrow \emptyset, A \leftarrow \mathcal{V}$
1: **for each** $e \in \mathcal{E}$ **do** ▷ Remove non-causal edges
2: **if** e is a wasRevisionOf edge **then**
3: $\mathcal{E} \leftarrow \mathcal{E} \setminus \{e\}$
4: **for each** $n \in N$ **do** ▷ Evidence n (note: $n \in \mathcal{V}, N \subset \mathcal{V}$)
5: $A_n \leftarrow \text{getAncestors}((\mathcal{V},\mathcal{E}),n)$ ▷ Set of ancestor nodes A_n
6: $A \leftarrow A \cap A_n$ ▷ Common ancestor set A
7: **for each** $a \in A$ **do** ▷ Common ancestor a
8: **if** a is an Agent node **then**
9: $Ag \leftarrow Ag \cup a$
10: **else if** a is an Activity node **then**
11: $Ac \leftarrow Ac \cup a$
12: **else**
13: $En \leftarrow En \cup a$
14: **return** (Ag, Ac, En) ▷ $Ag \subset \mathcal{V}, Ac \subset \mathcal{V}, En \subset \mathcal{V}$

Algorithm 3 Iterative Backward-Forward Trace

Input: graph $\mathcal{G}$, evidence n, root r
Output: affected difference function $\Delta : \mathcal{V} \rightarrow \mathfrak{P}(\mathcal{V})$
Initialize: $(\mathcal{V},\mathcal{E}) \leftarrow \mathcal{G}; \Delta(i) \leftarrow \emptyset, \forall i \in \mathcal{V}$
1: **for each** $e \in \mathcal{E}$ **do** ▷ Remove non-causal edges
2: **if** e is a wasRevisionOf edge **then**
3: $\mathcal{E} \leftarrow \mathcal{E} \setminus \{e\}$
4: $A_n \leftarrow \text{getAncestors}((\mathcal{V},\mathcal{E}),n)$ ▷ Evidence's ancestor set A_n
5: $D_r \leftarrow \text{getDescendants}((\mathcal{V},\mathcal{E}),r)$ ▷ Root's descendant set D_r
6: $\mathcal{V}_{intermediate} \leftarrow A_n \cap D_r$
7: **for each** $v_i \in \mathcal{V}_{intermediate}$ **do**
8: $\Delta(i) \leftarrow D_r \setminus \text{getDescendants}((\mathcal{V},\mathcal{E}),v_i)$
9: **return** $(\mathcal{V}_{intermediate}, \Delta)$

node's dictionary into a set of key-value pairs. The cleaner component can perform preprocessing to remove unnecessary or irrelevant nodes and edges. For instance, the cleaner removes singleton nodes that are not connected to anything; they may appear if objects are not being used. The cleaner removes nodes that are not relevant to an investigation; for instance, removing Statistic nodes about traffic counts may be useful if the investigation does not involve traffic counts. The topology augmenter adds edges into the graph (*e.g.,* wasDerivedFrom relations between PacketIns and PacketOuts) to define the data plane model; doing so achieves goal **G2**.

PICOSDN's data plane model algorithm is shown in Algorithm 1. We assume that the data plane's topology can vary over time, and for each variation, we say that the state is an *epoch* consisting of a topology that is valid between a start time and an end time (lines 1–2). For each PacketIn, we want to determine if it should link to a causally related PacketOut (line 4). PICOSDN filters temporally based on the current epoch (line 5), and it checks all PacketOuts during that epoch (line 6). We consider a PacketOut to be causally related to the PacketIn if all of the following conditions are met: 1) there is a link between the outgoing and incoming switches (line 7); 2) the specified packet headers are the same for both packets (line 8); 3) the PacketOut "happened before" the PacketIn (line 9); and 4) the timestamp differences between the PacketOut and PacketIn are within a specific threshold (line 9).

As PICOSDN is modular, Algorithm 1's data plane model can be replaced as needed. For instance, header space analysis [30] uses functional transformations to model how packets are transformed across the data plane (*e.g.,* packet modifications), and P4 [7] proposes a programmable data plane. Practitioners can write their own data plane model components that take those transformations into account.

5.2.2 Tracer

After the graph is prepared, the tracer component answers investigative queries. PICOSDN provides facilities to answer queries related to root cause analysis, network activity summarization, and network state evolution; these facilities achieve goal **G3**. We now describe each kind of query and under what scenarios a practitioner would want to use each kind.

As $\mathcal{G}$ is a DAG, we assume the use of standard graph functions in Algorithms 2–5 that can determine the ancestor and descendant nodes (*i.e.,* progeny) of a given node n, denoted by $\text{getAncestors}(\mathcal{G},n)$ and $\text{getDescendants}(\mathcal{G},n)$, respectively.

Root cause analysis After an attack, a practitioner wishes to investigate the attack's causes so as to determine what changes should be made to prevent such attacks from reoccurring. We assume that a practitioner has *evidence* of incorrect behavior, wants to find common causes, and wants to explore whether other evidence of incorrect behavior also exists. PICOSDN provides two interrelated algorithms to do achieve these goals: *common ancestry tracing* (Algorithm 2) and *backward-forward tracing* (Algorithm 3). Practitioners can iteratively use these tools to determine root causes efficiently.

Algorithm 2 shows the common ancestry tracing. We assume that our practitioner can pinpoint evidence of incorrect

Algorithm 4 Network Activity Summarization

Input: graph $\mathcal{G}$
Output: set of (activity a, flow rule f_{out}, packet p_{in}, data plane packets P_{in})
Initialize: $(\mathcal{V},\mathcal{E}) \leftarrow \mathcal{G}, S \leftarrow \emptyset$
1: **for each** $e \in \mathcal{E}$ **do** ▷ Remove non-causal edges
2: **if** e is a wasRevisionOf edge **then**
3: $\mathcal{E} \leftarrow \mathcal{E} \setminus \{e\}$
4: **for each** $a \in \mathcal{V}_{\text{class=Activity}}$ **do**
5: $f_{out} \leftarrow$ null, $p_{in} \leftarrow$ null, $P_{in} \leftarrow$ null
6: $P_{in} \leftarrow$ getAncestors$((\mathcal{V},\mathcal{E}),a)$
7: **for each** $p \in P_{in}$ **do**
8: **if** $p \notin \mathcal{V}_{\text{class=PacketIn}}$ **then**
9: $P_{in} \leftarrow P_{in} \setminus \{p\}$
10: **if** $\langle a \rightarrow (v \in \mathcal{V}_{\text{class}\neq\text{Activity}} \text{ or } e \in \mathcal{E})^* \rightarrow p \in \mathcal{V}_{\text{class=PacketIn}} \rangle$ backward trace path exists **then**
11: $p_{in} \leftarrow p$
12: **if** $\langle f \in \mathcal{V}_{\text{class=FlowRule}} \rightarrow (v \in \mathcal{V}_{\text{class}\neq\text{Activity}} \text{ or } e \in \mathcal{E})^* \rightarrow a \rangle$ backward trace path exists **then**
13: $f_{out} \leftarrow f$
14: $S \leftarrow S \cup \{(a, f_{out}, p_{in}, P_{in})\}$
15: **return** S

behavior, such as a set of packets or flow rules that appear suspicious. Our practitioner's goal is to see if such evidence has anything in common with past history. PICOSDN starts by discarding non-causal edges in the graph (lines 1–3). Then, for each piece of evidence, PICOSDN computes its set of ancestor nodes and takes the intersection of that ancestry with the ancestries of all previous pieces of evidence (lines 4–6). Once all the pieces of evidence have been examined, the set of common ancestors is partitioned into agent, activity, and entity nodes (lines 7–13). Thus, PICOSDN provides data-centric, process-centric, and agent-centric answers.

Algorithm 3 shows the iterative backward-forward tracing. Our practitioner has a piece of evidence and a suspected root cause (derived, perhaps, from Algorithm 2). Our practitioner's goal is to iteratively determine how intermediate causes (*i.e.,* those causes that lie temporally in between the evidence and the root cause) impact the evidence and other effects on the network's state. PICOSDN starts by discarding non-causal edges in the graph (lines 1–3). For the piece of evidence, PICOSDN determines all of its ancestors, or the set of all causally related entities, activites, and agents responsible for the evidence (line 4). For the suspected root cause, PICOSDN determines all of its descendants, or the set of all the entities and activities that the root cause affected (line 5). PICOSDN takes the intersection of those two sets (line 6) to examine only the intermediate causes that occurred as a result of the root cause. For each intermediate cause, PICOSDN derives the set of affected entities and activities that the root cause affected that the intermediate cause did not affect (lines 7–8). In essence, that lets the practitioner iteratively examine intermediate effects at each stage.

Network activity summarization One general provenance challenge is that graphs can become large and difficult to interpret even for simple activities, and that creates fatigue when one is analyzing such graphs for threats and attacks [20]. PICOSDN provides an efficient network-specific summarization.

Algorithm 4 shows the summarization approach. Our practitioner's goal is to answer questions of the form "Which data plane activities (*i.e.,* packets) caused flow rules to be or not be installed?" PICOSDN starts by discarding non-causal edges in the graph (lines 1–3). It collects each event listener or packet processor activity (line 4). For each activity, it derives all of the PacketIn packets that causally affected the activity (lines 5–9). Then, PICOSDN determines whether a PacketIn is a direct[5] cause by computing a backward trace path; if it is a direct cause, the packet is marked (lines 10–11). Similarly, PICOSDN determines whether a FlowRule is a direct effect of the activity; if it is, the flow rule is marked (lines 12–13).

Algorithm 4 allows practitioners to efficiently investigate instances in which flow rules were *not* created, too. For example, if an event listener used a packet but did not generate a flow rule, the resulting value for f_{out} would be null. Algorithm 4 also derives a set of all data plane PacketIn packets causally related to each activity; as we show later in § 7, this information is useful for diagnosing cross-plane attacks.

Algorithm 5 Network Identifier Evolution

Input: graph $\mathcal{G}$, network identifier i
Output: revision trace path t_r, affected nodes function F
Initialize: $(\mathcal{V},\mathcal{E}) \leftarrow \mathcal{G}$; $\mathcal{E}_{stash} \leftarrow \emptyset$; $F(i) \leftarrow \emptyset, \forall i \in \mathcal{V}$
1: **for each** $e \in \mathcal{E}$ **do** ▷ Remove and stash non-causal edges
2: **if** e is a wasRevisionOf edge **then**
3: $\mathcal{E} \leftarrow \mathcal{E} \setminus \{e\}$
4: $\mathcal{E}_{stash} \leftarrow \mathcal{E}_{stash} \cup \{e\}$
5: $n \leftarrow$ getMostRecentNode$(\mathcal{V}, i)$
6: $t_r \leftarrow \langle n \rangle$
7: $F(n) \leftarrow$ getDescendants$((\mathcal{V},\mathcal{E}),n)$
8: **while** $n \leftarrow$ getNextNode$(\mathcal{E}_{stash})$ and n is not null **do**
9: t_r.append(wasRevisionOf, n)
10: $F(n) \leftarrow$ getDescendants$((\mathcal{V},\mathcal{E}),n)$
11: **return** (t_r, F)

Network state evolution Given the attribution challenges of data plane host activities, practitioners will want to investigate whether any of the pertinent identifiers have been spoofed. Such spoofing can have systemic consequences on subsequent control plane decisions [13, 24, 49, 53]. PICOSDN efficiently tracks network identifier evolution (*i.e.,* the wasRevisionOf relation) and provides an algorithm to query it (Algorithm 5).

Algorithm 5 shows the network identifier evolution approach. Our practitioner's goal is to see whether any identifiers have evolved over time as a result of malicious spoofing, as well as the extent of damage that such spoofing has caused. PICOSDN starts by stashing non-causal edges in the

[5] In other words, without any intermediate Activity nodes in between. However, intermediate data derivations between Entity objects are permissible.

Table 3: List of PICOSDN hooks (*i.e.,* PICOSDN API calls).

PICOSDN API call	Description
recordDispatch(*activity*)	Mark the start of an event dispatch or packet processing loop
recordListen(*activity*)	Mark the demarcation (*i.e.,* start of each loop) of an event being listened to or a packet being processed
recordApiCall(*type*, *entity*)	Record a control plane API call of a *type* (*i.e.,* create, read, update, delete) on an *entity* (or *entities*)
recordDerivation(*entity*, *entity*)	Record an *object* derived from another *object*

graph, thus removing them from causality-related processing, but keeping them for reference (lines 1–4). For a given network identifier, PICOSDN determines the node most recently linked to that identifier (line 5) and adds it to a revision trace path (line 6). PICOSDN derives that node's descendants to determine the extent to which that network identifier causally affected other parts of the network state (line 7). That process is repeated back to the identifier's first version (lines 8–10).

Algorithm 5 produces a concise representation of an identifier's state changes over time. That allows the practitioner to easily determine when an identifier may have been spoofed, and that respective node in time can be used in Algorithm 3 as a root cause to perform further iterative root-cause analysis. Furthermore, the affected nodes that are returned by Algorithm 5 can be used as evidence in the common ancestry trace of Algorithm 2.

6 Implementation

We implemented PICOSDN in Java on ONOS v1.14.0. Our implementation is available at https://github.com/bujcich/PicoSDN. We modified ONOS in several key locations. We created a set of PICOSDN API calls, which are listed in Table 3. We created Java classes to represent Activity and Entity objects, and we made them into superclasses for relevant ONOS classes (*e.g.,* ONOS's Packet superclass is Entity). We wrapped the ONOS event dispatcher and packet processor by using the recordDispatch() and recordListen() calls, which represented the execution partitioning of PICOSDN. We hooked the ONOS core services'[6] public API calls by using the recordApiCall() calls.[7] For a given core service API call, if the return value was iterable, we marked each object within the iterable object with its own separate provenance record. For certain data whose processing spanned multiple threads, we used recordDerivation() calls to maintain the causal relations across threads. We implemented the ingester, modifier, and tracer on top of the JGraphT library.

Because of our design decisions, described in § 5.1, we did not need to perform an analysis on or make any modifications to the ONOS apps. Practitioners do not need to instrument each new app that they install in their network. Furthermore, PICOSDN's API and classes allow PICOSDN to be easily updated as new core services and objects are implemented in ONOS. Although we implemented PICOSDN on ONOS, the same conceptual provenance model and design can be implemented with minimal modifications on any event-based SDN controller architecture, and indeed the most popular controllers (*e.g.,* ODL and Floodlight) all use such architectures.

7 Evaluation

We now evaluate PICOSDN's performance and analysis capabilities. We have examined its performance overhead in terms of latency and storage (§ 7.1). We used recent SDN attacks to show that PICOSDN can capture and explain a broad diversity of SDN attacks (§7.2). We implemented all topologies using Mininet.[8] We ran experiments using a workstation with a four-core 3.30-GHz Intel Core i5-4590 processor and 16 GB of memory.

7.1 Performance Evaluation

Given the latency-critical nature of control plane decision-making, we benchmarked the latency that PICOSDN imposed on common ONOS API calls (Figure 8a). To further understand these costs, we microbenchmarked PICOSDN's hooks (Figure 8b) and benchmarked the overall latency imposed by a reactive control plane configuration (Figure 8c) as a function of the data plane's network diameter. We also measured the costs to store provenance graphs (Table 4).

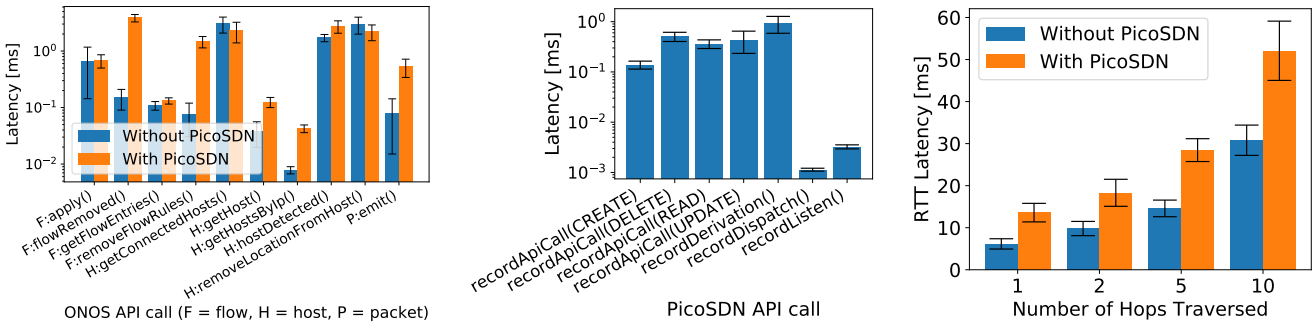
Benchmarks on ONOS Figure 8a shows the average latencies of common ONOS API calls with and without PICOSDN enabled. These calls were called most often in our security evaluation (§ 7.2) and relate to flow rules, hosts, and packets. Although certain calls generated significantly greater latency, that was expected for cases in which iterable objects require generation of individual provenance records.

Microbenchmarks To further analyze the benchmark results, we microbenchmarked PICOSDN's hooks (*i.e.,* PICOSDN's API calls). Figure 8b shows the average latencies of

[6] In ONOS, these core services are represented by classes that end in *Manager or *Provider. For instance, ONOS has a HostManager class and a HostProvider class that include public API calls related to hosts.

[7] As ONOS does not provide a reference monitor architecture that would allow us to wrap one central interposition point across *all* API calls, we had to add recordApiCall() hooks across 141 API calls to ensure completeness.

[8] We chose Mininet because it is common in prior work (*e.g.,* [52, 55]) and because it causes PICOSDN's runtime phase to record the same kind and amount of provenance information that would be captured in a real network. Real networks may differ in terms of imposed latency.



(a) Average latency per ONOS API call. (b) Average latency per PICOSDN hook. (c) Overall average latency per diameter.

Figure 8: PICOSDN latency performance results. (Error bars represent 95% confidence intervals.)

the PICOSDN API calls listed in Table 3, with the recordApiCall() calls broken down by call type. As shown in Figure 8b, event listening and dispatching are fast operations. We expected API calls to be slower, given the tracking operations within PICOSDN's internal state.

Overall latency We also measured the overall latency that PICOSDN imposes on control plane operations. We wanted to see what the additional incurred latency would be from the perspective of host-to-host communication, or the *time-to-first-byte* metric. This metric measures the total round-trip time (RTT) measured between data plane hosts (*e.g.,* via the `ping` utility) for the first packet of a flow. The RTT captures the latency of both data plane processing and control plane decision-making.

In *reactive* control planes, the first packet of a flow suffers high latency because it does not match existing flow rules, but once matching flow rules have been installed, the remaining packets of the flow use the data plane's fast path. Although SDN configurations can be *proactive* by installing flow rules before any packets match them, we measured a reactive configuration because it represents the *worst*-case latency that is imposed if the controller must make a decision at the time it sees the first packet. (See § 8 for a discussion of the differences.) In addition, the network's diameter (*i.e.,* the number of hops between data plane hosts) affects latency in reactive configurations if the first packet must be sent to the controller *at each hop*. Thus, we measured a reactive configuration and varied the number of hops to determine the effect on latency.

Figure 8c shows the average overall latencies imposed with and without PICOSDN on the first packet, varied by the number of hops. We performed each experiment over 30 trials. In contrast to prior work [52, 55], we parameterized the number of hops traversed to reflect different network topology diameters. We found that PICOSDN increased the overall latency on average from 7.44 ms for 1-hop (*i.e.,* same-switch) topologies to 21.3 ms for 10-hop topologies. That increase was expected, given that additional provenance must be generated for longer routes. For long-running flow rules, the one-time latency cost in the flow's first packet can be amortized. Thus, we find PICOSDN acceptable for practical implementation.

Storage costs Internally, PICOSDN maintains only the minimum state necessary to keep track of object changes. Thus, the state is as large as the number of objects representing the network's flow rules, topology, and system principals (*e.g.,* switches and hosts) at a given time.

We investigated the external provenance graph storage costs based on the network's characteristics, and we summarize our results in Table 4. Given the network diameter's impact on latency in reactive control planes, we focused the analysis on the network diameter's impact on storage costs. We set up a bidirectional, reactive, end-to-end flow between two hosts, and we parameterized the number of hops between those hosts. We defined the storage cost as being all of the related provenance needed to explain the origins of the connectivity between those two hosts (*e.g.,* flows, packets, hosts, topologies, events, apps, switch ports). We compared costs using the raw output of the runtime phase ("before cleaning") and the cleaned graph used for investigation ("after cleaning"). Since such storage reflects a single bidirectional flow, we considered the scalability of an enterprise-scale workload of 1,000 new bidirectional flows per second [55].

We found that the cleaned graph requires a significantly smaller amount of persistent storage space, with reductions of 95 to 98 percent. We optimized what provenance was kept by removing orphan nodes, redundant edges, activities without effects, and activities that did not impact flows; these options are configurable by practitioners. We found that the storage costs increased as the number of hops increased. This was expected, given that more objects (*e.g.,* packets) are generated and used with longer routes. PICOSDN generates an estimated 4 to 15 GB/h for an enterprise-scale network with 1,000 new bidirectional flows per second. Further provenance storage reduction can be implemented outside PICOSDN using existing provenance storage reduction systems and techniques [9, 21, 34].

We compare PICOSDN's storage requirements with the

Table 4: PICOSDN storage costs of a bidirectional flow's provenance.

Hops	Graph before cleaning			Graph after cleaning			Reduction in storage			Estimated storage cost of 1,000 new bidirectional flows per second [55]
	# Nodes	# Edges	Data [KB]	# Nodes	# Edges	Data [KB]	Nodes	Edges	Data	
1	822	400	23.0	67	95	1.1	-91.9%	-76.3%	-95.2%	3.96 GB/h
2	2,158	2,298	62.9	146	363	1.7	-93.2%	-84.2%	-97.3%	6.12 GB/h
5	4,299	2,674	110.2	267	495	2.6	-93.8%	-81.5%	-97.6%	9.36 GB/h
10	11,742	7,319	289.8	538	1,175	4.3	-95.4%	-84.0%	-98.5%	15.48 GB/h

most closely related work. FORENGUARD [55] generates 0.93 GB/h of metadata for a 10-switch, 100-host topology with 1,000 new flows per second. Although PICOSDN has higher storage costs, the additional metadata allows PICOSDN to handle more sophisticated analyses that FORENGUARD does not provide (*e.g.,* network identifier evolution, common ancestry trace). We illustrate this in our security evaluation section (§ 7.2). PROVSDN [52] does not evaluate storage costs. As the graphs produced by PROVSDN are optimized for IFC security label generation rather than for explaining root causes, the necessary metadata that must be kept (and, thus, storage costs) are not directly comparable to the metadata that PICOSDN keeps.

7.2 Security Evaluation

We used representative vulnerabilities found with EVENTSCOPE [53] and TOPOGUARD [24] to evaluate PICOSDN's security efficacy.

EVENTSCOPE CVE-2018-12691 We now revisit the motivating cross-plane attack example described in § 2.1. Our practitioner now examines the provenance data collected during the attack by PICOSDN's runtime phase, which is shown in abbreviated form in Figure 2b.

As our practitioner knows that hosts h_1 and h_2 communicated, they use the network activity summarization to derive the set of flow rules related to these hosts. Among the returned set, the practitioner sees the following: 1) the flow rule from fwd that allowed communication $(\mathsf{fwd}, f_3, p_4, \{p_3, p_2\})$; 2) acl's failure to install a flow denial rule, resulting from an invalid IP address $(\mathsf{acl}, \text{null}, \text{null}, \{p_1\})$; and 3) acl's failure to install a flow denial rule, resulting from the host event type's not being handled $(\mathsf{acl}, \text{null}, \text{null}, \{p_2\})$.

The practitioner uses the common ancestry trace of fwd and acl's actions to determine the common ancestors of the discovered flow rules. Among this set, the common ancestor is the switch port agent s_1 : port 1. Now equipped with a set of possible root causes, the practitioner issues a backward-forward trace from f_3 to the root of the switch port agent to see the differences in descendants (*i.e.,* impacts) that each intermediate cause affects. That allows the practitioner to discover that the relevant root cause can be traced back to the spoofed packet p_1. Starting there, the practitioner's forward traces show the effects that p_1 has on the network's subsequent activities, such as the corrupted host object $h_{1(v1)}$. PICOSDN identifies the root cause and effects of the spoofed packet, thus letting the practitioner know that host h_1 should be disconnected.

Improvements upon prior work: FORENGUARD and PROVSDN do not link data plane activities together. As a result, practitioners would miss the necessary causal dependency that is critical for understanding this attack's root cause. Furthermore, FORENGUARD and PROVSDN cannot diagnose causes related to the *absence* of effects (*e.g.,* acl's failure to install flow rules). As a result, practitioners using these tools would not be able to diagnose the class of attacks that use the absence of effects to accomplish the attacker's objectives. By contrast, PICOSDN's data plane model clearly links the data plane packets that result from fwd's installation across switches (Figure 2b). PICOSDN's network activity summarization efficiently identifies the activities that lack effects (*i.e.,* f_{out} is null). In this attack, practitioners can see the *presence* of a potential cause (*e.g.,* the execution of acl) and the *absence* of an expected effect (*e.g.,* a flow denial rule).

EVENTSCOPE CVE-2019-11189 We evaluated another vulnerability found by EVENTSCOPE, CVE-2019-11189. This attack bypasses an intended access control policy. It uses a malicious host to spoof a network identifier of a victim host, which causes installed flow rules associated with the access control policy to be uninstalled by the host mobility application, mobility. We refer the reader to [53] for a detailed description of the attack's mechanism.

PICOSDN is able to capture the installation of the flow rules associated with the access control policy, the triggering of the host mobility application because of spoofed packets, and the removal of the flow rules by the host mobility application. A practitioner who notices that undesired communication occurred between the malicious host and the victim host can use the graph to understand the causal relationships among all three activities and to pinpoint the spoofed packet as the actual root cause.

Improvements upon prior work: FORENGUARD and PROVSDN do not explicitly model the deletion of control plane state as a graphical relation. As a result, practitioners

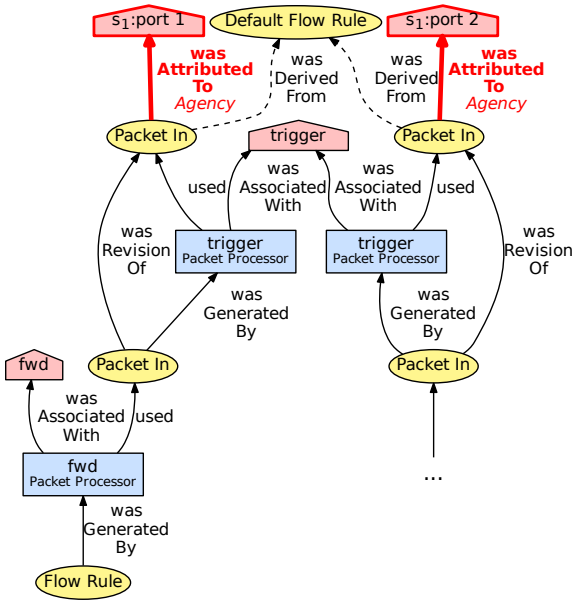


Figure 9: Relevant features of the graph from the cross-app attack. The graph shows that trigger modifies packets before fwd receives them.

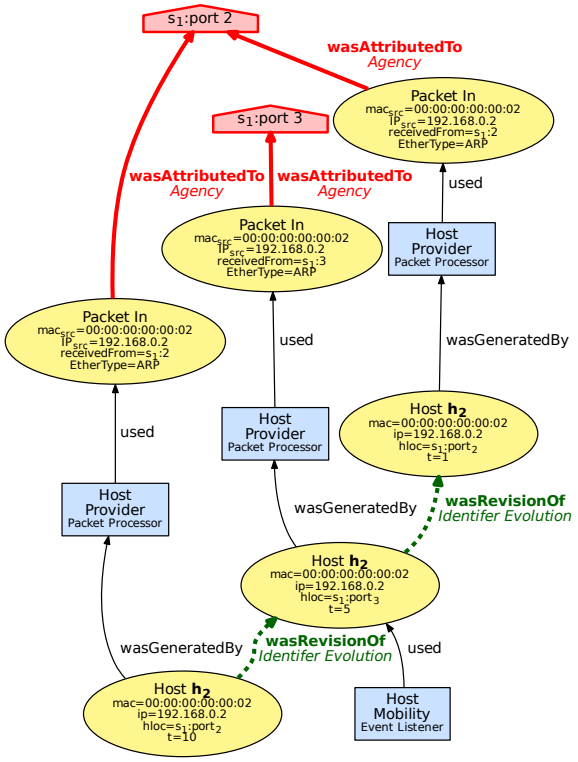


Figure 10: Relevant features of the host migration attack's graph showing the evolution of hosts that claimed to be h_2.

who use these tools would not be able to perform causal analysis over the deletions' dependencies. By contrast, PICOSDN's wasInvalidatedBy relation links control plane state objects to control plane activities. That augments PICOSDN's capabilities to trace common ancestors and to trace backward and forward iteratively. In this example, a practitioner sees that the removed flow rule can be tracked backward to mobility's use of a modified (*i.e.,* spoofed) host object.

PROVSDN Cross-App Poisoning Attack We also use PICOSDN to analyze a cross-app poisoning attack. This attack uses a malicious app to modify packets in the packet-processing pipeline, which subsequent apps use to make control plane decisions. We refer the reader to [52] for a detailed description of the attack's mechanism.

Figure 9 shows the important features of the graph. We can see that the packet changes as it is handed off from the triggering trigger (*i.e.,* malicious) app to the forwarding fwd (*i.e.,* benign) app in the processing pipeline. Since PICOSDN uses an event-based model, we can reduce the false dependencies. For instance, for each instance of trigger's event handler, the precise API calls that were used are embedded in the used and wasGeneratedBy relations for API read and write calls, respectively, on the PacketIns.

To understand how the attack occurred, a practitioner issues a network activity summarization query to find malicious flow rules and uses them in the common ancestry trace to look at the trigger agent. The practitioner then issues an iterative backward-forward trace query on the trigger app to determine the extent to which trigger has caused other undesired network activities. PICOSDN identifies the root cause and other effects of trigger, thus informing the practitioner that the app should be removed.

Improvements upon prior work: FORENGUARD and PROVSDN do not provide common ancestry tracing capabilities. As a result, practitioners using FORENGUARD or PROVSDN would need to manually determine overlapping root causes, which could significantly hinder any time-sensitive investigations, increase the attackers' dwell time, and increase the damage [12]. By contrast, PICOSDN uses its common ancestry trace in this example to efficiently determine that all of the malicious flows have trigger in common.

TOPOGUARD Host Migration Attack We consider another cross-plane-based host migration attack. This attack uses a malicious data plane host to trick the control plane into believing that a victim host has changed its location. We assume a three-host (h_1, h_2, and h_3) topology with one switch (s_1). Host h_3 attempts to masquerade as host h_2 so as to trick other hosts (*e.g.,* h_1) into sending traffic that was meant for h_2 to h_3 instead. We refer the reader to [24] for a detailed description of the attack's mechanism.

Our practitioner queries the network identifier evolution for h_2. Figure 10 shows a partial provenance graph of the relevant features. The evolution shows that h_2 appears to have switched network ports from s_1's port 2 to port 3; in reality,

h_3 spoofed h_2's identifier. The query returns the descendants (*i.e.,* the impacts) that each version of the identifier has had on the network. For instance, during the time that the spoofed location of h_2 was being used between times $t = [5, 10]$, old flow rules that directed traffic to h_2 were removed by the host mobility app. The practitioner can now efficiently see the attack's ramifications at each stage because of the combination of the network identifier evolution and the forward-tracing capabilities. PICOSDN identifies a cause in the spoofed packet used by the host provider, and also finds the other effects of the spoofed packet. The practitioner thus disconnects the malicious host from port 3.

Improvements upon prior work: FORENGUARD and PROVSDN do not store the additional relations needed to track network identifier evolution, and they do not provide the forward-tracing capabilities to determine the effects that spoofed identifiers have on other network activities. As a result, practitioners using these tools would not be able to quickly assess the extent of damage. By contrast, PICOSDN's network identifier evolution tool shows the network effects at each stage of identifier change.

8 Discussion

Reactive and proactive configurations PICOSDN is designed to work for both reactive and proactive SDN control plane configurations. We used reactive configurations in our case studies because recent SDN attacks have leveraged reactive configurations [24, 49, 53, 62], but we argue that PICOSDN is well-suited for proactive configurations, too. Proactive configurations install flow rules ahead of time. However, the time at which flow rules are inserted may be far removed from the time when data plane packets exercise these rules. As a result of the time gap, manual tracing by a practitioner would be a difficult task. That provides the motivation to create quality network forensics tools such as PICOSDN to maintain history.

Deployment Considerations Our work complements existing detection and investigation tools in the security monitoring pipeline. PICOSDN does not automatically detect attacks, but instead provides investigators with *insight* into control plane modifications and *analysis* of causal dependencies. This is a critical step for enterprise security, particularly as threat alerts are known to suffer from high rates of false alarm; some reports show that more than half of alerts are false alarms, and as few as 4% are properly investigated [16]. PICOSDN thus addresses a vital gap in existing investigation products; one such application of this technology would be to integrate it into existing SIEM products, e.g., Splunk, to allow analysts to observe SDN-related intelligence streams alongside other network telemetry data. SDN attack detection in particular is an open challenge, with past work examining expected semantic behavior [13, 24, 49] and pattern recognition of anomalous features or behavior [32, 53], but these pursuits are orthogonal to PICOSDN's aims.

9 Related Work

SDN control plane insight FORENGUARD [55] is the prior effort that is most closely related to PICOSDN. Like FORENGUARD, PICOSDN provides root cause analysis capabilities for SDN attacks. PICOSDN extends those capabilities with a data plane model and mitigates the data dependency explosions caused by default flow rules. PROVSDN [52] focuses on information flow control enforcement rather than root cause analysis, so its analysis capabilities are limited; it also uses an API-centric model rather than an event-centric model for execution partitioning, resulting in false dependencies that would not be generated in PICOSDN's provenance model. GitFlow [15] proposes a version control system for SDN; that influenced our decision to include revision relations. AIM-SDN [14] outlines the challenges in SDN, influencing our decisions on how to represent agency. Ujcich *et al.* [54] argue why provenance is necessary to ensure a secure SDN architecture.

Declarative network provenance has shown promise in automated bug removal [58], differential provenance [10, 11], meta provenance [57], and negative provenance [60, 61]. The various solutions use a declarative paradigm [36], which requires nontrivial translation for apps written in the imperative paradigm. A benefit of declarative programs is that they inherently capture the data plane model, which PICOSDN provides but PROVSDN and FORENGUARD do not.

The general research space of SDN security, including the set of potential attack vectors, is large and well-studied; we refer the reader to [63] for a survey of the area.

SDN debugging and verification We outline existing SDN debugging and verification tools, as they are complementary to provenance-based causal analysis tools.

Control-plane debugging tools include FALCON [35], Net2Text [6], among others. They record the network's state to identify unusual behavior and replay suspicious activities in a simulated environment. However, they assume that activity traces are dependent upon all previous states and/or inputs, whereas PICOSDN avoids that assumption through its dependency partitioning.

Data plane verification tools include Cocoon [48] and SDNRacer [43], and BEADS [27], among others. They prevent instantiation of incorrect configurations in the network according to a predefined policy, but such tools' prevention capabilities are dependent upon correct policy specifications. PICOSDN records known and unknown attacks so that practitioners can investigate how such attacks occurred.

Provenance and causality analysis The dependency explosion problem has been studied for host applications [39], binary analysis [22, 33], and host operating systems [20, 26, 29, 31, 37, 38, 40]. Provenance for attack causality analysis has also been well-studied [2–4, 19, 25, 42, 46, 50, 51, 56, 59]. PICOSDN's primary contributions to this area include 1) a provenance model for SDN control and data planes that focuses on SDN-specific dependency explosion factors (*e.g.,* default flow rule dependencies), and 2) relevant attack analysis techniques of particular interest to network practitioners (*e.g.,* network summarization).

10 Conclusion

We presented PICOSDN, a provenance-informed causal observation tool for SDN attacks. PICOSDN leverages a fine-grained provenance model to allow practitioners to reconstruct past control and data plane activities, to analyze them for root causes when control plane attacks occur, to understand the scope of attacks' effects on other network activities, and to succinctly summarize the network's activities and evolution. We evaluated PICOSDN using recent control plane attacks, and we found that PICOSDN is practical for runtime collection and offline analysis.

Acknowledgements

The authors thank our shepherd, Jelena Mirkovic, and the anonymous reviewers for their helpful comments, which improved this paper; the PERFORM and STS research groups at the University of Illinois for their advice and feedback; and Jenny Applequist for her editorial assistance. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1750024.

References

[1] Endpoint Detection and Response Solutions Market. https://www.gartner.com/reviews/market/endpoint-detection-and-response-solutions, 2019.

[2] Adam Bates, Kevin Butler, Andreas Haeberlen, Micah Sherr, and Wenchao Zhou. Let SDN be your eyes: Secure forensics in data center networks. In *NDSS SENT '14*, 2014.

[3] Adam Bates, Wajih Ul Hassan, Kevin Butler, Alin Dobra, Bradley Reaves, Patrick Cable, Thomas Moyer, and Nabil Schear. Transparent web service auditing via network provenance functions. In *WWW '17*, 2017.

[4] Adam Bates, Dave Tian, Kevin R. B. Butler, and Thomas Moyer. Trustworthy whole-system provenance for the Linux kernel. In *USENIX Security '15*, 2015.

[5] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. ONOS: Towards an open, distributed SDN OS. In *ACM HotSDN '14*, 2014.

[6] Rudiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin Vechev. Net2Text: Query-guided summarization of network forwarding behaviors. In *NSDI '18*, 2018.

[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, 2014.

[8] Jiahao Cao, Renjie Xie, Kun Sun, Qi Li, Guofei Gu, and Mingwei Xu. When match fields do not need to match: Buffered packets hijacking in SDN. In *NDSS '20*, 2020.

[9] Adriane Chapman, H.V. Jagadish, and Prakash Ramanan. Efficient provenance storage. In *ACM SIGMOD '08*, 2008.

[10] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Differential provenance: Better network diagnostics with reference events. In *ACM HotNets '15*, 2015.

[11] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *ACM SIGCOMM '16*, 2016.

[12] Crowdstrike. Why Dwell Time Continues to Plague Organizations. https://www.crowdstrike.com/blog/why-dwell-time-continues-to-plague-organizations/, 2019.

[13] Mohan Dhawan, Rishabh Poddar, Kshiteej Mahajan, and Vijay Mann. SPHINX: Detecting security attacks in software-defined networks. In *NDSS '15*, 2015.

[14] Vaibhav Hemant Dixit, Adam Doupé, Yan Shoshitaishvili, Ziming Zhao, and Gail-Joon Ahn. AIM-SDN: Attacking information mismanagement in SDN-datastores. In *ACM CCS '18*, 2018.

[15] Abhishek Dwaraki, Srini Seetharaman, Sriram Natarajan, and Tilman Wolf. GitFlow: Flow revision management for software-defined networks. In *ACM SOSR '15*, 2015.

[16] FireEye, Inc. How Many Alerts is Too Many to Handle? https://www2.fireeye.com/StopTheNoise-IDC-Numbers-Game-Special-Report.html, 2019.

[17] Jessica Goepfert, Karen Massey, and Michael Shirer. Worldwide Spending on Security Solutions Forecast to Reach $103.1 Billion in 2019, According to a New IDC Spending Guide. https://www.businesswire.com/news/home/20190320005114/en/, March 2019.

[18] S. R. Gomez, S. Jero, R. Skowyra, J. Martin, P. Sullivan, D. Bigelow, Z. Ellenbogen, B. C. Ward, H. Okhravi, and J. W. Landry. Controller-oblivious dynamic access control in software-defined networks. In *IEEE/IFIP DSN '19*, 2019.

[19] Ragib Hasan, Radu Sion, and Marianne Winslett. Preventing History Forgery with Secure Provenance. *Trans. Storage*, 5(4):12:1–12:43, 2009.

[20] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. NoDoze: Combatting threat alert fatigue with automated provenance triage. In *NDSS '19*, 2019.

[21] Wajih Ul Hassan, Mark Lemay, Nuraini Aguse, Adam Bates, and Thomas Moyer. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *NDSS '18*, 2018.

[22] Wajih Ul Hassan, Mohammad A. Noureddine, Pubali Datta, and Adam Bates. OmegaLog: High-fidelity attack investigation via transparent multi-layer log analysis. In *NDSS '20*, 2020.

[23] Tagato Hiroki, Sakae Yoshiaki, Kida Koji, and Asakura Takayoshi. Automated Security Intelligence (ASI) with Auto Detection of Unknown Cyber-Attacks. *NEC Technical Journal*, 11, 2016.

[24] Sungmin Hong, Lei Xu, Haopei Wang, and Guofei Gu. Poisoning network visibility in software-defined networks: New attacks and countermeasures. In *NDSS '15*, 2015.

[25] Md Nahid Hossain, Sadegh M Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R Sekar, Scott D Stoller, and VN Venkatakrishnan. SLEUTH: Real-time attack scenario reconstruction from COTS audit data. In *USENIX Security '17*, 2017.

[26] Md Nahid Hossain, Sanaz Sheikhi, and R Sekar. Combating dependence explosion in forensic analysis using alternative tag propagation semantics. 2020.

[27] Samuel Jero, Xiangyu Bu, Cristina Nita-Rotaru, Hamed Okhravi, and Sonia Fahmy. BEADS: Automated Attack Discovery in OpenFlow-based SDN Systems". In *Proceedings of RAID*, 2017.

[28] Samuel Jero, William Koch, Richard Skowyra, Hamed Okhravi, Cristina Nita-Rotaru, and David Bigelow. Identifier binding attacks and defenses in software-defined networks. In *USENIX Security '17*, 2017.

[29] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *ACM CCS '17*, 2017.

[30] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI '12*, 2012.

[31] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela Ciocarlie, et al. MCI: Modeling-based causality inference in audit logging for attack investigation. In *NDSS '18*, 2018.

[32] C. Lee, C. Yoon, S. Shin, and S. K. Cha. INDAGO: A new framework for detecting malicious SDN applications. In *IEEE ICNP '18*, 2018.

[33] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High Accuracy Attack Provenance via Binary-based Execution Partition. In *NDSS '13*, 2013.

[34] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. LogGC: Garbage collecting audit log. In *ACM CCS '13*, 2013.

[35] 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 *IFIP/IEEE IM '19*, 2019.

[36] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking: Language, execution and optimization. In *ACM SIGMOD '06*, 2006.

[37] Shiqing Ma, Kyu Hyung Lee, Chung Hwan Kim, Junghwan Rhee, Xiangyu Zhang, and Dongyan Xu. Accurate, low cost and instrumentation-free security audit logging for Windows. In *ACSAC '15*, 2015.

[38] Shiqing Ma, Juan Zhai, Yonghwi Kwon, Kyu Hyung Lee, Xiangyu Zhang, Gabriela Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Dongyan Xu, and Somesh Jha. Kernel-supported cost-effective audit logging for causality tracking. In *USENIX ATC '18*, 2018.

[39] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. MPI: Multiple perspective attack investigation with semantic aware execution partitioning. In *USENIX Security '17*, 2017.

[40] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. ProTracer: Towards practical provenance tracing by alternating between logging and tainting. In *NDSS '16*, 2016.

[41] Eduard Marin, Nicola Bucciol, and Mauro Conti. An in-depth look into SDN topology discovery mechanisms: Novel attacks and practical countermeasures. In *ACM CCS '19*, 2019.

[42] Sadegh M Milajerdi, Rigel Gjomemo, Birhanu Eshete, R Sekar, and VN Venkatakrishnan. HOLMES: Real-time APT detection through correlation of suspicious information flows. In *IEEE S&P '19*, 2019.

[43] Jeremie Miserez, Pavol Bielik, Ahmed El-Hassany, Laurent Vanbever, and Martin Vechev. SDNRacer: Detecting concurrency violations in software-defined networks. In *ACM SOSR '15*, 2015.

[44] Paolo Missier, Khalid Belhajjame, and James Cheney. The W3C PROV family of specifications for modelling provenance metadata. In *ACM EDBT '13*, 2013.

[45] Steve Morgan. Global Cybersecurity Spending Predicted To Exceed $1 Trillion From 2017-2021. https://cybersecurityventures.com/cybersecurity-market-report/, 2019.

[46] Kexin Pei, Zhongshu Gu, Brendan Saltaformaggio, Shiqing Ma, Fei Wang, Zhiwei Zhang, Luo Si, Xiangyu Zhang, and Dongyan Xu. HERCULE: Attack story reconstruction via community discovery on correlated log graph. In *ACSAC '16*, 2016.

[47] Phillip Porras, Steven Cheung, Martin Fong, Keith Skinner, and Vinod Yegneswaran. Securing the software-defined network control layer. In *NDSS '15*, 2015.

[48] Leonid Ryzhyk, Nikolaj Bjørner, Marco Canini, Jean-Baptiste Jeannin, Cole Schlesinger, Douglas B. Terry, and George Varghese. Correct by construction networks using stepwise refinement. In *USENIX NSDI '17*, 2017.

[49] 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 *IEEE/IFIP DSN '18*, 2018.

[50] Manolis Stamatogiannakis, Paul Groth, and Herbert Bos. Looking inside the black-box: Capturing data provenance using dynamic instrumentation. In *IPAW '15*, 2015.

[51] Dawood Tariq, Maisem Ali, and Ashish Gehani. Towards automated collection of application-level data provenance. In *USENIX TaPP '12*, 2012.

[52] Benjamin E. Ujcich, Samuel Jero, Anne Edmundson, Qi Wang, Richard Skowyra, James Landry, Adam Bates, William H. Sanders, Cristina Nita-Rotaru, and Hamed Okhravi. Cross-app poisoning in software-defined networking. In *ACM CCS '18*, 2018.

[53] Benjamin E. Ujcich, Samuel Jero, Richard Skowyra, Steven R. Gomez, Adam Bates, William H. Sanders, and Hamed Okhravi. Automated discovery of cross-plane event-based vulnerabilities in software-defined networking. In *NDSS '20*, 2020.

[54] Benjamin E. Ujcich, Andrew Miller, Adam Bates, and William H. Sanders. Towards an accountable software-defined networking architecture. In *IEEE NetSoft '17*, 2017.

[55] Haopei Wang, Guangliang Yang, Phakpoom Chinprutthiwong, Lei Xu, Yangyong Zhang, and Guofei Gu. Towards fine-grained network security forensics and diagnosis in the SDN era. In *ACM CCS '18*, 2018.

[56] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl Gunter. Fear and logging in the Internet of things. In *NDSS '18*, 2018.

[57] Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Automated network repair with meta provenance. In *ACM HotNets '15*, 2015.

[58] Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Automated bug removal for software-defined networks. In *NSDI '17*, 2017.

[59] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. Zeno: Diagnosing performance problems with temporal provenance. In *NSDI '19*, 2019.

[60] Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Answering why-not queries in software-defined networks with negative provenance. In *ACM HotNets '13*, 2013.

[61] Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Diagnosing missing events in distributed systems with negative provenance. In *ACM SIGCOMM '14*, 2014.

[62] Feng Xiao, Jinquan Zhang, Jianwei Huang, Guofei Gu, Dinghao Wu, and Peng Liu. Unexpected data dependency creation and chaining: A new attack to SDN. In *IEEE S&P '20*, 2020.

[63] Changhoon Yoon, Seungsoo Lee, Heedo Kang, Taejune Park, Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. Flow wars: Systemizing the attack surface and defenses in software-defined networks. *IEEE/ACM Trans on Networking*, 25(6):3514–3530, 2017.