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...
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.

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 forwarding application. The attack works as follows. A malicious data plane host, h1, wants to connect to a victim host, h2, but the data plane access control policy is configured to deny traffic
After an alert is raised, it is then the responsibility of a network practitioner or security analyst to manually investigate the nature, triggering threat alerts when a suspicious event occurs.

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 response. 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.

### 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 response. 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 h₁ to a known malicious domain. Unbeknownst to the practitioner, this flow represents an exfiltration of sensitive data from host h₂ to the open network via h₁, 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 h₁ and h₂, 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 h₁ and h₂ seems to have coincided with the security incident, they use this as an initial piece of evidence: a flow rule (f₃) was installed that allowed traffic from h₁ to h₂ on switch s₂. 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 h₁’s packets to s₁’s default flow rule (i.e., f₁) does capture past causality, the practitioner is easily overwhelmed when all packets over all time from any of s₁’s ports are also linked to that default flow rule. The practitioner also finds that switches s₁ and s₂ as principal agents become...
InboundPacket
PacketContext
within a system can create false dependencies. For instance, long-running processes or widely-used data structures correctly implies that one of the root causes of the attack is agency traces back to a switch, either responsibility (or agency) in the data plane. In Figure 2a, the similar challenge exists in the assignment of attribution Limitation (L2): Coarse-grained responsibility and false effects of a root cause must also be analyzed.

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 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 from \( r = [1, 3] \).

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 (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 ael’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. 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 \( G = (\mathcal{V}, \mathcal{E}) \), 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 involved in such data’s generation and use. Our unified approach accounts for app, control, and data plane activities, which allows us to reason holistically about SDN attacks.

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 \( t_b = (v_0 \rightarrow e_0 \rightarrow \cdots \rightarrow e_i \rightarrow v_j), e_0, \ldots , e_i \in \mathcal{E}_{\mathcal{V}, \mathcal{E}} \), 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 \( G \) is a DAG, nodes \( v_0, \ldots , v_{j-1} \) 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 \( t_r = (v_0 \rightarrow e_0 \rightarrow \cdots \rightarrow e_i \rightarrow v_j), e_0, \ldots , e_i \in \mathcal{E}_{\mathcal{V}, \mathcal{E}} \), 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.

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

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 invalidated by a given Activity and mitigate the dependency explosion.

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, s1 and s2. Figure 6a shows...
As we assume in our threat model (described 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.

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 provenance-informed causal observation for software-defined networking, 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.
**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, \( e_1 \), is processed *sequentially* by each interested event listener (i.e., \( e_1 \) is processed by listener \( l_1 \), then by \( l_2 \); different events, \( e_1 \) and \( e_2 \), may be processed *concurrently* (i.e., \( e_1 \) is processed by listener \( l_1 \) followed by \( l_2 \), while concurrently \( e_2 \) 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 interoperation 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 serializes before writing 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 ingester reads in the serialized graph. As most nodes contain additional details, the graph ingestor de-serializes the time limit for each event listener to avoid indefinite halting.
Algorithm 1 Data Plane Model

Input: graph $G$, data plane topology states $\mathcal{D}_{top}$, time window $\tau$, headers fields to match on $H$

Output: graph with data plane model $\tilde{G}$

Initialize: $(\mathcal{V}', \mathcal{E}) \leftarrow \tilde{G}$
1: for each $D \in \mathcal{D}_{top}$ do
2: $(\mathcal{N}, \tau_{start}, \tau_{end}) \leftarrow D$  \hspace{1em} \triangleright Data plane topology graph $\mathcal{N}$, epoch start
3: $((\mathcal{N}_{starts}, \mathcal{N}_{ends})) \leftarrow \mathcal{N}$
4: for each $p_{in} \in \mathcal{V}_{class} \rightarrow \text{PacketIn}$ do
5: if $\tau_{start} < p_{in}ts < \tau_{end}$ then
6: for each $p_{out} \in \mathcal{V}_{class} \rightarrow \text{PacketOut}$ do
7: if $(p_{out} \text{ switch}, p_{in} \text{ switch}) \in \mathcal{N}_{links}$ then
8: if $p_{out}H = p_{in}H$ then
9: if $p_{out}ts < p_{in}ts$ and $p_{in}ts - p_{out}ts < \tau$ then
10: $\mathcal{V}' \leftarrow \mathcal{V}' \cup \{p_{in}, p_{out}\}$
11: $\tilde{G} \leftarrow (\mathcal{V}', \mathcal{E})$
12: return $\tilde{G}$

Algorithm 2 Common Ancestry Trace

Input: graph $G$, evidence set $\mathcal{E}$

Output: agent set $A_e$, activity set $A_c$, and entity set $En$

Initialize: $(\mathcal{V}', \mathcal{E}) \leftarrow G$, $Ag \leftarrow \emptyset$, $Ac \leftarrow \emptyset$, $En \leftarrow \emptyset$, $A \leftarrow \mathcal{V}'$
1: for each $e \in \mathcal{E}$ do  \hspace{1em} \triangleright Remove non-causal edges
2: if $e$ is a wasRevisionOf edge then
3: $\mathcal{E} \leftarrow \mathcal{E} \setminus \{e\}$
4: for each $n \in \mathcal{E}$ do  \hspace{1em} \triangleright Evidence $n$ (note: $n \in \mathcal{V}', N \subset \mathcal{V}'$)
5: $A_e \leftarrow \text{getAncestors}((\mathcal{V}', \mathcal{E}), n)$  \hspace{1em} \triangleright Set of ancestor nodes $A_e$
6: $A \leftarrow A \cup A_e$  \hspace{1em} \triangleright Common ancestor set $A$
7: for each $a \in A$ do  \hspace{1em} \triangleright 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 $(Ag, Ac, En)$  \hspace{1em} \triangleright Ag $\subset \mathcal{V}'$, Ac $\subset \mathcal{V}'$, En $\subset \mathcal{V}'$

Algorithm 3 Iterative Backward-Forward Trace

Input: graph $G$, evidence $n$, root $r$

Output: affected difference function $\Delta: \mathcal{V}' \rightarrow \mathcal{P}(\mathcal{V}')$

Initialize: $(\mathcal{V}', \mathcal{E}) \leftarrow G$, $\Delta(i) \leftarrow \emptyset$, $\forall i \in \mathcal{V}'$
1: for each $e \in \mathcal{E}$ do  \hspace{1em} \triangleright Remove non-causal edges
2: if $e$ is a wasRevisionOf edge then
3: $\mathcal{E} \leftarrow \mathcal{E} \setminus \{e\}$
4: $A_e \leftarrow \text{getAncestors}((\mathcal{V}', \mathcal{E}), n)$  \hspace{1em} \triangleright Evidence’s ancestor set $A_e$
5: $D_r \leftarrow \text{getDescendants}((\mathcal{V}', \mathcal{E}), r)$  \hspace{1em} \triangleright Root’s descendant set $D_r$
6: $\mathcal{V}_{\text{intermediate}} \leftarrow A_e \cap D_r$
7: for each $v_i \in \mathcal{V}_{\text{intermediate}}$ do
8: $\Delta(i) \leftarrow D_r \setminus \text{getDescendants}((\mathcal{V}', \mathcal{E}), v_i)$
9: return $(\mathcal{V}_{\text{intermediate}}, \Delta)$

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 $G$ 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 getAncestors$(G, n)$ and getDescendants$(G, n)$, 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

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.
Algorithm 4  Network Activity Summarization

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

Algorithm 5  Network Identifier Evolution

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

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. PicoSdSn starts by discarding non-causal edges in the graph (lines 1–3). Then, for each piece of evidence, PicoSdSn computes its set of ancestor nodes and takes the intersection of this 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, PicoSdSn 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. PicoSdSn starts by discarding non-causal edges in the graph (lines 1–3). For the piece of evidence, PicoSdSn determines all of its ancestors, or the set of all causally related entities, activities, and agents responsible for the evidence (line 4). For the suspected root cause, PicoSdSn determines all of its descendants, or the set of all the entities and activities that the root cause affected (line 5). PicoSdSn 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, PicoSdSn 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 inter-

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]. PicoSdSn 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. PicoSdSn starts by stashing non-causal edges in the
Table 3: List of PicoSDN hooks (i.e., PicoSDN API calls).

<table>
<thead>
<tr>
<th>PicoSDN API call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>recordDispatch(activity)</td>
<td>Mark the start of an event dispatch or packet processing loop</td>
</tr>
<tr>
<td>recordListen(activity)</td>
<td>Mark the demarcation (i.e., start of each loop) of an event being listened</td>
</tr>
<tr>
<td>recordApiCall(type-entity)</td>
<td>Record a control plane API call of a type (i.e., create, read, update, delete) on an entity (or entities)</td>
</tr>
<tr>
<td>recordDerivation(entity,entity)</td>
<td>Record an object derived from another object</td>
</tr>
</tbody>
</table>

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’s public API calls by using the recordApiCall() calls. 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

---

8In 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.

9As ONOS does not provide a reference monitor architecture that would allow us to wrap one central interception point across all API calls, we had to add recordApiCall() hooks across 141 API calls to ensure completeness.

---

8We 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.
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.

<table>
<thead>
<tr>
<th>Hops</th>
<th>Graph before cleaning</th>
<th>Graph after cleaning</th>
<th>Reduction in storage</th>
<th>Estimated storage cost of 1,000 new bidirectional flows per second [55]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># Nodes</td>
<td># Edges</td>
<td>Data [KB]</td>
<td># Nodes</td>
</tr>
<tr>
<td>1</td>
<td>822</td>
<td>400</td>
<td>23.0</td>
<td>67</td>
</tr>
<tr>
<td>2</td>
<td>2,158</td>
<td>2,298</td>
<td>62.9</td>
<td>146</td>
</tr>
<tr>
<td>5</td>
<td>4,299</td>
<td>2,674</td>
<td>110.2</td>
<td>267</td>
</tr>
<tr>
<td>10</td>
<td>11,742</td>
<td>7,319</td>
<td>289.8</td>
<td>538</td>
</tr>
</tbody>
</table>

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 ($\text{fwd, f}_3, \text{p}_4, \{p_3, p_2\}$); 2) acl’s failure to install a flow denial rule, resulting from an invalid IP address ($\text{acl, null, null, \{p_1\}}$); and 3) acl’s failure to install a flow denial rule, resulting from the host event type’s not being handled ($\text{acl, null, null, \{p_2\}}$).

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 $h_{1(1)}$. 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., $f_{\text{out}}$ 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...
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 PacketIn.

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 (h1, h2, and h3) topology with one switch (s1). Host h3 attempts to masquerade as host h2 so as to trick other hosts (e.g., h1) into sending traffic that was meant for h2 to h3 instead. We refer the reader to [24] for a detailed description of the attack’s mechanism.

Our practitioner queries the network identifier evolution for h2. Figure 10 shows a partial provenance graph of the relevant features. The evolution shows that h2 appears to have switched network ports from s1’s port 2 to port 3; in reality,
h₃ spoofed h₂’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 h₂ was being used between times t = [5, 10], old flow rules that directed traffic to h₂ 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. AIMSDN [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 Coconu [48] and SDN NRacer [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]. PICO SDN’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 PICO SDN, a provenance-informed causal observation tool for SDN attacks. PICO SDN 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 PICO SDN using recent control plane attacks, and we found that PICO SDN 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


