Diagnosing Programmable Data Plane Attacks with Provenance Observability

Dhiraj Saharia
Georgetown University
Washington, DC, USA

Ashish Gehani
SRI International
Menlo Park, CA, USA

Vinod Yegneswaran
SRI International
Menlo Park, CA, USA

Benjamin E. Ujcich
Georgetown University
Washington, DC, USA

Abstract

The decoupling of the control and data planes in software-defined networking (SDN) has enabled flexible and high-performance network designs. At the same time, advances in data-plane programmability—most notably through languages such as P4—allow complex, stateful network functions to execute directly in the data plane. While powerful, this shift introduces new security risks: data-plane programs can make routing and policy decisions independently of centralized controllers, evading traditional monitoring, auditing, and analysis tools. These decisions may also have delayed or systemic effects on network behavior, without a clear record of how they were produced.

To address these limitations, we present ProvP4, a provenance-based cross-plane observer designed to observe and analyze stateful data plane attacks in programmable networks. ProvP4 includes a data plane provenance event collection module, a querying mechanism, and extensible analysis algorithms that collectively offer a unified view of network activities. Our evaluation demonstrates that ProvP4 imposes minimal overhead in terms of storage and performance, while significantly enhancing security analysis capabilities. We further validate ProvP4 through three case studies, highlighting its effectiveness and coverage in enabling detailed attack investigation and forensic analysis.

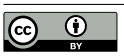
CCS Concepts

• **Networks** → **Programmable networks**; • **Security and privacy** → **Network security**.

Keywords

programmable data plane; network provenance; P4; software-defined networking

ACM Reference Format:
Dhiraj Saharia, Ashish Gehani, Vinod Yegneswaran, and Benjamin E. Ujcich. 2026. Diagnosing Programmable Data Plane Attacks with Provenance Observability. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3832594



ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3832594

1 Introduction

Software-defined networking (SDN) has revolutionized network design by enabling programmatic control over configurations, facilitating dynamic adaptation to evolving threats. In recent years, attention has shifted toward making the SDN data plane programmable. This shift emerged from the limitations of static configurability, prompting efforts to embed more intelligent decision-making capabilities directly into the data plane, which can process traffic at line rate and avoid the control plane's slower path.

One prominent example is Programming Protocol-independent Packet Processors (P4) [25], a data plane programming language that enables developers to specify how packets should be parsed, processed, and forwarded. The net effect is that control plane intelligence extends into the data plane via data plane programs that now handle functions once performed solely by the SDN controller. This makes the data plane a compelling target for attackers and is particularly vulnerable because the control plane processes untrusted input from the data plane (*e.g.,* packets from hosts) that can influence trusted control decisions [79, 81, 82, 90].

Unfortunately, understanding the attack surface in programmable data planes (PDPs) has not kept pace with this evolution. We identify several key trends and barriers to advancing this understanding:

1) *The data plane is a black box.* Unlike SDN controllers that run on commodity servers, switches[1] are often resource-constrained devices. Data plane programs are not typically designed to generate logs for understanding attacks, and their decision-making logic is highly dependent on incoming packets and internal state memory. This makes post-incident analysis difficult because there are often no observability mechanisms in place for trustworthy reasoning.

2) *Cross-plane interactions are poorly understood.* There is an incomplete understanding of how decision-making is distributed across and influenced by the interplay between the control plane and the increasingly complex data plane. While the control plane can push rules into the data plane, decisions made in the data plane are not always communicated back to the controller. Prior work [82, 90] has demonstrated that SDNs can be exploited through these cross-plane interactions. Information flows that affect the controller's view of the network may not reflect actual data plane behavior. Without visibility into the data plane, it is impossible to trace such dependencies. Existing network diagnosis tools that leverage network provenance fail to adequately capture the complexity of PDP programs written in P4 [80, 81, 85].

[1]We use "switch" and "target" (P4 terminology) interchangeably to refer to generic network forwarding devices that process and forward packets in a network.

3) *Traditional observability does not scale.* Naively implementing existing logging and dependency tracking mechanisms and approaches would be ineffective due to scale and performance constraints. The data plane must process vast volumes of packets at high speed and throughput. Tracking metadata for every event would introduce unacceptable overhead.

Overview. We present ProvP4, an approach for observing and diagnosing attacks on P4 programmable data planes. ProvP4 enables network and security operators to understand the root causes of attacks that exploit both the control and data planes of P4-enabled SDN. ProvP4 is designed to function as a network reference monitor for secure auditing to ensure that P4 programs cannot bypass or tamper with ProvP4's provenance collection mechanisms.

ProvP4 leverages network provenance techniques that track metadata to capture the dependencies and evolution of the network's behavior over time. This approach helps in understanding the processes involved in the generation, use, and derivation of network state (*e.g.,* forwarding rules, topology, switch memory state), and for attributing responsibility to specific system principals. Provenance precisely identifies historical actions that contributed to an observed effect, making it invaluable for attack investigations.

Several technical challenges arise in designing and building ProvP4. First, capturing a complete view of both planes must be done in a way that ensures completeness and security over metadata. Second, provenance modeling must be amenable to attack investigation, which requires minimizing false dependencies. ProvP4 optimizes for collecting provenance only on packets that meaningfully alter the data plane state. To ensure provenance collection that supports useful queries, ProvP4 employs program analysis and targeted instrumentation to collect only relevant provenance and repurposes unused switch memory for storage.

We demonstrate ProvP4's efficacy through three case studies of diagnosing attacks against P4-based network defense systems, selected to illustrate ProvP4's broad applicability across different types of data plane functionality. NetHCF [55] emphasizes control-data plane interactions, while SECAP [74] implements functionality almost entirely within the data plane. We show how various attacks can be carried out against such security-focused systems and how ProvP4 can be used to successfully diagnose their root causes.

In addition, we conduct a performance evaluation to assess ProvP4's impact on overhead costs. We find that ProvP4 imposes minimal performance penalties (up to 3.8 ms) on round-trip latency, while providing observability and diagnostic capabilities previously unavailable in programmable data plane architectures.

Contributions. Our paper makes the following contributions:

(1) the design of a programmable data plane provenance model that captures P4 semantics;
(2) the design and implementation of ProvP4, a tool to observe and reason about security attacks against programmable data planes; and
(3) a comprehensive security and performance evaluations demonstrating the efficacy of our approach.

2 Background and Motivation

The P4 domain-specific language enables customizable packet processing pipelines, supporting a wide range of offloaded network and

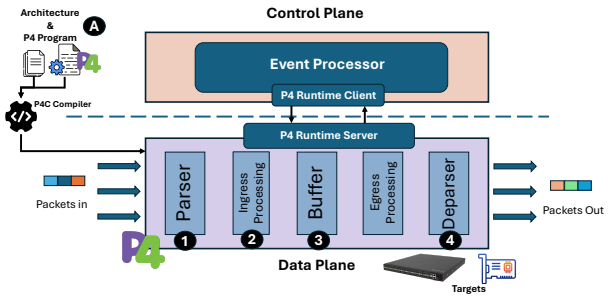


Figure 1: High-level overview of the P4 architecture.

security applications, including congestion control [36], load balancing [61], network consensus [28], and network telemetry [50]. The P4 specification [11] defines an *architecture* (Figure 1) as a collection of component interfaces exposed to programmers for implementing packet processing behavior.

P4 Pipeline. To specify the data plane's packet processing behavior, a programmer specifies three components in a *program*: parser, control, and deparser. The *parser* component extracts a packet's header fields, the *control* block specifies how to process packets using match-action tables (MATs), memory, and arithmetic logic units (ALUs), and the *deparser* specifies how packet headers need to be reconstructed and sent to the network.

P4 Targets. A P4 program and its architecture are compiled to a specific *target, i.e.,* a device implementation capable of executing the program in the data plane. Vendors typically provide both the architecture definition and compiler for their targets. The P4 community has also developed vendor-independent architectures such as Bmv2's v1model [10], PSA [13], the SmartNIC PNA model [14], and XDP/eBPF [12], alongside vendor-specific architectures including Intel Tofino's TNA [15], Xilinx FPGAs [16], and Nvidia/Netronome NICs [5, 18]. The bmv2 software switch using the v1model architecture acts as a vendor-independent reference for an *abstract forwarding model* [25] and is also the most popular and de facto target to develop and release P4 programs [37, 47].

P4 State. P4 exposes two types of stateful constructs: *tables* and *externs* [11]. Tables are read-only from the data plane, but externs can be read from and written to by the data plane. All stateful elements must be explicitly allocated at compile time to avoid dynamic memory allocation. In most targets, *registers* implement externs for stateful memory. The v1model exposes two methods, `read()` and `write()`, to read from and write to the registers, respectively.

P4 for Security. Given that P4 enables stateful packet processing, prior work has demonstrated the feasibility of offloading various security functions to the P4 data plane such as distributed denial-of-service (DDoS) detection [96] [59], link flooding prevention [92, 99], covert channel prevention [91], link failure detection [38], BYOD policy enforcement [64], and information flow control [22].

Security applications commonly rely on state to track features such as TCP connection state [55, 59, 91]. This state is updated in response to arriving packets, creating opportunities for adversaries to manipulate state transitions and violate system security properties [86]. We present a concrete example of such an attack and outline the observations required to analyze it.

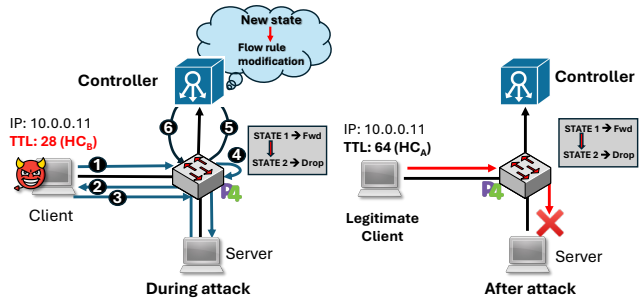
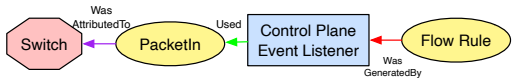
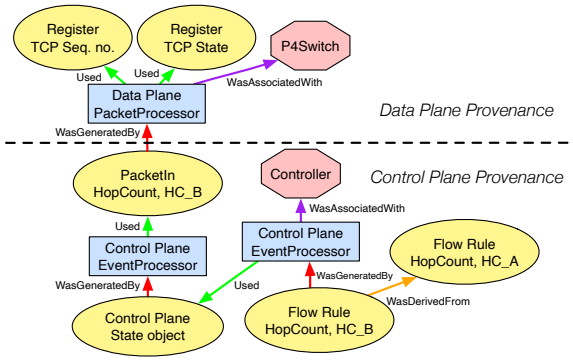


Figure 2: Motivating attack scenario using the NetHCF [55] system's P4-enabled data plane. By spoofing a TCP handshake, the adversary manipulates the data plane to change flow table state (*i.e.,* Fwd → Drop).



(a) Existing approach using PicoSDN: No observability of data plane stateful behavior that triggered the PacketIn packet.



(b) New approach using ProvP4: The flow rule state changed from HC_A to HC_B due to a PacketIn packet, which was triggered by the data plane that used the switch's register values.

Figure 3: Investigation using PicoSDN [81] and ProvP4.

2.1 Motivating Scenario

Suppose a network uses a P4-based data plane and a centralized SDN controller. We consider a security-oriented application that computes a hop count at runtime and signals to the controller about any change in the feature. The controller then modifies the match-action table (MAT) rules to incorporate the change into the current network. This general approach follows a common design paradigm seen in many P4-based security applications [45, 55, 99].

An adversary's goal is to manipulate the switch-local information using malicious packets in a way that poisons the data plane state, causing changes in its packet processing behavior. This attack undermines an application's security guarantees from the data plane, and observing only from the controller would be ineffective.

Attack. Figure 2 presents a concrete example of this class of attack, focusing on a P4-based implementation of the hop-count filtering (HCF) mechanism [42, 84], which aims to detect and mitigate IP spoofing by validating the expected hop count of incoming packets. NetHCF [55] is a high-performance, line-rate in-network system that realizes HCF functionality using P4. In NetHCF, the data plane dynamically identifies "hot" IP addresses and computes their hop counts by extracting time-to-live (TTL) values from IP headers and correlating them with known initial TTLs.

During the TCP three-way handshake, the data plane observes the sequence of packets to estimate the accurate hop count for each IP address. This locally computed hop count is then relayed to the SDN controller, which maintains a global mapping of IP addresses to their validated hop counts. Upon receiving this update, the controller installs or modifies flow rules in the data plane to enforce the correct hop count for subsequent packets, thereby mitigating potential spoofing attempts.

Specifically, ❶ **H1** sends a spoofed `TCP-SYN` packet to **H2** with the IP *source address* and *TTL* as shown in Figure 2. The switch forwards the packets ❷ and keeps track of the TCP handshake state. After the `ACK` ❸, the data plane uses the *TTL* value inside the packet to compute the hop count. Due to the hop count change (from HC_A to HC_B, due to spoofing), the switch state also changes ❹. This triggers a PacketIn to the controller ❺, and the controller issues a MAT rule modification into the data plane ❻ to check for the updated hop-count. Figure 2 shows the effect of the scenario. Whenever a host with the same IP address tries to join the network, the packets are dropped, leading to denial of service (DoS).

Investigation. An operator is alerted about the suspicious activity from the network's Security Information and Event Management (SIEM) system and wishes to determine the attack's root cause. Equipped with initial evidence, such as an IP address or link information, the operator queries a network provenance system, such as PicoSDN [81], to trace the steps leading to the incident to be able to use that information to thwart future similar attacks.

Table 1: Comparison of prior approaches to SDN attack diagnosis. (CP = control plane, DP = data plane.)

	DP Protocol	Reasoning Model	Semantic Capabilities	Stateful DP
ForenGuard [85]	OpenFlow	CP	✗	✗
ProvSDN [80]	OpenFlow	CP	✗	✗
PicoSDN [81]	OpenFlow	CP+DP	Topology	✗
P4-Tracking [53]	P4	DP	✗	✔
ProvP4	**P4**	**CP+DP**	**Program**	✔

2.2 Current Limitations and Opportunities

An operator encounters several challenges when investigating such a scenario, as mentioned in the previous section, shown in Figure 3a. Using the state-of-the-art PicoSDN tool for SDN attack analysis, the operator cannot observe the data plane's stateful behavior. Although the operator may be able to infer that a PacketIn packet was involved, prior work would fail to address why it was generated or which resources are responsible for it. Based on the lack of observability and the inability to answer investigative questions, we identify two key limitations of the current state of the art:

Limitation (L1): Incomplete view of internal data plane effects. The shift from fixed-function switches to PDPs [25] has

introduced complex, stateful constructs that traditional provenance systems cannot observe. While predecessors like OpenFlow [60] relied on control-plane-mediated decisions, P4 allows for high-throughput, autonomous state transitions directly within the data plane. As shown in Table 1, existing SDN provenance solutions [53, 81] focus on control plane events and fail to capture these internal P4 state dynamics and lack the stateful analysis required to reason about how untrusted packets modify internal switch memory and necessary querying capabilities.

This lack of visibility creates significant security blind spots. For example, NetHCF [55] monitors TCP handshakes in the data plane to validate traffic; however, an insider colluding with an external attacker can spoof handshakes to manipulate hop-count computations [86]. Because these state changes occur entirely within the PDP and never trigger control plane events, prior provenance tools remain oblivious to root causes.

Limitation (L2): Lack of stateful investigative capabilities. From Table 1, existing tools focus on an OpenFlow-based data plane, which lacks the concept of stateful packet processing. PicoSDN offers various investigative capabilities for root-cause tracing, activity summarization, and forward-backward tracing algorithms, but none of their features allow for understanding P4-based stateful components and would fail to provide the analytical capabilities necessary to understand how stateful components can maliciously manipulate data plane decisions.

Our Solution: ProvP4. We introduce ProvP4 as an improvement over prior approaches. ProvP4 overcomes limitations **L1** and **L2** by constructing a provenance graph that enables precise root-cause analysis of attacks originating from data-plane behavior.

Figure 3b illustrates the provenance graph generated by ProvP4 for such an attack. Given the victim's IP address as the *initial evidence*, ProvP4 performs a lineage query on the provenance graph to extract the attack context. The resulting query reveals that the attack was executed via PacketIn, triggered through interactions with data-plane registers (*i.e.,* TCP sequence number and TCP state).

By tracking and linking data plane state changes involved in the attack, ProvP4 provides enhanced observability into data plane behavior. This enables a more comprehensive investigation compared to prior work, which limited analysis to the controller's perspective.

3 ProvP4 Goals and Challenges

ProvP4's primary goal is to enable network and security operators to *collect* and *analyze* security-relevant network events as comprehensible provenance graphs. These graphs support querying via graph algorithms to diagnose data plane attacks and understand stateful packet processing. ProvP4's design goals are as follows:

Goal (G1): Complete Observability. ProvP4 must provide a comprehensive view of the network (data plane packet processing, control plane event processing, and the interactions between them) to correlate events across execution planes for analysis capabilities not found in prior work **(L1)**.

Goal (G2): Analysis Capabilities. ProvP4 should aid operators in performing investigative queries. Querying algorithms should assume operator minimal knowledge about the underlying data plane program, and should be generic to capture different kinds of attacks and work across different kinds of programs **(L2)**.

Goal (G3): Security. To provide trustworthy assurances about network state changes, ProvP4 should serve as a "reference monitor" [20] such that the P4 target operates as a trusted kernel-space with untrusted "userspace" programs that cannot tamper or modify the provenance collection to avoid observability (**G1**).

We identify several major technical challenges in satisfying the aforementioned design goals and sketch our solutions:

(TC1): No Clear Userspace/Kernelspace Boundary. In traditional host operating systems, the kernel provides a natural location for inserting provenance hooks in system calls, which captures system events such as `read()`, `write()`, and `open()` [23]. Similarly, existing network provenance systems that focus on the network's control plane, such as ForenGuard [85] and PicoSDN [81], hook all controller API requests within the network operating system.

Unfortunately, data plane targets lack an equivalent formally defined kernel, and thus lack a kernelspace/userspace boundary for security. Each target vendor may implement proprietary APIs that limit visibility and restrict standardized instrumentation. The absence of a centralized execution context complicates the choice of where provenance hooks should be inserted, which impacts what level of granularity is appropriate for downstream reasoning **(L1)** and the security assurances of the collected provenance.

Our solution: We identify the P4 program as the lowest common level of abstraction that works across the diversity of targets. ProvP4 uses program analysis to extract a program's memory declarations, operations, and control plane communications. ProvP4 uses program instrumentation to incorporate "kernel" provenance registers that are isolated by compilation (Section 6.1). We ensure that the original program cannot undermine the provenance collection capabilities (Section 5).

(TC2): Reasoning Capabilities. In earlier OpenFlow-based SDN, changes in topology, device information, and flow rules are the primary drivers of network behavior, and prior provenance systems capture these factors adequately. P4, however, introduces packet-driven logic and persistent mutable state that may occur entirely within the data plane, which existing provenance models fail to capture and that leaves one unable to answer state-dependent reasoning queries. For example, prior models may create disconnected provenance (sub)graphs that do not adequately capture dependencies necessary to reason about root causes. Accurate reasoning further requires proper partitioning and versioning of activities to tackle *dependency explosion* **(L2)**. Reasoning depends on a provenance model that is expressive enough to represent stateful processing yet comprehensible for operators performing diagnosis.

Our solution: ProvP4 models P4's registers, MAT rules, and counters as data plane artifacts. ProvP4 tracks these artifacts with an internal artifact state store for modifications via read/write operations. After the operations are identified, ProvP4 adds appropriate edges between the nodes to capture the relationship (Section 4).

(TC3): Data Plane Event Scaling Capabilities. Historically, control plane provenance has been sufficient for capturing network state transitions, as control plane events occur at a significantly lower frequency than data plane traffic. However, PDPs allow individual packets to modify internal switch states (e.g., registers). Capturing provenance for every packet fails to scale.

Table 2: ProvP4 provenance model node types.

Type	Description
Entity	A data or control plane object that incorporates state (e.g., `MATRule`, `Register`, `Counter`, `PacketIn`, and `PacketOut`).
Activity	An abstraction of processing events or packets inside the control or data plane (e.g., `PacketProcessor` and `EventProcessor`).
Agent	A system principal responsible for performing an activity (e.g., `Switch`, `Switch port`, `Control plane program`, and `Application`).

Our solution: We leverage the fact that only a small subset of data plane events actually impact state changes relevant to diagnosis. ProvP4 employs program analysis to identify the precise code paths and operations that modify these critical state variables. By instrumenting only these identified points, ProvP4 captures essential provenance data without full packet logging, thereby maintaining reasoning capabilities while ensuring line-rate scalability.

4 Provenance Model

We define a provenance model of relevant objects, states, identifiers, and system principals (or agents) for programmable data plane operations, inspired by W3C PROV [63] and incorporating programmable network semantics. Our model captures networking device metadata, packet interactions with device states, and state evolution. These interactions can be represented as provenance graphs to be used for investigative queries.

Definitions. A *provenance graph* is a directed acyclic graph (DAG) $G(V, E)$, where a node $v \in V$ represents an `Agent`, `Entity`, or `Activity`, with the various subtypes listed and described in Table 2. Entities denote network objects, activities represent packet or event processing, and agents control network actions. An edge $e \in E$ belongs to one of {`Used`, `WasGeneratedBy`, `WasDerivedFrom`, `WasAssociatedWith`}, capturing relationships such as read/write, modification, or attribution across data and control planes. For instance, Entities are *generated* or *used* by Activities, may be *derived from* other Entities, and Activities are *associated with* Agents. (See Appendix C for the complete provenance model.)

Partitioning and Versioning. A P4 program can be viewed as a long-running process executing multiple *actions* on incoming *packets*, causing various state changes. Without activity partitioning, operators face overwhelming and imprecise results from queries, causing the *dependency explosion* problem. Since packets are natural input units, ProvP4 partitions activities based on each packet's processing execution. To mitigate dependency explosion, ProvP4 uses a *version* counter to group state changes by packet execution, associating events with identical version numbers. Versioning also implicitly orders events, enabling query algorithms to efficiently prune irrelevant nodes from specific periods. For artifacts, ProvP4 uses *version-on-every-write* [65] such that if an entity is successively read without any intervening writes to it, all the reads will be from the same (last modified) version of the entity. ProvP4 adds a new entity node whenever there is a write operation.

5 Threat Model and Security Analysis

We assume that (i) P4-based targets (*i.e.,* the target firmware) and the P4 compiler are trusted and form part of the trusted computing base (TCB), and (ii) P4 programs are trusted but that their behavior may be exploited by untrusted input controlled by an adversary; both assumptions are consistent with prior work (*e.g.,* [80, 81, 85]).

ProvP4 is designed as a reference monitor for P4-based SDNs [20, 34]. We assume adversaries want to attack the intended operation of a program (e.g., manipulation of state through arbitrary packets) *Defense:* These attacks will not be prevented but ProvP4 guarantees all persistent data plane changes will be recorded. Furthermore, if we assume adversary is aware of ProvP4, they may attempt to undermine the provenance collection itself. *Defense:* Such attacks will be prevented through NEAT properties described below.

- **Non-bypassable.** We embed provenance capture statements within the instrumented P4 program's control flow. We assume that a program is instrumented prior to being installed on the target.[2] Thus, an attacker cannot trigger control flow statements without invoking ProvP4's instrumentation and cannot install an uninstrumented program on the target directly.
- **Evaluable.** We minimize the TCB and make ProvP4's source code available (redacted for submission) to support independent verification and assurance.
- **Always invoked.** Instrumentation is tightly coupled with state changes, thus ensuring provenance is consistently collected whenever state changes occur.
- **Tamper-proof.** The P4 target is included in the TCB, and ProvP4 does not expose APIs to the original "userspace" program for modifying provenance "kernelspace" registers. The compilation process enforces no unauthorized writes to kernel registers to ensure tamper-proof provenance.

Security of ProvP4. We also consider attacks by a knowledgeable adversary that target provenance security, integrity, availability and completeness.

- **Evasion:** An adversary cannot evade provenance partitioning or versioning by crafting packet sequences, as events are tied to packet execution and the version counter is incremented at pipeline entry, ensuring serialized packet processing.
- **Denial-of-service:** An adversary may generate high-volume stateful traffic to stress provenance storage. However, since provenance events are buffered via a stateful message broker, event loss is avoided assuming adequate resource provisioning.
- **Integrity:** Under the stated threat model assumptions and separation of kernel space, adversaries cannot inject forged labels into the provenance system, preserving event integrity.
- **Completeness:** ProvP4 focuses on persistent data plane state (registers) rather than transient per-packet metadata. As all data plane state changes are logged, any metadata written to registers for persistence is captured.

TCB Justification. ProvP4 trusts the P4 toolchain (compiler and target) as part of its TCB. While this enlarges the TCB, we justify this based on two established techniques: (1) *Compiler testing:*

[2]In our proof-of-concept implementation of ProvP4, we rely on a P4 program being correctly installed and verified as part of the network operator's policy of operational security practices. However, orthogonal work on using remote attestation (RA) [75] can be implemented without relying on redesigning ProvP4.

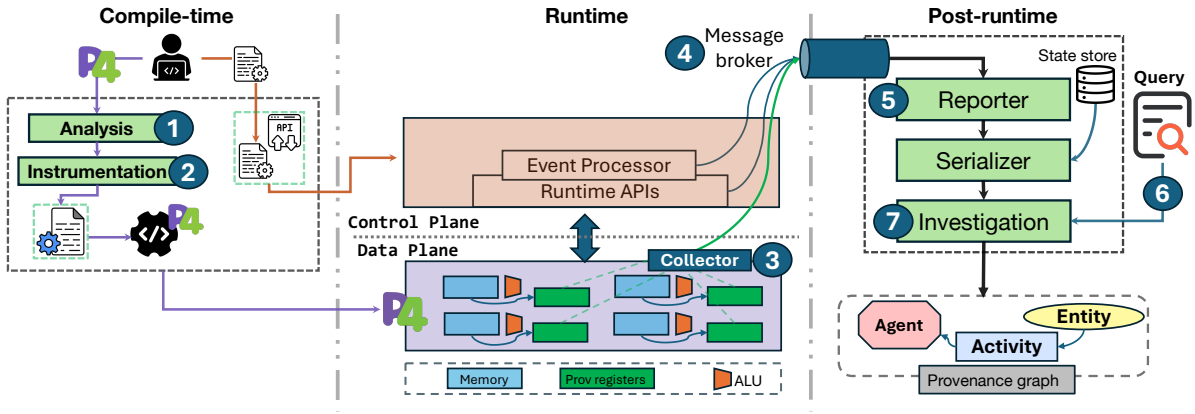


Figure 4: ProvP4 components and workflow. (1) The program analysis component parses the compiled P4 program to extract stateful events. (2) The program instrumentation component augments the P4 source to record provenance metadata at runtime. (3) A lightweight provenance collector extracts event contents and forwards them to a message broker (4) for serialization. (5) The ProvP4 reporter serializes the stateful events from the message broker after adding relevant node information to the provenance graph. The provenance kernel saves this information in a persistent database for storage and querying. During investigation, the operator supplies initial evidence and queries, which ProvP4 executes (6) and provides resulting answers (7).

prior work, such as Gauntlet [70] and P4Testgen [69], tests for compiler bugs and is directly integrated into the p4c compiler toolchain; (2) *Attestation:* remote attestation (RA) [75] can attest an augmented ProvP4 P4 program, and any deviation from the installed program would raise alerts. These independent techniques can be used alongside ProvP4 to secure the entire P4 ecosystem. Finally, a buggy P4 toolchain is a broader supply chain threat that affects the entire P4 ecosystem rather than ProvP4 specifically.

6 ProvP4 Design

ProvP4's design (Figure 4) incorporates a *pre-runtime* phase to instrument the intended P4 program to execute, a *runtime* phase that collects provenance during execution, and a *post-runtime* phase to investigate and analyze past actions for understanding attacks.

6.1 Pre-runtime Phase

P4 specifically facilitates stateful operations through memory units known as *registers* to maintain various stateful information, but collecting such information is challenging.

Naive Solution. One approach for collection would be to directly modify the P4 target to capture state changes. However, P4 targets are often proprietary, and vendors restrict modifications. Logging state changes of every packet incurs prohibitive overhead.

Our Solution. ProvP4 adopts a two-phase approach.

During *program analysis*, ProvP4 uses the p4c compiler's output to extract stateful information from the original P4 program, including register declarations, read/write operations, and control plane interactions. of the full data plane pipeline, and ProvP4 identifies stateful operations from p4c's JSON description of the pipeline. ProvP4 scans the *actions* keys for *register_read* and *register_write* references, extracting (for each operation) the index and value fields (name and size), line number, exact source fragment, and filename. For data-plane-to-control-plane communication, ProvP4 further extracts *clone* operations from the compiled JSON. ProvP4 then filters and retains only those stateful operations that modify the data plane via MATRules or registers, as these directly influence packet processing decisions[3]. As a result, ProvP4 achieves 100% instrumentation coverage for MATRule and register changes, ensuring complete observability (**G1**). (See Appendix D for more details.)

Table 3: ProvP4 control plane APIs.

API call	Description
RecordAPICall(*type*, *entity*)	Record a control plane API call of *type* on an *entity*
RecordActivity(*name*)	Record an event process function *name*

During *program instrumentation*, ProvP4 augments the original P4 code by adding statements that write stateful metadata to dedicated "kernelspace" registers (*provenance registers*) during runtime. The width of each provenance register is determined by the register index and value size, a unique identifier for the P4 action executing the stateful instruction, the operation type (read or write), and a version number. Due to target-specific restrictions of control plane APIs such as Thrift [1], ProvP4 records a maximum of 64 bits of stateful information per provenance metadata record. If program analysis determines that the width exceeds this limit, ProvP4 notifies the user to manually reduce the affected register sizes in the original P4 program. Additionally, when target-specific register limits are exceeded, ProvP4 can leverage external data plane memory [51]. (See Appendix E for an example.)

6.2 Runtime

The disparity between data plane throughput and control plane processing speeds makes metadata management challenging.

[3]P4 *counters* are ignored, as they cannot be read from the data plane, and any meaningful state change must ultimately be reflected in the control plane as either a register update or a MATRule modification, both of which ProvP4 tracks directly.

Naive Solution. One approach might attempt to continuously stream runtime provenance events from the data plane to a central controller or external storage in real-time. However, at line-rate speeds, the resulting metadata volume would saturate the control plane channel, leading to significant performance degradation, increased packet latency, and potential event loss.

Our Solution. ProvP4 employs a hybrid approach:

- For the control plane, lightweight APIs (Table 3) transmit events directly to a message broker, which minimizes the controller's processing load.
- For the data plane, ProvP4 periodically queries provenance registers and attaches metadata such as *event type*, *operation* (Read-/Write), and *device details*. The data is serialized per the provenance model (Table 2) and stored by the P4 reporter component.
- Provenance registers use a wrap-around strategy when full, which avoids resets and ensures line-rate state changes are captured without performance degradation.

The P4 reporter sequentially extracts events and adds provenance semantics by inserting appropriate nodes and edges from Table 2. It maintains an internal *state store* to track the evolution of artifacts such as registers and MATRules, ensuring that provenance edges are added only on state changes (*i.e.,* write operations). The reporter also captures the evolution of how artifacts change across versions, which is critical for meaningful investigation as we demonstrate in our case studies (Section 8.1).

6.3 Investigation

Querying complex provenance graphs with numerous nodes, edges, and annotations poses significant challenges for operators seeking clear, meaningful insights.

Naive Solution. Manual exploration or simple text-based searches are inefficient and do not scale to large graphs. Additionally, text-based searches would be unable to utilize the lineage relationships that are provided by provenance graphs.

Our Solution. ProvP4 addresses this via a dedicated *query surface*, which employs automated *query algorithms* to process investigative queries, reducing manual effort and improving accuracy. Operators can pose queries, such as:

- **Q1:** PacketTracer: What does the CP-DP communication look like at switch, S_i and port, P_i?
- **Q2:** LinkTracer: What does the traffic look like at a link, (Switch_s1:Port_1) ↔ (Switch_s2:Port_2)?
- **Q3:** MemoryTracer: How did the value at index change in a register, R?
- **Q4:** OrderTracer: What are the components active during a specific period of an event?

Operators provide *a priori* information to the query surface, and ProvP4 selects and applies the appropriate query algorithms to generate provenance graphs as outputs.

6.3.1 Query Surface. The query surface usage is shown below for the scenario when the operator wishes to execute query **Q2**. The operator provides LinkTracer as the query algorithm, with (Switch_s1:Port_1) ↔ (Switch_s2:Port_2) as *a priori* information or *initial evidence*:

```
--query_algorithm LinkTracer
--device_id Switch_s1 --port Port_1
--device_id Switch_s2 --port Port_2
--output DOT
```

The output of this query is a provenance graph that provides the complete egress traffic from S1, P1.

Algorithm 1 PacketTracer

Input: Provenance graph $\mathcal{G}_{\text{full}}$, Switch S_i, Port P_i, traffic_direction T
Output: Set of vertices V that match the input properties
1: **if** T == "egress" **then**
2: direction_key $D \leftarrow$ "egress_port"
3: entity_key $E \leftarrow$ "packet_out"
4: **else if** T == "ingress" **then**
5: direction_key $D \leftarrow$ "ingress_port"
6: entity_key $E \leftarrow$ "packet_in"
7: **end if**
8: node_constraint $C \leftarrow$ ("id" == S_i) ∧ ($D == P_i$) ∧ ("type" == E)
9: vertex $V \leftarrow$ getVertex($\mathcal{G}_{\text{full}}$, C)
10: **return** V

Algorithm 2 LinkTracer

Input: Link information (Switches: S_i, S_j; Ports: P_i, P_j)
Output: Combined traffic graph $\mathcal{G}$ for S_i:P_i and S_j:P_j
1: $\mathcal{E}_{S_i,P_i} \leftarrow$ packet_tracer("egress", P_i, S_i)
2: $\mathcal{I}_{S_j,P_j} \leftarrow$ packet_tracer("ingress", P_j, S_j)
3: traffic $\mathcal{E} \leftarrow$ getLineage($\mathcal{E}_{S_i,P_i}$, 'ancestors')
4: traffic $\mathcal{I} \leftarrow$ getLineage($\mathcal{I}_{S_j,P_j}$, 'ancestors')
5: $\mathcal{G} \leftarrow \mathcal{E} \cup \mathcal{I}$
6: **return** $\mathcal{G}$

6.3.2 Query Algorithms. The query surface uses Algorithm 1 and Algorithm 2 to query for information that the operator uses for their analysis.

PacketTracer. Given information about a specific switch and a port, ProvP4 invokes Algorithm 1 to query the control plane ↔ data plane communication via PacketIn and PacketOut. As the PacketIn is generated due to traffic from a switch port, we query the *ingress_port* and similarly *egress_port* for PacketOut. Finally, we create a constraint (line 8) to filter out nodes (line 9) and return the relevant nodes.

LinkTracer. Given information about a link, the network operator can check for abnormal behavior by invoking Algorithm 2. The main insight is that for a direct link between two endpoints, the traffic at both ends should look similar. ProvP4 queries the ingress and egress traffic (lines 1- 2) at the specified switch ports and returns the combined provenance graph, $\mathcal{G}$ (line 6). Therefore, the operator can use the graph $\mathcal{G}$ to isolate suspicious links and use this information as a starting point to analyze specific (MemoryTracer) or active components (OrderTracer).

MemoryTracer. Collecting provenance at the granularity of P4 stateful memory has the drawback that not all registers or MAT rules are equally relevant. For example, in NetHCF, only 5 out of 7 registers produce meaningful provenance data, while others track routine states or packet counts. Without filtering, operators

may face overwhelming provenance nodes. To address this, we enable querying by specifying stateful memory using specific names, indices, or values.

OrderTracer. After provenance collection, operators can focus on a specific temporal window. ProvP4 uses its versioning scheme (Section 4) to impose implicit ordering, allowing LinkTracer and MemoryTracer to isolate a target event and query events before or after it, thereby revealing switch state for the desired interval.

7 Implementation

We implemented ProvP4's program analysis, augmentation, collector, serializer, and query front-end using Python v3.10. The collector uses the Thrift control plane API to extract provenance events from the network devices and to send such events to a RabbitMQ message broker queue. A custom SPADE [33] module, *P4 reporter*, parses the messages from the queue, adds relevant provenance entity data, edge relations, and sends it to the SPADE kernel. For control plane provenance, ProvP4 exposes the APIs listed in Table 3, which are used to manually instrument the control plane program or controller with provenance collection capabilities. The P4 programs are compiled using the official p4c compiler v1.2.4.3. We implemented the querying engine on top of SPADE's QuickGrail [32]. Our implementation is available at https://github.com/GUSecLab/ProvP4.

8 Evaluation

We now evaluate ProvP4's efficacy at analyzing security attacks against security-oriented P4-based systems (Section 8.1) and its performance on storage and network resources (Section 8.2).

8.1 Security Case Studies

We evaluated ProvP4 using three representative P4-based systems: NetHCF [55] and SECAP [74] and P4Knocking [95]. These systems not only focus on network defense applications (an attractive target for attackers) but also serve as representative examples of two classes of common design patterns of P4-based systems:

(1) **Complex cross plane interactions** (*e.g.*, NetHCF): data plane and control plane communications that influence packet processing and routing behavior [21, 43, 45, 55, 56, 72].
(2) **Data plane driven decision-making** (*e.g.*, SECAP): data plane state primarily influences packet processing and routing behavior [59, 74, 95].

Case Study 1: NetHCF Spoofing Attack

System background. Revisiting the scenario in Section 2.1, NetHCF uses the data plane (*cache*) for managing the most active IP addresses and the control plane (*mirror*) for handling newer IP addresses and managing the IP2HC table. The two components share network state information and can directly modify each other's views. NetHCF operates in *learning* and *filtering* modes; in the learning mode, the `TCPSessionMonitor` tracks TCP state transitions and updates the `IP2HC` table upon successful handshakes to maintain accurate hop-count values.

Threat model and attack scenario. Consider the NetHCF deployment in Figure 2, where hosts **H1** and **H2** are connected to the NetHCF P4 data plane. **H1** and **H2** act as client and server (HTTP) respectively. The threat model [86] assumes the adversary controls **H2**, *i.e.*, they

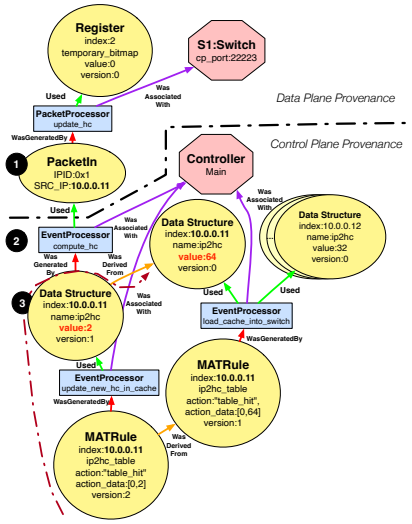


Figure 5: Provenance graph of lineage query on the malicious flow rule. The flow was installed due to a change in internal controller state, triggered by a PacketIn data plane event.

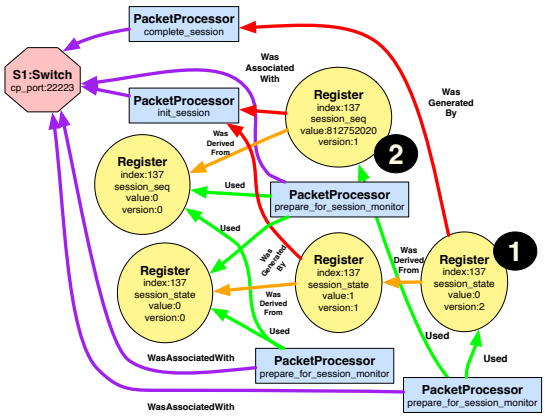


Figure 6: Artifacts used by the active components during PacketIn event.

may send and receive TCP traffic but cannot alter the NetHCF P4 program. The switch and controller remain trusted.

Investigation. An operator wants to understand the spoofing attack and find the root cause. We assume that the operator knows the legitimate host machine's IP address (`10.0.0.11`), which forms the *initial evidence*. The operator asks the following investigative queries (IQs):

- **IQ1:** Which data plane flow rules match the *initial evidence*?
- **IQ2:** What event triggered the installation of that flow rule?
- **IQ3:** What other network components were active during the malicious flow rule installation?

For **(IQ1)**, the operator provides the *initial evidence* as a query parameter. ProvP4 queries the provenance graph using the MemoryTracer component and searches for all the table rule entities

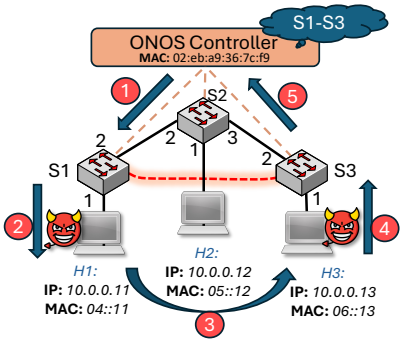


Figure 7: Experimental topology for the SECAP case study. Host H1 and H3 use an out-of-band channel to relay LLDP messages to fabricate the (S1:1) ↔ (S3:1) link.

with `10.0.0.11`. The operator knows the flow table state and proceeds to reason about what triggered the flow rule. For **(IQ2)**, the operator invokes the lineage component using the MemoryTracer and receives the flow rule's lineage output. Figure 5 shows the lineage of the flow rule. From the lineage query, the operator observes:

(1) The flow-rule installation was triggered by a PacketIn message from the data plane ❶.
(2) This PacketIn invoked the `compute_hc` processor, causing the hop count to be recomputed ❷.
(3) The recomputation updated the controller's hop-count state ❸.
(4) Since the PacketIn originated in the data plane, the operator next examines which data plane components were active.

This query provides the root cause of the attack. Now, the operator wants to understand the data plane components active up until PacketIn is triggered. For **(IQ3)**, the operator utilizes the OrderTracer (from Section 6.3.2). To gain additional insight, the operator queries the artifacts that were active during this event. Figure 6 shows that the two main components active were the two registers: *session_state* ❶ and *session_seq* ❷.

In summary, the operator learns that this suspicious event originated at the data plane, propagated to the control plane (via PacketIn), poisoned the global view of the IP2HC mappings, and resulted in a flow rule with incorrect hop count data that lead to packet drops and denial of service for a legitimate client.

Compared to prior approaches. While ForenGuard and PicoSDN provide only a control plane vantage point, they cannot answer investigative queries that span both planes. In contrast, ProvP4 incorporates data plane state and supports cross plane querying, giving operators the unified visibility that those systems lack.

Case Study 2: SECAP - Relay-type LFA

System background. SECAP [74] is a P4-native, in-network defense against topology-poisoning attacks such as data plane ARP cache poisoning (DACP) and relay-style link fabrication attacks (LFA). It avoids sending suspicious traffic to the control plane by verifying source IP–MAC information directly from ARP and LLDP headers. SECAP uses P4 stateful registers to track the mapping between switch ports and their connected devices.

Threat model and attack scenario. Figure 7 shows the experimental topology of a relay-based LLDP attack. An adversary controls hosts

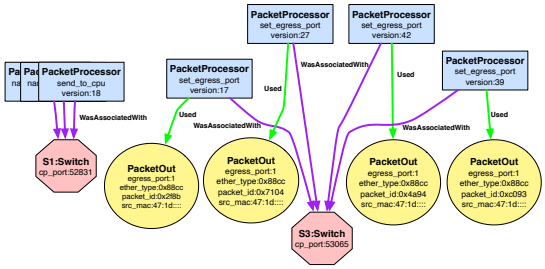


Figure 8: Relevant provenance graph for traffic at Switch S1 and S3 through Port 1. LLDP (EtherType: 0x88cc) packets are sent from one end of the link (S1-S3) via `S3:Port 1` but did not reach the other end `S1:Port 1`.

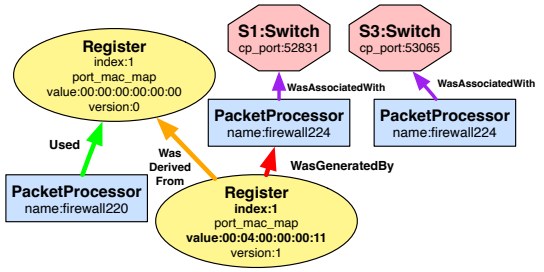


Figure 9: Provenance graph of SECAP's port-mapping registers. Switch 1's mapping register maps port 1 (index 1) to a MAC address, but Switch 2's mapping register is empty.

H1 and **H2**, and the hosts exchange packets through an out-of-band channel or by using a Generic Routing Encapsulation (GRE) tunnel. The adversary's main goal is to fabricate a direct link between switches **S1** and **S3** by poisoning the view of the SDN controller [4]. The switches are assumed to be trusted. Assume hosts **H1** and **H2** connected to their respective switches, and no corresponding SECAP mappings exist at initialization. Figure 7 shows the attack:

❶ ONOS issues a PacketOut containing an LLDP packet for a specified *egress_port*.
❷ Switch:**S1** decapsulates the LLDP packet and forwards it to that port (*i.e.,* Host:**H1**).
❸ Host:**H1** relays the packet unchanged to Host:**H3** via an out-of-band channel.
❹ Host:**H3** forwards the packet to Switch:**S3**.
❺ Switch:**S3** returns the LLDP packet to the controller through `CPU_PORT` using ONOS's default rule.

Investigation. The operator receives a topology change alert[5] and turns to ProvP4's analysis. Through the analysis, the operator wants to answer the following investigative queries (IQs):

- **IQ1:** Does the new link show any abnormal behavior? (*generic* question)
- **IQ2:** Are the P4 registers mapping consistent with the switch-local and packet views? (*program-specific* question)

[4] In this case, the ONOS SDN controller's `LLDPLinkProvider` module.
[5] Generated by ONOS's `TopologyManager`.

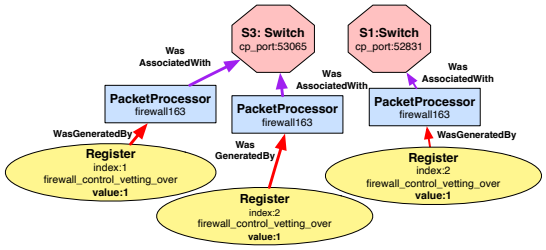


Figure 10: Relevant features for the link monitor register query. The switch-local view of the link (S1-S3) appears different in both switches.

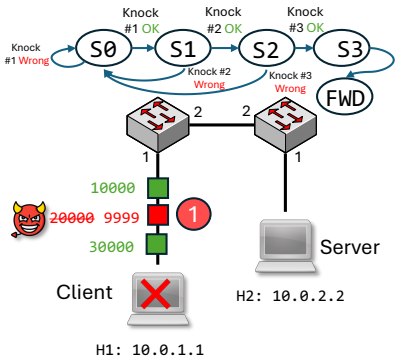


Figure 11: Experimental topology for the P4Knocking case study. The adversary changes the correct knock sequence from H1 (client) to H2 (server).

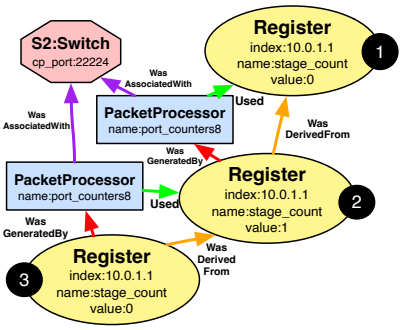


Figure 12: Relevant features for the memory tracer query. The knock count state machine received an incorrect sequence and reverted to start.

For **IQ1**, the operator supplies the link end-points as *initial evidence*, and invokes LinkTracer to test link symmetry. ProvP4 compares PacketOut–PacketIn control traffic at S3:port 1 and S1:port 1 using unique packet identifiers. As shown in Figure 8, some packets sent from S3:Port1 never arrive at S1:Port1, revealing that the link is asymmetric and warrants further investigation.

For **IQ2**, the operator checks whether SECAP's data plane view aligns with the packets observed at each port. The operator specifies the relevant SECAP artifacts, *i.e.,* mapping registers (`port_mac_map`, `port_ip_map`) and the status register (`vetting_over`) to the MemoryTracer query algorithm. Figure 9 shows the registers governing switch-local port–address mappings. The outputs reveal that S1 and S3 hold inconsistent views of the same link: S3's mapping registers are empty—indicating no traffic on its port, while S1 records a mapping for Port 1. The operator concludes that this mismatch signals abnormal behavior.

Using the `vetting_over` register (Figure 10), the operator further observes that SECAP treats S1:Port1 as a non-switch link, whereas S3:Port1 is classified as a switch link due to receiving an LLDP packet. The operator infers that a device on S3:Port1 relayed LLDP packets from S1:Port1, explaining both the incorrect local mappings and the observed link asymmetry.

Compared to prior approaches. Although ForenGuard and PicoSDN observe ONOS control plane activity, they provide no visibility into SECAP's data plane port mappings or LLDP processing, which would prevent **IQ1** and **IQ2** from being investigated. In contrast, ProvP4 enables direct querying of data-plane state, exposing SECAP's LLDP processing behavior.

Case Study 3: P4Knocking - Denial-of-Service (DoS)

System background. P4Knocking [95] offloads firewall functionality to the data plane by implementing the classic port-knocking algorithm in P4. P4Knocking utilizes stateful P4 registers to record the sequence of successful knocks and forwards connections only after all knocks are observed in order. Our case study focuses on the data plane only implementation [7].

Threat model and attack scenario. Figure 11 illustrates the attack topology. An adversary only needs the ability to inject or modify traffic destined for the P4Knocking data plane. By tampering with any packet in the knock sequence encoded as predefined TCP ports, the adversary can force the state machine to reset, thereby preventing the legitimate client (**H1**) from completing the sequence and connecting to the protected server (**H2**).

Investigation. The operator receives a complaint from a client about not being able to connect to the server and provides the IP address (`10.0.1.1`) as the *initial evidence*. The operator wants to answer the following investigative query (IQ):

- **IQ1:** How does the knocking sequence appear from the perspective of the data plane?

For **IQ1**, the operator leverages ProvP4's MemoryTracer to query the state of the knock sequence register (*stage_count*) and provides the `10.0.1.1` as *initial evidence*. The query output in Figure 12 shows the register advancing from state 0 (`Start` ❶) to state 1 (first correct knock ❷) and then reverting to state 0 ❸. The transition ❷ → ❸ reveals that the data plane received an incorrect knock. By comparing this with the client's original transmitted sequence, the operator can infer that the knock sequence was modified in transit, which caused denial of service.

Compared to prior approaches. Prior systems such as ForenGuard and PicoSDN fail to answer queries such as **IQ1** because they lack visibility into the P4 data plane, leaving practitioners unable to conduct meaningful investigations. In contrast, ProvP4 captures P4 state transitions directly.

Table 4: Storage cost comparison for PicoSDN and ProvP4. (PicoSDN values borrowed from PicoSDN's performance evaluation [81]).

Hops	System	Storage cost (1 Flow) [KB]	Estimated cost for 1000 flows [GB/hr].
1	PicoSDN	1.1	3.96
	ProvP4	0.464	1.67
2	PicoSDN	1.7	6.12
	ProvP4	0.640	2.30
5	PicoSDN	2.6	9.36
	ProvP4	0.865	3.11
10	PicoSDN	4.3	15.48
	ProvP4	1.342	4.83

8.2 Performance Evaluation

We further evaluate ProvP4's performance overheads on:

- **Storage**: What is the rate at which ProvP4 generates provenance metadata to store?
- **Network performance**: How much does ProvP4 impose additional latency overhead compared to a baseline without ProvP4?
- **Target overhead**: To what extent does ProvP4 consume data plane resources on the P4 target at runtime?

All the experiments were performed using an Intel Xeon E5-2640 v4 3.400 GHz CPU with 64 GB RAM and Ubuntu 22.04.1 OS with kernel version 5.15.0-56-generic. We conducted our evaluation using four P4-based systems: Heavy Hitter (HH) [2], NetHCF [55], SECAP [74] and P4Knocking [95]. The source code was obtained from their respective official open source repositiories [2, 4, 7, 9].

HH is a representative pathological stateful program, which performs per-packet stateful action (read and write). We used the reference bmv2 software switch [17] for evaluation due to its widespread popularity and common reference among P4-based system implementations [27, 38, 43, 48, 56, 64, 72, 74, 92]. We used linear network topologies to measure the network diameter's effect on provenance, similar to prior work [81].

Storage. ProvP4 maintains data structures to track artifact derivations. These structures are as numerous as stateful objects and depend on the specific P4 program. Rather than measuring the storage overhead of internal components, we measured the cost of external provenance graph storage using two metrics: the rate of *new provenance node* generation and *file storage size*. Finally, we compare the storage cost of ProvP4 with PicoSDN [81].

Setup. To evaluate the steady-state average number of new nodes generated per minute, we ran ProvP4 for 15 minutes across varying flow and network diameters, recording the events processed. We periodically measured the number of vertices, edges, and the size of the serialized provenance graph in compressed (gzip) DOT format. For the experiments comparing ProvP4 with PicoSDN, we used a reactive P4 forwarding program [6] to match PicoSDN's experimental setup.

Results. Figure 13 shows the storage cost results over varying number of flows [81, 85]. On average for a 15-switch linear topology with 1000 flows/min, ProvP4 produces 114, 956 nodes/min for NetHCF, 37, 321 nodes/min for SECAP, 86, 791 nodes/min for P4Knocking and 114, 676 nodes/min for HH. HH's node generation rate is higher since it is performing per-packet state changes as a pathological worst case. Figure 15 shows the compressed storage cost in megabytes (MB). Given that HH's generation rate is the highest, the slope is correspondingly highest. We note that the flow processing stop times are longer than 15 minutes, which we determined was caused by a bug of duplicate acknowledgements in ONOS outside of the scope of ProvP4.

Table 5: Resource usage comparison for different P4 programs. Here, CP usage: High (new per-flow), Low (independent of flow), and Medium (using threshold). w/ProvP4 column refers to original program + ProvP4 instrumentation.

Program	CP Usage	Total Registers		Extra Actions	Extra Metadata	LoC	
		Orig.	w/ProvP4			Orig.	w/ProvP4
NetHCF	High	7	17	3	2	575	672
SECAP	Low	11	18	3	3	1522	1585
HH	Medium	1	5	3	3	191	237
P4Knocking	-	1	3	3	2	346	366

Table 4 shows the estimated storage costs for ProvP4 and PicoSDN. We found ProvP4's storage costs are 57% lower than PicoSDN. ProvP4 generates less provenance than PicoSDN, though we note that each system captures provenance at different layers of the SDN environment.

Key Takeaways. Provenance storage depends significantly on the P4-based system and scales with a network's data plane diameter.

Network Performance. We measured the overall latency that ProvP4 adds to packet processing, specifically the average round-trip time (RTT) of sending traffic for host-to-host communication with and without ProvP4's provenance instrumentation.

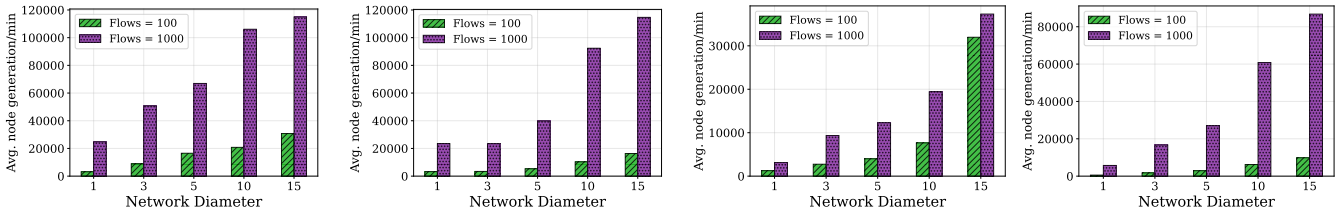
Setup. For HH, NetHCF, and P4Knocking, we used a *proactive* control plane—flow rules are directly installed to allow routing across the network. For SECAP, we used a *reactive*[6] control plane that utilizes the *fwd* (learning switch) ONOS application. We utilized `ping` [8] and `hping3` [3] to craft the necessary IP and TCP traffic for their respective applications and repeated each for 15 trials.

Results. Figure 14 shows the average latencies imposed with and without ProvP4, parametrized by the number of hops (*i.e.,* network diameter) and the P4 program running in the target switches. HH with a network diameter of 3 imposed the lowest (best) absolute and relative overheads across all configurations with an additional 0.052 ms (3.05%) cost imposed by ProvP4. NetHCF with a network diameter of 10 imposed the highest (worst) absolute and relative overheads with an additional 3.798 ms (26.7%). The increase in overall latency is expected as the increase in network diameter (*i.e.,* hops) requires additional provenance recording at each hop.

Key Takeaways. Across the systems and topologies evaluated, ProvP4 imposes a modest 3.05% relative latency overhead in the best case, a 0.052 ms absolute latency overhead in the best case, and a 3.798 ms absolute latency overhead in the worst case.

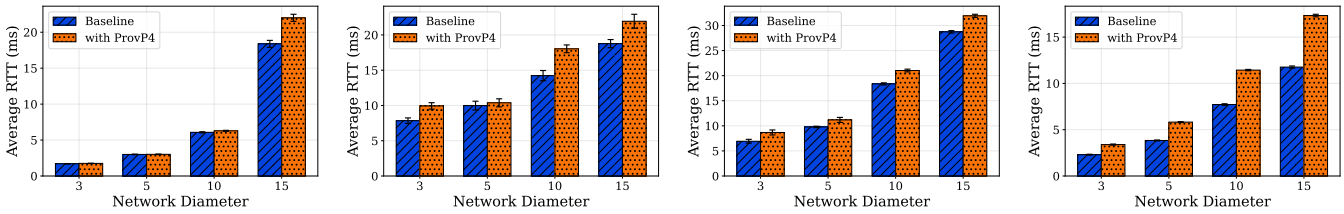
Target Overhead. As ProvP4 uses registers and metadata to record stateful events, we measured the associated memory overhead. Table 5 summarizes the additional usage. For a P4 program with **N**

[6]In *reactive* forwarding, the first packet of a previously unseen flow always incurs a higher latency as it does not match with any existing flow rules; the controller installs flow rules to handle future packets of the flow at data plane line rates.



(a) Heavy Hitter (b) NetHCF (c) SECAP (d) P4Knocking

Figure 13: ProvP4— storage cost rate for the four P4 systems, varied by network diameter and traffic rate.



(a) Heavy Hitter (b) NetHCF (c) SECAP (d) P4Knocking

Figure 14: ProvP4 latency overhead for the four P4 systems. (Error bars represent 95% confidence intervals.)

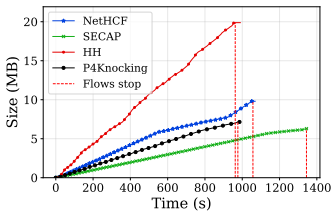


Figure 15: ProvP4 compressed file storage cost using a 5-switch linear topology with 1000 flows/min for 15 minutes.

registers, ProvP4 adds at most **2N+2** registers. Metadata and action additions are constant, used for tracking available slots and versioning. Provenance storage cost scales with the program's statefulness. In practice, overhead can be reduced by avoiding overly large register widths and leveraging external memory [51].

9 Related Work

SDN security. SDN security has been extensively studied, with defenses targeting control-plane manipulation [94], link fabrication attacks [29, 39, 41, 73], and race conditions [93]. These systems are designed for OpenFlow-based architectures and operate primarily from the controller's vantage point. As a result, they lack visibility into data plane execution semantics and persistent state, and cannot reason about attacks that exploit programmable, stateful P4 pipelines. ProvP4 directly addresses this gap by observing and analyzing attacks within the data plane itself.

Programmable Data Plane Security. A large body of work demonstrates that complex security functions can be offloaded to the PDP [22, 38, 45, 54, 55, 59, 74, 91, 92, 95, 96, 99]. These systems treat the data plane as an enforcement substrate and assume benign or well-behaved inputs. Few papers explicitly consider adversarial packet streams or malicious manipulation of data plane state [24, 44, 52, 71, 79, 86, 98], and these focus on demonstrating vulnerabilities rather than providing general-purpose observability. Recent prior work, such as TAP4S [19], also explored a new type system to perform information flow control analysis in detecting security policy violations for P4 programs. ProvP4 is orthogonal: it does not enforce a particular policy, but instead exposes the causal structure of stateful P4 execution under attack. Remote attestation [75] strengthens integrity guarantees for data plane execution. Attestation alone cannot explain how or why an attack manifests; ProvP4 provides this missing capability. Prior work such as P4Auth [67] employs cryptographic primitives to secure the control channel against adversarial manipulation by a compromised switch. However, such approaches operate under a fundamentally different threat model; their defensive guarantees are orthogonal to, and complementary with, those of ProvP4.

Provenance and Attack Analysis. Provenance has been widely used for attack causality analysis in systems and networks [40, 62, 66, 76]. Early work focused on explaining forwarding and routing decisions [26, 89, 100], while more recent efforts model the network as an operating system and capture cross-plane dependencies [58, 80, 81, 83, 85]. Prior systems assume OpenFlow-based data planes and abstract away the mutable state central to P4. Consequently, they cannot capture causality in attacks that exploit P4 control flow, registers, or custom state machines like ProvP4 does.

Network Verification and Debugging. Network verification [46, 49, 68, 97] checks correctness properties before or during execution, and P4 verification [30, 31, 57, 87] detects known bugs. Verification cannot diagnose emergent or adversarial behaviors at runtime. Network debugging [35, 77, 78, 88] assists with performance diagnosis but lacks P4 semantics. ProvP4 complements these approaches by enabling runtime, attack-driven analysis of P4 data plane behavior.

10 Conclusion

We introduced ProvP4, a provenance-based cross-plane observability framework for investigating and analyzing P4-based PDP attacks. By modeling the data plane's stateful activities and instrumenting programs for provenance runtime collection, ProvP4 constructs provenance graphs that empower operators to perform state-aware investigative queries. ProvP4 enhances the detection and analysis of complex attack scenarios. Our performance evaluations demonstrate that ProvP4 incurs modest overhead, while security case studies highlight its versatility in addressing various data plane attack scenarios within P4 systems.

Acknowledgments

This material is based upon work supported by the National Science Foundation through grants CNS-2229455 and CNS-2346499.

References

[1] [n. d.]. BMv2 Thrift API/Simple Switch. https://github.com/p4lang/behavioral-model/tree/main/targets/simple_switch.
[2] [n. d.]. Heavy Hitter GitHub. https://github.com/nsg-ethz/p4-learning/tree/master/examples/heavy_hitter.
[3] [n. d.]. hping3 tool. https://linux.die.net/man/8/hping3.
[4] [n. d.]. NetHCF GitHub. https://github.com/NetHCF/NetHCF.
[5] [n. d.]. Netronome NFP Architecture. https://old.hotchips.org/wp-content/uploads/hc_archives/hc25/HC25.60-Networking-epub/HC25.27.620-22nm-Flow-Proc-Stark-Netronome.pdf.
[6] [n. d.]. P4 L2 Learning Application. https://github.com/nsg-ethz/p4-learning/tree/master/exercises/04-L2_Learning/thrift.
[7] [n. d.]. P4Knocking Github. https://github.com/ederollora/NETPROC_P4Knocking.
[8] [n. d.]. Ping tool. https://linux.die.net/man/8/ping.
[9] [n. d.]. SECAP GitHub. https://github.com/smythtech/SECAP_Switch_Prototype.
[10] [n. d.]. v1model. https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4.
[11] 2017. P4-16 Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html.
[12] 2018. p4c-xdp. https://github.com/vmware-archive/p4c-xdp.
[13] 2021. P4-16 Portable Switch Architecture (PSA). https://p4.org/p4-spec/docs/PSA.html.
[14] 2021. P4 Portable NIC Architecture (PNA). https://p4.org/p4-spec/docs/PNA.html.
[15] 2021. Tofino Native Architecture. https://raw.githubusercontent.com/barefootnetworks/Open-Tofino/master/PUBLIC_Tofino-Native-Arch.pdf.
[16] 2024. AMD Vitis Networking P4 tool (VNP4). https://www.xilinx.com/content/dam/xilinx/publications/white-papers/wp555-vitis-networking-p4.pdf.
[17] 2024. BEHAVIORAL MODEL (bmv2). https://github.com/p4lang/behavioral-model.
[18] 2025. P4 Language Support in DOCA Pipeline Language (DPL). https://docs.nvidia.com/doca/sdk/p4+language+support+in+dpl/index.html.
[19] Anoud Alshnakat, Amir M Ahmadian, Musard Balliu, Roberto Guanciale, and Mads Dam. 2025. Securing p4 programs by information flow control. In *2025 IEEE 38th Computer Security Foundations Symposium (CSF)*. IEEE, 284–299.
[20] James P Anderson. 1972. Computer Security Technology Planning Study ESD-TR-73-51. *USAF Electronics Systems Division* (1972).
[21] Maria Apostolaki, Ankit Singla, and Laurent Vanbever. 2021. Performance-driven internet path selection. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*. 41–53.
[22] Osama Bajaber, Bo Ji, and Peng Gao. 2024. P4control: Line-rate cross-host attack prevention via in-network information flow control enabled by programmable switches and ebpf. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 4610–4628.
[23] Adam Bates, Dave Jing Tian, Kevin RB Butler, and Thomas Moyer. 2015. Trustworthy {Whole-System} provenance for the linux kernel. In *24th USENIX Security Symposium (USENIX Security 15)*. 319–334.
[24] Conor Black and Sandra Scott-Hayward. 2021. Adversarial exploitation of p4 data planes. In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 508–514.
[25] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.
[26] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2016. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 115–128.
[27] Alexandre da Silveira Ilha, Ângelo Cardoso Lapolli, Jonatas Adilson Marques, and Luciano Paschoal Gaspary. 2020. Euclid: A fully in-network, P4-based approach for real-time DDoS attack detection and mitigation. *IEEE Transactions on Network and Service Management* 18, 3 (2020), 3121–3139.
[28] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. 2015. Netpaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. 1–7.
[29] Mohan Dhawan, Rishabh Poddar, Kshiteej Mahajan, and Vijay Mann. 2015. Sphinx: detecting security attacks in software-defined networks.. In *Ndss*, Vol. 15. 8–11.
[30] Ryan Doenges, Mina Tahmasbi Arashloo, Santiago Bautista, Alexander Chang, Newton Ni, Samwise Parkinson, Rudy Peterson, Alaia Solko-Breslin, Amanda Xu, and Nate Foster. 2021. Petr4: formal foundations for p4 data planes. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–32.
[31] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. 2018. Uncovering bugs in p4 programs with assertion-based verification. In *Proceedings of the Symposium on SDN Research*. 1–7.
[32] Ashish Gehani, Raza Ahmad, Hassaan Irshad, Jianqiao Zhu, and Jignesh Patel. 2021. Digging into "Big Provenance" (with SPADE). *Commun. ACM* 64, 12 (2021), 48–56.
[33] Ashish Gehani and Dawood Tariq. 2012. SPADE: Support for Provenance Auditing in Distributed Environments. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 101–120.
[34] Dimitrios Gkounis, Felix Klaedtke, Roberto Bifulco, and Ghassan O Karame. 2016. Cases for including a reference monitor to SDN. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 599–600.
[35] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. 2014. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 71–85.
[36] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 29–42.
[37] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. 2023. A survey on data plane programming with p4: Fundamentals, advances, and applied research. *Journal of Network and Computer Applications* 212 (2023), 103561.
[38] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. 2019. Blink: Fast connectivity recovery entirely in the data plane. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 161–176.
[39] Sungmin Hong, Lei Xu, Haopei Wang, and Guofei Gu. 2015. Poisoning network visibility in software-defined networks: New attacks and countermeasures.. In *Ndss*, Vol. 15. 8–11.
[40] Md Nahid Hossain, Sadegh M Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R Sekar, Scott Stoller, and VN Venkatakrishnan. 2017. {SLEUTH}: Real-time attack scenario reconstruction from {COTS} audit data. In *26th USENIX Security Symposium (USENIX Security 17)*. 487–504.
[41] Samuel Jero, William Koch, Richard Skowyra, Hamed Okhravi, Cristina Nita-Rotaru, and David Bigelow. 2017. Identifier binding attacks and defenses in {Software-Defined} networks. In *26th USENIX Security Symposium (USENIX Security 17)*. 415–432.
[42] Cheng Jin, Haining Wang, and Kang G Shin. 2003. Hop-count filtering: an effective defense against spoofed DDoS traffic. In *Proceedings of the 10th ACM conference on Computer and communications security*. 30–41.
[43] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th symposium on operating systems principles*. 121–136.
[44] Qiao Kang, Jiarong Xing, and Ang Chen. 2019. Automated attack discovery in data plane systems. In *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*.
[45] Qiao Kang, Lei Xue, Adam Morrison, Yuxin Tang, Ang Chen, and Xiapu Luo. 2020. Programmable {In-Network} security for context-aware {BYOD} policies. In *29th USENIX Security Symposium (USENIX Security 20)*. 595–612.
[46] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header space analysis: Static checking for networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 113–126.

[47] Elie F Kfoury, Jorge Crichigno, and Elias Bou-Harb. 2021. An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends. *IEEE access* 9 (2021), 87094–87155.

[48] Xin Zhe Khooi, Levente Csikor, Dinil Mon Divakaran, and Min Suk Kang. 2020. DIDA: Distributed in-network defense architecture against amplified reflection DDoS attacks. In *2020 6th IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 277–281.

[49] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. 2012. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the first workshop on Hot topics in software defined networks*. 49–54.

[50] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, Lawrence J Wobker, et al. 2015. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, Vol. 15. 1–2.

[51] Daehyeok Kim, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, and Srinivasan Seshan. 2018. Generic external memory for switch data planes. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*. 1–7.

[52] Jiwon Kim, Dave Jing Tian, and Benjamin E. Ujcich. 2025. Chimera: Fuzzing P4 Network Infrastructure for Multi-Plane Bug Detection and Vulnerability Discovery. In *2025 IEEE Symposium on Security and Privacy (SP)*. 3088–3106. doi:10.1109/SP61157.2025.00194

[53] Suriya Kodeswaran, Mina Tahmasbi Arashloo, Praveen Tammana, and Jennifer Rexford. 2020. Tracking P4 program execution in the data plane. In *Proceedings of the Symposium on SDN Research*. 117–122.

[54] Patrick Tser Jern Kon, Aniket Gattani, Dhiraj Saharia, Tianyu Cao, Diogo Barradas, Ang Chen, Micah Sherr, and Benjamin E. Ujcich. 2024. NetShuffle: Circumventing Censorship with Shuffle Proxies at the Edge. In *2024 IEEE Symposium on Security and Privacy (SP)*. 3497–3514. doi:10.1109/SP54263.2024.00036

[55] Guanyu Li, Menghao Zhang, Chang Liu, Xiao Kong, Ang Chen, Guofei Gu, and Haixin Duan. 2019. NETHCF: Enabling line-rate and adaptive spoofed IP traffic filtering. In *2019 IEEE 27th international conference on network protocols (ICNP)*. IEEE, 1–12.

[56] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. {FlowRadar}: A better {NetFlow} for data centers. In *13th USENIX symposium on networked systems design and implementation (NSDI 16)*. 311–324.

[57] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. 2018. P4v: Practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on data communication*. 490–503.

[58] Ziwen Liu, Jian Mao, Jun Zeng, Jiawei Li, Qixiao Lin, Jiahao Liu, Jianwei Zhuge, and Zhenkai Liang. 2025. ProvGuard: Detecting SDN Control Policy Manipulation via Contextual Semantics of Provenance Graphs. In *NDSS*.

[59] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. 2021. Jaqen: A {High-Performance}{Switch-Native} approach for detecting and mitigating volumetric {DDoS} attacks with programmable switches. In *30th USENIX Security Symposium (USENIX Security 21)*. 3829–3846.

[60] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM computer communication review* 38, 2 (2008), 69–74.

[61] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 15–28.

[62] Sadegh M Milajerdi, Rigel Gjomemo, Birhanu Eshete, Ramachandran Sekar, and VN Venkatakrishnan. 2019. Holmes: real-time apt detection through correlation of suspicious information flows. In *2019 IEEE symposium on security and privacy (SP)*. IEEE, 1137–1152.

[63] Paolo Missier, Khalid Belhajjame, and James Cheney. 2013. The W3C PROV family of specifications for modelling provenance metadata. In *Proceedings of the 16th international conference on extending database technology*. 773–776.

[64] Adam Morrison, Lei Xue, Ang Chen, and Xiapu Luo. 2018. Enforcing {Context-Aware}{BYOD} Policies with {In-Network} Security. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*.

[65] Kiran-Kumar Muniswamy-Reddy and David A Holland. 2009. Causality-based versioning. *ACM Transactions on Storage (TOS)* 5, 4 (2009), 1–28.

[66] Kexin Pei, Zhongshu Gu, Brendan Saltaformaggio, Shiqing Ma, Fei Wang, Zhiwei Zhang, Luo Si, Xiangyu Zhang, and Dongyan Xu. 2016. Hercule: Attack story reconstruction via community discovery on correlated log graph. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*. 583–595.

[67] K Ranjitha, Medha Rachel Panna, Stavan Nilesh Christian, Karuturi Havya Sree, Sri Hari Malla, Dheekshitha Bheemanath, Rinku Shah, and Praveen Tammana. 2025. Securing In-Network Traffic Control Systems with P4Auth. In *2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 429–442.

[68] Sundararajan Renganathan, Benny Rubin, Hyojoon Kim, Pier Luigi Ventre, Carmelo Cascone, Daniele Moro, Charles Chan, Nick McKeown, and Nate Foster. 2023. Hydra: Effective Runtime Network Verification. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 182–194.

[69] Fabian Ruffy, Jed Liu, Prathima Kotikalapudi, Vojtech Havel, Hanneli Tavante, Rob Sherwood, Vladyslav Dubina, Volodymyr Peschanenko, Anirudh Sivaraman, and Nate Foster. 2023. P4testgen: An extensible test oracle for p4. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 136–151.

[70] Fabian Ruffy, Tao Wang, and Anirudh Sivaraman. 2020. Gauntlet: Finding bugs in compilers for programmable packet processing. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 683–699.

[71] Harish SA, K Shiv Kumar, Anibrata Majee, Amogh Bedarakota, Praveen Tammana, Pravein Govindan Kannan, and Rinku Shah. 2023. In-network probabilistic monitoring primitives under the influence of adversarial network inputs. In *Proceedings of the 7th Asia-Pacific Workshop on Networking*. 116–122.

[72] Dominik Scholz, Sebastian Gallenmüller, Henning Stubbe, Bassam Jaber, Minoo Rouhi, and Georg Carle. 2020. Me love (SYN-) cookies: SYN flood mitigation in programmable data planes. *arXiv preprint arXiv:2003.03221* (2020).

[73] Richard Skowyra, Lei Xu, Guofei Gu, Veer Dedhia, Thomas Hobson, Hamed Okhravi, and James Landry. 2018. Effective topology tampering attacks and defenses in software-defined networks. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 374–385.

[74] Dylan Smyth, Sandra Scott-Hayward, Victor Cionca, Sean McSweeney, and Donna O'Shea. 2023. SECAP switch—Defeating topology poisoning attacks using P4 data planes. *Journal of Network and Systems Management* 31, 1 (2023), 28.

[75] Nik Sultana, Deborah Shands, and Vinod Yegneswaran. 2022. A case for remote attestation in programmable dataplanes. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*. 122–129.

[76] Azadeh Tabiban, Heyang Zhao, Yosr Jarraya, Makan Pourzandi, Mengyuan Zhang, and Lingyu Wang. 2022. ProvTalk: Towards Interpretable Multi-level Provenance Analysis in Networking Functions Virtualization (NFV).. In *NDSS*.

[77] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. 2015. CherryPick: Tracing packet trajectory in software-defined datacenter networks. In *Proceedings of the 1st ACM SIGCOMM symposium on software defined networking research*. 1–7.

[78] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. 2016. Simplifying datacenter network debugging with {PathDump}. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 233–248.

[79] Benjamin E. Ujcich. 2025. A Systems Security Approach for Emerging Programmable Network Architectures. *IEEE Security and Privacy* 23, 6 (2025), 32–41. doi:10.1109/MSEC.2025.3603133

[80] Benjamin E Ujcich, Samuel Jero, Anne Edmundson, Qi Wang, Richard Skowyra, James Landry, Adam Bates, William H Sanders, Cristina Nita-Rotaru, and Hamed Okhravi. 2018. Cross-app poisoning in software-defined networking. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 648–663.

[81] Benjamin E Ujcich, Samuel Jero, Richard Skowyra, Adam Bates, William H Sanders, and Hamed Okhravi. 2021. Causal Analysis for {Software-Defined} Networking Attacks. In *30th USENIX Security Symposium (USENIX Security 21)*. 3183–3200.

[82] Benjamin E Ujcich, Samuel Jero, Richard Skowyra, Steven R Gomez, Adam Bates, William H Sanders, and Hamed Okhravi. 2020. Automated discovery of cross-plane event-based vulnerabilities in software-defined networking. In *Network and Distributed System Security Symposium*.

[83] Benjamin E Ujcich, Andrew Miller, Adam Bates, and William H Sanders. 2017. Towards an accountable software-defined networking architecture. In *2017 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 1–5.

[84] Haining Wang, Cheng Jin, and Kang G Shin. 2007. Defense against spoofed IP traffic using hop-count filtering. *IEEE/ACM Transactions on networking* 15, 1 (2007), 40–53.

[85] Haopei Wang, Guangliang Yang, Phakpoom Chinprutthiwong, Lei Xu, Yangyong Zhang, and Guofei Gu. 2018. Towards fine-grained network security forensics and diagnosis in the SDN era. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 3–16.

[86] Liang Wang, Prateek Mittal, and Jennifer Rexford. 2022. Data-plane security applications in adversarial settings. *ACM SIGCOMM Computer Communication Review* 52, 2 (2022), 2–9.

[87] Qinshi Wang, Mengying Pan, Shengyi Wang, Ryan Doenges, Lennart Beringer, and Andrew W Appel. 2023. Foundational verification of stateful P4 packet processing. In *14th International Conference on Interactive Theorem Proving (ITP 2023)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 32–1.

[88] Weitao Wang, Xinyu Crystal Wu, Praveen Tammana, Ang Chen, and TS Eugene Ng. 2022. Closed-loop network performance monitoring and diagnosis with {SpiderMon}. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 267–285.

[89] Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2014. Diagnosing missing events in distributed systems with negative provenance. *ACM SIGCOMM Computer Communication Review* 44, 4 (2014), 383–394.

[90] Feng Xiao, Jinquan Zhang, Jianwei Huang, Guofei Gu, Dinghao Wu, and Peng Liu. 2020. Unexpected data dependency creation and chaining: A new attack to SDN. In *2020 IEEE symposium on security and privacy (SP)*. IEEE, 1512–1526.
[91] Jiarong Xing, Qiao Kang, and Ang Chen. 2020. {NetWarden}: Mitigating network covert channels while preserving performance. In *29th USENIX Security Symposium (USENIX Security 20)*. 2039–2056.
[92] Jiarong Xing, Wenqing Wu, and Ang Chen. 2021. Ripple: A programmable, decentralized {Link-Flooding} defense against adaptive adversaries. In *30th USENIX Security Symposium (USENIX Security 21)*. 3865–3881.
[93] Lei Xu, Jeff Huang, Sungmin Hong, Jialong Zhang, and Guofei Gu. 2017. Attacking the brain: Races in the {SDN} control plane. In *26th USENIX Security Symposium (USENIX Security 17)*. 451–468.
[94] Changhoon Yoon, Seungsoo Lee, Heedo Kang, Taejune Park, Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. 2017. Flow wars: Systemizing the attack surface and defenses in software-defined networks. *IEEE/ACM Transactions on Networking* 25, 6 (2017), 3514–3530.
[95] Eder Ollora Zaballa, David Franco, Zifan Zhou, and Michael S Berger. 2020. P4Knocking: Offloading host-based firewall functionalities to the network. In *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. IEEE, 7–12.
[96] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qianqian Li, Mingwei Xu, and Jianping Wu. 2020. Poseidon: Mitigating volumetric ddos attacks with programmable switches. In *the 27th Network and Distributed System Security Symposium (NDSS 2020)*.
[97] Peng Zhang, Hao Li, Chengchen Hu, Liujia Hu, Lei Xiong, Ruilong Wang, and Yuemei Zhang. 2016. Mind the gap: Monitoring the control-data plane consistency in software defined networks. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. 19–33.
[98] Huancheng Zhou and Guofei Gu. 2025. Securing Networks with Programmable Data Planes: Opportunities and Challenges. *IEEE Security and Privacy* 23, 6 (2025), 42–50. doi:10.1109/MSEC.2025.3603621
[99] Huancheng Zhou, Sungmin Hong, Yangyang Liu, Xiapu Luo, Weichao Li, and Guofei Gu. 2023. Mew: Enabling large-scale and dynamic link-flooding defenses on programmable switches. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3178–3192.
[100] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. 2011. Secure network provenance. In *Proceedings of the twenty-third ACM symposium on operating systems principles*. 295–310.

A Ethical Considerations

This work focuses on the technical architecture of a provenance collection tool and does not involve human subjects or the analysis of sensitive user data. The provenance metadata collected consists of system-level network state changes (e.g., registers and packet metadata), which do not contain identity-revealing information beyond that which would otherwise be routinely collected within a network environment in which ProvP4 would be deployed (e.g., a controlled enterprise network setting). Furthermore, the tool was evaluated using publicly available open-source code repositories for P4-based systems. All experiments were conducted in controlled, isolated environments using non-sensitive data.

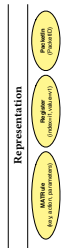
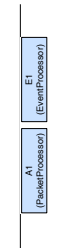
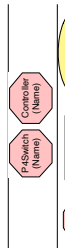
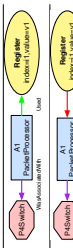
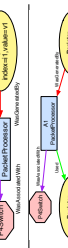
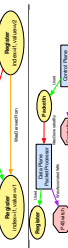
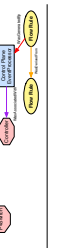
B Open Science

The core contributions of this work are supported by the following artifacts available at https://github.com/GUSecLab/ProvP4: ProvP4 codebase for data and control planes, written in Python3; instrumented P4 programs for the P4-based systems (NetHCF, SECAP, HH, and P4Knocking); and evaluation test scripts and plotting scripts for latency and storage costs.

C Detailed Provenance Model

Table 6 summarizes the ProvP4 provenance model objects (**Nodes**) and interactions (**Edges**) among various data plane and control plane objects, including common graph patterns representing P4 program semantics.

Table 6: Complete ProvP4 provenance model.

Description	P4 Example	Representation
Any object within the SDN environment that is generated or used for network decisions	MATRule(key, action, action_parameters), Register(index,value), PacketIn,PacketOut	
Abstraction of packet (data plane) or event (control plane) processing units	Data plane → action A1(){}; Control plane → Function (e.g., E1)	
SDN entity responsible for performing any action	Data plane → bmv2,Tofino; Control plane → ONOS, Thrift, P4Runtime	
Data plane *Activity* reading from a register	action A1 {register.read(v1, i1)}	
Data plane *Activity* writing to a register	action A1 {register.write(i1, v1)}	
Data plane *Activity* reading and writing to the same register	action A1 {register.read(v1, i1); register.write(i1, v2)}	
Data plane generating a PacketIn and/or control plane using it to modify DP flow rule	action A1 {send_to_controller()} → controller installs new flow rule	

D Provenance Instrumentation Coverage

We define *provenance instrumentation coverage* as the ratio of instrumented stateful instructions to the total number of stateful instructions in a P4 program. After program instrumentation, ProvP4 achieves the following coverage for case studies: NetHCF (77%), SECAP (100%), Heavy Hitter (100%), and P4Knocking (100%). ProvP4 instruments registers but not counters, resulting in less than full coverage of all stateful instructions for NetHCF. This design choice does not meaningfully impact ProvP4's reasoning or querying capabilities, since counters must be accessed via the control plane to affect data plane state changes, making data-plane counter instrumentation unnecessary.

E P4 Program Instrumentation

We apply provenance instrumentation to SECAP [74] below:

```
apply{
  // Retrieve the current epoch number
  get_epoch_counter();
  // Data plane performs a register write
  port_knocking_stage_count.write(
    (bit<32>)meta.pk_metadata.index, meta.pk_metadata.stage
  );
  /* Start Provenance capture */
  // Retrieve the next available slot
  get_prov_index(INDEX_PROV_PORT_KNOCKING_STAGE_COUNT);
  // Record the stateful metadata in the provenance register
  prov_port_knocking_stage_count.write(
    meta.prov_index_val,
    (bit<32>)meta.pk_metadata.index ++ meta.pk_metadata.stage ++
    (bit<8>) 14 ++ (bit<8>)meta.current_epoch ++ (bit<1>) 1
  );
  // Increment the current slot
  increment_prov_index(INDEX_PROV_PORT_KNOCKING_STAGE_COUNT);
  /* End Provenance capture */
}
```