I. TOPOLOGY

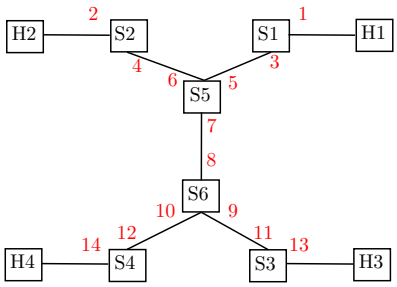

Figure 1. Race Condition

$$
\begin{aligned}
T \triangleq \quad & (pt = 1) + (pt = 2) + (pt = 13) + (pt = 14) \\
& (pt = 3 \;.\; pt \leftarrow 5) \;\; + (pt = 5 \;.\; pt \leftarrow 3) \\
& (pt = 7 \;.\; pt \leftarrow 8) \;\; + (pt = 8 \;.\; pt \leftarrow 7) \\
& (pt = 9 \;.\; pt \leftarrow 11) \;\; + (pt = 11 \;.\; pt \leftarrow 9) \\
& (pt = 4 \;.\; pt \leftarrow 6) \;\; + (pt = 6 \;.\; pt \leftarrow 4) \\
& (pt = 10 \;.\; pt \leftarrow 12) \;\; + (pt = 12 \;.\; pt \leftarrow 10)
\end{aligned} \tag{1}
$$

Safe state 1: H3 can be reachable from H1.
Safe state 2: H4 can be reachable from H2.
Fault behavior 1: H4 must not be reachable from H1.
Fault behavior 2: H3 must not be reachable from H2.

II. Logs

A. Safe State 1:

S1:

```
Type: OFPT_PACKET_IN (10)
switch: S1
controller: C
Reason: No matching flow (table-miss flow entry)
ip.src: 10.0.0.1
ip.dst: 10.0.0.3
```

Figure 2. Simplified Packet_In message for S1

```
Type: OFPT_FLOW_MOD (14)
switch: S1
controller: C
in port = 1
output port = 3
openflow.ofp_match.source_addr: 10.0.0.1
openflow.ofp_match.dest_addr: 10.0.0.3
```

Figure 3. Simplified Flow_Mod message for S1

```
Type: OFPT_PACKET_OUT (13)
switch: S1
controller: C
In port: 1
Actions type: Output to switch port (0)
Output port: 3
```

Figure 4. Simplified Packet_Out message for S1

S5:

```
Type: OFPT_PACKET_IN (10)
switch: S5
controller: C
Reason: No matching flow (table-miss flow entry)
ip.src: 10.0.0.1
ip.dst: 10.0.0.3
```

Figure 5. Simplified Packet_In message for S5

```
Type: OFPT_FLOW_MOD (14)
switch: S5
controller: C
In port = 5
output port = 7
openflow.ofp_match.source_addr: 10.0.0.1
openflow.ofp_match.dest_addr: 10.0.0.3
```

Figure 6. Simplified Flow_Mod message for S5

```
Type: OFPT_PACKET_OUT (13)
switch: S5
controller: C
In port: 5
Actions type: Output to switch port (0)
Output port: 7
```

Figure 7. Simplified Packet_Out message for S5

S6:

```
Type: OFPT_PACKET_IN (10)
switch: S6
controller: C
Reason: No matching flow (table-miss flow entry)
ip.src: 10.0.0.1
ip.dst: 10.0.0.3
```

Figure 8. Simplified Packet_In message for S6

```
Type: OFPT_FLOW_MOD (14)
switch: S6
controller: C
In port = 8
output port = 9
openflow.ofp_match.source_addr: 10.0.0.1
openflow.ofp_match.dest_addr: 10.0.0.3
```

Figure 9. Simplified Flow_Mod message for S6

```
Type: OFPT_PACKET_OUT (13)
switch: S6
controller: C
In port: 8
Actions type: Output to switch port (0)
Output port: 9
```

Figure 10. Simplified Packet_Out message for S6

S3:

```
Type: OFPT_PACKET_IN (10)
switch: S3
controller: C
Reason: No matching flow (table-miss flow entry)
ip.src: 10.0.0.1
ip.dst: 10.0.0.3
```

Figure 11. Simplified Packet_In message for S3

```
Type: OFPT_FLOW_MOD (14)
switch: S3
controller: C
In port = 11
output port = 13
openflow.ofp_match.source_addr: 10.0.0.1
openflow.ofp_match.dest_addr: 10.0.0.3
```

Figure 12. Simplified Flow_Mod message for S3

```
Type: OFPT_PACKET_OUT (13)
switch: S3
controller: C
In port: 11
Actions type: Output to switch port (0)
Output port: 13
```

Figure 13. Simplified Packet_Out message for S3

B. Safe State 2:

S2:

```
Type: OFPT_PACKET_IN (10)
switch: S2
controller: C
Reason: No matching flow (table-miss flow entry)
ip.src: 10.0.0.2
ip.dst: 10.0.0.4
```

Figure 14. Simplified Packet_In message for S2

```
Type: OFPT_FLOW_MOD (14)
switch: S2
controller: C
In port = 2
output port = 4
openflow.ofp_match.source_addr: 10.0.0.2
openflow.ofp_match.dest_addr: 10.0.0.4
```

Figure 15. Simplified Flow_Mod message for S2

```
Type: OFPT_PACKET_OUT (13)
switch: S2
controller: C
In port: 2
Actions type: Output to switch port (0)
Output port: 4
```

Figure 16. Simplified Packet_Out message for S2

S5:

```
Type: OFPT_PACKET_IN (10)
switch: S5
controller: C
Reason: No matching flow (table-miss flow entry)
ip.src: 10.0.0.2
ip.dst: 10.0.0.4
```

Figure 17. Simplified Packet_In message for S5

```
Type: OFPT_FLOW_MOD (14)
switch: S5
controller: C
In port = 6
output port = 7
openflow.ofp_match.source_addr: 10.0.0.2
openflow.ofp_match.dest_addr: 10.0.0.4
```

Figure 18. Simplified Flow_Mod message for S5

```
Type: OFPT_PACKET_OUT (13)
switch: S5
controller: C
In port: 6
Actions type: Output to switch port (0)
Output port: 7
```

Figure 19. Simplified Packet_Out message for S5

S6:

```
Type: OFPT_PACKET_IN (10)
switch: S6
controller: C
Reason: No matching flow (table-miss flow entry)
ip.src: 10.0.0.2
ip.dst: 10.0.0.4
```

Figure 20. Simplified Packet_In message for S6

```
Type: OFPT_FLOW_MOD (14)
switch: S6
controller: C
In port = 8
output port = 10
openflow.ofp_match.source_addr: 10.0.0.2
openflow.ofp_match.dest_addr: 10.0.0.4
```

Figure 21. Simplified Flow_Mod message for S6

```
Type: OFPT_PACKET_OUT (13)
switch: S6
controller: C
In port: 8
Actions type: Output to switch port (0)
Output port: 10
```

Figure 22. Simplified Packet_Out message for S6

S4:

```
Type: OFPT_PACKET_IN (10)
switch: S4
controller: C
Reason: No matching flow (table-miss flow entry)
ip.src: 10.0.0.2
ip.dst: 10.0.0.4
```

Figure 23. Simplified Packet_In message for S4

```
Type: OFPT_FLOW_MOD (14)
switch: S4
controller: C
In port = 12
output port = 14
openflow.ofp_match.source_addr: 10.0.0.2
openflow.ofp_match.dest_addr: 10.0.0.4
```

Figure 24. Simplified Flow_Mod message for S4

```
Type: OFPT_PACKET_OUT (13)
switch: S4
controller: C
In port: 12
Actions type: Output to switch port (0)
Output port: 14
```

Figure 25. Simplified Packet_Out message for S4

III. DyNetKAT

$$p \triangleq D1||C$$
$$ch \triangleq S5Reqflow1, S5Upflow1$$
$$S6Reqflow1, S6Upflow1$$

$$C \triangleq ((S5Reqflow1\ ?\ one)\ ;\ ((S5Upflow1!X'_{S5})\ ;\ C)) \oplus ((S6Reqflow1\ ?\ one)\ ;\ ((S6Upflow1!X'_{S6})\ ;\ C))$$

$$D1 \triangleq ((X_{S1} + X_{S5} + X_{S6} + X_{S3} + X_{S2} + X_{S4})\ .\ (T))^*; D1 \oplus (S5Reqflow1!one); D1 \oplus (S5Upflow1?\ X'_{S5}); D3 \oplus (S6Reqflow1!one); D1 \oplus (S6Upflow1?X'_{S6}); D2$$

$$D2 \triangleq ((X_{S1} + X_{S5} + X'_{S6} + X_{S3} + X_{S2} + X_{S4})\ .\ (T))^*; D2 \oplus (S5Reqflow1!one); D2 \oplus (S5Upflow1?X'_{S5}); D4 \oplus (S6Reqflow1!one); D2 \oplus (S6Upflow1?X'_{S6}); D2$$

$$D3 \triangleq ((X_{S1} + X'_{S5} + X_{S6} + X_{S3} + X_{S2} + X_{S4})\ .\ (T))^*; D3 \oplus (S5Reqflow1!one); D3 \oplus (S5Upflow1?X'_{S5}); D3 \oplus (S6Reqflow1!one); D3 \oplus (S6Upflow1?X'_{S6}); D4$$

$$D4 \triangleq ((X_{S1} + X'_{S5} + X'_{S6} + X_{S3} + X_{S2} + X_{S4})\ .\ (T))^*; D4 \oplus (S5Reqflow1!one); D4 \oplus (S5Upflow1?X'_{S5}); D4 \oplus (S6Reqflow1!one); D4 \oplus (S6Upflow1?X'_{S6}); D4$$

$$X_{S1} \triangleq pt = 1\ .\ pt \leftarrow 3$$
$$X_{S2} \triangleq pt = 2\ .\ pt \leftarrow 4$$
$$X_{S3} \triangleq pt = 11\ .\ pt \leftarrow 13$$
$$X_{S4} \triangleq pt = 12\ .\ pt \leftarrow 14$$
$$X_{S5} \triangleq pt = 5\ .\ pt \leftarrow 7$$
$$X_{S6} \triangleq pt = 8\ .\ pt \leftarrow 9$$
$$X'_{S5} \triangleq pt = 6\ .\ pt \leftarrow 7$$
$$X'_{S6} \triangleq pt = 8\ .\ pt \leftarrow 10$$

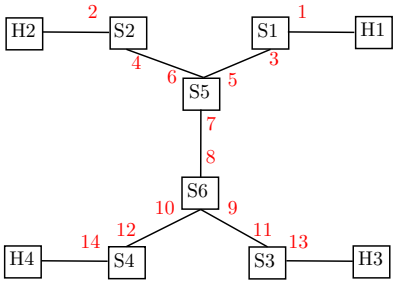

Figure 26. Race Condition

Consider R is a set of reconfigurations:

$$R = \left\{ r_1 = \text{rcfg}_{S5\text{Upflow1},X'_{S5}}, r_2 = \text{rcfg}_{S6\text{Upflow1},X'_{S6}} \right\}$$

Suppose we want to define a subset of reconfigurations where the order of elements is significant.

we define such ordered subsets of reconfigurations as follows:

$$S \subseteq R, \quad S = (s_1, s_2, \ldots, s_k),$$
$$\text{where } s_i \in R, \quad \text{and } s_i \neq s_j \text{ for } i \neq j$$

Safe state 1: H3 can be reachable from H1.

$$(pt = 1)\ .\ head(p)\ .\ (pt = 13) \not\equiv 0$$

Safe state 2: H4 can be reachable from H2.

$$(pt = 2)\ .\ head(tail(p, R))\ .\ (pt = 14) \not\equiv 0$$

Fault behavior 1: H4 must not be reachable from H1.

$$\exists S\ .\ (pt = 1)\ .\ head(tail(p, S))\ .\ (pt = 14) \equiv 0$$

In this example:

$$S = \left\{ r_2 = \text{rcfg}_{S6\text{Upflow1},X'_{S6}} \right\}$$
$$(pt = 1)\ .\ head(tail(p, S))\ .\ (pt = 14) \equiv 0$$

Fault behavior 2: H3 must not be reachable from H2.

$$\exists S\ .\ (pt = 2)\ .\ head(tail(p, S))\ .\ (pt = 13) \equiv 0$$

In this example:

$$S = \left\{ r_1 = \text{rcfg}_{S5\text{Upflow1},X'_{S5}} \right\}$$
$$(pt = 2)\ .\ head(tail(p, S))\ .\ (pt = 13) \equiv 0$$

Property	type	Result	output_json_path
h1h3	!0	Satisfied	./Expriments/Fault_Scenarios/result_race_condition/DyNetKAT_race_condition_h1h3_0.json
		Satisfied	./Expriments/Fault_Scenarios/result_race_condition/DyNetKAT_race_condition_h1h3_1.json
		Satisfied	./Expriments/Fault_Scenarios/result_race_condition/DyNetKAT_race_condition_h1h3_2.json
		Violated	./Expriments/Fault_Scenarios/result_race_condition/DyNetKAT_race_condition_h1h3_3.json
		Violated	./Expriments/Fault_Scenarios/result_race_condition/DyNetKAT_race_condition_h1h3_4.json
h2h4	!0	Violated	./Expriments/Fault_Scenarios/result_race_condition/DyNetKAT_race_condition_h2h4_0.json
		Violated	./Expriments/Fault_Scenarios/result_race_condition/DyNetKAT_race_condition_h2h4_1.json
		Violated	./Expriments/Fault_Scenarios/result_race_condition/DyNetKAT_race_condition_h2h4_2.json
		Satisfied	./Expriments/Fault_Scenarios/result_race_condition/DyNetKAT_race_condition_h2h4_3.json
		Satisfied	./Expriments/Fault_Scenarios/result_race_condition/DyNetKAT_race_condition_h2h4_4.json
h1h4	0	Satisfied	./Expriments/Fault_Scenarios/result_race_condition/DyNetKAT_race_condition_h1h4_0.json
		Satisfied	./Expriments/Fault_Scenarios/result_race_condition/DyNetKAT_race_condition_h1h4_1.json
		Violated	./Expriments/Fault_Scenarios/result_race_condition/DyNetKAT_race_condition_h1h4_2.json
		Satisfied	./Expriments/Fault_Scenarios/result_race_condition/DyNetKAT_race_condition_h1h4_3.json
		Violated	./Expriments/Fault_Scenarios/result_race_condition/DyNetKAT_race_condition_h1h4_4.json
h2h3	0	Satisfied	./Expriments/Fault_Scenarios/result_race_condition/DyNetKAT_race_condition_h2h3_0.json
		Violated	./Expriments/Fault_Scenarios/result_race_condition/DyNetKAT_race_condition_h2h3_1.json
		Satisfied	./Expriments/Fault_Scenarios/result_race_condition/DyNetKAT_race_condition_h2h3_2.json
		Violated	./Expriments/Fault_Scenarios/result_race_condition/DyNetKAT_race_condition_h2h3_3.json
		Satisfied	./Expriments/Fault_Scenarios/result_race_condition/DyNetKAT_race_condition_h2h3_4.json

Figure 27. race condition result

Fault Prediction in Software-Defined Networks

Mohammadreza Ghobakhlou

Abstract—In this work, we tackle fault prediction in Software-Defined Networks (SDNs) using the DyNetKAT framework. Our approach extracts DyNetKAT terms from OpenFlow log files to analyze the sequence of switch reconfigurations that may lead to faults. A network is initially designed and deployed with a set of rules and configurations to ensure its correct and secure operation. Over time, multiple updates to switch and controller configurations are applied, and the network is expected to remain in a "safe" state. However, certain sequences of updates may inadvertently push the network out of its secure state, leading to faults.

We present FPSDN [GitHub], a tool that leverages DyNetKAT terms for formal verification, enabling precise assessment of network reachability and dynamic reconfiguration management. FPSDN was evaluated across eight experiments and successfully predicted faults automatically, demonstrating its effectiveness in proactive fault detection for SDNs.

IV. Introduction

Traditional network architectures have consistently relied on specialized hardware, limiting adaptability and prolonging configuration times, with core design principles remaining largely unchanged for four decades [Ham09], [Slo13].
software-defined networking has been standardized, which separates the data plane from the control plane, unlike traditional architectures that necessitate manual device configuration to ensure proper traffic flow. One of the key advantages of software-defined networking is its establishment of open standards, allowing for implementation by any vendor. For instance, the OpenFlow API [MAB+08] delineates the capabilities and behavior of switch hardware and provides a standardized low-level language for configuring these devices. As with software engineering, the networking field is recognizing an increasing need for validation, prompted by innovative research in software-defined networking.
Fault prediction in SDNs has emerged as a critical area of study due to the increasing complexity and dynamism of modern networks. As SDNs continue to evolve, the flexibility and programmability that define these networks also introduce new challenges, particularly in ensuring consistent and fault-free operation. Despite the advantages of SDNs, existing methods often fall short in providing comprehensive verification tools that can preemptively identify potential faults during network reconfigurations. Our work aims to provide a systematic approach to fault detection, focusing on the verification of network behaviors in dynamic settings where traditional validation methods may fail.

V. Languages for SDNs

In the rapidly evolving landscape of software-defined networks(SDNs), the ability to accurately specify and model network behaviors is crucial for ensuring system reliability and performance. Precise specification is not only essential for managing the dynamic and complex nature of SDNs but also plays a critical role in fault prediction and validation, which are vital for maintaining network stability and preventing system failures.

A. NetKAT

NetKAT is an algebraic language specifically designed for reasoning about software-defined networks. It achieves this by utilizing Kleene Algebra with Tests(KAT), offering a framework for analyzing network behaviors. NetKAT operates on a set of policies, which consist of filter predicates(tests) and field assignments(actions), and these policies can be combined to model controller applications. The NetKAT framework enables formal reasoning about network architecture and its correctness through a sound and complete axiomatization, allowing the proof or disproof of equalities over policies. The comprehensive syntax of NetKAT is detailed in its foundational paper [AFG+14]. A significant limitation of NetKAT is its stateless nature, which restricts its ability to model concurrency within networks, thereby preventing the modeling of scenarios where state changes influence packet flow. Additionally, NetKAT lacks the capability to facilitate dynamic updates to flow tables in SDNs.

B. DyNetKAT

To address the limitations of NetKAT, a dynamic extension known as DyNetKAT has been introduced. DyNetKAT preserves the foundational strengths of NetKAT while incorporating additional operators to enhance its capabilities. In contrast to NetKAT, DyNetKAT is specifically designed for dynamic software-defined networks(SDNs), enabling the modeling of communications between the data and control planes to accurately represent SDN behavior. The details of these new operators, along with their semantics and formal proofs, are thoroughly documented in the original paper [CHMT21].

In Figure 28, we recall the NetKAT and DyNetKAT syntax. The predicate for dropping a packet is denoted by **0**, while passing on a packet(without any modification) is denoted by **1**. The predicate checking whether the field f of a packet has value n is denoted by $(f = n)$; if the predicate fails on the current packet it results on dropping the packet, otherwise it will pass the packet

NetKAT Syntax:

$$\begin{array}{lll} Pr & ::= & \mathbf{0} \mid \mathbf{1} \mid f = n \mid Pr + Pr \mid Pr \cdot Pr \mid \neg Pr \\ N & ::= & Pr \mid f \leftarrow n \mid N + N \mid N \cdot N \mid N^* \end{array}$$

DyNetKAT Syntax:

$$\begin{array}{lll} D & ::= & \bot \mid N\,;D \mid x?N\,;D \mid x!N\,;D \mid D \,||\, D \mid D \oplus D \mid X \\ & & X \triangleq D \end{array}$$

Figure 28. NetKAT and DyNetKAT Syntax

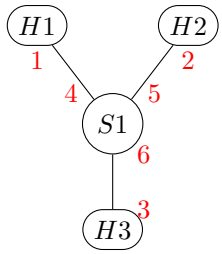

Figure 29. One Switch Example

on. Disjunction and conjunction between predicates are denoted by $Pr + Pr$ and $Pr \cdot Pr$, respectively. Negation is denoted by $\neg Pr$. The policy that modifies the field f of the current packet to take value n is denoted by $(f \leftarrow n)$. A multicast behaviour of policies is denoted by $N+N$, while sequencing policies(to be applied on the same packet) are denoted by $N \cdot N$. The repeated application of a policy is encoded as N^*.

By $\bot$ we denote a dummy policy without behaviour. Our new sequential composition operator, denoted by $N\,;D$, specifies when the NetKAT policy N is applicable to the current packet has come to a successful end and, thus, the packet can be transmitted further and the next packet can be fetched for processing according to the rest of the policy D. Communication in DyNetKAT, encoded via $x!N\,;D$ and $x?N\,;D$, consists of two steps. In the first place, sending and receiving NetKAT policies through channel x are denoted by $x!N$, and $x?N$. Secondly, as soon as the sending or receiving messages are successfully communicated, a new packet is fetched and processed according to D. The parallel composition of two DyNetKAT policies(to enable synchronisation) is denoted by $D \,||\, D$. Finally, one can use recursive variables X in the specification of DyNetKAT policies, where each recursive variable should have a unique defining equation $X \triangleq D$.

C. Running Example

To illustrate the principles and methodologies discussed in this paper, we employ a simplified network topology as a running example29. This example serves as a consistent reference point throughout our analysis and modeling efforts. The example network consists of a single switch (S1) connected to three distinct hosts. Host 1 (H1) is connected to port 4 of switch S1 through its own port 1. Similarly, Host 2 (H2) connects to port 5 of switch S1 via its port 2, and Host 3 (H3) is connected to port 6 of switch S1 through its port 3. In this example, we assume that the flow table of the switch (S1) is preconfigured with a rule that dictates the forwarding behavior. Specifically, if the switch receives a packet from port 4, it is allowed to forward the packet to port 6. This predefined rule in flow table of switch can be formally expressed in the DyNetKAT model as

$$X_{S1} \triangleq pt = 4 \;.\; pt \leftarrow 6 \tag{2}$$

, where $pt = 4$ denotes the receipt of the packet on port 4, and $pt \leftarrow 6$ indicates the forwarding of the packet to port 6.

In the context of our example, where the switch (S1) is initially configured to forward packets from port 4 to port 6, dynamic updates may necessitate changes to the flow table. For instance, if network conditions change or a new policy is introduced—such as forwarding packets from port 5 to port 6—the switch must update its flow table to reflect these new routing requirements. In such scenarios, the DyNetKAT model plays a crucial role in validating the correctness of these dynamic updates. The model ensures that the updated flow table, now incorporating the new policy, continues to maintain the desired network behavior, thus preventing potential faults that could arise from incorrect or incomplete configurations.

VI. Motivation

In the realm of Software-Defined Networking, ensuring the reliability and correctness of network behavior is paramount. OpenFlow [The12], as a widely adopted protocol for SDN, allows for the flexible management of network flows through programmable controllers. However, a critical challenge persists: the ability to write OpenFlow-compatible controller applications does not inherently provide the means to rigorously validate the correctness of the resulting dynamic network configurations.

The prevalence of bugs in dynamic network systems presents a significant challenge, often leading to operational inefficiencies and security vulnerabilities [AFLV08]. Despite extensive testing and verification efforts [BGMW17], [ZKVM12], many dynamic networks are launched with hidden bugs that only manifest under certain conditions, causing unexpected failures and disruptions. These issues underscore the need for more proactive and intelligent solutions capable of monitoring network behavior in real-time [GJVP+14], [EV02].

In response to this critical need, our research introduces a system designed to operate alongside dynamic networks, continuously listening to network communications and analyzing OpenFlow logs. By leveraging this real-time data, our system can detect potential faults and anomalies, offering early warnings before these issues escalate into more serious problems. This proactive approach to fault detection not only improves the reliability of dynamic

```
Type: OFPT_PACKET_IN (10)
switch: S1
controller: C
In port: 5
Reason: No matching flow (table-miss flow entry)
eth.dst: aa:a4:91:a5:71:28
eth.src: 7e:c8:93:0d:7c:71
ip.src: 10.0.0.2
ip.dst: 10.0.0.3
```

Figure 30. Simplified Packet_In message for one switch example 29

```
Type: OFPT_FLOW_MOD (14)
switch: S1
controller: C
In port: 5
openflow.eth_src: 7e:c8:93:0d:7c:71
openflow.eth_dst: aa:a4:91:a5:71:28
openflow.ofp_match.source_addr: 10.0.0.2
openflow.ofp_match.dest_addr: 10.0.0.3
```

Figure 31. Simplified Flow_Mod message for one switch example 29

networks but also enhances their overall stability, ensuring that they can meet the demands of increasingly complex and fast-changing environments.

VII. Log File and Topology

In this section, we explore the relationship between the network's OpenFlow log files and its underlying topology. Log files provide a detailed record of the interactions between switches and the controller, particularly capturing the dynamic flow table changes that occur in SDNs. Based on the nature of these dynamic changes, and the communication between the control plane and the data plane, we have assumed that our log file consists exclusively of OpenFlow packets. By analyzing this log file, we can infer the behavior of the network and gain insights into its structure and operational patterns, despite not capturing direct communication between switches or pre-installed flows.

This section is divided into two parts: Subsection VII-A provides an in-depth look into the structure and content of the OpenFlow log files, detailing how network events are captured. Subsection VII-B discusses the process of topology extraction, where we map the relationships between network devices and reconstruct the connectivity using the information gathered from the OpenFlow log files.

A. Into the log file

In SDNs, the OpenFlow protocol is a communications protocol that governs the interaction between the control plane and the data plane. It enables the controller to dynamically manage and update the flow tables of switches, directing how packets are processed and forwarded through the network. Among the various types of OpenFlow packets, Packet_In, Flow_Mod, and Packet_Out messages are the focus of our analysis. By serving as the communication bridge, OpenFlow allows the controller to issue instructions via Flow_Mod messages, respond to Packet_In queries from switches, and even directly manage traffic with Packet_Out messages. This real-time communication is essential for maintaining network flexibility and adaptability, ensuring that traffic is efficiently routed based on the controller's centralized logic.

1) Packet_In messages: These messages are a fundamental type of OpenFlow packet, sent from a switch to the controller when the switch receives a packet that does not match any of its flow table entries or when the packet explicitly triggers a Packet_In due to a matching flow entry with the "Send to Controller" action. This message typically contains a portion of the received packet, as well as metadata about the packet, such as its input port and other relevant details. Packet_In messages serve as a mechanism for the switch to offload decision-making to the controller, enabling the controller to instruct the switch on how to handle packets that are not explicitly matched in the flow table. Analyzing Packet_In logs provides insight into the frequency and nature of controller-invoked decisions, helping to identify patterns or inefficiencies in flow table configurations.

In our example (Figure 29), the Packet_In message interact between the switch (S1) and the controller (C) within the SDN environment. When the switch receives a packet on port 5 but lacks a corresponding flow entry in its flow table, it generates a Packet_In message and sends it to the controller for further processing. This action prompts the controller to analyze the packet and potentially update the flow table with new rules. The corresponding log file for this Packet_In message is provided in Figure 30.

2) Flow_Mod messages: These messages are sent by the controller to the switch in response to Packet_In messages or other events requiring a change in the switch's flow table. These messages instruct the switch on how to process future packets that match specific criteria, effectively allowing the controller to modify the switch's behavior dynamically. A Flow_Mod message can add, modify, or delete flow entries within the switch's flow table, and it includes details such as match fields, priority levels, and actions to be taken on matching packets. We can gain insight into how the controller shapes the switch's packet-forwarding decisions over time through the analysis of Flow_Mod packets. By examining these packets, one can assess the controller's strategy for managing traffic and optimizing network performance.

In our example (Figure 29), the Flow_Mod message facilitates the dynamic updating of the switch's (S1) flow table by the controller (C). After analyzing a Packet_In message, the controller may determine the need to install

```
Type: OFPT_PACKET_OUT (13)
switch: S1
controller: C
In port: 5
Actions type: Output to switch port (0)
Output port: 6
```

Figure 32. Simplified Packet_Out message for one switch example 29

a new flow rule. In this case, the controller sends a Flow_Mod message to the switch, instructing it to forward packets received on port 5. The corresponding log file for this Flow_Mod message is provided in Figure 31.

3) Packet_Out messages: These messages represent the final step in the process initiated by a Packet_In message. After receiving a Packet_In and possibly sending a Flow_Mod to update the switch's flow table, the controller may decide to send a Packet_Out message, which instructs the switch to forward a specific packet out of a designated port. This packet could be the original one that triggered the Packet_In, or it could be a different packet crafted by the controller. Packet_Out messages enable the controller to exert fine-grained control over packet forwarding, allowing for custom handling of specific traffic flows. In Wireshark logs, analyzing Packet_Out messages reveals how the controller directs traffic in scenarios where immediate packet forwarding is necessary, often providing insights into the handling of exceptional or high-priority traffic.

In simple switch example (Figure 29), After receiving a Packet_In message and processing the packet, the controller may decide to forward the packet immediately. It does so by sending a Packet_Out message to the switch, specifying the output action, such as forwarding the packet from port 5 to port 6. The corresponding log file for this Packet_Out message is provided in Figure 32.

As a result, each type of packet serves a distinct role in determining how packets are processed within the network, and their interplay is vital for maintaining the desired network behavior.

B. Topology Extraction

To extract the network topology from OpenFlow log files, we adopt a systematic approach, leveraging the dynamic configurations captured in these logs. Given that the log file consists solely of OpenFlow messages, such as Packet_In, Flow_Mod, and Packet_Out messages, we base our analysis entirely on the interactions between the controller and switches. Since we assume no prior knowledge of the network's behavior, this method allows us to infer the topology by focusing on the real-time communication and flow updates initiated by the controller. Also, these OpenFlow messages include critical details such as the source host, destination host, and the switch name, all of which are essential for mapping the network's structure. By storing each device and their respective connections as nodes and edges within a directed graph, we can accurately represent the relationships and paths between the various elements of the network. Subsequently, ports are assigned to the outgoing edges of each node, corresponding to the physical interfaces through which the devices are interconnected.

In Algorithm 1, the steps for extracting the network topology from log files by constructing a directed graph are presented. To provide a more detailed explanation of Algorithm 1, it outlines the process of extracting the network topology from log files by constructing a directed graph G. The algorithm begins by iterating through each Packet_In message in the log file. For every Packet_In message, it checks whether the source host, destination host, or switch involved in the packet are already represented as nodes in the graph G. If any of these elements are not yet present, the algorithm adds the corresponding node to the graph. Next, the algorithm defines the required edges of the graph based on the interactions captured in the Packet_In messages. These edges represent the connections between the hosts and switches, illustrating how traffic flows through the network. By systematically adding nodes and edges, the algorithm incrementally builds a comprehensive topology that reflects the structure and operational patterns of the network as observed from the log file.

This approach ensures that every element involved in packet forwarding is accurately represented in the graph, enabling a clear visualization of the network's topology. The directed graph G ultimately serves as a foundational structure for further analysis and understanding of the network's behavior.

In Algorithm 2, the process of assigning port numbers to the outgoing edges of each switch node in the network graph G is detailed. Initially, an empty dictionary is defined to store the mapping between each pair of connected nodes (i.e., the edges of the graph) and their corresponding port numbers. Additionally, a variable p is initialized to 1, which will be used to incrementally assign port numbers. For each switch node in the graph G, the algorithm iterates through its neighboring nodes (i.e., the nodes directly connected to it by an edge). For each neighbor, a tuple in the form of (node, neighbor) is created to represent the edge between the two nodes. This tuple is then assigned a port number in the dictionary. After assigning a port number to the current edge, the p is incremented by 1, ensuring that each subsequent neighbor is assigned a unique port number. This process continues until all neighbors of all switch nodes have been assigned port numbers, completing the port allocation for the entire network.

In our example Figure 29, the network topology can be effectively modeled within the DyNetKAT framework. Based on the provided log files and the associated network

Algorithm 1 Topology Extraction

```
Require: Log File L
G ← Empty directed graph G
Prev_SrcHost ← ∅
Prev_DstHost ← ∅
Prev_SwitchName ← ∅
Flag ← False
for each Packet_In message(m) in the L do
    SrcHost ← Source host in the m
    DstHost ← Destination host in the m
    SwitchName ← switch name in the m
    if SrcHost is not in G then
        Add node SrcHost to G
    if DstHost is not in G then
        Add node DstHost to G
    if SwitchName is not in G then
        Add node SwitchName to G
    if Prev_DstHost == DstHost and Flag then
        Add edge Prev_SwitchName → Prev_DstHost to G
        Flag = False
    if Prev_SrcHost != SrcHost then
        Add edge SrcHost → SwitchName to G
        Flag = True
    else if Prev_SwitchName == SwitchName then
        Add edge Prev_SwitchName → SwitchName to G
    Prev_SrcHost ← SrcHost
    Prev_DstHost ← DstHost
    Prev_SwitchName ← SwitchName
return Topology G
```

Algorithm 2 Port Allocation

```
Require: Topology G
PortMap ← Empty dictionary
p = 1
for node in G do
    if node.type == "switch" then
        for n in node.neighbors do
            PortMap[(node,n)] ← p
            p ← p+1
return PortMap
```

behavior, the DyNetKAT topology for this example can be formally defined as shown in Equation 3.

$$\begin{aligned} T = \quad & (pt = 1 \;.\; pt \leftarrow 4) \; + (pt = 4 \;.\; pt \leftarrow 1) \\ & (pt = 2 \;.\; pt \leftarrow 5) \; + (pt = 5 \;.\; pt \leftarrow 2) \\ & (pt = 6 \;.\; pt \leftarrow 3) \; + (pt = 3 \;.\; pt \leftarrow 6) \end{aligned} \tag{3}$$

With the topology graph established and ports allocated, we can seamlessly define the network topology as a string representation, adhering to the formalism outlined in the NetKAT framework [AFG+14].

VIII. Extraction DyNetKAT rules from log file

We can uncover the behavior of the network by extracting information from log files. This section delves into the methods and strategies used to extract specific rules and patterns from log files.

A. Overview

In DyNetKAT, SDN behaviors are captured through expressions that combine flow tables, topologies, and controller actions. The overall network behavior is described by a composite expression that accounts for the communications between switches and controllers. The framework ensures that changes in the network's state, such as updates to flow tables, are only visible to new packets, maintaining the integrity of in-flight packets. This capability is vital for accurately modeling and verifying the real-time behavior of SDNs. The overall network behavior is expressed as

$$SDN \triangleq D_{X_1,\ldots,X_m} \,||\, C, \tag{4}$$

where $D_{X_1,\ldots,X_m}$ represents the collective operations of the data plane, consisting of m switches, encoded in terms of their corresponding flow tables $X_1, \ldots, X_m$, while C refers to controller. This parallel structure reflects the synchronous communication between the data plane and the controller, ensuring that updates to the network state, such as modifications to flow tables, are coherently propagated and enforced across the system. Also, we write the data plane expression as

$$\begin{aligned} D_{X_1,\ldots,X_m} \quad \triangleq \quad & ((X_1 + \ldots + X_m) \cdot T)^*; D_{X_1,\ldots,X_m} \oplus \\ & \textstyle\sum_{X_i' \in FT}^{\oplus} m_i?X_i'; D_{X_1,\ldots,X_i',\ldots,X_m} \end{aligned} \;. \tag{5}$$

As previously mentioned, an SDN consists of m switches, each encoded through their respective flow tables $X_1, \ldots, X_m$, and a network topology T. These terms collectively define the SDN's operational structure, with the flow tables specifying the forwarding rules at each switch and the topology T delineating the interconnections and communication pathways among the switches. The communication between the data plane and the control plane in an SDN is inherently synchronous, ensuring that any updates to the network's operational state are executed in a coordinated manner. Specifically, the controller issues updates to a flow table X_i, modifying it to a new state X_i', through a message dispatched via the designated communication channel m_i. This synchronous communication mechanism guarantees that changes are consistently propagated across the network, allowing for precise control over packet forwarding and other network behaviors.

It is assumed that switch i is capable of listening or reading its updated flow table on the communication channel m_i. This update is designed to be visible only to the new packets that are queued for processing, adhering

to the semantics of the sequential composition operator(;). In the DyNetKAT framework, flow table updates operate under the assumption that in-flight packets interact with only a single, consistent set of flow tables at any given time. The set FT represents the entirety of all possible flow tables that could be employed within the network, ensuring that updates are seamlessly integrated without disrupting ongoing network operations.

B. Rules Extraction

In this section, we present a methodology for extracting DyNetKAT specifications from SDN datasets, specifically focusing on Packet_In, Flow_Mod, and Packet_Out messages. The approach involves parsing the SDN log file to derive the DyNetKAT rules that define the network's behavior. By systematically analyzing these messages, we can construct a DyNetKAT specification that approximates the interaction between the data plane and controller, encapsulated in the expressions 4 and 5.
We begin by making several key assumptions for our methodology. First, the OpenFlow log file is extracted using Wireshark, which serves as the input for our tool. Second, during the preprocessing step, the network topology is extracted following the procedure outlined in Algorithm 1. Additionally, it is assumed that initially, no flow tables are installed on the switches, and the DyNetKAT controller expression C is initially set to $\perp$. Furthermore, the control plane and data plane are assumed to communicate synchronously. Consequently, the right-hand side summand of $\oplus$ in expression 5 updates switch flow table X_i to X_i' only when the controller initiates it through a message sent on channel m_i.
As discussed in the section VII-A, there are three critical types of messages exchanged between switches and the controller: Packet_In, Flow_Mod, and Packet_Out. In the following, we will define the corresponding DyNetKAT encodings for each of these message types.

1) Packet_In(sw, mid, o_mids): This indicates that a switch sw has transmitted the OpenFlow message mid to the controller, identified by o_mids. The corresponding DyNetKAT encoding,

$$D_{X_1,\ldots,X_m} \oplus = \Sigma^{\oplus}_{o\_mid \in o\_mids} o\_mid!1; D_{X_1,\ldots,X_m} \tag{6}$$

facilitates synchronous communication with the SDN controller. Encoding 6 formalizes the interaction between the switch and the controller, ensuring that messages are correctly relayed and processed within the network's control framework.

2) Flow_Mod(sw, mid', o_mids'): This indicates that the controller has received and processed the OpenFlow message mid', subsequently generating the OpenFlow messages identified in o_mids' as a response. It is important to note that mid' corresponds to one of the $o\_mid \in o\_mids$ elements. The DyNetKAT encoding for this operation is represented as

$$C \oplus = mid'?1; (\sum^{\oplus}_{o\_mid^i_{sw} \in o\_mids'} o\_mid^i_{sw}!ft\_mid^i_{sw}; C) \tag{7}$$

where $ft\_mid^i_{sw}$ is calculated based on the controller decision. Encoding 7 captures the precise interaction between the controller and the data plane, ensuring that the flow table modifications are correctly communicated

3) Packet_Out(sw, mid, m_type, ops): This signifies that the switch sw has received and processed the OpenFlow message mid of type m_type. The processing of this message involves executing the operations ops specified in the associated flow table. These operations ops dictate how the switch manages incoming packets, updating its forwarding behavior accordingly. The corresponding DyNetKAT representation is denoted as

$$D_{X_1,\ldots,X_i,\ldots,X_m} \oplus = mid?X_i^{'}; D_{X_1,\ldots,X_i',\ldots,X_m} \tag{8}$$

with $X_i^{'} = ft$ and ft computed according to ops. Encoding 8 encapsulates the data plane's response to the controller's directives. By formalizing this process, DyNetKAT ensures a precise and verifiable model of the switch's operational behavior within the SDN.

C. Running Example

In the context of our single switch example depicted in Figure 29, the extraction of DyNetKAT terms is directly informed by the observed network messages (VII-A), as recorded in the log file. The behavior of the Software-Defined Network is described as the parallel composition of the data plane $D_{X_{S1}}$ and the controller C, which together are represented by equation 9. We assume the existence of a predefined rule, as outlined in Equation 2, within switch S1 ($X_{S1} \triangleq pt = 4 \;.\; pt \leftarrow 6$). Furthermore, the behavior of the controller remains unknown ($C \triangleq \perp$). The topology T, as defined in Equation 3, has been extracted following the methodology described in Section VII-B.

$$\begin{gathered} SDN \triangleq D_{X_{S1}} \,||\, C \\ D_{X_{S1}} \triangleq ((X_{S1}) \cdot T)^*; D_{X_{S1}} \end{gathered} \tag{9}$$

When a Packet_In message (Figure 30) is observed, the data plane updates itself by adding a send communication term according to rule 6, as represented in equation 10.

$$\begin{gathered} D_{X_{S1}} \triangleq ((X_{S1}) \cdot T)^*; D_{X_{S1}} \oplus \\ ch!1; D_{X_{S1}} \end{gathered} \tag{10}$$

Subsequently, the controller C updates its own terms based on rule 7 and Flow_Mod message 31, as indicated in equation 11.

$$C \triangleq (ch?1; (ch'!(pt = 5.pt \leftarrow 6); C)) \tag{11}$$

Finally after the Packet_out message 32, the data plane and the flow table of the switch are updated following rule 8, leading to the formulation shown in equation 12.

$$X'_{S1} \triangleq (pt = 5.pt \leftarrow 6)$$

$$\begin{aligned} D_{X_{S1}} \triangleq ((X_{S1}) \cdot T)^*; D_{X_{S1}} \oplus \\ ch!1; D_{X_{S1}} \oplus \\ ch'?X'_{S1}; D_{X'_{S1}} \end{aligned} \tag{12}$$

$$\begin{aligned} D_{X'_{S1}} \triangleq ((X'_{S1}) \cdot T)^*; D_{X'_{S1}} \oplus \\ ch!1; D_{X'_{S1}} \oplus \\ ch'?X'_{S1}; D_{X'_{S1}} \end{aligned}$$

This example demonstrates the step-by-step process by which DyNetKAT terms are systematically extracted from network events, reflecting the dynamic behavior of the SDN as it responds to incoming traffic and controller directives.

IX. Implementation

In the implementation of Fault Prediction in Software-Defined Networks(SDNs), our focus is on developing a robust framework that leverages formal methods to ensure network reliability and resilience. To achieve this, we have developed a prototype tool named FPSDN, which is designed to predict and mitigate potential faults in SDN environments.

We devised a mechanism for extracting DyNetKAT specifications from these real log files, enabling us to identify possible faulty behavior in a fully automated fashion(FPSDN GitHub repository). Our tool, FPSDN, is built on the DyNetKAT framework, which itself is based on Maude, the NetKAT decision procedure, and Python. This combination of technologies allows us to harness the formal verification capabilities of DyNetKAT while integrating them with practical tools for network analysis and management. The use of DyNetKAT in conjunction with real log data ensures that our fault detection mechanism is both accurate and comprehensive, as it considers the actual conditions observed in the network. The input to FPSDN is a .pcapng Wireshark log file, which contains the network traffic data captured from the SDN. Upon receiving this input, the tool performs several preprocessing steps: it first removes any duplicated packets to ensure data accuracy, then it proceeds to extract the network topology and allocate ports to the corresponding nodes. Following this, the tool applies predefined extraction rules, as outlined in Section VIII-B, to derive the DyNetKAT terms that model the network's behavior.

X. Experiments

To evaluate the effectiveness and performance of FPSDN, we conducted a series of experiments using various network topologies and their corresponding log files. The experiments were designed to assess the tool's ability to process real network data and accurately extract DyNetKAT terms, which are crucial for fault prediction. In all experiments, the network initially resides in a safe state, and after multiple switch reconfigurations, it remains safe. To define the experimental scenarios, we use the notation $Path(X, Y)$ to indicate that device Y is reachable from device X. Similarly, $Add\_Rule(X, Y, Z)$ represents an update applied to switch X, establishing a connection between device $Y$and Z. Additionally, the exclamation mark (!) denotes negation; thus, $!Path(X, Y)$ signifies that device Y is not reachable from device X.

A. Star Experiment

In this experiment, we implement a star topology (Figure 33) consisting of a central switch connected to six hosts. The scenario begins with Host 1 initially having communication access only to Host 2. Through a sequence of four updates to the switch's flow table, Host 1 gradually gains access to all other hosts in the network.

Initial safe state: Host 1 can only reach Host 2 ($Path(H1, H2)$).

Secondary safe states: After applying the updates, Host 1 can communicate with all other hosts ($Path(H1, H3), Path(H1, H4), Path(H1, H5),$ *and* $Path(H1, H6)$).

Dynamic updates: $Add\_Rule(S7, S1, S3)$, $Add\_Rule(S7, S1, S4), Add\_Rule(S7, S1, S5),$ *and* $Add\_Rule(S7, S1, S6)$.

Safety properties: This experiment aims to validate the correctness of our tool in identifying and maintaining both the initial and secondary safe states. Since the updates are controlled and do not introduce an unsafe transition, no fault is defined in this experiment.

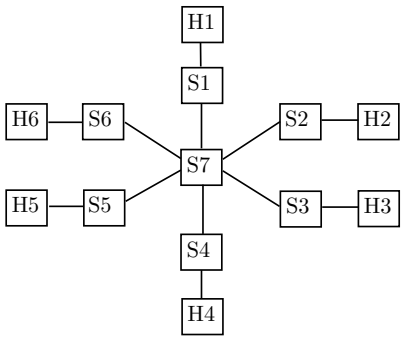

Figure 33. Star Experiment Topology

B. Mesh Experiment

In this experiment, we implement a mesh topology (Figure 34) consisting of six independent switches, each connected to a dedicated host. The scenario begins with Host 1 (connected to Switch 1) initially having

communication access only to Host 2 (connected to Switch 2). Through a sequence of four updates to the flow table of Switch 1, Host 1 progressively gains access to all other hosts in the network.

Initial safe state: Host 1 can only reach Host 2 ($Path(H1, H2)$).

Secondary safe states: After applying the updates, Host 1 can communicate with all other hosts ($Path(H1, H3), Path(H1, H4), Path(H1, H5),$ *and* $Path(H1, H6)$).

Dynamic updates: $Add\_Rule(S1, H1, S3)$, $Add\_Rule(S1, H1, S4), Add\_Rule(S1, H1, S5),$ *and* $Add\_Rule(S1, H1, S6)$.

Safety properties: This experiment is designed to validate the correctness of our tool in maintaining both the initial and secondary safe states. Since the updates do not introduce an unsafe transition, no fault is defined in this experiment.

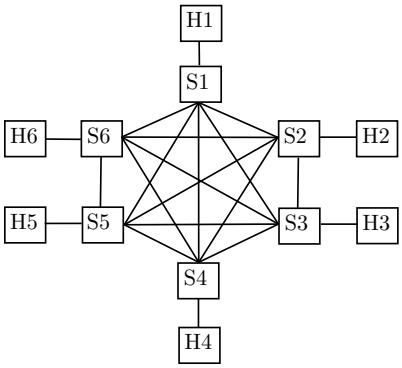

Figure 34. Mesh Experiment Topology

C. Ring Experiment

In this experiment, we implement a ring topology (Figure 35) consisting of six switches, each connected to a dedicated host. The scenario begins with Host 1 initially having communication access only to Host 5 (connected to Switch 5). Through two updates to the flow tables of the switches, Host 3 gains the ability to communicate with Host 6.

Initial safe state: Host 1 can reach Host 5 ($Path(H1, H5)$).

Secondary safe state: After applying the updates, Host 3 can reach Host 6 ($Path(H3, H6)$).

Dynamic updates: $Add\_Rule(S3, H3, S4)$ *and* $Add\_Rule(S5, S4, S6)$.

Safety Properties: After the updates, Host 3 must not be able to communicate with Host 5 ($!Path(H3, H5)$).

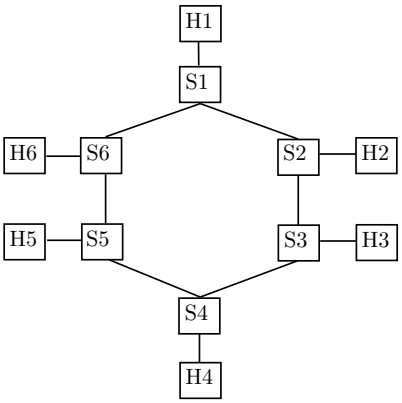

Figure 35. Ring Experiment Topology

D. Black Hole Experiment

In this experiment, we implement a linear topology (Figure 38) consisting of ten switches, each connected to a dedicated host. The scenario begins with Host 1 initially having communication access to Host 4, while Host 2 is unable to communicate with Host 3. After applying two updates to the flow tables of the switches, Host 2 gains the ability to communicate with Host 3.

Initial safe state: Host 1 can reach Host 4. Host 2 cannot reach Host 3 before the updates ($Path(H1, H4)$)

Secondary safe state: After applying the updates, Host 2 can reach Host 3 ($Path(H2, H3)$).

Dynamic updates: $Add\_Rule(S2, S1, H2)$ *and* $Add\_Rule(S3, S2, H3)$.

Safety Properties: After the updates, Host 3 must not be able to communicate with Host 1 ($!Path(H1, H3)$), and Host 4 must not be able to communicate with Host 2 ($!Path(H2, H4)$).

E. Race Condition Experiment

In this experiment, we implement a network topology (Figure 36) consisting of six switches and four hosts. The scenario begins with Host 1 initially having communication access to Host 3. After applying two updates to the flow tables of the switches, Host 2 gains the ability to communicate with Host 4.

Initial safe state: Host 1 can reach Host 3 ($Path(H1, H3)$).

Secondary safe state: After applying the updates, Host 2 can reach Host 4 ($Path(H2, H4)$).

Dynamic updates: $Add\_Rule(S5, S2, S6)$ *and* $Add\_Rule(S6, S5, S4)$

Safety properties: After the updates, Host 4 must not be

able to communicate with Host 1 ($!Path(H1, H4)$), and Host 3 must not be able to communicate with Host 2 ($!Path(H2, H3)$).

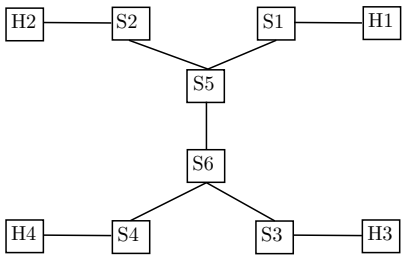

Figure 36. Race Condition Experiment Topology

F. Isolation Experiment

In this experiment, we implement a linear topology (Figure 37) consisting of six switches and six hosts, where every two adjacent switches and their corresponding hosts are considered as a group. The scenario begins with Host 1 (from Group 1) having communication access to Host 4 (from Group 2). After applying two updates to the flow tables of the switches, Host 3 (from Group 2) gains the ability to communicate with Host 6 (from Group 3).

Initial safe state: Host 1 can reach Host 4 ($Path(H1, H4)$).

Secondary safe state: After applying the updates, Host 3 can reach Host 6 ($Path(H3, H6)$).

Dynamic updates: $Add\_Rule(S3, H3, S4)$ *and* $Add\_Rule(S4, S3, S5)$.

Safety properties: After the updates, Host 6 must not be able to communicate with Host 1 ($!Path(H1, H6)$).

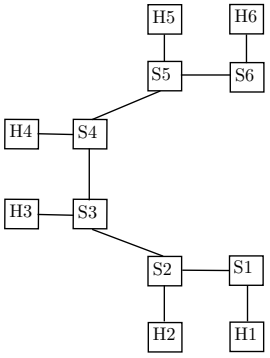

Figure 37. Isolation Experiment Topology

G. Faulty Linear Experiment

In this experiment, we implement a linear topology (Figure 38) consisting of ten switches, each connected to a corresponding host. The scenario begins with Host 7 initially having communication access to Host 10. After applying three updates to the flow tables of the switches: Host 6 gains the ability to communicate with Host 10. Host 4 gains the ability to communicate with Host 8.

Initial safe state: Host 7 can reach Host 10 ($Path(H7, H10)$).

Secondary safe state: After applying the updates, Host 6 can reach Host 10 ($Path(H6, H10)$).

Dynamic updates: $Add\_Rule(S7, S6, S8)$, $Add\_Rule(S6, S5, S7)$, *and* $Add\_Rule(S8, S7, H8)$.

Safety properties: After the updates, Host 10 must not be able to communicate with Host 4 ($!Path(H4, H10)$), and Host 8 must not be able to communicate with Host 6 ($!Path(H6, H8)$).

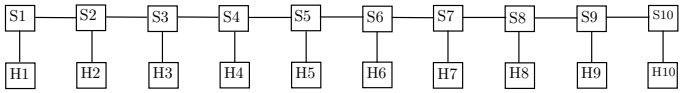

Figure 38. Linear Topology

H. Faulty Fattree Experiment

In this experiment, we implement a Fattree network topology (Figure 39). The scenario begins with Host 2 having communication access to Host 8. After applying three updates to the flow tables of the switches:
Host 5 gains the ability to communicate with Host 7.
Host 1 gains the ability to communicate with Host 8.

Initial safe state: Host 2 can reach Host 8 ($Path(H2, H8)$).

Secondary safe states: After applying the updates, Host 5 can reach Host 7 ($Path(H5, H7)$) and Host 1 can reach Host 8 ($Path(H1, H8)$).

Dynamic updates: $Add\_Rule(T1, H1, A1)$, $Add\_Rule(C1, A5, A7)$, *and* $Add\_Rule(A7, C1, T7)$

Safety properties: After the updates, Host 7 must not be able to communicate with Host 1 ($!Path(H1, H7)$), Host 8 must not be able to communicate with Host 5 ($!Path(H5, H8)$), and Host 7 must not be able to communicate with Host 2 ($!Path(H2, H7)$).

To simulate the network and these topologies, we employed Mininet [KA15], a popular network emulator that allows for the creation and testing of virtual networks. The network was managed using the POX controller [PPS+22], an open-source controller platform that supports OpenFlow, enabling us to control and monitor the network's behavior effectively. By integrating POX with Mininet, we could simulate real-world scenarios and dynamically manage network flows, which is critical for testing the fault prediction capabilities of FPSDN.

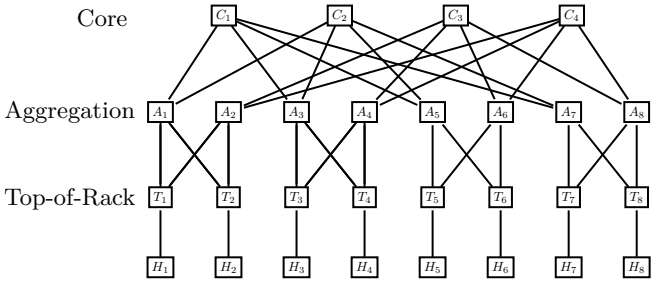

Figure 39. Fattree Experiment Topology

XI. Results

The detailed results of our experiments are available for download in the result_with_detail.xlsx file, which can be accessed in the GitHub repository of our tool. Additionally, Table I provides a summary of the key outcomes. In the following sections, we analyze the results for each experiment individually.

A. Star Experiment

In this experiment, a total of 17 different sequences of switch flow table updates were automatically tested, each evaluated against five predefined security properties. The results confirm the correctness of the tool in maintaining expected network behaviors:
Initial Safe State Validation: The reachability of Host 2 from Host 1 was correctly maintained across all tested update sequences.
Secondary Safe State Validation: After the updates, the reachability of Hosts 3, 4, 5, and 6 from Host 1 was correctly preserved in all cases.
Since this experiment was designed to validate the tool's accuracy in modeling safe network updates, no faults were detected, confirming that the expected safe states remained intact throughout all tested scenarios.

B. Mesh Experiment

In this experiment, a total of five different sequences of switch flow table updates were automatically tested, each evaluated against five security properties. The results confirm that the expected network behaviors were correctly maintained:
Initial Safe State Validation: The reachability of Host 2 from Host 1 was successfully maintained in all tested cases.
Secondary Safe State Validation: After the updates, the reachability of Hosts 3, 4, 5, and 6 was correctly established as expected.
Since the primary objective of this experiment was to validate the accuracy of the tool in modeling and verifying safe network updates, no faults were detected, confirming that the system consistently upheld the defined security properties.

C. Ring Experiment

In this experiment, a total of five different sequences of switch flow table updates were automatically tested, each evaluated against three security properties. The results indicate the following key findings:
Initial Safe State Validation: The reachability of Host 5 from Host 1 was correctly maintained in the initial network configuration.
Secondary Safe State Validation: After the updates, the reachability of Host 6 from Host 3 was also correctly preserved.
Fault Detection: The experiment defined a fault condition as the connectivity between Host 5 and Host 3.
The results show that there exists at least one specific update sequence that violates this property, leading to a connectivity.

D. Black Hole Experiment

In this experiment, a total of five different sequences of switch flow table updates were automatically tested, each evaluated against five security properties. The key observations from this experiment are as follows:
Initial Safe State Validation: The reachability of Host 4 from Host 1 was correctly maintained before any flow table updates. The non-reachability of Host 3 from Host 2 in the initial state was also correctly enforced.
Secondary Safe State Validation: After the updates, the reachability of Host 3 from Host 2 was correctly established, ensuring the expected connectivity was achieved.
Fault Prediction and Detection: The fault prediction mechanism aimed to identify any update sequence that could violate two security properties: Non-reachability of Host 3 from Host 1 and non-reachability of Host 4 from Host 2. The results indicate that at least one specific update sequence exists that leads to the violation of these properties, demonstrating a potential misconfiguration scenario.

E. Race Condition Experiment

In this experiment, a total of five different sequences of switch flow table updates were automatically tested, each evaluated against four security properties. The results of the experiment are summarized as follows: // Initial Safe State Validation: The reachability of Host 3 from Host 1 was successfully maintained in the initial configuration without any updates. // Secondary Safe State Validation: After the updates, the reachability of Host 4 from Host 2 was correctly preserved, ensuring the expected network connectivity. // Fault Prediction and Detection: The fault condition was defined as finding an update sequence that could violate the following security properties: Non-reachability of Host 4 from Host 1. Non-reachability of Host 3 from Host 2. The results indicate that at least one specific update sequence exists that leads to the violation of these properties, causing unintended disruptions in the network.

F. Isolation Experiment

In this experiment, a total of five different sequences of switch flow table updates were automatically tested, each evaluated against three security properties. The results can be summarized as follows:
Initial Safe State Validation: The reachability of Host 4 from Host 1 was correctly maintained without any updates, confirming the initial network configuration was secure.
Secondary Safe State Validation: After the updates, the reachability of Host 6 from Host 3 was correctly preserved, despite the sequence of flow table updates.
Fault Prediction and Detection: The fault prediction focused on identifying an update sequence that could violate the following security property: Non-reachability of Host 6 from Host 1. The results indicate that there exists at least one specific update sequence that leads to the violation of this property, disrupting the connectivity as expected.

G. Faulty Linear Experiment

In this experiment, a total of ten different sequences of switch flow table updates were automatically tested, each evaluated against four security properties. The results of the experiment can be summarized as follows:
Initial Safe State Validation: The reachability of Host 10 from Host 1 was correctly maintained without any updates, ensuring the initial secure state of the network configuration.
Secondary Safe State Validation: After the updates, the reachability of Host 10 from Host 6 was correctly established, even after a single switch flow table update, confirming that secondary connectivity was preserved.
Fault Prediction and Detection: The fault prediction was focused on identifying an update sequence that could violate the following security properties: Non-reachability of Host 10 from Host 4. Non-reachability of Host 8 from Host 6. The experiment revealed that at least one specific update sequence exists that leads to the violation of these properties, causing connectivity issues as anticipated.

H. Faulty Fattree Experiment

In this experiment, a total of ten different sequences of switch flow table updates were automatically tested, with each sequence evaluated against six security properties. The results of the experiment are summarized as follows:
Initial Safe State Validation: The reachability of Host 8 from Host 2 was correctly maintained without any updates, ensuring the initial secure state of the network configuration.
Secondary Safe State Validation: After the updates, the reachability of Host 7 from Host 5 and the reachability of Host 8 from Host 1 were successfully established, confirming that secondary network connectivity was appropriately maintained after the flow table updates.
Fault Prediction and Detection: The fault prediction in this experiment focused on identifying an update sequence that could violate the following security properties: Non-reachability of Host 7 from Host 1. Non-reachability of Host 8 from Host 5. Non-reachability of Host 7 from Host 2. The results indicate that at least one specific update sequence exists that leads to the violation of these three security properties, causing connectivity issues as predicted.

These experiments validate the correctness of the verification tool in identifying both expected and unintended behaviors in network updates, reinforcing its effectiveness in predicting security violations in Software-Defined Networks (SDNs). Specifically, by evaluating multiple update sequences in various network topologies, the tool successfully confirmed the initial security properties, such as host reachability and isolation, while also identifying conditions under which these properties could be violated. In scenarios where sequential flow table updates altered connectivity constraints, the tool detected security breaches, including unintended host communication and loss of isolation guarantees. The results demonstrate that the tool is capable of systematically analyzing different update orders, predicting potential faults, and ensuring that network policies remain intact despite dynamic modifications.

As shown in the results table available in the code repository, a total of eight experiments were conducted, evaluating 35 security properties, including initial secure state properties, secondary secure state properties, and fault prediction properties. The total number of packets before preprocessing across these eight experiments was 22,451, which was reduced to 352 after preprocessing—representing a 98.43% reduction in packet volume. This significant reduction facilitates the learning of Software-Defined Network (SDN) behaviors. In terms of time efficiency, the total preprocessing time for these experiments was 60.94 seconds, while the time required for SDN behavior extraction was 8.36 milliseconds. Additionally, a total of 285 different update sequences were examined across the 35 security properties. The evaluation of these 285 sequences required 761.20 seconds in total.

The only scenarios where our tool was unable to provide a response were observed in the Star experiment, where DyNetKAT was unable to analyze 10 different update sequences in the flow tables. All of these analyses were conducted with a maximum of two updates (configurations) in the network. This limitation arises from the underlying infrastructure of the DyNetKAT language, which faces computational constraints when analyzing properties with a higher number of configurations. As the number of updates increases, the computation time grows to the extent that the program halts due to excessive runtime. This highlights the need for future optimization of algorithms and methods for analyzing more complex scenarios. Different scenarios have been explored and presented.

We conducted the experiments on a computer running Ubuntu 23.04, equipped with an Intel i5-4210U processor and 4 GB of RAM. This hardware setup reflects a typical mid-range environment, ensuring that the results are both practical and replicable in similar settings. To replicate these results, you can follow the detailed instructions provided in the FPSDN GitHub repository[1]. The repository includes all necessary scripts, data files, and additional documentation to help you set up the environment, preprocess the log files, and extract DyNetKAT rules, allowing you to verify the outcomes discussed in this report.

XII. Conclusion

In conclusion, we successfully implemented fault prediction in Software-Defined Networks(SDNs) using the DyNetKAT framework and our FPSDN tool. Tested on a simulated FatTree topology, our tools demonstrated the ability to identify network faults introduced by dynamic reconfigurations. These findings suggest that as SDNs become more integral to critical infrastructure, tools like FPSDN will be indispensable for preemptively identifying and addressing faults before they can lead to network failures.

References

[AFG+14] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: semantic foundations for networks. In Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, page 113–126, New York, NY, USA, 2014. Association for Computing Machinery.

[AFLV08] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication, SIGCOMM '08, page 63–74, New York, NY, USA, 2008. Association for Computing Machinery.

[BGMW17] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '17, page 155–168, New York, NY, USA, 2017. Association for Computing Machinery.

[CHMT21] Georgiana Caltais, Hossein Hojjat, Mohammad Reza Mousavi, and Hünkar Can Tunç. Dynetkat: An algebra of dynamic networks. CoRR, abs/2102.10035, 2021.

[EV02] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. SIGCOMM Comput. Commun. Rev., 32(4):323–336, aug 2002.

[GJVP+14] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. Opennf: enabling innovation in network function control. In Proceedings of the 2014 ACM Conference on SIGCOMM, SIGCOMM '14, page 163–174, New York, NY, USA, 2014. Association for Computing Machinery.

[Ham09] James Hamilton. Networking: The last bastion of mainframe computing. https://perspectives.mvdirona.com/2009/12/networking-the-last-bastion-of-mainframe-computing/. 2009.

[KA15] Faris Keti and Shavan Askar. Emulation of software defined networks using mininet in different simulation environments. In 2015 6th International Conference on Intelligent Systems, Modelling and Simulation, pages 205–210, 2015.

[MAB+08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. SIGCOMM Comput. Commun. Rev., 38(2):69–74, mar 2008.

[PPS+22] Riya Patel, Parth Patel, Parth Shah, Bansari Patel, and Dweepna Garg. Software defined network (sdn) implementation with pox controller. In 2022 3rd International Conference on Smart Electronics and Communication (ICOSEC), pages 65–70, 2022.

[Slo13] Timon Sloane. Software-defined networking: the new norm for networks. open networking foundation. (may 2, 2013). retrieved june 20, 2023 from https://opennetworking.org/sdn-resources/whitepapers/software-defined-ne tworking-the-new-norm-for-networks/. 2013.

[The12] The Open Networking Foundation. OpenFlow Switch Specification, Jun. 2012.

[ZKVM12] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies, CoNEXT '12, page 241–252, New York, NY, USA, 2012. Association for Computing Machinery.

[1] https://github.com/mghobakhlou/FPSDN

Table I
Summary of Experiment Results

Experiment Name	Packets Before Preprocessing	Packets After Preprocessing	Preprocessing Time (Seconds)	Rule Extraction Time (Milliseconds)	Property Name	Verification Time by DyNetKAT (Seconds)	Result
Ring	975	36	2.53	0.61	Path(h1,h5)	0.33	Satisfied
					Path(h3,h6)	0.91	Satisfied
					!Path(h3,h5)	0.36	Violated
Mesh	1767	40	4.42	1.22	Path(h1,h2)	0.38	Satisfied
					Path(h1,h3)	0.44	Satisfied
					Path(h1,h4)	0.43	Satisfied
					Path(h1,h5)	0.44	Satisfied
					Path(h1,h6)	0.42	Satisfied
Race Condition	529	32	1.41	0.58	Path(h1,h3)	0.34	Satisfied
					Path(h2,h4)	0.94	Satisfied
					!Path(h1,h4)	0.35	Violated
					!Path(h2,h3)	0.34	Violated
Isolation	2347	32	6.16	0.66	Path(h1,h4)	0.34	Satisfied
					Path(h3,h6)	0.94	Satisfied
					!Path(h1,h6)	0.34	Violated
Blackhole	1893	24	5.19	0.56	Path(h1,h4)	0.34	Satisfied
					Path(h2,h3) before_rcfg	0.33	Satisfied
					Path(h2,h3) after_rcfgs	0.82	Satisfied
					!Path(h2,h4)	0.33	Violated
					!Path(h1,h3)	0.33	Violated
Fattree Faulty	11713	60	32.99	1.04	Path(h2,h8)	0.5	Satisfied
					Path(h5,h7)	23.33	Satisfied
					Path(h1,h8)	0.71	Satisfied
					!Path(h1,h7)	23.47	Violated
					!Path(h5,h8)	0.55	Violated
					!Path(h2,h7)	0.64	Violated
Star	1567	72	4.01	2.48	Path(h1,h2)	0.69	Satisfied
					Path(h1,h3)	1.05	Satisfied
					Path(h1,h4)	1.02	Satisfied
					Path(h1,h5)	1.13	Satisfied
					Path(h1,h6)	1.02	Satisfied
Linear Faulty	1660	56	4.23	1.21	Path(h7,h10)	0.5	Satisfied
					Path(h6,h10)	0.54	Satisfied
					!Path(h4,h10)	23.58	Violated
					!Path(h6,h8)	23.63	Violated
Sum	22451	352	60.94	8.36			
Average	2806.37	44	7.61	1.04			