Exploiting Temporal Vulnerabilities for Unauthorized Access in Intent-Based Networking

Ben Weintraub
MIT Lincoln Laboratory
Lexington, MA, USA
Northeastern University
Boston, MA, USA
weintraub.b@northeastern.edu

Jiwon Kim
Purdue University
West Lafayette, IN, USA
kim1685@purdue.edu

Ran Tao
Georgetown University
Washington DC, USA
rt822@georgetown.edu

Cristina Nita-Rotaru
Northeastern University
Boston, MA, USA
c.nitarotaru@northeastern.edu

Hamed Okhravi
MIT Lincoln Laboratory
Lexington, MA, USA
hamed.okhravi@ll.mit.edu

Dave (Jing) Tian
Purdue University
West Lafayette, IN, USA
daveti@purdue.edu

Benjamin E. Ujcich
Georgetown University
Washington DC, USA
bu31@georgetown.edu

ABSTRACT

Intent-based networking (IBN) enables network administrators to express high-level goals and network policies without needing to specify low-level forwarding configurations, topologies, or protocols. Administrators can define intents that capture the overall behavior they want from the network, and an IBN controller compiles such intents into low-level configurations that get installed in the network and implement the desired behavior.

We discovered that current IBN specifications and implementations do not specify that flow rule installation orderings should be enforced, which leads to temporal vulnerabilities where, for a limited time, attackers can exploit indeterminate connectivity behavior to gain unauthorized network access.

In this paper, we analyze the causes of such temporal vulnerabilities and their security impacts with a representative case study via the ONOS IBN implementation. We devise the Phantom Link attack and demonstrate a working exploit to highlight the security impacts. To defend against such attacks, we propose `Spotlight`, a detection method that can alert a system administrator of risky intent updates prone to exploitable temporal vulnerabilities. `Spotlight` is effective in identifying risky updates using realistic network topologies and policies. We show that `Spotlight` can detect risky updates in a mean time of 0.65 seconds for topologies of over 1,300 nodes.

CCS CONCEPTS

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

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10.
https://doi.org/10.1145/3658644.3670301

KEYWORDS

Intent-based Networking, Software-defined Networking, Access Control, Vulnerability, Timing Attack

ACM Reference Format:
Ben Weintraub, Jiwon Kim, Ran Tao, Cristina Nita-Rotaru, Hamed Okhravi, Dave (Jing) Tian, and Benjamin E. Ujcich. 2024. Exploiting Temporal Vulnerabilities for Unauthorized Access in Intent-Based Networking. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3670301

1 INTRODUCTION

Over the last decade, software-defined networking (SDN) has been deployed in a variety of production settings to ease the burden on network administrators in managing control plane operations without manual intervention on every switch[1]. More recently, intent-based networking (IBN) allows an administrator to express network policies without configuring individual network devices with specific flow rules. Instead, administrators define intents that capture the overall behavior they want from the network, then an SDN controller translates them into low-level flow rules and communicates with the network devices to implement the desired behavior. IBN has been standardized by the Open Networking Foundation (ONF) [53], the Internet Engineering Task Force (IETF) [21, 45], and the 3rd Generation Partnership Project (3GPP) [5–7].

Once an administrator defines their desired policy, two steps take place prior to enforcing such policy. First, each intent is compiled into a set of flow rules through *intent compilation*. Then, the individual flow rules corresponding to each intent are sent to the

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

[1]We use "switch" and "forwarding device" synonymously throughout the paper.

devices through *intent installation*. Given the complexities of the inherently distributed system, it becomes challenging to ensure that the intended behavior of all flow rules becomes actionable and consistent at the same time when flows are updated.

Addressing such update inconsistencies is critical for security. For instance, given a security network policy, a host should never have connectivity to a host to which it is not explicitly granted access. When a host does become connected to an unauthorized host that it was not connected to before the update started and will not be connected to after the update completes, we refer to this as a *temporal vulnerability*. If such connectivity is the result of a malicious action we refer to it as a *temporal attack*. We also refer to the resulting additional connectivity as *unauthorized connectivity*.

Although prior work has studied the problem of inconsistent updates in SDN, no work has focused on the security implications to the IBN layer or the security implications of inconsistent data plane updates. Several models for consistent updates have been proposed [48, 51, 61], but unfortunately none of them have been widely adopted, likely due to both memory and time overhead. The introduction of intents as a new abstraction presents additional attack vectors, while inconsistent updates and their potential impact on security continue to remain an unsolved problem. As a feature, IBN provides conditions under which flow rules will be recomputed and redeployed. This implies an expanded threat model in which an attacker can influence when and how this redeployment occurs. These capabilities increase the attacker's control over the system, and have not been considered in prior work. IBN-related security has focused on secure ACL update plans [71], provenance analysis [72], and automatic bug discovery [42], but none have identified any temporal vulnerability during intent updates. To the best of our knowledge, no work has studied temporal vulnerabilities in IBN caused by inconsistent updates.

Overview. In this paper, we focus on IBN temporal attacks. We explore different approaches that can be exploited by an attacker to conduct such attacks: monitoring flow installation, delaying installation of specific rules, or forcing intents recompilations by an honest controller, all with the goal of creating an inconsistency that gives the attacker unauthorized access.

We demonstrate a concrete delay attack, which we refer to as the Phantom Link attack. In this attack, we show that an attacker-controlled host is able to create transient connections to other hosts that are not specified by the intent. In other words, despite an intent dictating where a host is able to connect, we are able to create short-lived connections to other hosts. Additionally, we are able to extend this *unauthorized connectivity* for long enough to show that an exploit can be delivered. We are, to the best of our knowledge, the first to show how an attacker can gain unauthorized connectivity via inconsistent flow rule updates without access to the controller or switches.

Solving the problem of inconsistent updates is challenging. On one side, existing IBN standards [5–7, 21, 45] do not specify update consistency guarantees or how updates ought to be implemented. As a result, developers may design or implement update consistency differently. Defenses implemented in popular controllers, like barriers and stripe keys, are insufficient to protect against our attacks. Stripe keys can only enforce the ordering when sending flow rule operations. They do not guarantee the install ordering among switches. An attacker can exploit this by selectively delaying updates to some switches. Likewise, barrier messages can only enforce the flow rule processing order *on a single switch*. We exploit improper flow rule update ordering between multiple switches, so barrier messages are not an effective defense. We demonstrate our attacks for the ONOS IBN implementation. Changing standards takes time; meanwhile, administrators are left either unaware or unable to deal with such inconsistencies and the vulnerabilities they introduce.

On the other side, enforcing consistent updates in the entire network is costly and not always necessary. We propose a solution designed to empower administrators without paying the network cost of enforcing global consistency on all the flow rule updates. Our approach instead is to identify high-risk intent updates before attempting to install the resulting flow rules on the network.

We propose `Spotlight`, a system that identifies updates prone to temporal vulnerabilities. We first show a strawman approach that finds all possible orders in which flow rules can be installed, which we then inspect individually to verify if it would lead to a transient connection outside the expectations of the IBN administrator. This approach is costly and is only feasible on small networks, but finds all such risky updates. We then present a faster method based on *link prediction* algorithms that scales better to large network graphs. Although this faster algorithm does not guarantee to find all risky updates, we show that, with proper tuning, we are able to identify risky updates in all properly tuned trials, even on topologies with over 1,300 nodes in a mean time of 0.65 seconds—this presents a speedup of 76.8× over the exhaustive search approach.

Empowered with `Spotlight`, an administrator can decide how to handle such risky updates. Our experimental evaluation shows that their number is small, thus for example, the administrator might decide to use a consistent update enforcement for those particular intents such as those previously proposed [48, 51, 61].

Contributions. We make the following contributions:

- We explore the space of temporal attacks in IBN and show different ways in which an attacker can exploit them: by monitoring intent installation, by forcing intent recompilation, or by delaying intent installation.
- We demonstrate an exploitable concrete temporal attack that controls the order of flow rules by delaying the installation of an intent at a particular switch. We refer to this attack as the Phantom Link attack.
- We design the `Spotlight` detection method to identify high-risk IBN updates.
- We implement and evaluate `Spotlight`, and show that it can detect risky updates in a mean time of 0.65 seconds for topologies of over 1,300 nodes. We open source `Spotlight` for the benefit of the community[2].

Ethics. We identified a novel vulnerability in the well-known ONOS SDN controller. We responsibly disclosed our discovery to the ONOS security response team of developers. Our attack was assigned CVE-2024-24270.

[2]https://zenodo.org/doi/10.5281/zenodo.11642349

2 BACKGROUND

2.1 Software-defined Networking (SDN)

Software-defined networking (SDN) differs from traditional networking by separating how traffic forwarding decisions are made (i.e., the *control plane*) from the traffic itself (i.e., the *data plane*), centralizing the control into a logically centralized *SDN controller*, and exposing programmable APIs to developers who want to extend the control plane functionality [43].

2.1.1 Programming the network. To program the network, the SDN controller sends *flow rules* to the network's forwarding devices (e.g., switches and routers). Flow rules contain "match-action" fields that specify parameters of incoming packets that should be matched and a specific action to be taken. For instance, a typical flow rule may specify (1) an ingress port to match, (2) an egress port to forward the packet out on, and (3) a set of header fields (e.g., IP destination address for routing) to match. A flow rule is triggered if the packet is received on the ingress port and matches the specified pattern, in which case the packet is sent out on the egress port.

Flow rules can be installed or removed by the controller at any time, either actively or passively (e.g., timeout due to lack of matching packets). A controller may send out many flow rule *updates* (i.e., installations, removals, or modifications) as part of a single *logical update* in which the goal is to establish or remove connectivity between at least two endpoints (i.e., hosts).

2.1.2 Update consistency in the data plane. Although the SDN architecture is logically centralized, it is a distributed system consisting of SDN controller instances and switches. Distributed systems often aim to provide *consistency* a property dictating that all participants see the same view of operations and execute intended state changes in a well-defined order.

The *update consistency problem* in the data plane[3] arises from the ordering of flow rule updates [62]. If an SDN controller sends multiple flow rule updates at one time, some switches may receive and install the updates before others. That leads to an inconsistent (and insecure) forwarding state, since in-transit packets may be subject to some indeterminate mixture of the old and new rule set. Foerster et al. [25] describe consistency in terms of ensuring that there are no loops and no blackhole routing (connectivity consistency), that certain configured properties must remain satisfied at all times (policy consistency), and that no flows should be created that violate link capacities (congestion-aware consistency). Additionally, *per-packet* consistency [61] ensures that every packet flowing through the network should behave as though it is being routed before *any* updates occurred, or after *all* updates have occurred.

To ensure update consistency, McGeer [51] proposes two-phase commits, which trades off space (preserving scarce TCAM memory on switches) with communication (extra communication cost). Reitblatt et al. [62] propose installing new flow rules before removing old ones and tagging packets with information that directs them on which set of flow rules to follow. Liu et al. [48] propose a faster algorithm for consistent updates, but this comes at the cost of a relaxed model of consistency.

[3]A similar update consistency challenge arises in the control plane, but prior work (e.g., [17, 56, 64]) has addressed this challenge and we leave it out of scope. We refer the reader to Bannour et al. [14] for a survey on distributed SDN control planes.

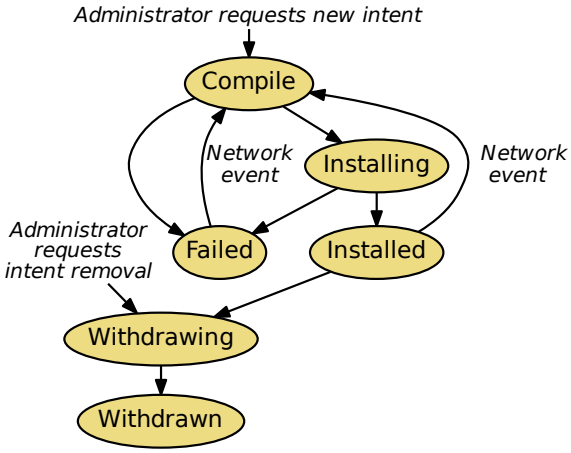


Figure 1: Intent compilation and installation of a single intent within a generic intent state machine model. Intents (and their low-level updates) are impacted by administrator events (e.g., requesting a new intent be installed, withdrawing an existing intent) or by network events (e.g., topology change).

2.2 Intent-based Networking (IBN)

Intent-based networking extends the programmability of SDN but aims to simplify network management through abstraction. Rather than focusing on "how" the network should implement some desired outcome or policy, IBN enables administrators to describe "what" they want the network to do through *intents*. Administrators can define high-level intents (e.g., "connect host A to B with minimum bandwidth X") without concerning themselves with implementation details in the data plane such as flow rules, network configuration protocols, or underlying topology.

IBN has been standardized by the Open Networking Foundation (ONF) [53], the Internet Engineering Task Force (IETF) [21, 45], and the 3rd Generation Partnership Project (3GPP) [5–7]. IBN shares a common model where a logically centralized *intent controller* (or *IBN controller*) coordinates and manages all intents based on an *intent state machine*. IBN has been implemented in open-source software such as the Open Network Operating System (ONOS) [54], OpenDaylight (ODL) [47], and the Open Network Automation Platform (ONAP) [46]. Proprietary IBN controllers have also been developed by Juniper [35], Cisco [20], Huawei [31], and IBM [32].

We use the ONOS IBN implementation as a running example throughout the rest of this paper because of its representative features and production-quality implementation.

When an IBN controller attempts to fulfill an administrator's intent, the intent is *compiled* from a high-level abstraction to low-level flow rules and *installed* into the network by the controller sending those flow rules to the network's switches. Figure 1 shows the intent compilation and installation within the intent state machine. An administrator requests an intent to be installed, which gets compiled into low-level flow rules. The compilation or installation can fail if the intent's requirements do not meet the network's current resources. Network events, such as topology changes, can impact failed intents (i.e., resources may become available allowing the intent to be installed) or impact installed intents (i.e., resources

may become unavailable preventing the intent from being fulfilled). An administrator can withdraw an intent when no longer needed.

3 CURRENT PRACTICES OF SECURE UPDATES IN IBN

Since IBN abstracts away many of the low-level implementation details in terms of flow rule compilation and installation, we now consider the security impacts of flow rule update consistency on the specification, design, and implementation of IBN.

We found a gap between the academic literature on programmable network updates and how they are implemented in IBN. We analyze the ONOS IBN implementation as an illustrative example of how update consistency is implemented, its shortcomings, and its security impacts. Based on the ONOS case study, we highlight several challenges in addressing these impacts, and we sketch our contributions to how we solve those challenges.

3.1 Case Study: ONOS IBN implementation

We examined ONOS v2.7.1 to understand how updates are implemented within the compilation and installation phases.

3.1.1 Compilation. Within the compilation stage, an administrator's intent is compiled into "middle intents" (i.e., intents that are not yet ready to be installed and need further refinement) and "installable intents" (i.e., intents that are ready to be installed). The compilation iteratively processes middle intents until they become installable intents. Installable intents are added to a queue to be installed. As a result, installable intents that are generated earlier during the compilation will be ordered before the installable intents that are generated later in the final queue.

Security implications: No ordering mechanism exists to ensure that intents are compiled in a particular order (e.g., FIFO). That prevents an administrator from being able to have fine-grained control over or assurances about potentially conflicting intents.

3.1.2 Installation with striping. Within the installation phase, for each switch, ONOS assembles a list of flow rule operations (i.e., the flow rule and whether it should be installed or deleted). Flow rule operations are separated into batches based on the switch to which they are being sent. ONOS implements "stripe keys" such that each batch can be assigned a key, and "[batches] with the same key will be executed sequentially. Operations [batches] without a key or with different keys might be executed in parallel. This parameter is useful to correlate different operations [batches] with potentially conflicting writes" [4].

We found that ONOS does correctly implement the in-order sequential execution of flow rule operation batches with the same stripe key—but only if a stripe key is assigned.[4] If utilized, the stripe key mechanism is an effective way to enforce a certain order when *sending* flow rule operation batches to the switches, but it does not enforce any ordering on the *receiving* and processing of those batches among the switches.

[4]Specifically, ONOS selects the thread that will send the batch based on the batch's stripe key, where the thread index equals to the stripe key mod the number of threads (i.e. the remainder of dividing the stripe key with the thread count). This ensures the batches with the same stripe key will be sent out sequentially on the same thread, even if the batches are intended for different switches.

Critically, we also found that when the batches are assembled, *no stripe keys are actually assigned.* In other words, the stripe key mechanism is implemented and available in the code, but not used when installing or deleting flow rules via the IBN subsystem.

Security implications: Although flow ordering mechanisms exist in ONOS, they are not used by the IBN subsystem. Furthermore, even if they were used, there is no assurance that the flows will be applied in a consistent order once they are updated on the switches. That lack of assurance can cause indeterminate behavior through race conditions and allow unauthorized data plane connectivity.

3.1.3 Installation with barrier messages. ONOS implements message ordering through "barriers" that separate groups of requests that should be processed. The controller sends an initial group of requests, followed by a barrier request, followed by a second group of requests. The switch may reorder any requests within the initial group but must complete all requests and send a barrier reply to the controller before attempting to process the second group. We found that ONOS places a barrier request between each batch of flow rule operations sent to a single switch. However, barriers are not used to enforce ordering for batches sent to *multiple* switches.

Security implications: Although single-switch updates can be ordered, multi-switch updates (which are common for intents) are not ordered. The lack of ordering could cause indeterminate behavior and introduce security-critical race condition vulnerabilities.

3.1.4 Automated recompilation and redeployment. A network event can cause intents in ONOS to be recompiled. Network events could include changes to topology, workload, or security administration. All intents are recompiled after a network event, but flow rule updates will only be distributed if they differ from the flow rules before the event. A network event does not imply that any flow rule updates will occur, but flow rule updates do imply that either an administrator made an intentional policy change at the controller, or a network event occurred.

Security implications: Intents provide convenience to ostensibly ease the responsibilities of network and security administrators. However, they do so by offloading some responsibility of flow rule management to network event detection modules. These network events can be manipulated to intentionally fool the controller into performing an intent recompilation and flow rule redeployment in a way that would be much more difficult if updates relied on a human administrator.

3.2 Challenges to Secure Updates

Based on the insights from our investigation into the ONOS IBN implementation, we consider several fundamental challenges about why inconsistent updates pose security risks in IBN and what the current obstacles are in defending against them.

3.2.1 Challenge 1: Inconsistent updates cause exploitable network conditions. Although data plane update consistency properties and guarantees have been proposed in academic literature [16, 25, 44, 61, 62, 66, 78], we discovered that major open-source SDN and IBN controllers like ONOS do not explicitly verify or enforce any notion of consistent ordering in traditional SDN, much less IBN.

In contrast to relatively static traditional networks, SDN and IBN enable more dynamic changes that theoretically allow for frequent network reconfiguration. Prior work has shown how such frequent

reconfiguration can lead to denial of service impacts [28]. As a result, attackers could leverage the inconsistencies in frequent updates to gain unauthorized communication access within the data plane.

Our contributions: We demonstrate the efficacy of an attacker exploiting such inconsistencies by proposing three temporal attacks (Section 4.2) and devising a working exploit in the ONOS IBN implementation (Section 4.3) to demonstrate the impact of such attacks on overall network security.

3.2.2 Challenge 2: Does IBN promise consistent updates? (If not, should it?) Given that IBN implementations are susceptible to attacks that leverage update inconsistencies, does the root cause come from the implementation, design, or specification? We surmise that the ONOS IBN implementation does not provide consistent updates in the data plane because there is no well-defined specification that says that it ought to provide such properties, nor do code or network traces indicate the presence of a consistent update mechanism.

Unfortunately, the existing IBN standards [5–7, 21, 45] do not specify update consistency guarantees or how updates ought to be implemented. As a result, developers may design or implement update consistency differently. If not properly specified, administrators could incorrectly assume that their IBN implementation implements update consistency securely when it may not.

Our contributions: We propose a set of invariant conditions (Section 5.2) that can be used to verify when inconsistent updates within IBN cause security-critical impacts on the network.

3.2.3 Challenge 3: Efficiently detecting security-impacting updates. Even if the right conditions can be detected, the state space of all possible updates (i.e., the set of all possible flow rules for all possible data plane paths that enable connectivity for an intent) becomes difficult to manage. However, not all updates will cause security-critical vulnerabilities to manifest. Even if consistent updates were implemented, such updates could impose a performance penalty on updates that do not cause vulnerabilities.

Our contributions: We propose a defense, `Spotlight`, and design a fast detection algorithm (Section 5.5) that can reduce the searchable state space for only security-relevant consistent updates.

4 TEMPORAL ATTACKS IN IBN

We describe how an adversary can execute temporal attacks in IBN by exploiting inconsistencies in intent installation and in the remote connection between the controller and switches. We first describe the attacker model including objective and capability. Then we introduce different types of temporal attacks and demonstrate a concrete temporal attack, Phantom Link. Finally, we reproduce the Phantom Link on the emulated network and show that by exploiting this attack, an attacker can construct unauthorized connectivity between a victim and a malicious server.

4.1 Threat Model

Network policy with intents. We refer to network devices comprising hosts, switches, and controllers. Hosts are end devices and may be workstations, servers, databases, or any device with which a user might want to interact. Switches are forwarding devices that receive network packets and forward them according to installed flow rules. Controllers are the control plane devices that decide which flow rules should be installed on which switches.

We assume that a network's security policy is defined and enforced using intents, for example as in NetViews [12, 13]. All intents are assumed to be correctly defined by the administrator, which is to say that the steady-state final connectivity after all updates are made is the intended configuration.

Temporal attacks. Given the system description above, we assume a default-deny model: a host should never have connectivity to a host to which it is not explicitly granted access. When a host does become connected to an unauthorized host that it was not connected to before the update started and will not be connected to after the update completes, we refer to this as a *temporal vulnerability*. If such connectivity is the result of a malicious action we refer to it as a *temporal attack*. We also refer to the resulting additional connectivity as *unauthorized connectivity*.

Attacker model. We consider an IBN network controlled by an IBN controller, which may be built upon an SDN controller and act as a subsystem within such an SDN controller [24, 47, 54]. We assume hosts in the data plane may be malicious but the controller and switches are not. An adversary may also have the ability to influence intent installation, but cannot install arbitrary intents directly. We assume that a malicious host has a goal of connecting to one or more hosts that are disallowed by the security policy. Even short-lived temporal vulnerabilities are considered a successful attack in the adversary's eyes if the connection is long enough to deliver a malicious payload. Additionally, we assume the attacker can hide its abnormal behavior from detection by using multiple machines and can retry the attack multiple times if prior attempts fail.

Table 1 shows a toolkit of possible attack capabilities leveraged by the attacker based on existing known attack methods. The attacker can learn the network's flow rules by using SDNMap [10] and infer intents from the learned rules. To saturate a target link, the attacker can find a virtual network topology by learning a "layer-3 link map" through `traceroute` [36]. In the case of SDN with in-band control, the attacker can discover "a shared link" used for both the control and the data plane and attack a target switch [18]. By fingerprinting SDN applications, the attacker may be able to execute the Switch Delay attack (Section 4.2.3), which depends on a specific application (e.g., ARP proxy in ONOS [54] and ODL [47]).

4.2 Temporal Attacks

We show how temporal vulnerabilities can occur not only when installing an intent that connects two endpoints[5] (e.g., two hosts), but also while modifying or recompiling an intent, in which case the controller may induce temporal vulnerabilities.

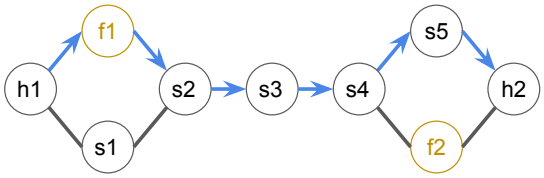
As described earlier in Section 3.2, the inconsistency of flow rule installation allows temporal vulnerabilities in IBN. Because the intents are installed in a random and indeterminate order, unexpected behavior can happen in terms of network connectivity. Such behavior can be characterized as:

- Lack of connectivity: Examples include blackholes where packets are dropped, or loops where packets are forwarded without ever reaching their destination. The outcome is a denial of service of a victim(s), or the entire network.

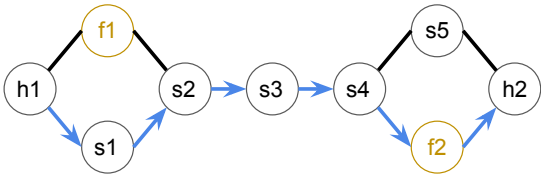
[5]Unless specified, we assume an intent is constrained by its endpoints.

Table 1: Existing attack tools, tools' capabilities, and tools' use in temporal attacks.

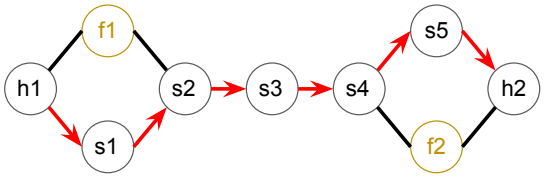
Tools	Capabilities	Use
SDNMap [10]	Fingerprinting flow rules	Intent speculation
Crossfire [36], Crosspath [18]	Disrupting a link or a switch	Intent recompilation (Section 4.2.2)
Cao et al. [19]	Fingerprinting SDN apps	Switch Delay attack (Section 4.2.3)



(a) The initial path compiled from the waypoint intent.



(b) The final path compiled from the waypoint intent.



(c) An intermediate path that bypasses both firewalls.

Figure 2: An indeterminate ordering installation of flow rules can cause a modification to a waypoint intent to suffer from temporarily bypassing the waypoints, which violates the intended security policy.

- Additional connectivity: The random order of installation can also result in additional connectivity between devices in the network that was not intended by the network security policy. In this case, the outcome is unauthorized access between two or more hosts.

We note that just because the controller does not enforce a certain order of the flow rules installation does not necessarily mean that unauthorized access is allowed. An attacker can either discover such unauthorized access by monitoring the installation of intents, or can enforce a certain order by delaying flow installation at certain switches, or by forcing the controller to create them by triggering the recompilation of intents.

We describe these methods in more detail:

4.2.1 Discovering unauthorized connectivity by network monitoring. A controller can modify an existing intent to change any endpoint of the intent. When two ends of the intent have been changed, the controller may induce unauthorized connectivity. For example, suppose an intent I allows a connection from $h_a \rightarrow h_b$ and the controller modifies I to allow a connection from $h_c \rightarrow h_d$, which shares part of its path with $h_a \rightarrow h_b$. While updating the intent I, if the switch that connects h_b and h_d is updated first, it will allow $h_a \rightarrow h_d$. If the switch that connects h_a and h_c is updated first, it will allow $h_c \rightarrow h_b$. As a result, there are two short-lived indeterminate connectivity possibilities that enable unauthorized access: $h_a \rightarrow h_d$ or $h_c \rightarrow h_b$.

In addition, the controller can also update the waypoint of a waypoint intent[6]. Figure 2 shows an example where the victim network has initially deployed a waypoint intent that requires the traffic between hosts $h1$ and $h2$ to go through the firewall $f1$. The assumed path is $h1-f1-s2-s3-s4-s5-h2$, as shown in Figure 2a. Some time later, the network operator modifies this intent so that the traffic between the two hosts is required to go through the firewall $f2$. The eventual new path is $h1-s1-s2-s3-s4-f2-h2$, as shown in Figure 2b.

Attack mechanism: If the flow rule updates for $s4$ and $s5$ are significantly delayed, then there will be an intermediate path where the traffic goes through $h1-s1-s2-s3-s4-s5-h2$, which bypasses both firewalls, as shown in Figure 2c. Although this simple topology may not be exactly the same in real deployed topologies, such bypasses can occur as long as the simple topology is a subgraph within a realistic topology, e.g. if $s2$, $s3$ and $s4$ are connected to other switches and hosts.

Attack implications: While intent modification can allow temporal attacks, an attacker has to meet strong requirements in order for this attack to succeed. The attacker needs to have the capability to monitor intent updates in the control plane to calculate the difference between the old set and the new set of intents.

4.2.2 Triggering unauthorized connectivity by intent recompilation. If an intent can be implemented on one of multiple paths, the controller can recompile the intent when the established path is no longer valid. Figure 3 shows an example of temporal connectivity when resolving a down link. We assume that the controller has two connectivity intents. The intent with a priority of 200 (I_A) sends packets destined to an IP subnet 10.0.10.0/24 from H_1 to H_3. The intent with a priority of 100 (I_B) forwards packets destined to an IP subnet 10.0.0.0/8 from H_2 to H_4. If the controller finds the shortest path for each intent, the path of I_A will be $H_1 - S_1 - S_2 - H_3$ and the path of I_B will be $H_2 - S_1 - S_3 - H_4$.

Attack mechanism: If the link $S_1 - S_2$ is disconnected, the controller will receive a link down event and recompile the intent I_A to avoid the failed link, $S_1 - S_2$ ❶. Since another valid path for I_A exists (i.e. $H_1 - S_1 - S_3 - S_2 - H_3$), the controller can still implement

[6]A *waypoint intent* adds additional constraints to an intent by specifying intermediate nodes (i.e., waypoints) in the network that the end-to-end path must traverse. Such waypoint intents can enforce network security policies such as "all traffic between A and B must go through a firewall".

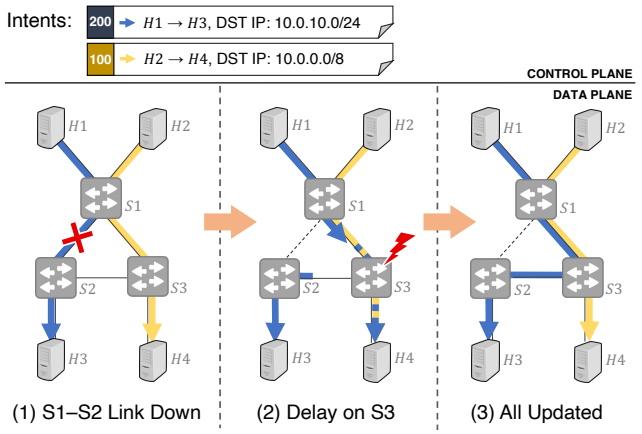


Figure 3: Execution of a temporal attack while handling a down link.

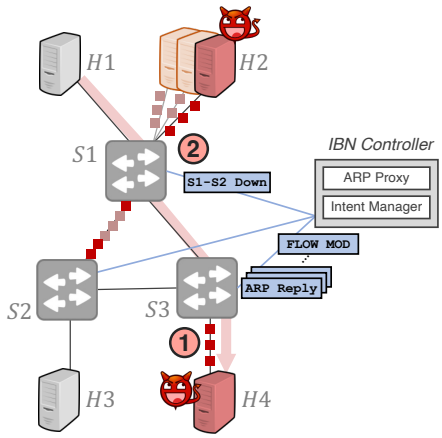


Figure 4: Execution stages of the Phantom Link attack from Figure 3. The attacker located in H_2 and H_4 executes steps 1 and 2 to allow unauthorized access.

I_A by removing outdated rules from the switches in the previous path and adding new rules to the switches in the new path. If the flow rule updates for S_3 are delayed, S_3 has outdated rules while S_1 and S_2 have new rules ❷. Since S_3 forwards packets destined to an IP subnet 10.0.0.0/8 to H_4 because of I_B, packets from H_1 are also delivered to H_4, not H_3, until S_3 handles delayed updates from the controller ❸.

Attack implications: Unlike the intent modification, attackers residing in a remote host can provoke intent recompilation at will. To disconnect a link or a switch, the attacker can saturate the target link [18, 36] or the control channel between the controller and the target switch [65, 75]. However, the random order of flow rule installation from intent recompilation does not always guarantee a successful temporal attack that would allow unauthorized access.

4.2.3 Constructing unauthorized connectivity by delaying installation of intents. During intent updates, a remote host attacker can execute the temporal attack by choosing the target switch to delay during rule installation. We refer to this as the Switch Delay attack.

Attack mechanism: To delay rule installation, the attacker can attempt to perform denial of service to one of the three parties involved in the rule installation: the controller, the control channel, and/or the switch.

First, attacking the controller requires flooding packets sent from many servers like in a botnets, considering that the controllers show high performance with thread scalability [22]. Although such DDoS attacks could succeed, they would paralyze the whole network.

Second, the attacker could saturate the control channel between the controller and the target switch. The SDN controller can receive packets of interest from a switch (e.g., `PACKET_IN` messages in OpenFlow[7]). To check link status, the SDN controller listens for link-layer discovery protocol (LLDP) packets. In our setup, by flooding 1300-byte LLDP packets with `tcpreplay`, the remote attacker can achieve 1400 Mbps bandwidth, which can saturate the control-channel uplink from the switch[8]. However, this does not affect the delay of the flow-rule update message, since the control-channel downlink to the switch is not affected. Unless the controller saturates the control-channel downlink by itself, the attacker in the data plane cannot delay the rule installation.

Finally, the adversary attacks the target switch. If the target switch crashes, the controller simply removes the erroneous switch and supports existing intents on the remaining switches. Instead, the attacker can overload the slow path of the target switch. The SDN controller listens to ARP packets to monitor hosts, and also to send ARP reply packets on behalf of the destination host (e.g., `PACKET_OUT` messages in OpenFlow). A remote attacker on a host connected to the target switch can flood ARP request packets. Due to the small size of ARP packets, the bandwidth of ARP flooding is less than 40 Mbps, much lower than the bandwidth of the control channel. The controller can handle such low-bandwidth `PACKET_IN` messages and send back corresponding `PACKET_OUT` messages to the target switch. However, the switch becomes slow in handling control messages while processing flooded `PACKET_OUT` messages.

An attacker does not necessarily need to gather information to flood ARP. An attacker can flood crafted ARP request packets by setting one machine as the destination and another as the source without needing the victim's information. Since the IBN controller will respond with ARP reply packets for these request packets, the path between the target switch and the controller becomes saturated, which is within the IBN threat model. Tools like SDNMap increase the success likelihood of our exploit but are not strictly required, and if the attack fails due to insufficient knowledge, the attacker can always try again.

Attack implications: The attacker can delay the flow rule installation request (e.g., `FLOW_MOD` in OpenFlow) by flooding ARP request packets.

[7]We use OpenFlow [52] and its message types as a representative example of the data plane configuration protocol, but the attack can generalize to other data plane configuration protocols (e.g., P4Runtime [55]).

[8]According to a specification of the Pica8 40GbE OpenFlow hardware switch [57], the bandwidth of the control channel is 1 Gbps.

4.3 Executing Phantom Link

Among temporal attacks described in Section 4.2, we introduce Phantom Link, which allows the attacker to exploit temporal attacks at the discretion of the attacker.

Figure 4 shows the process of executing the Phantom Link attack. First, the attacker on H_4 executes the Switch Delay attack by flooding ARP request packets ❶. That forces S_3 to handle PACKET_OUT messages that include ARP reply packets sent from the ARP proxy in the controller. Then, the attacker in H_2 floods packets to saturate the target link ❷. That cuts off the $S_1 - S_2$ link, and the controller will receive a link-down message from S_1. Since I_A uses the disconnected link, the intent manager in the controller will update the flow rules of I_A to bypass the failed link. However, while processing flooded ARP reply messages, S_3 delays the FLOW_MOD message sent from the controller. Therefore, the attacker can receive packets from H_1 until the flow rules have been updated in S_3.

4.3.1 Reproducing Phantom Link. To demonstrate that the attacks are not just theoretical, we reproduced the Phantom Link attack. We ran two virtual machines in Google Cloud Platform: 4 vCPUs, 32 GB memory, and 120 GB balanced persistent disk. On one VM, we ran ONOS v2.7.1 as an IBN controller (controller VM). On the other VM, we executed network emulation based on Mininet v2.3.0 with Open vSwitch v2.14.0 (switch VM) to emulate the scenario described in Figure 3. Each emulated switch connects to the controller through OpenFlow 1.3 over TCP. These two virtual machines are connected to the same network, which shows nearly 8 Gbits per second. We limited the bandwidth of each management connection to 1 Gbps by referring to the specification of the OpenFlow hardware switch [57].

To disconnect the target link, the attacker can execute the link-flooding attack [36] by flooding packets that traverse the target link. We emulated this attack by simply disabling an emulated interface of the target link. We executed 10 times to measure the average.

We flooded ARP request packets from H_4 to delay S_3. The bandwidth of ARP showed 40 Mbps, which is much lower than the 1Gbps bandwidth of the control channel. After a few seconds for the target switch to process control messages, we disconnected the $S_1 - S_2$ link. ONOS recompiled the I_A intent and sent the FLOW_MOD and the BARRIER_REQ messages to S_1, S_2, and S_3. The time difference between these messages was less than 10 ms, while the order of messages was random. However, the target switch (S_3) sent the corresponding BARRIER_REPLY message to ONOS after 1304.5 ms, compared to 157.9 ms taken in the remaining switches.

Thus, S_3 allowed temporal connectivity from when normal switches responded to when the target switch responded, or 1146.6 ms.

4.3.2 Security implications. An attacker can use the Phantom Link attack as a stepping stone in a larger attack campaign where a tactical goal is to gain unauthorized access within a network. The attacker's ultimate goal may be to send malicious payloads to a victim host (e.g., malware campaign) or to learn about the network's configuration or the traffic being sent across it.

5 Spotlight: TEMPORAL VULNERABILITY DETECTION FOR IBN

We provide some insights into the causes of temporal vulnerabilities, and propose Spotlight—a defense module that can successfully mitigate our Phantom Link attack. We describe both the design and proof-of-concept realization of Spotlight.

5.1 System Model and Goals

Our model considers two roles, each of whom will be interacting with the system in different ways and under different constraints. The first role is that of *policy administrators* who are charged with implementing network connectivity policies. The policy administrator enforces whatever network policy has been selected. In the context of IBN, this means that the policy administrator can add, remove, or modify intents, as well as perform other network actions necessary to adhere to the requirements. We assume that policy administrators are not malicious. The second role is that of *users*. They are human actors or software programs at any of the network hosts endpoints. Users are blind to the constitution of the network fabric; their only visibility into which is through side-channel inferences based on network operations such as initiating (or receiving) connections or sending data. Additionally, our model assumes that the dominant cause of delay in flow rule updates is link latency, so we treat all updates on a single switch as atomic.

System goals. We aim to design a detector system that can alert policy administrators if the intent being added may be a *high-risk update*. We define high-risk updates to be the modification of intents such that, if switch flow rules are not updated consistently, a temporal vulnerability will occur. Spotlight returns an alert to the policy administrator that the submitted intent has a temporal vulnerability. Alternatively, it can return all possible orders that flow rule updates could be installed in that would yield a temporal vulnerability.

5.2 Causes of Temporal Vulnerabilities

In Section 4.2, we presented several examples of temporal vulnerabilities. In this section, we characterize similarities between them, and identify insights about when a temporal vulnerability may occur. We use these insights to guide our solution, Spotlight.

For completeness, we define three terms: *starting state* $\mathcal{S}_0$, *action* $\mathcal{A}$, and *finishing state* $\mathcal{S}_f$. The starting state $\mathcal{S}_0$ is a stable state of the IBN; all intents have been compiled and all pending flow rules have been installed on their respective switches. The finishing state $\mathcal{S}_f$ is the quiescent stable state achieved by the system after the action $\mathcal{A}$ has been executed on the starting state $\mathcal{S}_0$. The action $\mathcal{A}$ is the addition, removal, of modification or intents, as well as any changes to the topology; it represents a transition from $\mathcal{S}_0 \rightarrow \mathcal{S}_f$.

We represent an intent being added, removed, or modified as $i \in I$, where $i \in \mathcal{S}_0 \vee i \in \mathcal{S}_f$. The source and destination of a connection as specified by an intent are represented as $i.src$, $i.dest \forall i \in I$. An intent that was added to $\mathcal{S}_f$ is represented as i, and an intent removed from $\mathcal{S}_0$ is represented as i'. We define a function $\mathcal{D}$ executed by the controller that returns the set of switches that need to be updated with new flow rules to satisfy a connectivity intent. Each intent has a boolean pattern matching function written $i.pattern \ \forall i \in I$, which returns whether or not a particular packet source or destination will match the flow rules comprising the intent. Using this terminology, we list below the requirements for extra connectivity:

(1) At least one intent must have already been installed in the starting state

$$|\mathcal{S}_0| > 0$$

(2) The action must include at least one intent being removed and one being installed in a single update batch—this could also be the modification of a single existing intent

$$|\mathcal{S}| > 0, \mathcal{S} = \mathcal{S}_f - \mathcal{S}_0$$

(3) There must be partially disjoint paths between the original intent and new intent that is being installed

$$0 < |\mathcal{D}(i_1) \cap \mathcal{D}(i_2)| < |\mathcal{D}(i_1) \cup \mathcal{D}(i_2)|$$

(4) The two intents—original and new—must differ in at least one endpoint or waypoint

$$i.src \neq i'.src \vee i.dest \neq i'.dest \vee i.way \neq i'.way \quad \forall i \in \mathcal{S}_0, \forall i' \in \mathcal{S}_f$$

(5) The packet filter rules must match the source and destination of both the original and new intents for every switch on the path. In other words, a temporal vulnerability can only occur for packets that would be affected by the intent update

$$i.pattern(h_1) \wedge i.pattern(h_2) \wedge i'.pattern(h_1) \wedge i'.pattern(h_2)$$
$$\exists h_1 \in H, h_2 \in H, h_1 \neq h_2$$

5.3 Spotlight Architecture

Our detection module sits logically after the IBN controller compilation stage. The workflow, as indicated in Figure 5, begins with a policy administrator either modifying an existing intent, or removing an existing intent and then adding a new one ❶. This is a policy change in which network connectivity is altered between at least two hosts. The compilation of the intent into flow rules is unmodified, but after compilation is complete, flow rules are sent out to the Spotlight detection module instead of being propagated to the switches over the network ❷.

The detection module processes the flow rules as described in Section 5.4 and Algorithm 2. If a high-risk update is detected, the compiled flow rules are cached and details of the vulnerability are reported to the policy administrator who can decide to pursue safer alternatives ❸. We have shown with our attack in Section 4 that updates that have not been fully vetted can entail security risks. In response, it makes sense to pre-approve updates to ensure that they are safe. If Spotlight does not find a temporal vulnerability, the update is considered safe and is forwarded on to the controller ❹, which then distributes the flow rules as usual ❺.

Administrators have several options for what to do with detected temporal vulnerabilities. We discuss those options and trade-offs in detail in Section 7.

The Spotlight architecture is designed as a standalone module and is agnostic to the underlying controller implementation. All that is required for adapting Spotlight to a different controller is to import the currently installed flow rules as well as the flow rules to be installed using Spotlight's program interface.

5.4 Strawman Detection Algorithm

We first present a strawman detection algorithm that when given as inputs (1) the flow rules presently installed on the switches in the network, and (2) the flow rules changes resulting from the IBN controller's compilation of the new intent, performs an exhaustive search of the state space for possible temporal violations.

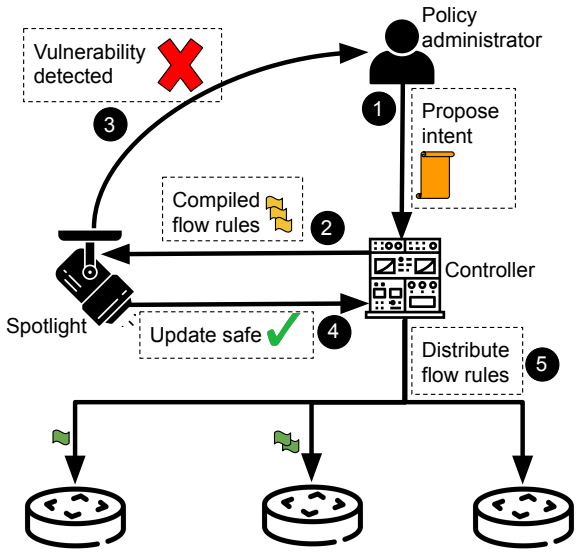


Figure 5: Spotlight architecture showing typical workflow of an administrator proposing an intent to be compiled and installed into the IBN-based network.

Algorithm 1 Strawman detection algorithm

1: **Input:** F_0, F_Δ, H, S
2: **Output:** $true \cup false$
3: $G_0 \leftarrow initGraph(H, S, F_0)$
4: $G' \leftarrow initGraph(H, S, F_0 + F_\Delta)$
5: **for** $S' \subset S$ **do**
6: $\quad G_u \leftarrow initGraph(H, S, F_0 + flowrules(S'))$
7: $\quad$ **for** $h_1 \in H, h_2 \in H, h_1 \neq h_2$ **do**
8: $\quad\quad p \leftarrow path(G_u, h_1, h_2)$
9: $\quad\quad$ **if** $\mathbb{1}(p) = \mathbb{1}(path(G_0, h_1, h_2))$ **then**
10: $\quad\quad\quad$ next host pair // hosts were connected before update
11: $\quad\quad$ **else if** $\mathbb{1}(p) = \mathbb{1}(path(G', h_1, h_2))$ **then**
12: $\quad\quad\quad$ next host pair // hosts will be connected after update
13: $\quad\quad$ **else if** $match(h_1, h_2, flowrules(s)), \forall s \in p$ **then**
14: $\quad\quad\quad$ **return** $true$ // temporal vulnerability detected
return $false$

We show the pseudocode in Algorithm 1, where F_0 and F_Δ represent the initial set of flow rules in the system, and the set of changes to the flow rules, respectively, H represents the set of hosts and (S) represents the set of switches. A flow rule is a tuple of an ingress port I, an egress port E, and a matching pattern M. We assume the physical topology is static, so all egress ports are connected to exactly one ingress port on another switch or host. This allows us to build a graph from the flow rules.

The procedure starts by creating a directed graph $G_0 = (H+S, F_0)$ where the vertices are the hosts and switches, and the edges are the set of flow rules before the update F_0 (line 3). Next, we create a

graph $G' = (H + S, F_0 + F_\Delta)$ where the vertices are the same, but the edges are the set of flow rules after the update ($F_0 + F_\Delta$) (line 4).

Following the graph generation, we need to generate the exhaustive set of unique update orderings. As discussed in Section 5.1, we assume that the updates on a single switch are atomic. Therefore, we only need to consider the unique subsets of switches $S' \subset S$, which, when updated before the other switches, result in temporal vulnerabilities (line 5). Note that this is a strict subset, because if the entire set S is updated then the update is complete and any connectivity is either intentional or a mistake by the administrator, which is out of scope for our system. In practice, when generating the unique subsets, we restrict S to only switches that will receive flow rule updates for the new intent.

On line 6, we build a new graph with the initial flow rules plus any changes to the flow rules in the update of switch S'. We then use the three graphs we have built to test connectivity between all pairs of hosts. If a host pair is connected in graph G_u but not in G_0 or G', then extra connectivity may be possible. We test connectivity by running a shortest path algorithm between each host pair on all three graphs, which we indicate in Algorithm 1 lines 8, 9, and 11 with the function $path()$. As flow rules define directed edges, we must test the connectivity in both directions.

For any host pairs for which we deem a temporal vulnerability to be possible, we need to make sure that there exist packets that will not be filtered out by the flow rules' pattern matching field. We show this part of the algorithm on lines 13–15 using the $match()$ function which returns $true$ if h_1 and h_2 match the pattern in at least one flow rule on switch s in $flowrules(s)$. Concretely, we iterate through the switches on the connected path and check if the source and destination hosts on the path match the patterns of all the new flow rules on the path.

5.5 Spotlight Detection Algorithm

The strawman approach described above is guaranteed to find all temporal vulnerabilities that result from an update. However, this comes at a significant computational cost.

One approach to improve the strawman algorithm is to use *link prediction* algorithms for graphs. These schemes rely on calculating some *similarity* score for some subset of node pairs, and then estimating the likelihood of a connection based on the pairs' score [50]. Our insight is that in representing the topology as a graph with the hosts H and switches S as nodes, the similarity between a switch $s \in S$ and a host $h \in H$ is related to the likelihood that the host h is impacted by a flow rule change on the switch s.

While there are many algorithms calculating such similarity scores, we selected Panther [77] because Panther can quickly calculate similarity for nodes in a topology T, which includes all hosts H, switches S, and their physical connectivity. Panther works by calculating random paths of length p from a node v, and returning the proportion of paths that include each of the top k most traversed nodes. These proportions are the similarity scores for each node in the top k. The similarity score computed by Panther is directly relevant to our use case—when a switch frequently shares a randomly generated path with a host, we might reasonably expect changes to the switch flow rules to impact the connectivity of frequently connected hosts.

Algorithm 2 `Spotlight` fast detection with similarity cache

```
1:  Input: F_0, F_Δ, H, S
2:  Output: true ∪ false
3:  G_0 ← initGraph(H, S, F_0)
4:  G' ← initGraph(H, S, F_0 + F_Δ)
5:  for S' ⊂ S do
6:      G_u ← initGraph(H, S, F_0 + flowrules(S'))
7:      for h_1 ∈ H, h_2 ∈ H, h_1 ≠ h_2 do
8:          if h_1 ∉ P ∧ h_2 ∉ P[s]   ∀s ∈ S' then
9:              next host pair   // neither host is similar to switches in S'
10:         p ← path(G_u, h_1, h_2)
11:         if 𝟙(p) = 𝟙(path(G_0, h_1, h_2)) then
12:             next host pair   // hosts were connected before update
13:         else if 𝟙(p) = 𝟙(path(G', h_1, h_2)) then
14:             next host pair   // hosts will be connected after update
15:         else if match(h_1, h_2, flowrules(s)), ∀s ∈ p then
16:             return true   // temporal vulnerability detected
    return false
```

For our system, `Spotlight`, we use Panther to calculate a *similarity cache* which includes the top k most similar nodes to each switch. If we assume that the physical network topology is unlikely to change often, we can precompute the similarity cache a single time. We provide an additional argument P to our fast algorithm, which represents the similarity cache.

The difference between the strawman and the fast algorithm in Algorithm 2 is that we use the preprocessed similarity cache P to inform us about which shortest path calculations can be skipped. *This significantly reduces the state space that needs to be searched.*

While the fast algorithm scales much better than the strawman approach, because it is based on heuristics instead of exhaustive search, it does not guarantee that it will always find a temporal vulnerability if it exists. We show in Section 6.2 that the algorithm is quite successful at detecting the vulnerabilities in practice.

5.6 Implementation

We implemented `Spotlight` as a Python3 program in 966 lines of code. We use Python generators to enumerate the unique update orderings S one at a time, so that the potentially large set of orderings to check never resides in memory in its entirety.

As described in Algorithm 1 and Algorithm 2, instead of modeling the topology graph based on the physical connectivity, we opt to create edges between nodes that have flow rules connecting them. This is because we rely on the assumption that the IBN controller's compilation stage is correct, and thus it cannot output flow rules that forward packets on non-existent links. For calculating shortest paths on these graphs, we use the `shortest_path` function included in the NetworkX [27] library, which uses Dijkstra's Algorithm [23]. We cache the results of the shortest path calculations we perform on the initial graph G_0 and the final graph G' for all pairs of hosts, which we repeatedly use in lines 9 and 11 of Algorithm 1. We note that our algorithm can be parallelized for additional execution speedup.

We perform the preprocessing step using the `panther_similarity` function included with NetworkX. We perform this preprocessing

step only once. In practice, an administrator would want to run this step during topology changes, which we expect to be less frequent than intent changes.

6 EVALUATION

We evaluate Spotlight on performance and its ability to detect temporal vulnerabilities. We aim to answer the following questions:

- **Q1** How does Spotlight's performance compare to a baseline exhaustive search algorithm?
- **Q2** How does Spotlight scale with the topology size?
- **Q3** How likely is Spotlight to find a temporal vulnerability if one exists?
- **Q4** Can we increase detection likelihood with proper tuning?

6.1 Experimental Setup

For evaluating the performance of Spotlight, we must both consider the topologies and the intents installed on the system we are measuring. We execute our experiments on a machine running Ubuntu 22.04. Our machine has an Intel Xeon Silver 4114 2.20GHz CPU with forty cores and 187GB of memory. We run each of the configurations discussed below ten times.

6.1.1 Fat tree topologies. Variations in network topology will impact the time it takes to detect temporal vulnerabilities. However, topology size is independent of the pattern of connectivity within the network. For this work, we perform our evaluations using the fat tree topology as described by Al-Fares et al. [11] because its short paths and redundant links make it representative of data center networks (e.g., Google [67]) and because its single parameterized input f_k allows us to generate topologies of increasing size while maintaining some degree of similarity in connectivity patterns.

The fat tree topology is a tree architecture with four layers. The top layer is called the core, below which is the aggregate layer followed by the edge layer. These three layers all consist of switches. The aggregate and edge switches are organized into f_k pods each containing f_k switches. We define the pods notationally as $p_i \in \{p_0, p_1, \ldots, p_{f_k-1}\}$. Likewise, we define edge switches of pod p_i as $p_i.e_j \in \{e_0, e_1, \ldots, e_{m-1}\}$, where m is the number of edge switches in each pod. The fourth layer, connected to the southbound interface of the edge switches contains the hosts. We index hosts using a dot notation of the form $p_i.e_j.h_n \in \{h_0, h_1, \ldots, h_{r-1}\}$, where r is the number of hosts in each pod. There are no direct connections between pods—pods can only communicate through core switches.

We wrote a Python script to generate all of our topologies using the NetworkX library [27]. These fat tree topologies vary in size from $f_k = 4$ to $f_k = 16$ (36 to 1,344 total nodes). See Table 2 for more details.

6.1.2 Stanford backbone topology. While the fat tree topology is popular, we consider other topologies for a broader evaluation. We tested Spotlight on the Stanford backbone topology, a common topology used in networking research [12, 13, 38, 76] due to its large size[9]. The topology consists of two core switches, ten aggregate switches, and fourteen edge switches. These are arranged in a tree with the core switches at the root, the aggregate switches connected as children to the core switches, and the edge switches as children to the aggregate switches. We honor precisely the reported connectivity between switches.

[9]The Stanford network serves at least 17,000 students and faculty [38].

Table 2: Topologies used for evaluation and probabilities of finding a temporal vulnerability along with preprocessing times for all the experiments.

Type	f_k	Switches	Hosts	Prob.	Preprocessing (s)
Fat tree	4	20	16	0.98	0.399
Fat tree	7	58	85	0.86	4.062
Fat tree	10	125	250	0.86	49.09
Fat tree	13	205	549	0.88	694.17
Fat tree	16	320	1,024	0.78	644.23
Stanford	-	25	26	1.00	0.219
Cisco	-	8	12	1.00	0.0442

6.1.3 Cisco topology. We also consider a topology provided by Cisco [12, 13, 76]. Unlike the fat tree and Stanford backbone topologies, which are intended to support large networks, the Cisco topology is designed to act as a distributed firewall. We chose this topology because it is one of the few known enterprise topologies, and its structural differences from the Stanford and fat tree topologies test the versatility of Spotlight. The Cisco topology is the smallest of the three with twelve hosts and eight switches. However, while the Stanford and fat tree topologies are tree-based, Cisco is not, which allows us to evaluate Spotlight in a less hierarchical network.

6.1.4 Testing intents. To make meaningful comparisons among topologies, we need to model networks that have similar sets of intents installed. We model only host-to-host intents such as those available on ONOS, which simply create connections between any pair of hosts. We choose to measure the upper bound of detection time. As such, we choose initial and modified intents that initiate connections with the number of hops equal to the graph diameter. For our fat tree graphs, we can follow the same simple procedure for graphs of all sizes. For the initial intent, we choose the source host $p_0.e_0.h_0$ and a destination host $p_{f_k-1}.e_{m-1}.h_{r-1}$. For the updated intent, we use the same source host but replace the destination host with $p_{f_k-1}.e_0.h_0$. For the Stanford backbone and Cisco topologies, we also use intents that span the entire graph diameter.

6.1.5 Metrics. The metrics we measure are:

(1) the time to detect the first temporal vulnerability,
(2) the number of temporal vulnerabilities detected per run, and
(3) the preprocessing costs required by Spotlight.

The time to detect the first temporal vulnerability does not include preprocessing time, which only needs to be done once for a topology. The clock starts before the state space search begins, and stops after connectivity is verified between two hosts that should not have been connected. We have defined a topology and set of intents such that there are exactly four temporal vulnerabilities that could lead to unauthorized connectivity. The baseline algorithm is guaranteed to find all vulnerabilities because it performs an exhaustive search.

For each experimental run of Spotlight, we collect the number of temporal vulnerabilities that were verifiably detected. The preprocessing time is the time it takes to run the Panther similarity algorithm on the topology. The clock starts before preprocessing begins and ends immediately after.

6.2 Performance of Spotlight

Figure 6 shows the time it took to detect the first temporal vulnerability in the fat tree topology. These are restricted to only the cases where at least one temporal vulnerability was found, which as we will see in Section 6.3 was the case for most runs. The x-axis tracks the number of hosts and switches for each run and the y-axis, drawn on a log scale, is the time in seconds to detect the first temporal vulnerability.

We compare the detection time of Spotlight (blue points) to that of the baseline (red points) (**Q1**). The detection time for both detector implementations increases with the size of the topology. This growth, however, is subexponential, as discussed in **??**. Additionally, the baseline algorithm has much greater variance, with a standard deviation on the largest topology of 17.5 seconds compared to 0.859 seconds for Spotlight. This is likely because there is a larger space of update orderings over which the temporal vulnerabilities are distributed. In the largest topology with 1,344 nodes, Spotlight's detection time was 0.65 seconds, while the baseline detection time was 49.9 seconds, a 76.8× speedup (**Q2**). Spotlight had a mean detection time of 0.002 seconds on the Stanford backbone topology, which contains 51 nodes. The median time to detect the first violation was 0.00192 seconds for Spotlight and 0.0013 seconds for the naïve baseline. This resulted in a median speedup of 6.77× and a maximum speedup of 31.13×. Results were similar for the Cisco topology, which contains 20 nodes. On this topology, the mean detection time was 0.00641 seconds—consistent with fat tree topologies of similar size.

In Table 2, the preprocessing time varies from an average of 0.399 seconds for a small fat tree network to over ten minutes for the largest topology. The Stanford backbone an Cisco topologies were preprocessed even faster, 0.219 seconds and 0.0442 seconds, respectively. The longest preprocessing time was 5,682.21 seconds (over ninety minutes), though this was an outlier. This can be an expensive operation but only needs to be computed once per topology. If topologies are unlikely to change frequently, such cost could be amortized over the lifespan of a single topology. For smaller networks, even those than can support tens of thousands of clients like Stanford's backbone, Spotlight's preprocessing time is quick and could be recomputed ad hoc at very little cost.

We also tracked the memory usage of Spotlight. We found that Spotlight used approximately 176.3 MB for our largest 1,300-node topology. This computation is performed on the controller node, rather than a switch, and thus is not subject to the same resource constraints as switch processes.

6.3 Accuracy of Spotlight

Unlike the strawman algorithm, which is guaranteed to find temporal vulnerabilities if they exist, Spotlight uses a probabilistic method based on graph heuristics and so may not find all vulnerabilities.

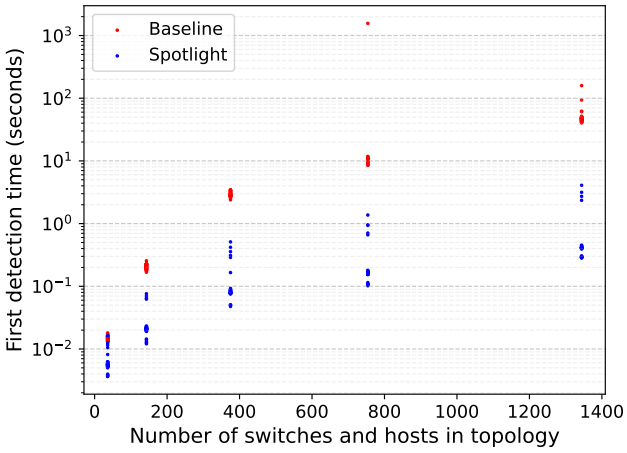


Figure 6: Time taken to detect temporal vulnerability as a function of the number of hosts in the fat tree topology.

In Table 2 column two, we see the probability of Spotlight finding at least one temporal vulnerability across all trials for each node count (**Q3**). For small networks, Spotlight finds a temporal vulnerability with high probability (98.0%), however the probability drops as the topology scales in size. This column, however, represents the probabilities over all configurations and tuning parameters. For both the Stanford backbone and Cisco topologies, Spotlight found all temporal vulnerabilities in all trials. These results suggest that Spotlight is highly reliable for small networks, even without optimal parameter tuning.

When we isolate the cases where Panther parameter k (representing the size of the similarity cache) is set to twenty, we see that all temporal vulnerabilities were found in all fat tree topology trials. This is further substantiated in Figure 7. In this heatmap, the x-axis is Panther's k, and the y-axis is the number of vulnerabilities found. Each square represents the proportion of runs of Spotlight with parameter k that found the corresponding number of vulnerabilities on the y-axis. There is a strong positive trend with increasing k suggesting that increasing the size of the similarity cache yields more found vulnerabilities at a slight cost of storing more data. Any additional computational cost is borne during preprocessing and thus does not factor significantly into detection timing.

There is also notably a pattern of stratification where each detector run finds either zero, two, or four temporal vulnerabilities, but never one or three. The most likely cause of this is the proximity of the updated switches in the topology graph. In the intents used in the evaluation, there are two updated switches near one host, and two near the other host. The similarity cache is thus likely to contain the switches in pairs, if at all.

Ultimately, we see that Spotlight can be tuned to find at least one temporal vulnerability with a very high probability (**Q4**).

7 DISCUSSION

Mitigating temporal attacks with least-privileges access control. Sufficiently precise pattern matching on the flow rules would prevent

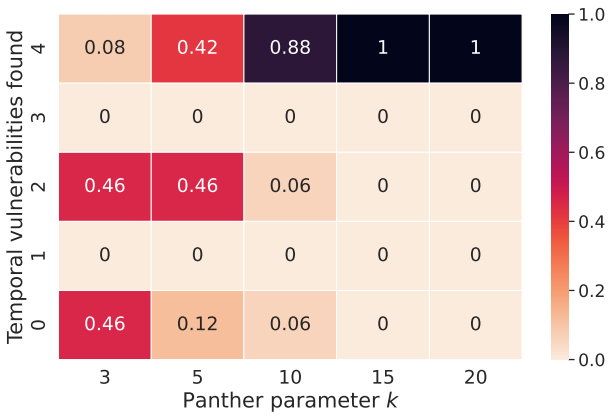


Figure 7: Sensitivity of finding vulnerabilities to the Panther algorithm's k parameter representing the size of the similarity cache.

unauthorized connectivity even in the presence of inconsistent flow rule updates. However, this puts an additional burden on the individual security and system administrators. A secure-by-design model, such as `Spotlight`, is more effective at securing sensitive systems because it offers protection even in the presence of overly permissive access control policies.

Additionally, some security models intentionally group hosts (e.g., identified by IP addresses) together within flow rule matching patterns as a data plane optimization to save TCAM space in switch memory [12], which is an intentional design decision for which more restrictive policies may not be an option.

Mitigating temporal attacks with update consistency. While consistent update schemes would mitigate temporal attacks, consistent updates come at significant performance cost. Liu et al. [48] measured several proposed update schemes and found that even the fastest scheme increased the median update time by more than 50% even on medium to small fat tree topologies ($f_k = 8$) with a relaxed notion of consistency. For more rigorous models, Liu et al. [48] reported an overhead of more than 400%. These consistent update schemes also come at a cost of additional TCAM usage, with even the lowest footprint model using more than 10% additional memory—already an expensive and scarce resource.

We hypothesize that a mixed-modal solution could provide a realistic balance. A temporal vulnerability detector like `Spotlight` could be used on all intent installations and recompilations, and only when a temporal vulnerability is detected a consistent update could be triggered. However, future work is necessary to identify the likelihood of high-risk updates appearing in the wild, as well as a comparison of the cost of consistent updates with the cost of executing a temporal vulnerability detector.

Other vulnerable IBN implementations. Our evaluations use ONOS as a representative case study. However, we suspect that several other well known controllers may also be vulnerable. To be robust against a Phantom Link during intent installation, an IBN controller must have a mechanism to control the flow rule install ordering across *multiple switches*. The popular IBN controller, OpenDaylight (ODL) [47] does not contain such a mechanism, and so to the best of our knowledge, it is also vulnerable. ODL is highly scalable [22] and has a built-in ARP proxy app which can also delay the installation of rules on the switch; these features provide the same footholds we used to exploit ONOS. Several other ostensibly independent SDN environments—ONAP [1], OpenStack [2], and Cisco Open SDN [3]—use ODL as the underlying SDN controller, and therefore share the same shortcomings as ODL. We are unable to acquire or examine the code of closed-source IBN controllers like Juniper Apstra or Cisco Meraki, so we cannot infer their vulnerability status.

Non-static topologies. While no topology is ever truly static, we argue that this is not necessarily a limitation for `Spotlight`. Intentional topology changes (e.g., adding links) takes non-negligible time, during which the Panther algorithm precomputation of the new topology is possible. Conversely, when the topology changes quickly due to link or switch failure, the naïve algorithm will still find all temporal vulnerabilities because the controller is alerted of the failure; as a result, the intent recompilation will be triggered. Meanwhile, the Panther algorithm will not see any reduction in likelihood of finding any new temporal vulnerabilities because link/switch failures do not create any new paths through the network topology which were not already taken into account.

8 RELATED WORK

IBN security. IBN abstracts away many implementation details to simplify network management and can be utilized to enhance the security of the overall network [41]. The LAI language [71] automatically generates ACL update plans that satisfy the network operators' intents. Herbaut et al. [29] propose an efficient conformance checking approach based on intents. Intent-based cloud services [40] provides security services to both the service providers and consumers to apply security policies without security expertise. While such approaches can improve the network's security, they do not consider the attack surface of the IBN architecture itself and do not mitigate temporal vulnerabilities.

Several security tools have been proposed for understanding IBN's attack surface. ProvIntent [72] generates a provenance of how IBN events affect the network state while bridging the semantic gap between high-level intent and low-level implementation. Intender [42] proposes a semantically-aware fuzzing framework with a new feedback mechanism, intent-state transition guidance. While these tools are complementary to `Spotlight`, they cannot identify temporal vulnerabilities during intent updates.

Network verification. Network verification checks that network policies are satisfied and that no constraints are violated. Early work in verification, such as header space analysis [37, 38], Anteater [49] and VeriFlow [39], check for network invariants (e.g., blackhole routing and loops) in the control path between an SDN controller and switches. However, such tools focus on singular updates (i.e., one flow rule at a time) rather than groups of updates (e.g., multiple flow rules for end-to-end connectivity) that are typical of IBN updates. Plankton [58] builds on VeriFlow's computation of packet equivalence classes and provides model checking for the forwarding behavior in the converged states of routers. Tiramisu [8]

provides verification algorithms for policies about failure resilience, quantitative path metrics, and path existence.

More recently, runtime verification checks network conditions in real time. Tools such as Delta-net [30], P4Consist [66], VeriDP [78], and Hydra [63] check data plane consistency, increasingly within the P4 [15] programmable data plane. None of this prior work focuses on the unique considerations and semantics of IBN—they only examine lower levels of abstraction.

Policy composition. Policy composition analyzes how to satisfy multiple new and existing policies, as well as identify and report unresolvable conflicts to users. Policy Graph Abstraction (PGA) [59] expresses user-submitted networking policies in the form of input graphs and provides algorithms to combine those input graphs into a conflict-free graph, which represents the final composed policy. PGA also maintains the invariants specified in each user-submitted policy. Unresolvable conflicts are reported to the user. Janus [9] extends PGA to additionally support QoS and dynamic policies. Genesis [70] proposes policy composition and compilation in multi-tenant networks. Policies are described in a new language called the Genesis Policy Language. A modular SMT-based algorithm is designed to enforce the policies. Although the aforementioned prior work addresses the correctness and optimality of policy composition, none of the prior work focuses on the ordering of operations and the temporal faults that may arise due to improper ordering.

SDN security. Both attacks and defenses of SDNs have been studied numerous times in the literature [26, 33, 34, 60, 65, 68, 69, 73, 74]. Since IBNs are an extension of SDNs, these attacks and defenses are also applicable to IBNs, but the focus of this paper is on temporal attacks that are especially pernicious on IBNs.

9 CONCLUSION

In this paper, we identified a vulnerability in IBN caused by the reordering of low-level flow rules from intents. The vulnerability can be exploited by an attacker to obtain unauthorized access to a host. We discussed means to achieve such attacks and demonstrate one type of attack, Phantom Link. We proposed `Spotlight` a detection method that can alert a system administrator of risky intents prone to temporal faults that can be exploited by an attacker. We demonstrate that `Spotlight` is fast and effective in identifying such risky intents using realistic network topologies and policies.

ACKNOWLEDGMENTS

The authors thank our anonymous shepherd and the reviewers for their helpful comments, which improved this paper. The first author would also like to thank his friends, Marcus and Bekah Coenen: your courage in these times is an inspiration. This material is based upon work supported by the National Science Foundation under Grant No. CNS-2339882.

REFERENCES

[1] [n. d.]. https://wiki.onap.org/display/DW/Controllers
[2] [n. d.]. https://docs.openstack.org/networking-odl/ocata/installation.html
[3] [n. d.]. https://www.cisco.com/c/en/us/td/docs/net_mgmt/open_sdn_controller/1-1/admin/guide/b_OSC11_Admin_Guide/b_OSC11_Admin_Guide_chapter_00.pdf
[4] [n. d.]. ONOS GitHub FlowRuleOperations.java. https://github.com/opennetworkinglab/onos/blob/master/core/api/src/main/java/org/onosproject/net/flow/FlowRuleOperations.java. Accessed: 2024-01-12.
[5] 3GPP. 2020. Study on scenarios for Intent driven management services for mobile networks (Release 17). https://www.3gpp.org/DynaReport/28812.htm.
[6] 3GPP. 2022. Management and orchestration; Intent driven management services for mobile networks (Release 17). https://www.3gpp.org/DynaReport/28312.htm.
[7] 3GPP. 2022. Study on enhanced intent driven management services for mobile networks (Release 18). https://www.3gpp.org/DynaReport/28912.htm.
[8] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Tiramisu: Fast multilayer network verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 201–219.
[9] Anubhavnidhi Abhashkumar, Joon-Myung Kang, Sujata Banerjee, Aditya Akella, Ying Zhang, and Wenfei Wu. 2017. Supporting Diverse Dynamic Intent-based Policies using Janus. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*. ACM, 296–309.
[10] Stefan Achleitner, Thomas La Porta, Trent Jaeger, and Patrick McDaniel. 2017. Adversarial network forensics in software defined networking. In *Proceedings of the Symposium on SDN Research*. 8–20.
[11] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*. ACM, 63–74.
[12] Iffat Anjum, Daniel Kostecki, Ethan Leba, Jessica Sokal, Rajit Bharambe, William Enck, Cristina Nita-Rotaru, and Bradley Reaves. 2022. Removing the Reliance on Perimeters for Security using Network Views. In *Proceedings of the 27th ACM SACMAT*. ACM, New York NY USA, 151–162.
[13] Iffat Anjum, Jessica Sokal, Hafiza Ramzah Rehman, Ben Weintraub, Ethan Leba, William Enck, Cristina Nita-Rotaru, and Bradley Reaves. 2023. MSNetViews: Geographically Distributed Management of Enterprise Network Security Policy. In *Proceedings of the 28th ACM SACMAT (SACMAT '23)*. ACM, 121–132.
[14] Fetia Bannour, Sami Souihi, and Abdelhamid Mellouk. 2017. Distributed SDN control: Survey, taxonomy, and challenges. *IEEE Communications Surveys & Tutorials* 20, 1 (2017), 333–354.
[15] 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.
[16] Kai Bu, Xitao Wen, Bo Yang, Yan Chen, Li Erran Li, and Xiaolin Chen. 2016. Is every flow on the right track?: Inspect SDN forwarding with RuleScope. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 1–9.
[17] Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. 2015. A distributed and robust SDN control plane for transactional network updates. In *2015 IEEE conference on computer communications (INFOCOM)*. IEEE, 190–198.
[18] Jiahao Cao, Qi Li, Renjie Xie, Kun Sun, Guofei Gu, Mingwei Xu, and Yuan Yang. 2019. The {CrossPath} Attack: Disrupting the {SDN} Control Channel via Shared Links. In *28th USENIX Security Symposium (USENIX Security 19)*. 19–36.
[19] Jiahao Cao, Zijie Yang, Kun Sun, Qi Li, Mingwei Xu, and Peiyi Han. 2019. Fingerprinting {SDN} applications via encrypted control traffic. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 501–515.
[20] Cisco Networks. 2024. Intent-Based Networking: Cisco. https://www.cisco.com/c/en/us/solutions/intent-based-networking.html.
[21] Alexander Clemm, Laurent Ciavaglia, Lisandro Granville, and Jeff Tantsura. 2022. Intent-Based Networking - Concepts and Definitions (RFC 9315). https://datatracker.ietf.org/doc/rfc9315/.
[22] Mohamad Darianian, Carey Williamson, and Israat Haque. 2017. Experimental evaluation of two openflow controllers. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*. IEEE, 1–6.
[23] Edsger Wybe Dijkstra. 1959. A note on two problems in connexion with graphs:(Numerische Mathematik, 1 (1959), p 269-271). (1959).
[24] Andrew D Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, et al. 2021. Orion: Google's {Software-Defined} Networking Control Plane. In *18th USENIX NSDI*. 83–98.
[25] Klaus-Tycho Foerster, Stefan Schmid, and Stefano Vissicchio. 2019. Survey of Consistent Software-Defined Network Updates. *IEEE Communications Surveys & Tutorials* 21, 2 (2019), 1435–1461. https://doi.org/10.1109/COMST.2018.2876749
[26] Steven R. Gomez, Samuel Jero, Richard Skowyra, Jason Martin, Patrick Sullivan, David Bigelow, Zachary Ellenbogen, Bryan C. Ward, Hamed Okhravi, and James W. Landry. 2019. Controller-Oblivious Dynamic Access Control in Software-Defined Networks. In *2019 49th Annual IEEE/IFIP DSN*. 447–459.
[27] Aric Hagberg and Drew Conway. 2020. Networkx: Network analysis with python. *URL: https://networkx. github. io* (2020).
[28] Robert Hanmer, Sheng Liu, Lalita Jagadeesan, and Muntasir Raihan Rahman. 2019. Death by babble: Security and fault tolerance of distributed consensus in high-availability softwarized networks. In *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 266–270.

[29] Nicolas Herbaut, Camilo Correa, Jacques Robin, and Raul Mazo. 2021. SDN Intent-based conformance checking: application to security policies. In *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*. IEEE, 181–185.

[30] Alex Horn, Ali Kheradmand, and Mukul Prasad. 2017. Delta-net: Real-time network verification using atoms. In *14th USENIX NSDI*. 735–749.

[31] Huawei. 2024. Huawei Launches the Intent-Driven Networking for CloudFabric Solution. https://www.huawei.com/en/news/2018/6/Intent-Driven-Networking-CloudFabric-Solution.

[32] IBM Newsroom. 2021. IBM Brings AI-Powered Automation Software to Networking to Help Simplify Broad Adoption of 5G.

[33] Samuel Jero, Xiangyu Bu, Cristina Nita-Rotaru, Hamed Okhravi, Richard Skowyra, and Sonia Fahmy. 2017. BEADS: Automated Attack Discovery in OpenFlow-Based SDN Systems. In *Research in Attacks, Intrusions, and Defenses*. Springer, 311–333.

[34] 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)*. USENIX Association, Vancouver, BC, 415–432.

[35] Juniper Networks. 2024. Juniper Apstra. https://www.juniper.net/us/en/products/network-automation/apstra.html.

[36] Min Suk Kang, Soo Bum Lee, and Virgil D Gligor. 2013. The crossfire attack. In *2013 IEEE symposium on security and privacy*. IEEE, 127–141.

[37] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. 2013. Real time network policy checking using header space analysis. In *10th USENIX NSDI*. 99–111.

[38] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header space analysis: Static checking for networks. In *9th USENIX NSDI*. 113–126.

[39] 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.

[40] Jinyong Kim, Eunsoo Kim, Jinhyuk Yang, Jaehoon Jeong, Hyoungshick Kim, Sangwon Hyun, Hyunsik Yang, Jaewook Oh, Younghan Kim, Susan Hares, et al. 2020. Ibcs: Intent-based cloud services for security applications. *IEEE Communications Magazine* 58, 4 (2020), 45–51.

[41] Jiwon Kim, Hamed Okhravi, Dave (Jing) Tian, and Benjamin E. Ujcich. 2024. Security Challenges of Intent-Based Networking. *Commun. ACM* 67, 7 (2024).

[42] Jiwon Kim, Benjamin E. Ujcich, and Dave (Jing) Tian. 2023. Intender: Fuzzing Intent-Based Networking with Intent-State Transition Guidance. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4463–4480.

[43] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. 2014. Software-defined networking: A comprehensive survey. *Proc. IEEE* 103, 1 (2014), 14–76.

[44] Maciej Kuzniar, Peter Peresini, and Dejan Kostić. 2014. Providing reliable FIB update acknowledgments in SDN. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 415–422.

[45] Chen Li, Olga Havel, Aiana Olariu, Peo Martinez-Julia, Jéferson Campos Nobre, and Diego Lopez. 2022. *Intent Classification*. Number RFC 9316. https://doi.org/10.17487/RFC9316

[46] Linux Foundation. 2024. Open Network Automation Platform (ONAP). https://www.onap.org/.

[47] Linux Foundation. 2024. OpenDaylight (ODL). https://www.opendaylight.org/.

[48] Sheng Liu, Theophilus A Benson, and Michael K Reiter. 2019. Efficient and safe network updates with suffix causal consistency. In *Proceedings of the fourteenth EuroSys conference 2019*. 1–15.

[49] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the data plane with anteater. *ACM SIGCOMM CCR* 41, 4 (2011), 290–301.

[50] Víctor Martínez, Fernando Berzal, and Juan-Carlos Cubero. 2017. A Survey of Link Prediction in Complex Networks. *Comput. Surveys* 49, 4 (Dec. 2017), 1–33.

[51] Rick McGeer. 2012. A safe, efficient update protocol for openflow networks. In *Proceedings of the first workshop on Hot topics in SDNs*. ACM, 61–66.

[52] 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.

[53] ONF. 2016. Intent NBI – Definition and Principles. https://opennetworking.org/wp-content/uploads/2014/10/TR-523_Intent_Definition_Principles.pdf.

[54] Open Networking Foundation. 2024. Open Network Operating System (ONOS). https://opennetworking.org/onos/.

[55] P4.org API Working Group. 2024. P4Runtime Specification. https://github.com/p4lang/p4runtime.

[56] Aurojit Panda, Wenting Zheng, Xiaohe Hu, Arvind Krishnamurthy, and Scott Shenker. 2017. {SCL}: Simplifying Distributed {SDN} Control Planes. In *14th USENIX NSDI*. 329–345.

[57] Pica8. 2014. Pica8 P-5401 Specification. https://www.pica8.com/wp-content/uploads/pica8-datasheet-32x40gbe-p5401.pdf

[58] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. 2020. Plankton: Scalable network configuration verification through model checking. In *17th USENIX NSDI*. 953–967.

[59] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. 2015. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. ACM, London United Kingdom, 29–42.

[60] Leila Rashidi, Daniel Kostecki, Alexander James, Anthony Peterson, Majid Ghaderi, Samuel Jero, Cristina Nita-Rotaru, Hamed Okhravi, and Reihaneh Safavi-Naini. 2021. More than a Fair Share: Network Data Remanence Attacks against Secret Sharing-based Schemes. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'21)* (San Diego, CA).

[61] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. 2012. Abstractions for network update. *ACM SIGCOMM Computer Communication Review* 42, 4 (Sep 2012), 323–334. https://doi.org/10.1145/2377677.2377748

[62] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. 2011. Consistent updates for software-defined networks: change you can believe in!. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*. ACM, Cambridge Massachusetts, 1–6. https://doi.org/10.1145/2070562.2070569

[63] 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.

[64] Liron Schiff, Stefan Schmid, and Petr Kuznetsov. 2016. In-band synchronization for distributed SDN control planes. *ACM SIGCOMM Computer Communication Review* 46, 1 (2016), 37–43.

[65] Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. 2013. Avantguard: Scalable and vigilant switch flow management in software-defined networks. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 413–424.

[66] Apoorv Shukla, Seifeddine Fathalli, Thomas Zinner, Artur Hecker, and Stefan Schmid. 2020. P4consist: Toward consistent p4 sdns. *IEEE Journal on Selected Areas in Communications* 38, 7 (2020), 1293–1307.

[67] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. *ACM SIGCOMM Computer Communication Review* 45, 4 (Sept. 2015), 183–197.

[68] Richard Skowyra, Kevin Bauer, Veer Dedhia, and Hamed Okhravi. 2016. Have No PHEAR: Networks Without Identifiers. In *Proceedings of the 2016 ACM Workshop on Moving Target Defense (MTD '16)*. ACM, 3–14.

[69] 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)*. 374–385.

[70] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. 2017. Genesis: synthesizing forwarding tables in multi-tenant networks. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, 572–585.

[71] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, et al. 2019. Safely and automatically updating in-network ACL configurations with intent language. In *Proceedings of ACM SIGCOMM*. 214–226.

[72] Benjamin E. Ujcich, Adam Bates, and William H. Sanders. 2020. Provenance for intent-based networking. In *2020 6th IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 195–199.

[73] 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* (Toronto, Canada) *(CCS '18)*. ACM, New York, NY, USA, 648–663.

[74] Benjamin E. Ujcich, Samuel Jero, Richard Skowyra, Steven Gomez, Adam Bates, William H. Sanders, and Hamed Okhravi. 2020. Automated Discovery of Cross-Plane Event-Based Vulnerabilities in Software-Defined Networking. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'20)*.

[75] Haopei Wang, Lei Xu, and Guofei Gu. 2015. Floodguard: A dos attack prevention extension in software-defined networks. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 239–250.

[76] Tianlong Yu, Seyed Kaveh Fayaz, Michael P Collins, Vyas Sekar, and Srinivasan Seshan. 2017. PSI: Precise Security Instrumentation for Enterprise Networks.. In *NDSS*.

[77] Jing Zhang, Jie Tang, Cong Ma, Hanghang Tong, Yu Jing, and Juanzi Li. 2015. Panther: Fast Top-k Similarity Search on Large Networks. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, Sydney NSW Australia, 1445–1454.

[78] 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.