Network Reconnaissance Attack Countermeasures: Occluding Occult Network Surveillance in SDN

Akash Verma
North Carolina State University
averma3@ncsu.edu

Isaac Polinsky
North Carolina State University
ipolins@ncsu.edu

ABSTRACT

This work examines previously unseen methods of active network reconnaissance made possible due to the advent and popularization of Software-Defined Networking (SDN). The architecture of SDN allows an adversary, controlling a compromised switch inside the network, to perform stealthy active network reconnaissance and gather intelligence about the network. This includes trying to identify sensitive resources, learn about the services, identify network policies, and discover potential vulnerabilities that can be exploited. This information can then be used in forming an attack plan to exfiltrate sensitive data. We propose Network RACOONS, a security application running on the controller, to detect active reconnaissance attacks. Network RACOONS is implemented in Python and runs on top of a Pox controller in a software defined network, it was tested for its ability to quickly and accurately detect network reconnaissance in a small simulated environment. Our testing shows Network RACOONS effectively detects network reconnaissance with no observed false negatives or false positives and an overhead of less than 3% on network throughput and latency. This work demonstrates the need to proactivley identify new network threats introduced by SDN before it gains a more widespread deployment.

1. INTRODUCTION

Over the years, attacks on networks have become increasingly more complex and covert and thus harder to detect. Post-mortem reports have shown network adversaries often spend upwards of 200 days inside the network before being discovered, and in some cases, more than 250 days [10]. As long as the adversary continues to go undetected, these sophisticated network attacks allow an adversary to collect sensitive data and exfiltrate it from the network, resulting in severe financial damages.

To successfully carry out an attack on a network, the adversary might typically perform a series of steps that goes like the following: (1) get a foothold within the network, (2) using that foothold identify both what the sensitive resources are and how to get to the sensitive resources, (3) exfiltrate the data, and (4) optionally have a plan to remove any evidence of their presence in the network and end the attack, or simply continue to stay hidden in the network until they are detected. In this work, we analyze Step (2), also known as *network reconnaissance*, in deeper detail. More specifically, we look at an active probing mechanism for network reconnaissance in a Software-Defined Network and defense mechanisms to detect such activities. Although network reconnaissance alone does not cause much harm to the network, it is often followed by an attack. Thus any attempt to perform network reconnaissance is a strong indication of a future attack on the network. We note simply preventing or detecting network reconnaissance alone is not enough for maintaining a secure network, but our work can be integrated into a defense in depth approach to network security.

As mentioned in step 2 of the sample attack lifecycle, one of the goals of network reconnaissance is to identify both the sensitive resources and to access the sensitive resources before being able to exfiltrate data. Adversaries also try to gather information about the network itself, such as the types of security middleboxes or other network policies present. Identifying which hosts internal to the network contain sensitive information may be a nontrivial task, but stealthily accessing the sensitive resources can be challenging. For example, the foothold acquired in step 1 may not have access to the targeted resource directly or there is a network defense between the adversary and the resource that will trigger an alarm of their presence. To mitigate this defense, the adversary must identify vulnerable hosts safely reachable from their foothold and also contain a vulnerability allowing them to move further into the network. These discovered vulnerable hosts act as *stepping stones* to the targeted resource from the adversary's foothold and it may require multiple stepping stones to be compromised in order to access the targeted resource. Once an adversary obtains a path to the targeted resources, which may take multiple rounds of reconnaissance, they may begin exfiltrating data from the network.

Network reconnaissance and methods to prevent and detect its activity has been an area of research since the late 1990's [14, 5]. Prior work in this domain has looked at network scanning attacks [5, 1, 19, 3] that use spoofed IP packets to gather intelligence in traditional networks. These active network reconnaissance attacks and the challenges in detecting the probes has also been well discussed in other

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

previous works [15]. However, as Software Defined Networks are gaining popularity, we find it valuable to examine new network reconnaissance techniques introduced by their architecture. Network reconnaissance has yet to be reported as a security problem in SDN, but in 2013 Song et al. [17] discussed the possibility of network scanning attacks in SDN. This work is motivated to take proactive measures in identifying new security threats before SDN is deployed on a large scale. In addition, by the nature of the proposed stealthy reconnaissance techniques, it is possible these attacks are actively being used against SDN in the wild and there are no detection mechanisms currently able to detect and report them as a problem.

To illustrate the importance of identifying security problems during the early stages of system deployment, consider the field of Operating System Security. When operating systems were first developed, there were no concerns of security and it was assumed all users and programs would behave as expected in a benign manner; however, as we all know, this quickly became an issue and systems needed to be re-engineered to provide much needed security features. As SDN is in its early phases, with the potential for wide spread adoption, we have the opportunity to not only tackle well known security issues seen in traditional networks but we can also identify potential new attack vectors before they have the chance to cause any harm to deployed systems.

In this work we analyze network reconnaissance in Software Defined Networks and make the following key contributions:

1. We present a new network reconnaissance attack in SDN that gives an adversary, controlling a compromised switch, an enhanced view of the entire network and enables information gathering in a way infeasible in traditional networks.

2. We present the design of an SDN application and implement a prototype to detect the reconnaissance attack introduced above. Our prototype successfully detected reconnaissance probes with no false positives or false negatives with an overhead around 3% for both network throughput and latency.

3. We setup a small scale but realistic lab environment to run our experiments and present the results of our experiments.

2. BACKGROUND

In this section, we first introduce the architecture and operation of Software-Defined Networking (SDN). Next we give an overview of network reconnaissance, its utility to attackers, and the impact of successful reconnaissance on an enterprise network's security. Finally we discuss how the architecture of SDN enables an adversary to perform reconnaissance in ways that were not feasible in traditional networks.

2.1 Software-Defined Networking

SDN decouples the data plane of a network from the control plane enabling dynamic configuration and management of network devices through programmable SDN applications that run on a network controller. The SDN architecture can be divided into 4 different planes, which serve distinct purposes and have their own defined functions. These planes

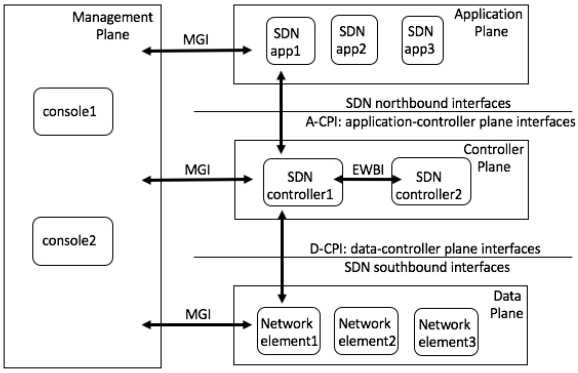

Figure 1: SDN Architecture

contain one or more entities or components working independently or together to perform the tasks of a given plane. Communication between the planes is made possible through various interfaces via open APIs and protocols.

2.1.1 *Planes/layers identified in SDN*

We now describe the components and functions for three of the four different planes, which are relevant to this work.
Data plane: The data plane is comprised of a set of one or more network elements (switches etc.) each of which contains a set of traffic forwarding or traffic processing resources. The forwarding rules present on the switches are stored in flow tables as flow-entries that are set by the SDN controller. The data plane incorporates the resources that deal directly with customer traffic and some supporting resources to ensure proper virtualization, connectivity, security, availability, and quality.
Controller plane: The controller plane is comprised of a set of SDN controllers that have exclusive control over a set of resources that are exposed by one or more network elements in the data plane. The major role of an SDN controller is to execute requests of the SDN applications it supports and keep applications isolated from each other. Since a network may have more than one SDN controller, a controller may communicate with peer controllers, subordinate controllers, or non-SDN environments, to carry out tasks on behalf of applications.
Application plane: The application plane is comprised of one or more applications, each of which has exclusive control of a set of resources exposed by one or more SDN controllers. Several applications can be operational at the same time performing various different tasks and can alter the state of network resources or alter the behavior of the data plane via the controller plane. An application may invoke or collaborate with other applications and can also act as an SDN controller in its own right.

2.1.2 *Interfaces identified in SDN*

To enable smooth communication between the planes and their underlying components, there are four interfaces present in the SDN architecture. Here we describe just two of the four, which are relevant to this work.
Northbound Interfaces (NBI): The northbound interface exists between the controller plane and the application plane and enables communication between them. Several

different applications can be operational in the application plane and can communicate with the controllers using an exposed API. The controllers also respond to applications' requests and pass on information, events etc. to the applications using this interface.

Southbound Interfaces (SBI): The southbound interface exists between the controller plane and the data plane and enables communication between the controllers and the network devices or elements. The communication across this interface uses an open standard protocol, such as the OpenFlow protocol. The controllers as well as the network devices both the support this messaging protocol. The most common communication across this interface are control messages sent by the controller to alter the behavior of the underlying network and notifications from the network devices when they come online.

Whenever a new flow is encountered by a switch, it first looks at its flow table to check for a flow entry matching the given flow. If such a match is found, the switch applies the action stored in flow table for the matching flow entry. If no match is found, the switch holds the packet in a buffer and sends the header information to the controller in the form of a *packet_in* message. This message transfer between the data plane and the control plane generally happens via OpenFlow [8] protocol, which is a widely popular SDN messaging protocol. Along with the packet header, the packet_in message contains information about the physical port on the switch the packet was received on. The controller has one or more SDN applications registered with it that run in the application plane and register event listeners for certain messages from the data plane, such as packet_in messages. As soon as the controller receives the packet_in message, the controller fires a packet_in event and the applications registered for the corresponding handler are notified. The applications then extract the flow information and perform certain logic to decide the action to be done to the flow the packet belongs to. This information is then passed to the controller from the applications by using the NBI. Finally, the controller sends down control messages to the switch in the data plane instructing what should be done to the flow by using the SBI.

2.2 Network Reconnaissance

Before carrying out a major attack against a network, the attackers generally attempt to gather as much information as possible. Most importantly, they are focused on learning about the resources available, their weaknesses and vulnerabilities, which resources are high value targets, and what existing defense mechanisms are in the network and how they are distributed. This intelligence gathering phase is known as network reconnaissance. The attacker can use various methods such as surveillance, eavesdropping on and intercepting communications, and launching probes to gather information about the computer systems, policies, and other resources. Network reconnaissance either falls into one of two categories:

Passive Network Reconnaissance: In this type of reconnaissance, usually the adversary already establishes a foothold at a very strong vantage point. For example, a strong vantage point in a network is at core switch. A core switch ties the network together, seeing a lot of flows crossing its boundaries. An adversary controlling one or more such switches can learn a lot about the topology of network,

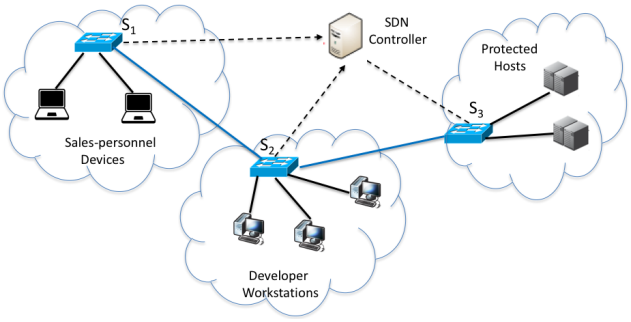

Figure 2: A subset of Typical Enterprise SDN Deployment

placement of security middleboxes, policies regarding communication between hosts, etc. by just passively observing the traffic passing through the switch. Therefore they can learn a lot without the need to perform any extra actions, and if and when required, may exfiltrate the gathered intelligence out of the network. Since the attacker does not perform any suspicious actions, attacks falling into passive reconnaissance are very hard to detect in a network.

Active Network Reconnaissance: In this type of reconnaissance, an adversary controls a less advantageous vantage points, which gives only a local view of the network. Passively observing flows from this vantage point prevents an adversary from learning about the entire network as only a small subset of flows are observable. Therefore, the adversary uses specially crafted network packets called probes to learn about other parts of the network. Adversaries subtly send probes at low frequencies to avoid triggering a defense mechanism using thresholds to identify anomalies in the network.

Adversaries aim to learn the following information while performing network reconnaissance:

- The topology of the network, including the locations of sensitive resources and security middleboxes.
- Active hosts accessible to external networks.
- The services and applications running in the network and any vulnerabilities these services and applications may have that can be exploited to gain unauthorized access.

In traditional networks, network reconnaissance using active probes has been done by network port scanning or by directly interacting with other hosts and devices on the network. Some of the techniques used for active probing of a network include making use of mechanisms such as the TCP handshake to judge a host's liveness, fingerprinting the protocol stack to try to figure out the operating system the host is running, probing DNS servers, and grabbing service banners volunteering information about the host.

The advent and popularization of SDN can lead adversaries to look for new ways of network reconnaissance and different vantage points that were not available or difficult to perform in a traditional network.

Typical Enterprise Network (SDN): Figure 2 shows a subset of a typical enterprise network, comprised of several different categories of hosts that perform different tasks,

or are owned by employees performing different types of tasks. Generally, these categories include developer workstations, sales and marketing devices, HR owned laptops, and other highly sensitive resources such as Git servers, database servers, DNS servers, etc. Enterprises generally keep these different categories of hosts in separate subnets for easy management and separation. In each subnet there are one or more subnet switches and the entire network has one or more SDN controllers depending on the number of hosts. In such a setup, if the developer subnet switch is compromised by adversary, they would ideally want to learn more about the network before performing more serious attacks. They might check if hosts in the sales subnet have connectivity with a protected resource they are targeting. This information is useful for a number of reasons. For example, if sales devices are running an OS that is different than the OS running on the developer workstations and the sales devices have an unpatched vulnerability, using the sales device to access the protected resource may be easier than using the developer workstation. This scenario illustrates the need to detect network reconnaissance in a network, the next section explicitly states all assumptions about the adversary and the network environment we consider in this work.

3. PROBLEM

In this section we introduce our threat model, as well as all assumptions we make about the environment. We defer all discussion about our assumptions and threats to the validity of our research to Section 7. Given our threat model and assumptions, we then define the problem statement this work aims to solve at the end of this section.

Threat Model: This work considers an adversary performing active reconnaissance from a compromised switch, in a software defined network. Further, it is assumed at most one switch within the network is compromised for the entirety of the attack, nor is the compromised switch a core switch in the network. This assumption prevents an adversary from maintaining a passive view of the entire network and forces the adversary to perform active reconnaissance to get a complete view of the network. The adversary not only has the ability to compromise the switch, but is also capable of crafting spoofed packets at the switch, which result in illegitimate packet_in messages to be sent to the controller and be processed by the applications in the application plane. In addition, the adversary is capable of ignoring flow_mod messages and dropping packets matching a specific flow, particularly to prevent their spoofed packet from propagating throughout the network. Finally, the adversary maintains full knowledge of our solution and the algorithms used to detect reconnaissance activity. With these stated capabilities, the adversary aims to gather intelligence about the network, which is later used to form an attack plan to exfiltrate data from sensitive resources inside the network.

We define our Trusted Computing Base (TCB) to be the SDN controller and applications as well as the communication channels used by the North Bound and South Bound Interfaces, note only the channels are trusted and not the data relayed within the channels. That is to say we guarantee which switch is actually sending data but we don't trust what that data actually reports. Our assumed environment is a fully SDN enabled enterprise network, whose topology was configured using well known best practices.

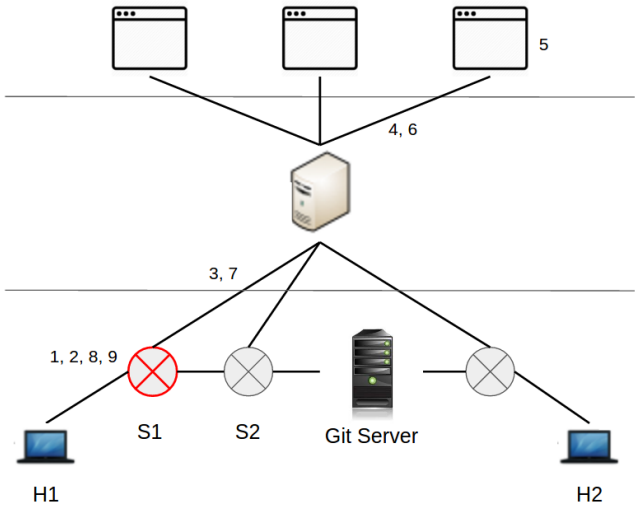

Figure 3: Overview of a reconnaissance attack abusing packet_in messages.

Problem Statement: *Software Defined Networks introduce a new perspective, for adversaries controlling a compromised switch, to perform sophisticated and stealthy network reconnaissance. Detecting such network reconnaissance techniques is crucial to the security of the network, since the information gathered may be used to launch devastating attacks against the network.*

4. OVERVIEW

Identifying network reconnaissance is an important piece of a defense in depth approach the network security. The information gathered during network reconnaissance can lead to damage to the network and financial losses. Therefore, reconnaissance activity is a strong indicator of a network breech and allows for administrators to respond before an adversary can carry out the rest of their attack. In this section, we first give a walkthrough of a previously undetected network reconnaissance attack from a compromised switch in a software defined network, and then we propose an approach to detect this activity.

As described in Section 2 every packet not matching any flows installed in a switch's flow table will result in a switch sending a packet_in message to the controller, which is then processed by one or more applications. Once processed, the application sends a flow_mod to be installed in the switch's flow table dictating what should be done with the packet and all packets matching the same flow. Using this mechanism it is possible to infer information about the network policy from the responses to packet_in messages generated at a switch.

At a high level an adversary can perform network reconnaissance from a compromised switch by treating the security and routing applications as a black box, probing it with spoofed packets that generate packet_in messages and learning the network policy from the resulting flow_mod messages sent by the applications. To a naive application, these illegitimate packet_in messages are indistinguishable from any other packet_in message and processed normally.

Figure 3 shows how an adversary can use packet_in messages to discover the network connectivity in subnets whose

traffic does not flow through their compromised switch. In the following example we walkthrough how an adversary, who compromised switch S1, can determine if host H2 can connect to the Git server. First the adversary generates a spoofed packet with source address H2 and destination address of the Git server. In step 2, the adversary picks a spoofed source switch port for the message before sending a packet_in message to the controller in step 3. Once receiving this message, the controller notifies all apps listening for packet_in events (step 4). Applications then process the packet (step 5) and send control messages to the controller (step 6). After receiving a control message, the controller sends flow_mod messages to the switch installing the flow rule generated by the application (step 7). Finally, in step 8, the adversary determines the network policy applied for the spoofed packet from the flow_mod message. Optionally, an adversary can take a ninth step to prevent the packet from being propagated throughout the network. In step 9, the adversary drops the spoofed packet at the switch and refuses to install the flow rule generated by the application. This additional measure reduces the adversary's footprint within the network and gives a better chance at staying in the network undetected.

We now propose a solution to detect packet_in abuse for network reconnaissance of an SDN. First it is important to note this attack is made possible because the controller and the apps often trust the data reported by a switch and make no attempt to verify the legitimacy of the reported data. Therefore, if each packet received from a switch is verified, we can identify both a spoofed packet and the switch where the packet was generated, with a few exceptions we mention in Section 7.

The key identifying characteristic of network reconnaissance using packet_in abuse is the adversary must spoof the event source port for the packet_in message. Using this characteristic, it is possible to detect spoofed packet_in messages by verifying the event source port for packet_in a message is the expected event port for a given source IP address in a packet. If the event source port does not match the expected physical switch source port for a given packet then an alarm can be raised alerting an administrator a reconnaissance packet has been discovered and a switch is potentially compromised.

The approach described above adequately provides a detection for packet_in message abuse, but it does not come without its challenges. We briefly describe challenges faced by our approach here and then further address how we overcome these challenges in Section 5.

1. **Detection Accuracy:** Since reconnaissance attacks may send only a few probes per day or even week, it is crucial that all probes are accurately detected to alert administrators of a network intrusion. Additionally, if a large number of false positives are present legitimate reconnaissance probes may go unnoticed by administrators defeating the purpose of our defense.

2. **Detection Time:** The quicker our solution is able to detect probe packets and raise an alarm for network administrators, the sooner actions can be taken to prevent the adversary's reconnaissance from being used in an attempt to exfiltrate data from the network.

3. **Low Overhead:** Any defense implementing our proposed solution should not cause excessive overhead on network latency or throughput. If our solution is implemented in a real time network, even the smallest amount of overhead may deem our defense mechanism useless.

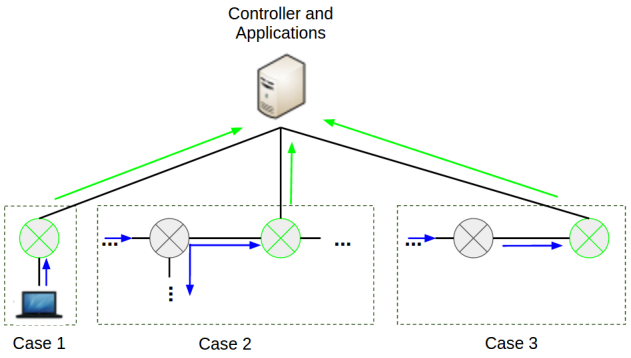

Figure 4: Three cases of packet_in messages being generated at a switch

5. DESIGN AND IMPLEMENTATION

This section introduces Network RACOONS, our proposed detection mechanism for network reconnaissance using packet_in abuse. First we discuss the high level design decisions for both Network RACOONS and the design of a packet_in abuse reconnaissance attack. The design will address the challenges listed at the end of Section 4 and briefly discuss the challenges and justify our implementation of a packet_in abuse reconnaissance attack. Further discussion about the trade-offs and their limitations will be covered in Section 7.

5.1 Design

Before getting into the low implementation details of Network RACOONS and the reconnaissance attack, we first examine the design of the pieces needed to properly implement Network RACOONS and perform a packet_in abuse network reconnaissance attack.

Network RACOONS: As described in Section 4 the key characteristic used to identify illegitimate packet_in messages is the mapping between the source IP and the physical switch the packet was received on. Also recall packet_in messages are generated by switches when an incoming packet does not match any flow installed in its flow table. This results in three cases in which a packet_in message can be generated at a switch. Each of these three cases must be handled differently in order to detect an illegitimate packet_in message. In Case 1, a host directly connected to a switch sends a packet not matching any flows in the switch's flow table and generates a packet_in message. This is the simplest case as a source IP can be directly mapped to a physical host attached to a switch's physical port.

Packets received from other switches make validation more complex. When a packet not matching any flows is received on a port connected to another switch, it is necessary to examine the previous switch to see how the packet was forwarded. The are two ways the previous switch may have forwarded the packet as described by Cases 2 and 3. In Case 2, the previous switch is instructed by the controller to flood a packet on all ports other than the port it was received on. Case 3 covers the situation where the previous

switch forwards a packet based on a match in its flow table.

Figure 4 visualizes each of these three cases using blue arrows to represent the flow of a packet and green arrows to represent the generation of a packet_in message from a switch (also colored in green) after receiving a packet that does not match any flows in its flow table.

In each case a packet_in message must be validated differently. Case 1 is trivial because it simply checks if the source IP matches the expected IP address of the host connected at that switch's physical port. Cases 2 and 3 do not have the luxury of easily tracing a packet to its origin. Instead of tracing a packet all the way back to its source hop by hop, the information at the previous switch may be used to validate an incoming packet. Validation of a packet_in message in Case 2 can be done with knowledge of packet_in messages generated by the previous switch. If an application used the controller's NBI to instruct a switch to flood a packet, then the packet_in messages at the next hop are considered legitimate. Similarly in Case 3, if the previous switch forwarded a packet using a rule from its flow table, then the presence of a flow installed on the switch can validate a packet_in message on the next hop.

Tracking the information needed to validate packet_in messages, such as topology information and a log of previous flow_mod messages, can already be tracked using existing applications; however Network RACOONS tracks this information without the need for a trusted third party application. This requires additional programming effort and computational overhead, but keeps the Trusted Computing Base minimalistic. To prevent high overheads on network latency and throughput, Network RACOONS processes each packet_in event in parallel with other applications. This allows other applications to quickly send control messages to the controller to install flows on a switch while Network RACOONS is validating the packet_in message.

Network RACOONS opts to detect network reconnaissance and raise alarms rather than preventing the attacks. This decision is two fold, first the cost of prevention is too high when considering false positives. False positives are discussed in more detail in Section 7, but an approach that isolates or reboots switches after an alarm is raised may cause severe service disruption to the network. Secondly, the damage done by network reconnaissance alone is not severe enough to justify taking preventative measures, recall network reconnaissance is an attack phase happening just before data exfiltration, so raising an alarm for manual analysis is an appropriate action.

Reconnaissance Attack: To ensure full coverage of the detection capabilities of Network RACOONS, the attack is constructed to spoof packets and their physical switch port to seem like the packets are coming from physical hosts connected to a switch and other switches. We simulate a reconnaissance attack using packet_in abuse by implementing a malicious app in the application plane to edit the event port of packet_in messages from certain IP ranges before they are processed by other applications and Network RACOONS. As discussed in the implementation, it is ensured the malicious app does not give any advantage to Network RACOONS in detecting attacks compared to attacks launched from a compromised switch.

5.2 Implementation

We now discuss the implementation of 1) the malicious SDN app used to simulate spoofed packets from a compromised switch that lead to fake packet_in messages being sent to the controller and 2) Network RACOONS, our network reconnaissance detection SDN app. The implementation is based on the design discussed in the previous section. We have used POX [11] as the SDN controller, which is a Python-based SDN controller, and both the malicious and detection SDN apps are implemented in python.

Malicious App: The malicious SDN app is implemented as a single python file. After being registered with the POX controller, it first reads the IP addresses of the host(s) being used to create the reconnaissance packets on behalf of a compromised switch. It then registers an event listener for packet_in messages from switches. For each packet_in event, it checks if the source IP address matches any IP in the malicious hosts list. If a match is found, the malicious app arbitrarily changes the event's physical port number to a random value, simulating a spoofed packet created from a compromised switch. The app then fires a custom event, using the capabilities of the event manager provided by POX, and forwards the changed packet information as the custom event argument. Other SDN apps running in the application plane register their handlers for the new custom event instead of the packet_in event and receive the new message from the malicious app. If the source IP is not matched to a malicious host then the malicious app does not change the event source port and passes the unmodified message to the other apps in the custom event. The apps listening for this custom event then process the packet and decide an action to be taken for the current message.

Detection App: The detection SDN app is implemented as a single python file in over 530 lines of source code. The detection app first registers with the controller and waits for the switches to connect to the controller by registering a handler for the connection_up event, which is raised whenever a new switch connects to the controller. When such an event occurs, the detection app uses the discovery module from POX to send control messages requesting switches broadcast LLDP packets. This allows the detection app to learn about the configuration of switches in the network and the links between different switches. The app then creates separate objects for each switch and maintains information about which physical ports on a given switch are connected to another switch.

The app also registers a handler for the custom event fired by the malicious app in response to any packet_in message from the switches. This handler maintains logic for the forwarding of packets and then also builds the mapping of hosts and switches to physical ports they are connected to for each of the switches. This mapping is referred to as the local map. When the forwarding logic instructs a switch to flood a packet, that information is added to an ephemeral flood map. This flood map is pruned on a fixed interval preventing the lookup from getting too large.

Using the data structures created from the forwarding logic, the handler asynchronously calls a validation function, passing the packet information to the validation function as an argument. The validation function extracts the flow information and looks at the local map and flood map to find any previous event that may lead to the generation of the current packet_in message. First looking at the local map, the app learns if the physical port, on which the new packet was received, is connected to another switch or a host. If

Table 1: Network Performance Overhead

Property	Baseline	Network RACOONS	Overhead
Bandwidth Achieved	19.98 ±.04 Mb/sec	19.55 ±.07 Mb/sec	2.21%
Average Latency	6.89 ±.31 ms	7.11 ±.28 ms	3.16%

Table 2: Accuracy of Network RACOONS (Row 4 with induced corner cases)

Packet_ins Inspected	Probes Detected	False Positives	False Negatives
1133	6	0	0
1744	11	0	0
2137	17	0	0
1108	13	2	0

the local map reports the event port is connected to a host it verifies the source IP matches the expected IP address for that switch's physical port. If the event port is connected to another switch, it then does a look up in its ephemeral flood map to see if the previous switch was instructed to flood the packet generating the packet_in message. If the lookup to the flood map fails, the app then uses the NBI to request flow statistics from the switches, which it listens for by registering a handler for a flow_stats_received event. Once the flow stats are received by the app, the validate function iterates through the flow entries on the previous switch looking for a flow that forwarded the packet to the current switch. If a matching flow cannot be found, the packet_in message is flagged along with the corresponding switch. During the processing of the packet_in messages the app does not block, so any other apps trying to perform operations in response to the event can go perform their tasks without interruption.

6. EVALUATION

We evaluated *Network RACOONS* by performing various experiments aimed at answering the following research questions.

1. **RQ1: Network Overhead** What is the overhead on the bandwidth and the latency introduced on the network by the detection app?

2. **RQ2: Accuracy** What is the accuracy of detecting attacks?

3. **RQ3: Detection Time** How long does it take for the detection app to detect a reconnaissance attempt after a fake packet_in is received?

We setup a small scale SDN environment to run our experiments. Our testing environment is built with 1 virtual switch (using OpenVSwitch), 1 HP 2920-24G SDN enabled switch, and 5 virtual machines (Ubuntu 16.04) running on 2 physical hosts (Ubuntu 16.04) using VirtualBox. The SDN controller was run on a Macbook Pro running OSX. Benign traffic was generated using 3 BeagleBone Blacks running Ubuntu, acting as low resource hosts. One of the three BeagleBone Blacks was used in the simulation of a compromised switch and created the probe packets that were changed by the malicious SDN app. The remaining two BeagleBone Blacks used a combination of D-ITG and iperf to generate traffic during testing.

Table 3: Detection Time

Flow_statistics Required	Average Detection Time
No	1.07 ±.11 milliseconds
Yes	72.3 ±9.2 milliseconds

Network Overhead: We used iperf to measure the bandwidth achieved with and without the detection app running in the presence of other benign traffic. Since the malicious app is not a part of the detection mechanism but is required to simulate fake packet_in messages, we ran the baseline with the malicious app running along with a forwarding app. The baseline bandwidth achieved was then compared to the bandwidth achieved with both the malicious and detection apps running. To measure the changes in network latency, ping was used between hosts while other benign traffic ran in the background. The overhead introduced by the detection app was around 2.2% in the achievable bandwidth and around 3% in the latency of the network as seen in Table 1.
Accuracy: We evaluated Network RACOONS for accuracy, specifically for false positives and false negatives. To test the accuracy, the same environment described above was used and benign traffic between the hosts was generated in the same way. While benign traffic was being generated, spoofed packets were created at the malicious host and the resulting packet_in messages were further modified by the malicious app. The number of packet_in messages processed by the detection app and the number of packet_in messages flagged as network reconnaissance packets were recorded for the duration of the experiments.

Table 2 shows the results of the accuracy testing. Rows 1 to 3 were run under normal configuration of the detection app, row 4 is discussed below. For each test of the normal configuration the detection app successfully detected spoofed packet_in messages with no false positives or false negatives. The duration of each test run was longer than the last, allowing the app to capture and inspect as close to 1,000, 1,500, and 2,000 packet_in messages as possible.

The experiment run in Row 4 of Table 2 simulates race conditions that may cause false positives. These conditions are very unlikely and did not appear during the testing of Network RACOONS. To ensure the conditions are met to cause a false positive, Network RACOONS was modified to send flow entries to switches with hard timeouts of less than .01 ms. Under this condition, when the next switch sends a packet_in message and the previous switch is queried, the corresponding flow will have expired from the switch. Using this delay, two situations were simulated. First a situation where a switch is overwhelmed and takes a longer time than the prune interval of the ephemeral flood map to generate a packet_in message for a packet flooded by a previous switch. Second it simulated the situation where a packet matches a flow, right before the flow expires and when the app checks for the installed flow it does not find it. Both of these simulated conditions lead to false positives. The results of testing with these induced corner cases can be seen in row 4 of Table 2.
Detection Time: Because our solution does not proactively block the reconnaissance or probe packets, it is important to know how much time it takes to detect such a packet and assess how much information may have been learned by the attacker on a compromised switch before being detected.

In order to measure the time, we record the time as soon a packet_in message is received by the detection app and subtract this time from the time when a message is classified as benign or malicious.

Table 3 shows the average detection time when the previous switch does not need to be queried is around 1 millisecond and in the case where the detection app has to request flow statistics from the previous switch, the average detection time is around 72.3 milliseconds. This shows the adversary has limited time to send a reconnaissance probe, analyze the result and launch an attack before their compromised switch is flagged.

7. DISCUSSION

In this section we discuss the limitations of our approach, including attacks against our solution, and the impacts of the trade-offs made during the implementation of Network RACOONS and the network reconnaissance attack simulation.

7.1 Limitations

We now focus on limitations of both the proposed network reconnaissance attack and the proposed solution at a high level, including how the assumptions of our threat model may impact real world applications in an SDN.

Fully Passive Adversaries: If an adversary controls a core switch in the network, he may monitor the network policy by passively viewing the switch's flow table and does not need to send reconnaissance packets. Thus Network RACOONS cannot detect the adversary in the network. We argue Network RACOONS is part of a defense in depth approach to network security, as it is not always guaranteed the adversary will be able to compromise a core switch and may need to launch an active reconnaissance attack. Similarly related, network administrators often configure all switches in their network in the same way and the same attack vectors may lead to compromise of more than one switch. With this in mind, we still argue for the need for Network RACOONS as an adversary is not always guaranteed to compromise more than one switch in a network.

VM Migration/Network Churn: Network RACOONS is not implemented to handle a high rate of network churn, i.e. virtual machine migration. One of SDN's advantages is the ease in which VM's can be migrated throughout the network, but in our current implementation we require the network to have very little network churn. Network RACOONS can be extended to allow for VM migration, but it may have an adverse effect on network overhead. This is left for future work.

Attack Feasibility and Effectiveness: We have already addressed the issue of a fully passive adversary. We also identify a situation where the attack does not effectively gather good intelligence of the network from the flow_mod message. Since the adversary is only guessing the network policy based on the resulting flow_mod messages, it is possible a forwarding decision may mislead an adversary into believing a host can connect to the desired destination. For example, a security application may want all suspicious looking connection attempts to divert into a honeynet or be forwarded to a blackhole that is more than one hop away. Since the adversary can only determine connectivity based on the next hop the information gathered from reconnaissance is effectively useless. If the blackhole or honeynet is directly connected to the compromised switch it is possible for the adversary to notice a difference in the expected output port and the output port specified in the flow rule. From this information the adversary can glean the position of middleboxes and other security defenses.

False Positives: As seen in Section 6, Network RACOONS successfully detected all attempted network reconnaissance with no false positives. In order to generate false positives, we intentionally introduced delays in Network RACOONS's processing or lowered timeout intervals. We note this delay can happen naturally in a few situations. If a non-compromised switch is overwhelmed or experiencing networking/processing delays, it is possible for the information of a previous switch used by Network RACOONS to expire before processing a packet_in message associated with that information. When our app goes to access that information, it is not found and the non-compromised switch is falsely identified as compromised. Similarly, flows are installed on a switch for a short period of time, if a flow expires directly after being used to forward a packet then the next switch will be mislabeled as malicious when its packet_in message is verified. In the first case we argue the alarm could be used to identify a misconfiguration in the network and the false positive can still provide useful information to network administrators. The race condition in the second case should be so rare, as we have not experienced during testing, that it will not cause too much of a hassle to inspect the offending packet manually.

Attacks on Network RACOONS: In addition to the false positives which can occur accidentally, we have identified one attack on Network RACOONS making it appear another switch is compromised. Note this attack does not allow an adversary to gather any intelligence about the network, but it may be used to hide their compromised switch in a large number of alerts generated by a non-compromised switch. According to our threat model the adversary has full knowledge of our solution and maintains root access on the compromised switch. Considering these assumptions, an adversary may launch an attack on neighboring switches to make them raise alarms and be tagged as malicious. This is accomplished by an adversary creating a packet and directly sending it out on a port connected to another switch. When the target switch receives the packet it generates a packet_in message to be processed by Network RACOONS. Since the packet was received on a port connected to a switch, Network RACOONS checks if the compromised switch either flooded the packet or used a flow table rule; however the packet was neither flooded or forwarded using flow table rules so the targeted switch is falsely reported as compromised. At worst, an adversary can only make neighboring switches generate false alarms. If a large number of false alarms are generated an administrator may still be alerted of potential malicious activity, but it is likely an adversary can use this hide his reconnaissance activity in a flood of false positives.

7.2 Trade-offs

Attack Implementation: In the interest of time, the attack was simulated using a malicious app on the controller rather than fully implementing malicious logic on the switch itself. The design of the app carefully ensured it did not introduce any characteristic differences making detection easier. The biggest effect introducing the malicious app has on

the evaluation is increased processing time at the controller. Before a packet_in message is processed by any application it is first modified by the malicious app, causing a short delay. Accounting for this, our baseline involved running the malicious app; however we did not explore the overhead introduced by the malicious app alone.

Timeout Thresholds: As mentioned in the limitations, we had to artificially introduce false positives, although these race conditions may happen naturally. The timeout thresholds, used by Network RACOONS to determine if a packet_in message with an event port connected to another switch is spoofed, may increase the likelihood of a race condition happening if they are set too low. In our experiments we did not see a performance overhead nor an increase of false positives when varying the timeout threshold. Future work may evaluate Network RACOONS under an increased load of packet_in messages while varying the threshold timeouts to see how it effects the network overhead and number of false positives. We note that an overwhelming number of packet_in events is very abnormal for a network, and stress testing in this manner may not apply to realistic environments.

Synchronous vs Asynchronous Processing: In its current implementation Network RACOONS processes packet_in events asynchronously with other applications running on the controller. This allows the adversary to receive the flow_mod message and gather intelligence about the network. In a synchronous approach, Network RACOONS would validate a packet_in message before other applications process the message. This has the obvious drawback of increased overhead on latency and network throughput, but prevents an adversary from gathering information using a spoofed packet if Network RACOONS drops the packet. If a prevention approach is taken, the cost of false positives become much higher than in a detection approach, although we note the conditions necessary to produce a false positive rarely occur. In the event of a false positive, the connection may be retried from the host, increasing the latency seen by the two hosts. In the very unlikely case, the same connection would continuously encounter the race condition and lose its service in the network. Since detection satisfies our needs, we opted for just detection and avoided unnecessary overhead.

8. RELATED WORK

Network Reconnaissance [15, 3] using various techniques has been identified as a threat to networks as it empowers an adversary to gather crucial information about the network, the devices present in the network, and the possible vulnerabilities that can be used for further attacks or intrusions against the network. Prior work has mostly focused on looking at the different reconnaissance techniques used in traditional networks. Network scanning using *Nmap* [13] has been discussed as a threat to the network [3, 2, 1] as it allows an adversary to learn about open ports on and liveliness of hosts in a network. Allman et al. [1] studied the onset of network scanning in the late 1990's and its evolution in terms of characteristics such as the number of scanners, targets, and probing patterns. Their work provided a preliminary examination of the network scanning phenomenon. Anbar et al. [2] presented an investigative study on network scanning techniques and identified many existing scanning methods. They also discussed how malicious code adopts scanning methods to find vulnerable hosts and services and explored the current approaches to detect the presence of these scanning methods in networks. Shaikh et al. [15] discussed different ways of performing network reconnaissance using probes and presented the general characteristics of such probes being used in the wild. In another work, Luckie et al. [12] used *traceroute* under different IP protocols (TCP, UDP, ICMP,...) and analyzed the routes taken when connecting to a set of routable IP addresses. In this work they identified a list of known routers and well-known websites and explored how the paths taken by a probe varied with the choice of the protocol used for a probe. Templeton et al. [18] observed adversaries tend to use spoofed packets for network reconnaissance and discussed a wide variety of methods for detecting spoofed packets, including both active and passive host-based methods as well as the more commonly discussed routing-based methods. Xiaobing et al. [19] also looked at this problem and analyzed the existing scanning methods to draw the conclusion that the then present detection and protection of scanning mainly aimed at information concealment. They then presented a novel system of the detection and protection named IEDP.

Although prior work has focused on the dangers of network reconnaissance on networks, they have been mostly targeted on the traditional networks. With the increasing popularization of Software-Defined Networks [7, 4], it's important to look at the similar problems in SDN and how it allows for previously unseen ways of performing active network reconnaissance, which were infeasible in traditional networks. There have been a number of proposals using SDN to enhance network security. Avant-Guard [16] specifically focuses on addressing the communication bottleneck between the control and data planes by identifying malicious traffic, including network scanning and denial-of-service. Song et al. [17] were the first to look at scanning attacks in SDN and proposed a solution by reactively hiding critical resources in response to such attacks. The authors implemented a security service that responds to network scanning predefined policies by redirecting attackers to a honeynet and confuse attackers by providing fake scanning results.

The idea of network devices, such as switches, being compromised and used for attacks against a Software-Defined Network was discussed in 2015 [6, 9]. Hong et al. [9] studied the popular and major SDN controllers and found that a lot of them are subject to the Network Topology Poisoning Attacks. They investigated mitigation methods against Network Topology Poisoning Attacks and presented *TopoGuard*, which is a security extension to SDN controllers and provides automatic and real-time detection of Network Topology Poisoning Attacks. Dhawan et al. [6] focused on several attacks targeting SDN controllers that violate network topology and data plane forwarding, and can be mounted by compromised network entities, such as end hosts and soft switches. They presented *SPHINX*, which uses flow graphs and a custom policy language, to detect both known and potentially unknown attacks on network topology and data plane forwarding originating within an SDN environment.

These previous works have only focused on detecting spoofed packets from physical hosts and do not consider a compromised switch generating spoofed packets. Since packets from a host are only checked on the switch they are directly connected to and not at each hop in the network, previous approaches are unable to detect packets generated at a switch.

Making detection even more difficult, the probes packets in our attack are not propagated to other switches or hosts in the network thus detection lies solely on identifying the probe packets as they are first made. Our work extends the ideas of network reconnaissance in SDN and looks at it from a different perspective. We identify how an adversary can leverage the architecture of SDN to launch network probes from a compromised switch in ways not seen in traditional networks and undetectable by previous SDN security application. We then present a way to detect these reconnaissance attacks and implement the design for an SDN app that can perform this detection.

9. CONCLUSION

In this work we introduced a novel network reconnaissance attack that exploits the mechanisms and architecture of a Software Defined Network, giving adversaries an enhanced view of a network compared to reconnaissance attacks in traditional networks. We then proposed a defense mechanism, Network RACOONS, which successfully detected our proposed attack with less than 3% overhead on both network latency and network throughput while observing no false positives or false negatives. Therefore, we believe Network RACOONS is an integral piece to a defense in depth approach to securing SDN and the efforts in this paper motivate future research exploring new vulnerabilities in the quickly evolving field of SDN.

10. REFERENCES

[1] M. Allman, V. Paxson, and J. Terrell. A brief history of scanning. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 77–82. ACM, 2007.

[2] M. Anbar, A. Manasrah, S. Ramadass, A. Altaher, A. Aljmmal, and A. Almomani. Investigating study on network scanning techniques. *International Journal of Digital Content Technology and its Applications*, 7(9):312, 2013.

[3] R. J. Barnett and B. Irwin. Towards a taxonomy of network scanning techniques. In *Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology*, pages 1–7. ACM, 2008.

[4] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: taking control of the enterprise. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 1–12, Aug 2007.

[5] M. De Vivo, E. Carrasco, G. Isern, and G. O. de Vivo. A review of port scanning techniques. *ACM SIGCOMM Computer Communication Review*, 29(2):41–48, 1999.

[6] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann. Sphinx: Detecting security attacks in software-defined networks. In *NDSS*, 2015.

[7] N. Feamster, J. Rexford, and E. Zegura. The road to sdn: an intellectual history of programmable networks. In *ACM SIGCOMM Computer Communication Review*, volume 44(2), pages 87–98. ACM, Apr 2014.

[8] O. N. Foundation. Openflow switch specification, version 1.5.1. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf, 2015. [Online; accessed 10-November-2016].

[9] S. Hong, L. Xu, H. Wang, and G. Gu. Poisoning network visibility in software-defined networks: New attacks and countermeasures. In *NDSS*, 2015.

[10] Infosecurity. Hackers spend 200+ days inside systems before discovery. http://www.infosecurity-magazine.com/news/hackers-spend-over-200-days-inside/.

[11] O. N. Lab. Pox controller wiki. https://openflow.stanford.edu/display/ONL/POX+Wiki, 2015. [Online; accessed 10-November-2016].

[12] M. Luckie, Y. Hyun, and B. Huffaker. Traceroute probe method and forward ip path inference. In *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, pages 311–324. ACM, 2008.

[13] G. F. Lyon. *Nmap network scanning: The official Nmap project guide to network discovery and security scanning*. Insecure, 2009.

[14] Phrack. Distributed information gathering. hhttp://phrack.org/issues/55/9.html#article.

[15] S. A. Shaikh, H. Chivers, P. Nobles, J. A. Clark, and H. Chen. Network reconnaissance. *Network Security*, 2008(11):12–16, 2008.

[16] S. Shin, V. Yegneswaran, P. Porras, and G. Gu. Avant-guard: Scalable and vigilant switch flow management in software-defined networks. In *ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 413–424, New York, NY, USA, Nov 2013. ACM.

[17] Y. Song, S. Shin, and Y. Choi. Network iron curtain: Hide enterprise networks with openflow. In *International Workshop on Information Security Applications*, pages 218–230. Springer, 2013.

[18] S. J. Templeton and K. E. Levitt. Detecting spoofed packets. In *DARPA Information Survivability Conference and Exposition, 2003. Proceedings*, volume 1, pages 164–175. IEEE, 2003.

[19] G. Xiaobing, Q. Depei, L. Min, Z. Ran, and X. Bin. Detection and protection against network scanning: Iedp. In *Computer Networks and Mobile Computing, 2001. Proceedings. 2001 International Conference on*, pages 487–493. IEEE, 2001.