SDN-GUARD: Protecting SDN Controllers Against SDN Rootkits

Dennis Tatang, Florian Quinkert, Joel Frank, Christian Röpke, and Thorsten Holz
Horst Görtz Institute for IT-Security (HGI)
Ruhr-University Bochum, Germany
Email: {firstname.lastname}@rub.de

***Abstract*—Software-defined networking (SDN) addresses pressing networking problems such as network virtualization and data center complexity. By separating the control plane from the data plane, SDN introduces a new abstraction layer. This new abstraction layer is typically implemented by means of a so-called *SDN controller*. SDN applications can interact with the controller to ensure network functionality. This new paradigm has multiple advantages, particularly in the fields of network automation and security. Recent work, however, has shown that existing SDN solutions lack adequate security properties; in particular, SDN rootkits allow attackers to take over entire networks by compromising SDN controllers.**

In this paper, we present SDN-GUARD, a novel system for detecting and mitigating SDN rootkits. The basic idea is to perform a dual-view comparison that detects malicious network programming attempts. An evaluation of our system demonstrates both its effectiveness and its flexibility in terms of application, along with its relatively small performance overhead.

I. INTRODUCTION

The *software-defined networking* (SDN) paradigm enables automation and flexibility, two important qualities that are necessary to solve problems currently encountered in operating networks [1]. Therefore, SDN decouples the *control plane*, in which control programs decide where to forward network packets, from the *data plane*, in which the forwarding hardware simply carries out these decisions. Before the development of SDN, such control programs were tightly coupled with each device distributed over the network. In contrast, SDN focuses on providing networking functionality by employing a logically centralized SDN controller that communicates with programmable network devices via a so-called *southbound interface* and with SDN applications via what is commonly referred to as a *northbound interface*. An example of a successful implementation of the southbound interface is the OpenFlow protocol [2], which is one of the key SDN boosters. Using this protocol, SDN controllers can provide a global view of a network, including network statistics, to SDN applications, while such applications can re-program the network based on the information provided. The major advantage of SDN is the fact that third-party control programs are able to make networking decisions in software that is solely programmed to the hardware through open interfaces.

In fact, the industry already benefits from SDN: examples of its implementation include link utilization, the reduction of operational costs [3], and improved network security [4]. The improvements it offers include dynamic flow control, which makes it possible to easily separate malicious from benign network traffic; network-wide visibility, which allows for simplified network-wide monitoring; easy development of network security applications through network programmability; and simplification of the data plane, making the addition of lightweight security modules easier on this layer. However, an inherent property of SDN is the ability of SDN controllers to govern an entire network; this, apparently, makes it an attractive target for attackers. Recently, an SDN rootkit [5] has been presented that subverts an open-source SDN controller (i. e., OpenDaylight [6]) in order to provide remote control to an attacker. As OpenDaylight aims to provide a reference framework for enterprise SDN controllers, this rootkit can potentially take over a large variety of controllers. In order to do so, it uses Java reflection to subvert critical controller functions; it is commanded remotely from an infected host that resides in the data plane.

In this paper, we propose SDN-GUARD which is an SDN-rootkit detection and mitigation system. It is based on the observation that one of the primary characteristics of SDN rootkits is to conceal their attempts to adversely re-program a network, e. g., by adding malicious flow rules. To effectively conceal itself, an SDN rootkit must suppress such manipulations from the SDN controller's view of the network. Such rootkits have a high demand on hiding malicious flow rules that have already been added. Without such attempts at concealment, a monitoring application or an administrator using a visualization application could easily detect such unwanted network manipulations. From the OpenFlow-based network device's perspective, the state of a network is represented by flow rules and corresponding statistics, which are stored in the network appliance. Therefore, reprogramming the network implies that flow rules must be added, modified, or deleted. Thus, the controller's view of the network provided through the northbound interface and the actual network state provided by network devices through the southbound interface are distinguishable. We take advantage of this observation by comparing these two views in order to reveal hidden network manipulations. Consequently, our system is capable of protecting against SDN rootkits from manipulating networks in an adverse manner.

To summarize, our main contribution tackles the problem of SDN rootkits as a whole. We propose a new protection system called SDN-Guard. Utilizing a dual-view comparison, it is capable of detecting and mitigating SDN rootkits, regardless of the technique used to subvert SDN controllers. We test our prototype implementation of SDN-Guard using different SDN rootkits and multiple SDN controllers in order to demonstrate its generic benefits. Our evaluation also demonstrates that we can protect SDNs with only a relatively small overhead.

II. Background

Before we go into detail about SDN-Guard, we recap the importance of SDN controllers and the security mechanisms intended to protect them. In addition, we survey existing SDN malware and explain the importance of securing against such attacks.

A. SDN Controllers

By design, SDN controllers have absolute control over a network. They communicate with their assigned switches via an (optionally encrypted) command channel and implement network actions such as packet switching and routing with the aid of SDN applications. This is mainly achieved by providing a global view of the network (including flow rules within network devices and corresponding statistics) to SDN applications, which are thereby able to make networking decisions. It is important to stress that this unique characteristic makes SDN controllers the highest priority target in SDN-based networks.

In order to protect SDN controllers from misbehaving or malicious SDN applications, several mechanisms have been proposed. Most notably, SDN application sandboxing makes it possible to restrict access to critical controller functions such as exiting the controller process or manipulating internal data structures [7]–[9]. In addition, flow rule checking makes it possible to detect dynamic flow rule tunneling and network invariants such as loops or black holes. This can be achieved from both outside of the SDN controller process [10], [11] and from within [11]–[13]. Furthermore, network event checking helps to guard the network topology from, e. g., poisoning attacks [14], [15]. Moreover, SDN application analysis [16], [17] provides relevant insights before its installation, and implementing a TCP proxy within network devices enables mitigating the impact of denial-of-service attacks that target SDN controllers from within the data plane [18]. Another important security feature, which is also used in modern operating systems, is signing SDN applications in order to avoid the installation of malicious software [19].

B. SDN Malware

Researchers already demonstrated possible attacks through malicious and misbehaving SDN applications. In the following, we give a brief overview of some. Porras et al. presented a new kind of attack that allows evading already installed flow rules [12], [13]. The authors show a way to reach a network host, even though an existing rule explicitly prohibits such connections. Röpke et al. introduced an SDN rootkit for OpenDaylight [5]. This SDN rootkit is able to manipulate internal data structures and can gain control of components responsible for programming the network. In addition, it can read the internal network state. The authors confirm with a proof-of-concept implementation that a potential attacker can add and conceal malicious flow rules, as well as remove legitimate flow rules, without these changes being visible to an administrator. Further malicious or misbehaving SDN applications abuse critical operations on the system level to damage SDN controllers [7], [8], [20]. These papers emphasize a great need for security mechanisms.

III. SDN-Guard

We now present SDN-Guard, our generic and modular defense mechanism intended to protect against SDN rootkits. First, we present our attacker model. Thereafter, we provide an introduction to our approach, which is followed by a description of our architecture. Finally, we describe our proof-of-concept implementation.

A. Attacker Model

Throughout the rest of this paper, we assume an attacker model similar to the situation described in Section II-B: An attacker has successfully compromised an SDN controller by installing a rootkit. He intends to change the network state, e. g., in order to extract information, and, at the same time, alter the controller's view to conceal his actions from network operators and monitoring applications.

B. Approach

In such a scenario, there exists a detectable difference between the SDN controller's view of the network and the actual network state. We exploit this observation in SDN-Guard to detect maliciously added and deleted flow rules. We generate two network views, the *controller's view* and the *network's view*, and compare them with each other. We collect the controller's view directly from the SDN controller's northbound interface by asking the controller for the state of the network. The network view is generated by observing the control channel between the controller and the switches, which we scan for *flow_mod* messages in order to register changes in the network. When SDN-Guard detects malicious behavior, it automatically restores maliciously removed flow rules or deletes maliciously added flow rules. This automated approach guarantees a rapid reaction and minimizes the impact of an SDN rootkit.

C. Architecture Overview

The following section describes SDN-Guard's architecture. The tool consists of two concurrently operating components: the *proxy unit* and the *decision unit*. The *proxy unit* acts as a reverse proxy between the controller and the switches. It proxies all of the command traffic and extracts flow_mod messages for review. The *decision unit* then compares this network view with the controller's view in order to detect malicious changes.

Figure 1 illustrates SDN-Guard's workflow. When receiving new command traffic (1), SDN-Guard's *proxy unit*

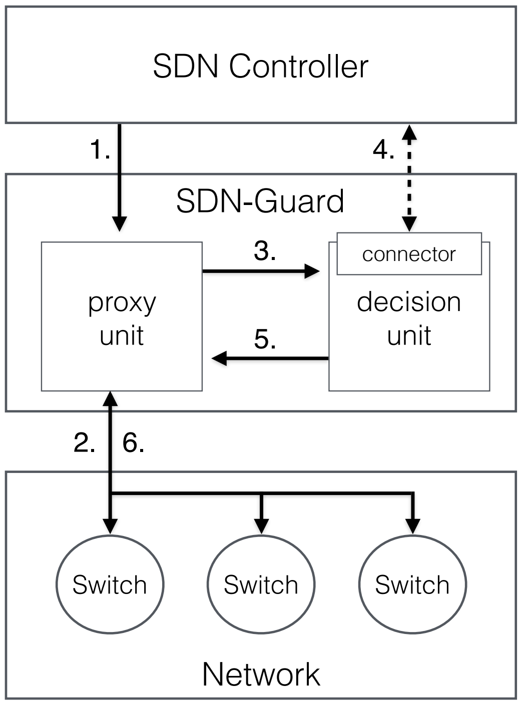

Figure 1. SDN-GUARD Architecture

forwards it to the corresponding switch (2) and, in the case of flow_mod messages, sends a copy to the *decision unit* (3). Since the vast majority of flow mod commands are valid, rapid forwarding of the flow mod commands to the switches is desired. Additionally, our evaluation in Section IV shows that SDN-GUARD is able to both rapidly detect malicious flow mod commands and undo their changes. The *decision unit* requests the SDN controller's view (5) and compares it with the network view observed by the *proxy unit*. In addition, it includes a control plane *connector* module, which can be adjusted for different SDN controllers. If the *decision unit* detects a difference, it notifies the *proxy unit* (5), which then generates flow mod commands to re-establish the originally desired network structure (6).

This design offers multiple benefits: SDN-GUARD intercepts every flow mod command sent by an SDN controller and therefore ensures by its design that each and every malicious flow mod command will be detected. It is possible to deploy SDN-GUARD on a separate host from the SDN controller in order to ensure that it functions correctly, even though the SDN controller's underlying host may be compromised. SDN-GUARD is even able to intercept flow mod commands sent via an encrypted connection between SDN controller and the switches, provided that the necessary keys are made available. Furthermore, SDN-GUARD is independent of a concrete SDN controller.

D. Implementation Details

We implemented SDN-GUARD in Go as a proof-of-concept in order to verify its feasibility and effectiveness. The data plane connector currently supports OpenFlow 1.3, using the gopenflow library [21]. Note, however, that, due to our modular design, other libraries, e.g., those that support different OpenFlow versions, can also be utilized. To further demonstrate this modularity, we also added support for three different SDN controllers: one proprietary, the HP VAN SDN controller, and two well-known open-source controllers, namely Opendaylight and Ryu. It is important to note that we only had to replace the module that contacts the SDN controller, thus demonstrating the easy portability of SDN-GUARD between the various SDN controllers. Since we did not want to spam the controller with requests, we implemented the comparison process as a loop. While one comparison task was being processed, newly received flow rules were stored in a buffer and handled in the next iteration. This allowed us to process updates in batches. Thus, we only needed to request and compare a subset view of the network, i.e., the view of all of the switches that had received new rules during a particular round, which significantly reduced the workload of SDN-GUARD.

IV. EVALUATION

The following section describes the performed evaluation. First, we present our evaluation setup, which is followed by a description of our test cases; finally, we describe the results of our evaluation.

A. Evaluation Setup

We performed our evaluation on a server system with two Intel Xeon X5650 CPUs (each of 2.67GHz) and 48 GBytes of RAM. The server emulated two virtual machines: one running an SDN controller and our SDN-GUARD and a second simulating an OpenFlow network, employing the popular network simulator Mininet [22]. For our evaluation, we configured Mininet with a tree topology of depth six, leading to a network with 63 switches.

B. Test Cases

We perform the following test cases to both demonstrate SDN-GUARD's performance and general applicability:

1) Performance test: SDN-GUARD was required to detect and delete malicious flow rules as quickly as possible in order to effectively protect SDNs. We used the HP VAN SDN controller, version 2.7.10, installed 10,000 malicious flow rules within 1,000 seconds using a rootkit-simulating application, and measured two time intervals. First, we measured the time elapsed from sending a malicious flow rule until it was detected by SDN-GUARD and the corresponding delete flow rule was sent. We refer to this interval as *detection time*. Second, we measured the processing time in the *proxy unit*, which we call *delay time*. We performed this test with various numbers of pre-installed flow rules on the switches. By default, each switch has eight flow rules installed; we refer to this set of rules as the *initial set*. In addition, we conducted the test with 250, 500, 750 and 1,000 pre-installed rules in order to demonstrate that SDN-GUARD remains reliable even when it has to perform a large number of comparisons.

2) General Applicability: We demonstrated the general applicability of SDN-GUARD by further evaluating it using three different controllers and installed rootkits or applications that simulate rootkit behavior:

- HP VAN SDN controller (version 2.7.10) with a rootkit simulating application;
- OpenDaylight (ODL) SDN controller (version helium sr1) and the rootkit proposed by Röpke et al. [5]; and
- Ryu SDN controller (version 4.4) and an SDN application that simulates rootkit behavior.

In our test cases, we sent both malicious and benign flow rules to each connected switch. We then verified whether SDN-GUARD correctly detected and deleted all of the malicious flow rules by checking if a delete flow rule was sent for each malicious flow rule sent. Thus, we automatically verified whether or not SDN-GUARD produced any false positives, i. e., whether it generated delete flow rules in response to benign flow rules sent over the network.

C. Results

The following sections describe the results of both the performance and general applicability tests.

1) Performance: Table IV-C1 shows the mean, variance and standard deviation for *detection time* and *delay time* as a function of the number of pre-installed flow rules. The mean of the *detection time* scales linearly with the number of pre-installed flow rules since the number of comparisons increases. This behavior is desirable since it shows that the approach scales for even larger numbers of pre-installed rules acceptable.

In contrast, the mean of the *delay time* is very small and is independent of the number of pre-installed flow rules; it converges, as indicated by the decreasing variance over time, towards an almost constant value. This underlines the small overhead of our implementation.

2) General Applicability: During the execution of the general applicability test, we verified that each maliciously added flow rule was correctly deleted. Additionally, we demonstrated that SDN-GUARD can be used with three different controllers.

V. LIMITATIONS AND DISCUSSION

SDN-GUARD can effectively reveal hidden flow rules as long as a rootkit is unable to identify SDN-GUARD's calls to the controller's northbound interface; in such a case, the rootkit could provide our SDN-GUARD with the correct network state and other applications with the manipulated one. However, this does not seem to be feasible in practice, as we only used standard northbound calls, which would be utilized by all other applications. A rootkit could potentially correlate such calls with a previously sent flow_mod message, which would be subject to review by SDN-GUARD. Theoretically, this could make it possible to identify our tool, if, for instance, no other SDN application is installed. In a real-world scenario, however, we claim that such a method of identification is not feasible since other SDN applications would use the same northbound call to a significant extent, and each of them would send flow_mod messages independently.

Our current implementation assumes that SDN controllers know the current network state when it is requested. Therefore, the HP controller sends flow-stats requests to switches on demand while ODL manages an internal flow rule database representing the current network state. In case SDN controllers implement an internal flow rule database which periodically receives network state messages (e. g., like for newer versions of ONOS and ODL), we must take the corresponding timing into account. For example, SDN-GUARD needs to wait until network modifications via flow-mod messages are included in state update messages.

SDN-GUARD also depends on a clean operating system. In case of a compromised system, an SDN rootkit would be able to subvert not only the controller's process but also the underlying operating system; it could also manipulate SDN-GUARD's processes. While it should be noted that none of the currently known malicious SDN applications can do so, there is a theoretical chance that this could happen in the future. In order to avoid such attacks, SDN-GUARD could be run on other hardware, including other operating systems. As this implies greater delays for sending and receiving network packets via the control channel, such other hardware should be placed near to the SDN controller in order to minimize the delay.

VI. FUTURE WORK

While we have demonstrated SDN-GUARD's ability to adequately protect large SDN networks, it operates reactively. Thus, attackers still have a small, albeit negligible, window of opportunity during which their actions can take effect. It would be desirable to remove this window in order to enhance SDN-GUARD's protective capabilities and transform it into an active protection mechanism; the current design could be reused for this purpose. We could introduce new functionality by intercepting flow_mod messages. In contrast to directly passing such a message, our tool can postpone its transmission to the network until SDN-GUARD has confirmed its authenticity. However, adopting such functionality is more challenging than it might seem at first. We do not change the network state; thus, when the controller is asked for its view, it will correctly return an empty network state. SDN-GUARD would conclude that a rootkit is hiding these rules and drop all changes to the network. We can circumvent this by modifying the flow_stats messages that are used by controllers to request the state of a switch. We could modify these messages according to the flow_mod messages awaiting approval, thus removing or adding dummy entries to the flow_stats messages. Thus, the controller, or a malicious rootkit, is led to believe that its changes took effect. As a result, we could stop SDN

# Rules	*detection time* Mean	Variance	Std.	*delay time* Mean	Variance	Std.
initial	37.06 ms	91.22 ms	9.55 ms	0.02 ms	0.57 ms	0.75 ms
250	51.07 ms	96.51 ms	9.82 ms	0.32 ms	0.67 ms	0.82 ms
500	63.84 ms	114.44 ms	10.70 ms	0.28 ms	0.38 ms	0.62 ms
750	76.02 ms	123.87 ms	11.13 ms	0.32 ms	0.61 ms	0.78 ms
1000	88.68 ms	126.34 ms	11.28 ms	0.31 ms	0.57 ms	0.75 ms
1500	100.18 ms	88.56 ms	9.41 ms	0.12 ms	0.04 ms	0.20 ms
2000	134.81 ms	240.46 ms	15.50 ms	0.12 ms	0.04 ms	0.21 ms

Table I
Detection time AND *delay time* AS A FUNCTION OF THE NUMBER OF PRE-INSTALLED FLOW RULES

rootkits before the network is manipulated by re-using the proposed architecture. We have almost finished a prototype of this design. However, it can only function for a relatively small-sized network of 400 rules per switch (assuming 63 switches) and still requires some engineering work.

VII. Related Work

There exist only a few studies on protecting SDNs against malicious applications such as SDN rootkits. The main countermeasures addressing this threat are sandbox approaches [7], [8], [23] and rule conflict resolution mechanisms [12], [13]. But, all of these approaches run inside an SDN controller and might be deceived by SDN malware such as our rootkit implementation. In case a rootkit is able to subvert an SDN controller appropriately, such countermeasures may not be able to achieve their security goals. To avoid a cat and mouse game, we focus on a mechanism independent of the SDN controller process. Policy checking can help here as it includes operating from the outside of SDN controllers [10], [24]. However, the main focus of current policy checkers [11], [25] is not on protecting networks against SDN rootkits but on finding network invariants such as forwarding loops and black holes. In addition, policy checkers as mentioned earlier depend on a complete set of security policies. In case they are missing, an attacker can exploit this by manipulating the network programming despite an active policy checker. In practice, policy checkers have to be maintained and configured, which might be error prone. However SDN-Guard avoids these issues, it can be deployed quickly and does not need to be maintained beyond that point.

VIII. Conclusion

In this paper, we tackled the problem of SDN rootkits in general, and we discussed both the architecture and implementation of a tool called SDN-Guard. When compared to existing work, SDN-Guard demonstrates considerable advantages in terms of detecting and mitigating the effects of SDN rootkits. Our evaluation shows that our approach can be implemented in a robust, flexible, and efficient way.

Furthermore, we extended our design to an active defense mechanism that evaluates flow rules before they come into effect. Initial tests have already demonstrated its feasibility, and we intend to refine this concept in the future.

Acknowledgments

This work has been performed in the framework of the CELTIC EUREKA project SENDATE-PLANETS (Project ID C2015/3-1), and it is partly funded by the German BMBF (Project ID 16KIS0469). The authors alone are responsible for the content of the paper.

References

[1] N. Feamster, J. Rexford, and E. Zegura, “The Road to SDN,” *ACM Queue: Tomorrow’s Computing Today*, 2013.
[2] Open Networking Foundation, “OpenFlow Switch Specification,” www.opennetworking.org.
[3] U. Hölzle, “OpenFlow @ Google,” Open Networking Summit, 2012.
[4] S. Shin, L. Xu, S. Hong, and G. Gu, “Enhancing Network Security through Software Defined Networking (SDN),” in *Proceedings of The 25th International Conference on Computer Communication and Networks (ICCCN’16)*, August 2016.
[5] C. Röpke and T. Holz, “SDN Rootkits: Subverting Network Operating Systems of Software-Defined Networks,” in *Symposium on Recent Advances in Intrusion Detection*, 2015.
[6] OpenDaylight Project, www.opendaylight.org.
[7] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang, “Rosemary: A Robust, Secure, and High-Performance Network Operating System,” in *ACM SIGSAC Conference on Computer and Communications Security*, 2014.
[8] C. Röpke and T. Holz, “Retaining Control Over SDN Network Services,” in *International Conference on Networked Systems*, 2015.
[9] K. SRI International, “Security-Mode Open Network Operating System,” wiki.onosproject.org/display/ONOS/Security-Mode+ONOS.
[10] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, “Real Time Network Policy Checking Using Header Space Analysis,” in *USENIX Symposium on Networked Systems Design and Implementation*, 2013.
[11] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey, “VeriFlow: Verifying Network-Wide Invariants in Real Time,” in *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2012.
[12] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, “A Security Enforcement Kernel for OpenFlow Networks,” in *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2012.
[13] P. Porras, S. Cheung, M. Fong, K. Skinner, and V. Yegneswaran, “Securing the Software-Defined Network Control Layer,” in *Symposium on Network and Distributed System Security*, 2015.
[14] S. Hong, L. Xu, H. Wang, and G. Gu, “Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures,” in *Symposium on Network and Distributed System Security*, 2015.
[15] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, “SPHINX: Detecting Security Attacks in Software-Defined Networks,” in *Symposium on Network and Distributed System Security*, 2015.
[16] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford, “A NICE Way to Test OpenFlow Applications,” in *USENIX Symposium on Networked Systems Design and Implementation*, 2012.
[17] C. Lee and S. Shin, “SHIELD: An Automated Framework for Static Analysis of SDN Applications,” in *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, 2016.

[18] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, “AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-Defined Networks,” in *ACM Conference on Computer and Communications Security*, 2013.

[19] Hewlett-Packard, “HP VAN SDN Controller,” www.hp.com.

[20] C. Röpke and T. Holz, “On Network Operating System Security,” *International Journal of Network Management*, 2016.

[21] Hiroaki Kawai, “gopenflow,” github.com/hkwi/gopenflow.

[22] B. Lantz, B. Heller, and N. McKeown, “A Network in a Laptop: Rapid Prototyping for Software-Defined Networks,” in *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2010.

[23] X. Wen, Y. Chen, C. Hu, C. Shi, and Y. Wang, “Towards a Secure Controller Platform for OpenFlow Applications,” in *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2013.

[24] C. Röpke, “SDN Malware: Problems of Current Protection Systems and Potential Countermeasures,” in *Sicherheit*, 2016.

[25] P. Kazemian, G. Varghese, and N. McKeown, “Header Space Analysis: Static Checking for Networks,” in *USENIX Symposium on Networked Systems Design and Implementation*, 2012.