Preventing Malicious SDN Applications From Hiding Adverse Network Manipulations

Christian Röpke
Ruhr-University Bochum
christian.roepke@rub.de

Thorsten Holz
Ruhr-University Bochum
thorsten.holz@rub.de

ABSTRACT

In Software-Defined Networks (SDN), so called *SDN controllers* are responsible for managing the network devices building such a network. Once such a core component of the network has been infected with malicious software (e.g., by a malicious SDN application), an attacker typically has a strong interest in remaining undetected while compromising other devices in the network. Thus, hiding a malicious network state and corresponding network manipulations are important objectives for an adversary. To achieve this, rootkit techniques can be applied in order to manipulate the SDN controller's view of a network. As a consequence, monitoring capabilities of SDN controllers as well as SDN applications with a security focus can be fooled by hiding adverse network manipulations.

To tackle this problem, we propose a novel approach capable of detecting and preventing hidden network manipulations *before* they can attack a network. In particular, our method is able to drop adverse network manipulations before they are applied on a network. We achieve this by comparing the actual network state, which includes both malicious and benign configurations, with the network state which is provided by a potentially compromised SDN controller. In case of an attack, the result of this comparison reveals network manipulations which are adversely removed from an SDN controller's view of a network. To demonstrate the capabilities of this approach, we implement a prototype and evaluate effectiveness as well as efficiency. The evaluation results indicate scalability and high performance of our system, while being able to protect major SDN controller platforms.

Keywords

Software-defined networking; SDN controller security; malicious SDN applications

© 2018 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

1. INTRODUCTION

Malicious SDN applications are capable of subverting SDN controllers in a way adverse network manipulations can be hidden. As a result, corresponding attacks remain undetected [22, 20]. For example, an attacker can add malicious flow rules to a network and use rootkit techniques to hide them from the global network view, which is provided by SDN controllers. In the same manner, legitimate flow rules can be adversely removed while they remain visible within the SDN controller's view of a network.

This problem is currently tackled by either sandbox mechanisms or policy checking methods. In case of sandbox systems, an SDN application is put into a sandbox to deny critical operations which result, for example, in unwanted SDN controller modifications [23, 1, 21]. For example, an SDN application can be prevented from executing critical operations, which are also known to be used by SDN rootkits. Since current rootkits abuse via Java reflection and Aspect-Oriented Programming (AOP) capabilities, corresponding critical operations can be easily denied. However, as critical operations are intended to be used in a benign way, we have to distinguish between their legitimate and adverse use. As current sandbox systems do not provide such a security mechanism, these systems are limited regarding their overall protection capabilities. In addition, critical operations—which can also be misused in a malicious way—must be known in advance in order to take advantage of sandboxing. As several studies indicate that rootkit techniques other than via Java reflection and AOP are possible [16, 17, 5], current sandbox systems face another limitation.

In case of policy checking methods, a network programming attempt is validated whether it violates a given security policy or not [19, 18, 9, 8, 7]. For example, a flow rule which performs a dynamic flow tunneling attack via OpenFlow's `set` and `goto` instructions can be dropped before it is programmed into a network. Similarly, flow rules which would introduce loops or black holes, or grant guest hosts unwanted access to an isolated network segment, can be prevented. However, in case such a system works as part of a compromised SDN controller, an attacker may be able to bypass this protection mechanism. With respect to security, policy checking is more robust if corresponding algorithms run independently of an SDN controller. Corresponding systems typically sit between a network and SDN controllers being able to intercept all programming attempts for validation. In case of policy violations, corresponding programming attempts can be dropped while legitimate ones are forwarded to a network. To prevent adverse network manipulations, however, corresponding security policies must be configured in advance. Similar to managing firewall rules, this can be achieved, for example, by programming flow rules to a network only if a security policy allows this explicitly. Nevertheless, in complex networks with more than 100k of forwarding rules [6, 8] avoiding misconfigurations (and also the possibility of adverse network manipulations) remains challenging.

To complement current sandbox and policy checking approaches, we propose a security mechanism which neither requires to distinguish between malicious and benign use of a system capability, nor does it depend on additional and error-prone configurations. Based on the assumption that an attacker has a strong interest in hiding malicious network manipulations as they would immediately present artifacts which can be used for detection purposes, we compare the actually programmed network state with the one provided by a potentially compromised SDN controller. As malicious flow rules must be included within the former network state to be effective, an SDN rootkit would remove malicious artifacts from the latter one. We take advantage of this observation and prevent SDN applications as well as compromised SDN controllers from hiding adverse network manipulations by finding differences in these states. Based on this, we protect networks in a proactive manner, i.e., we drop hidden manipulations before they can reach a network device. As a result, we are able to improve SDN security where current sandbox and policy checking systems have limitations.

2. TECHNICAL BACKGROUND

As our approach requires the network state provided by a potentially compromised SDN controller, we briefly describe how a global network view is typically generated by an SDN controller. In addition, we outline the attacker model we use throughout the rest of this paper.

2.1 Providing the Global Network View

A global network view typically includes discovered hosts and links as well as flow rules presenting the network's forwarding behavior. As we are particularly interested in this forwarding behavior, we outline how popular SDN controllers such as ONOS [11], OpenDayLight [13] and FloodLight [3] collect information necessary to provide corresponding flow rules. Typically, this is achieved by means of using an internal database which, among others, contains the installed flow rules to present.

The Open Networking Operating System (ONOS), for instance, manages such a flow rule database in the following way: In case an SDN application intends to add a flow rule to a network device, an entry with a certain state information (*i. e.*, *PENDING_ADD*) is added to this database. Then in case of OpenFlow, a so-called flow-mod and barrier-request message is sent to a network commanding a network device to insert a corresponding flow rule. In turn, this device responds with error messages (if necessary) and a so-called barrier-reply message. The database entry's state is changed to *ADDED* if no error message but a corresponding barrier-reply message is received. Similarly, in case of flow rule removals the corresponding states are changed to *PENDING_REMOVE* first. Then, after sending a corresponding flow-mod and barrier-request message followed by receiving only a corresponding barrier-reply message, corresponding flow rules are removed from this database. In addition to such updates, this database can be queried, *e. g.*, by SDN applications regardless of flow rules state without generating additional control channel traffic.

In case of OpenDaylight, two separate databases are managed in order to implement different flow rule states. On the one hand, a so-called *config data store* is used to represent an intended network state. SDN applications can use associated methods to add, modify or remove entries which results in programming a network accordingly (*e. g.*, via flow-mod messages). On the other hand, a so-called *operational data store* represents the actual network state including its flow rules. It is periodically updated by sending and receiving additional control channel traffic, among others, via flow-stats-request and reply messages. Both data stores can be queried without generating further control traffic, except the one needed to update the operational data store. Note that the data stores are not synchronized, thus, the config data store does only contain flow rules added via associated methods. Also note that the operational data store provides network changes only with a delay. In the standard configuration, the default is four seconds.

Floodlight implements a database, among others, to maintain flow rules which are added, modified and removed via a so-called *staticentrypusher* service. Similar to ONOS and OpenDaylight's config data store, this database can be permanently queried for up-to-date information without generating additional control traffic. To collect information about flow rules which are installed other than by the staticentrypusher, network statistics including adequate flow rule data can be queried separately. In contrast to Floodlight's aforementioned database but similarly to OpenDaylight's operational data store, this generates additional control channel traffic. To the contrary of OpenDaylight, querying network statistics for Floodlight generates corresponding control messages each time the associated method is called.

2.2 Attacker Model

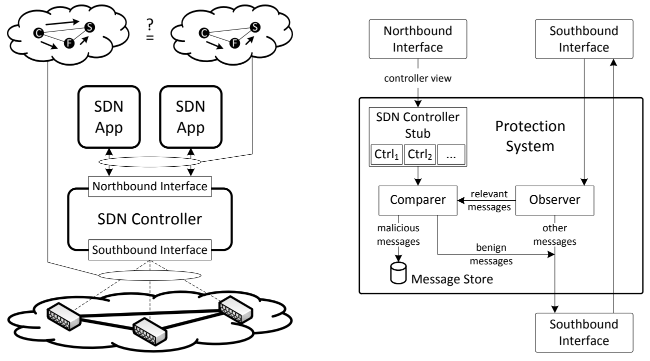
Throughout the rest of this paper, we assume that (i) an attacker is capable of compromising an SDN controller via a malicious SDN application and (ii) that such an attack includes adverse modifications to manipulate the provided global network view. This can be achieved, for example, by fooling administrators into installing a Trojan version of a legitimate SDN application, which applies rootkit techniques [22, 20]. Security mechanisms which are supposed to protect against such attacks (*e. g.*, cryptographic signatures) can be bypassed, for example, by generating a valid signature via a stolen certificate [2], by exploiting software vulnerabilities [14, 15], or via exploiting design flaws [22, 20].

3. PROTECTION SYSTEM

In the following, we describe the high-level idea of our approach, briefly outline several challenges that need to be solved, and finally provide a brief overview of the proposed architecture.

3.1 Approach

As explained before, the basic idea is to compare the actual network state with the state provided by a potentially compromised SDN controller to reveal and prevent adverse and hidden network manipulations. The former network state can be determined by observing the control channel between SDN controllers and network devices. On the one hand, this allows recognizing all network programming attempts whether they are benign or malicious. This is because hiding malicious activities from this state means that an attacker is actually not manipulating a network. On the other hand, malicious programming attempts can be intercepted and dropped to protect a network. The latter network state can be determined by querying the SDN controllers' view of a network. In contrast to the actual network state, malicious network changes are hidden from this view in case of an attack. Figure 1(a) illustrates our approach. Note that the actual network state (upper left cloud) includes a malicious flow rule which allows client C to directly reach server S unintendedly. On the contrary, the global network view provided by the SDN controller (upper right cloud) does not include this malicious forwarding behavior.



(a) Approach (b) Architecture

Figure 1: System Overview

3.2 Challenges

As we intend to interfere with the southbound protocol behavior, we must consider corresponding protocol specifics. In particular, intercepting network programming commands such as OpenFlow flow-mod messages can raise network reliability problems. This is the case, for example, when a flow-mod message is followed by a barrier-request message, which allows message ordering as such a request is supposed to block the control channel until a corresponding barrier-reply message is received. Such barrier messages are used, for instance, to program a certain forwarding rule not until it is certain that a previously programmed forwarding rule is inserted into a network device. To preserve network reliability, message ordering mechanisms must be considered when intercepting southbound traffic. Another challenge is to let the component, which is responsible for requesting the global network view, appear as a normal SDN application. Otherwise, an attacker could leave the network view untouched for such requests while hiding adverse manipulations from the network view provided to other SDN applications. This is important as, if not met, an attacker could bypass the proposed architecture while remaining undetected by other security tools. Furthermore, retrieving the SDN controller's view must be performed in an efficient way. This is mandatory in order to provide proactive network security, which heavily depends on a minimal delay of network programming attempts. Finally, we must also find a data structure, which allows efficient network state comparisons, in order to achieve high performance.

3.3 Architecture

Our architecture is depicted in Figure 1(b) and consists of the following four components: (i) an observer, (ii) an SDN controller stub, (iii) a comparer, and (iv) a message store. The observer unit intercepts the messages sent by an SDN controller, which are able to change the current network state. Such relevant messages are then provided as input for the comparer unit. In case of OpenFlow, this can be, for example, flow-mod as well as barrier-request messages. Other messages are forwarded towards a network device. The SDN controller stub implements a controller-specific method to query the network view provided to SDN applications, which is then hand over to the comparer as well. The comparer unit transforms both inputs to a data structure which can be compared efficiently. The comparison check consists of validating whether a network programming attempt in question is made visible by an SDN controller. In case such a request is indeed included within the SDN controller's view of the network, it is considered benign and, thus, forwarded to a network. In case it is hidden from the SDN controller's view of a network, it is considered malicious and, thus, dropped in order to protect a network. To enable forensic investigations, we finally store dropped manipulation attempts to a message store such that they can be analyzed after an incident was detected.

4. PROTOTYPE IMPLEMENTATION

We implemented a prototype of the proposed approach in Python. We use an open-source library [4] to handle southbound protocol specifics. It supports current releases of major SDN controller platforms (*i. e.*, ONOS Magpie/v1.12.0, OpenDaylight Nitrogen SR1/v0.7.1, and Floodlight in its master version) and OpenFlow version 1.3. Thereby, we currently take up the aforementioned challenges in the following ways. To preserve network reliability we block the control channel when receiving a barrier-request message, finish all network state comparisons depending on this request, and finally pass benign (or drop malicious) programming attempts followed by sending the barrier message to a network. Furthermore, we use the SDN controllers' northbound interface provided via REST API to collect the controller's view of a network. On the one hand, this significantly complicates to identify if a global network view request is initiated by our prototype implementation or another SDN application. On the other hand, this limits validation performance to the REST interface speed. To minimize delay of REST calls, we re-use TCP connections and use REST calls, which ideally query only a small set of flow rules. This can be achieved, for example, by querying a flow rule which has the same cookie as the one provided by the flow rule in question. This is especially beneficial as it avoids parsing large sets of flow rules which is the case for complex networks consisting of thousands of flow rules per switch. Finally, we generate a hash value per flow rule which allows efficient comparisons of the flow rule in question and the flow rule(s) included in a potentially manipulated global network view.

5. EVALUATION

We evaluate our current prototype implementation and measure effectiveness as well as efficiency penalties. Therefore, we use Mininet [10] to simulate networks of different topologies and take the popular SDN controllers ONOS, OpenDaylight and Floodlight to control these networks. In between, we run our prototype implementation as a proxy supporting OpenFlow version 1.3. All performance tests are performed on a Fujitsu Lifebook with 2 Intel Core i7-6600U CPU and 2 cores running at 2.60 GHz, 32 GB of RAM with Ubuntu Linux 16.04 LTS installed.

5.1 Effectiveness

To demonstrate effectiveness, we run compromised versions of aforementioned SDN controllers and attack networks in a way an attacker would do via a malicious SDN application using rootkit techniques. In particular, we manipulate the SDN controllers such that maliciously added flow rules are actively hidden from the global network view, while adversely removed flow rules are pretended to remain active. Then, we connect networks of different topologies and add malicious flow rules as well as remove legitimate ones. As a result, we observe that all relevant OpenFlow messages are intercepted correctly by our observer unit and that all adverse network manipulations are categorized as malicious by the comparer unit. In addition, we add and remove legitimate flow rules and validate that none of them is wrongly classified as an attack.

5.2 Efficiency

We test efficiency by measuring the delay time introduced by our prototype. Since our system checks each network programming attempt for each switch separately, we focus on network load (i.e., installed flow rules) rather than on large network topologies in these experiments. In particular, we install thousands of legitimate flow rules step by step and thereby increase the network load. Then, we add and delete several tens of flow rules at each step and measure the runtime performance of our prototype. Figure 2 shows measurement results for different SDN control platforms as a function of different network loads when adding flow rules. Note that our results look very similar in case we delete flow rules. Also note that we observe similar results in case of detecting and dropping adverse network programming attempts (*e. g.*, adding malicious flow rules and pretending existence of maliciously removed flow rules).

More specifically, we measure the delay time of important factors of our prototype which are (i) receiving and sending network packets from an SDN controller and to a network device (rx/tx), (ii) dispatching a network programming attempt in question to a thread which performs a validation check, (iii) querying a global network view via a REST call, (iv) parsing the flow rule in question and the global network view, and (v) comparing parsed messages in order to reveal and prevent hidden network manipulations. As a result, we observe that querying a potentially manipulated network view is the most time-consuming validation phase followed by receiving and sending OpenFlow messages and, in case of Floodlight, parsing a queried network view.

While the rx/tx delay time can be reduced by optimizing our prototype, for example, by using an efficient proxy library, reducing the two other ones is more challenging. This is mainly because of two reasons. On the one hand, the way we currently collect the global network view (*i. e.*, via REST API) is beneficial with respect to let our system appear like a normal SDN application. While this prevents bypass attacks based on distinguishing between network view queries initiated by our prototype and other SDN applications, using

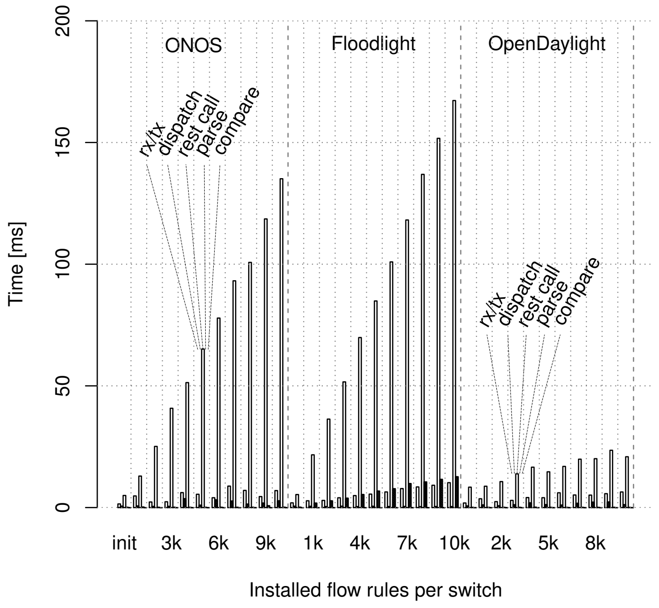


Figure 2: Delay time when adding flow rules

REST has speed limitations. A possible solution could be to implement an SDN application which can be triggered via a more efficient mechanism than REST. In such a case, however, it must be ensured that an attacker running within a compromised SDN controller must not be able to distinguish between such an SDN application and others. On the other hand, parsing a queried global network view can become challenging in case all flow rules of an SDN switch can be queried only. With respect to the tested SDN control platforms this is the case for Floodlight. For growing network loads this means that an increasing amount of flow rules must be parsed which results in a performance reduction. Similar to ONOS and OpenDaylight this can be addressed by extending Floodlight's northbound API such that individual flow rules can also be queried. For example, one could reuse the cookie value of flow-mod messages as an identifier for a corresponding flow rule managed by an SDN controller, which is the case for ONOS.

Nevertheless, current results indicate scalability and high performance even though there is room for optimizations. Regarding scalability, we observe that the introduced delay increases only moderately for ONOS and Floodlight but slowly for OpenDaylight with respect to the network load. In particular, the delay time for ONOS increases on average in the same way the network load growths, *i. e.*, by about 31 percent. In addition, the worst case delay is noticed at 225 milliseconds when adding flow rules at a network load of 9k flow rules and at 229 milliseconds when deleting flow rules at highest load. With respect to performance, our results indicate almost real-time capabilities for low network loads. For example, the average delay time of validating an ONOS programming attempt for up to thousand flow rules per network device is between 7 and 19 milliseconds on average. In contrast, current real-time policy checkers [9, 7] need less than a millisecond for adding a flow rule up to a few seconds for adding a new link. For higher network loads, however, the average delay time of our prototype for adding and deleting flow rules to ONOS increases up to 145 milliseconds per validation check at the highest network load. Though, in comparison to a system providing a similar mechanism in a reactive fashion [24], we are about 5 times faster while our system is capable of providing proactive security. Note that scalability and performance of our prototype is similar when using Floodlight and even better for OpenDaylight (see Figure 2).

As mentioned before, performance of our current implementation significantly depends on REST. Thus, Figure 3 contrasts collecting a global network view for aforementioned SDN controllers in detail. In case of ONOS, we observe

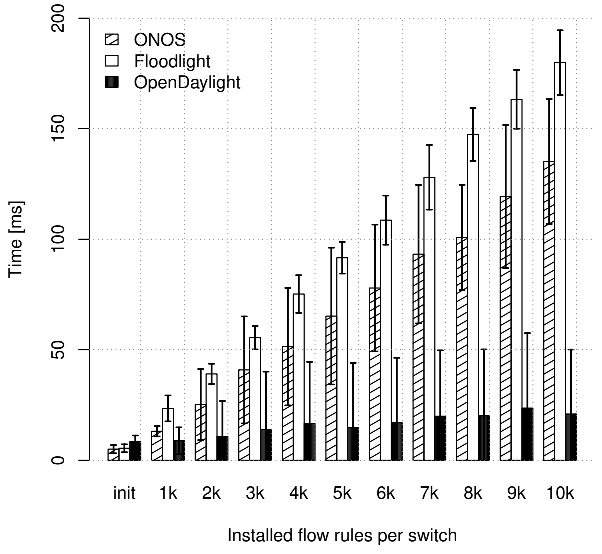


Figure 3: Delay for getting global network view

that the delay time for such queries increase depending on the network load, even though our system queries for a single flow rule. In particular, we check if ONOS can present information about a certain flow rule, which corresponds to a programming attempt containing a certain cookie within a flow-mod message. Note that we also recognize such an increasing delay time when running ONOS without our protection system. In case of Floodlight, only all installed flow rules of a single network device can be requested via REST, although a smaller set of flow rules would be sufficient for our purposes. Thus, a lot of data is transferred for each query, which must then also be parsed in order to select the flow rule relevant for a corresponding validation check. This causes the increasing growth of the delay time with respect to the network load. In case of OpenDaylight, our experiments show the best results regarding collecting a global network view. However, when installing a flow rule we must ensure that (i) it contains an identifier which can be queried via REST (*e. g.*, the flow rule id) and (ii) that this identifier corresponds to a value which is included in a programming attempt sent via a southbound message (*e. g.*, the cookie field in case of OpenFlow). Otherwise, our system is forced to query and parse all flow rules of a network device which would increase the delay time depending on network load in a similar way as for Floodlight.

6. LIMITATIONS AND DISCUSSION

While we are capable of fully protecting ONOS, our current prototype protects Floodlight and OpenDaylight only partially when using OpenFlow. In particular, we detect and prevent hidden network manipulations in case Floodlight's staticentrypusher service and OpenDaylight's config data store is used to make network changes. The reason for this implementation limitation lies in additionally control traffic, which can not be exchanged in case of a blocked control channel. For example, if an SDN application adds a flow rule using a controller service other than Floodlight's staticentrypusher or OpenDaylight's config data store, no entry is added to corresponding databases. Thus, we can not use them anymore to query a correct global network view. Instead, we can request Floodlight and OpenDaylight to query network statistics of installed flow rules as these also includes the information we need for our validation checks. However, such queries depend on sending and receiving additional messages (*e. g.*, flow-stats request/reply messages) over a control channel, which can be blocked by an SDN controller via a barrier-request message, while we want to run a validation check. As a consequence, the required additional control traffic can not be sent until a corresponding barrier-reply message is received [12]. Note that in case of OpenDaylight, we cannot use its operational data store as it also depends on additional control traffic for being updated. To overcome this limitation, we can use so-called auxiliary OpenFlow connections which can be established between SDN controllers and SDN switches in addition to a main connection [12]. For example, in case OpenFlow's main connection (or an auxiliary connection) is blocked, flow-stats request and reply messages can be exchanged via an unblocked auxiliary connection. Since such reply message do not contain information about the network programming attempt under validation, we can inject data to enable an SDN controller to provide corresponding information within its global network view. Although querying network statistics can also be performed by any other SDN application, there is no need to clean up after injecting data about a flow rule which is actually not installed on a network. The reason for this is twofold. First, the corresponding network view is correct in case of a benign manipulation which we apply on a network. Second, the received view does not contain hidden flow rule information in case of malicious manipulation attempts. As a result, this approach can be used to fully protect OpenDaylight and Floodlight as well.

7. RELATED WORK

To the best of our knowledge, no malicious SDN application has been found in the wild yet. Nevertheless, several studies demonstrate that a single malicious SDN application is able to significantly harm an SDN controller and cause substantial damage on an associated network [23, 21, 19, 18]. Especially, SDN rootkits [22, 20] are able to subvert SDN controllers while applied techniques aim at hiding malicious actions such as re-programming a network in an adverse manner. In order to detect and prevent such attacks, several countermeasures have been already proposed including sandbox mechanisms [23, 25, 21] and real-time policy checking [9, 7]. While putting an SDN application into a sandbox and restricting access to critical operations require a priori knowledge about attacks for a correct configuration as well as a list of used critical operations to avoid SDN application crashes, generating a complete set of security policies to protect dynamic networks is challenging with respect to policy checking. Complementary to aforementioned countermeasures, we therefore propose a system which is neither error-prone regarding its configuration, nor does it require a priori data about attacks, nor do we need to distinguish between benign and malicious use of system functionalities. Closest to our work is SDN-Guard [24] which is a reactive system. In particular, adverse network manipulations are initially programmed to a network and removed after a certain amount of detection time. In contrast, our system is able to prevent such manipulations before they are applied on a network. In addition, we achieve a performance which is about five times faster than SDN-Guard. Note that validation checks of network programming attempts can also be performed on the control layer [19, 18]. To the contrary of real-time policy checkers, such systems typically run within an SDN controller's execution environment which makes them vulnerable to bypass attacks [22].

8. CONCLUSION

In this paper, we present a novel approach to improves SDN security for major SDN control platforms where existing protection mechanisms have limitations. In particular, we protect networks in a proactive fashion and, thus, prevent the installation of adversely hidden network manipulations. Complementary to sandbox and policy checking mechanisms, we provide a scalable system with high performance to improve protection against malicious SDN applications and compromised SDN controllers.

Acknowledgements

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.

9. REFERENCES

[1] B. Chandrasekaran, B. Tschaen, and T. Benson. Isolating and Tolerating SDN Application Failures with LegoSDN. In *ACM Symposium on SDN Research*, 2016.

[2] N. Falliere, L. O. Murchu, and E. Chien. W32. stuxnet dossier. *White paper, Symantec*, 2011.

[3] Floodlight Project. Floodlight Controller. www.projectfloodlight.org/floodlight/.

[4] Floodlight Project. LoxiGen Tool to Generate OpenFlow protocol libraries. https://github.com/floodlight/loxigen.

[5] P. Holzinger, S. Triller, A. Bartel, and E. Bodden. An In-Depth Study of More Than Ten Years of Java Exploitation. In *ACM Conference on Computer and Communications Security*, 2016.

[6] U. Hölzle. OpenFlow @ Google. Open Networking Summit, 2012.

[7] 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.

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

[9] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *USENIX Symposium on Networked Systems Design and Implementation*, 2013.

[10] 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.

[11] ONOS Project. Open Networking Operating System. onosproject.org/.

[12] Open Networking Foundation. OpenFlow Switch Specification. www.opennetworking.org.

[13] OpenDaylight Project. OpenDaylight. www.opendaylight.org.

[14] Oracle. Security Alert for CVE-2012-4681. www.oracle.com, 2012.

[15] Oracle. Security Alert for CVE-2013-0422. www.oracle.com, 2013.

[16] P. Parrend and S. Frénot. Java Components Vulnerabilities - An Experimental Classification Targeted at the OSGi Platform. *Technical report 6231, Institut National de Recherche en Informatique et en Automatique*, 2007.

[17] P. Parrend and S. Frénot. More Vulnerabilities in the Java/OSGi Platform: A Focus on Bundle Interactions. *Technical report 6649, Institut National de Recherche en Informatique et en Automatique*, 2008.

[18] 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.

[19] 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.

[20] C. Röpke. SDN Ro^{2}tkits: A Case Study of Subverting A Closed Source SDN Controller. In *GI Sicherheit*, 2018.

[21] C. Röpke and T. Holz. Retaining Control Over SDN Network Services. In *International Conference on Networked Systems*, 2015.

[22] C. Röpke and T. Holz. SDN Rootkits: Subverting Network Operating Systems of Software-Defined Networks. In *Symposium on Recent Advances in Intrusion Detection*, 2015.

[23] 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 Conference on Computer and Communications Security*, 2014.

[24] D. Tatang, F. Quinkert, J. Frank, C. Röpke, and T. Holz. SDN-Guard: Protecting SDN controllers against SDN rootkits. In *IEEE Workshop on Network*

Function Virtualization and Software Defined Networks, 2017.

[25] C. Yoon, S. Shin, P. Porras, V. Yegneswaran, H. Kang, M. Fong, B. O'Connor, and T. Vachuska. A Security-Mode for Carrier-Grade SDN Controllers. In *Anual Computer Security Applications Conference*, 2017.