# SHIELD: An Automated Framework for Static Analysis of SDN Applications

Chanhee Lee
KAIST
mitzvah@kaist.ac.kr

Seungwon Shin
KAIST
claude@kaist.ac.kr

## ABSTRACT

Software-Defined Network (SDN) is getting popular and increasingly deployed in both of academia and industry. As a result of which, its security issue is being magnified as a critical controversy, and some pioneering researchers have investigated the vulnerabilities of SDN to discover the feasibility of compromising SDN networks. Especially, they prove that a simple malicious/buggy SDN application running on an SDN controller can kill an SDN control plane because it usually has a right to access the resources of SDN controller. To address this issue, we focus on the malicious SDN application themselves (i.e., how to understand if an SDN application is malicious). In this context, we consider analyzing SDN applications before running in a static manner. We present SHIELD, a new automated framework for static analysis of SDN applications carefully considering SDN abilities. SHIELD provides the Control-Flow Graph (CFG) and critical flows of SDN applications. We evaluate the effectiveness of SHIELD with 33 real world applications (both benign and malicious applications), and from the results, we define 10 malicious behaviors of SDN applications.

## 1. INTRODUCTION

Software-Defined Networking (SDN) is now considered as one of the promising future networking technologies, and this trend can be clearly verified by observing activities of both industry and academia. In industry, we can easily observe that many major network vendors, such as Cisco and Juniper, are actively supporting SDN functionalities. In addition, several network service providers have deployed SDN-enabled networks in their services (e.g., Google B4 and Microsoft SWAN). This trend can also be noticeable in academia. Recently, many SDN-related research outputs have been published [8] and announced [6].

As SDN is getting popular, its security issue is being magnified as a critical controversy. In this context, some pioneering researchers have investigated the security issues of SDN to comprehend what kinds of vulnerabilities are existing in SDN [14] [7] [12]. Those studies verify that current SDN architectures are not safe and there are many possible security breaches, and we discover that most attack cases are about an SDN control plane (a.k.a., controller), which manages connected underlying networking devices. It is not so surprised because the SDN control plane will handle most complicated functions in a network. Moreover, network applications running on an SDN controller will have unlimited power to control/access resources of a system (i.e., hosting an SDN control plane) and network components [15] [12]. Thus, some malicious/buggy network applications can kill an SDN control plane, which can lead to shut down a whole network.

To address this issue, some researchers have proposed new SDN control plane architectures [15] [12], and they can reduce the effect of malicious/buggy network applications. However, their approaches still have problems. They assume that network applications will be managed by a certified developer and signed by his signature to denote its identity, which is not so feasible if there many third-party developers. Moreover, so far there is no noticeable certified deputy agency for SDN applications. In addition, they assume that applications can be monitored and controlled in run-time. It is an interesting approach to restricting the malicious behaviors of untrusted network applications, but sometimes it can cause serious performance overhead if an application conducts complicated functions.

This situation motivates us to propose a method of analyzing SDN applications before running (i.e., static analysis of SDN applications), and it does not require any certified agencies and does not cause the overhead of run-time monitoring. This static analysis of an application enables network administrators, who want to launch SDN application, to carefully examine applications before launch (or installation), and thus they can sift out malicious/buggy applications beforehand.

Analyzing a program statically (i.e., static analysis) is not a new idea, and it is commonly used in diverse areas (e.g., finding malicious functions of a javascript application). However, when we apply this concept to a different domain, we need to consider domain-specific features. For example, if we analyze an Android application, we need to understand its internal operations. Likewise, when analyzing an SDN application statically, we should carefully consider the characteristics of SDN. For example, since most SDN applications will be invoked by events delivered from a data plane, we should carefully investigate which event (from the data plane) triggers which application. This operation is quite



*SDN-NFVSec'16, March 11 2016, New Orleans, LA, USA*



DOI: http://dx.doi.org/10.1145/2876019.2876026

different from other legacy applications, such as Android applications, which are activated by user inputs or timers. Moreover, their operations truly depend on the abilities to support SDN features (e.g., OpenFlow protocol [4]), and it implies that our analysis method should understand those abilities in analyzing the operations of SDN applications.

In this paper, we present SHIELD, an automated framework for static analysis of SDN applications that are providing the Control-Flow Graph (CFG) and critical flows of an SDN application. When designing SHIELD, we consider SDN abilities (i.e., OpenFlow-based on SDN functions) to address challenges mentioned above. For tracing the source code, we set one of entry points as a receiver or a trigger for an event delivered from a data plane. We also regard each logic of OpenFlow message handling and its internal operations as the critical flows of an SDN application. To infer the critical flows, we first build the critical API table from some previous research addressed SDN vulnerabilities and the design specification of popular SDN controllers. SHIELD discovers the critical flows of a target application based on the table, and such flows denoted as behavior graphs.

We analyze total 33 real world applications of both benign and malicious applications of OpenDaylight [11] and Floodlight [3]. We collect and analyze the malicious applications to understand what are the malicious behaviors of SDN applications, and then we categorize several malicious behaviors of SDN applications.

In summary, this paper makes the following contributions:

- We present SHIELD, a new automated framework for static analysis of SDN applications. SHIELD carefully considers characteristics of SDN to analyze functions of an SDN application.
- We evaluate SHIELD with the 33 real world applications, which operate on popular SDN controllers (i.e., OpenDaylight, Floodlight). We also describe the principal differences in those application's activities.
- We define/formalize the malicious behaviors from the results of our evaluation and then, we can use these behaviors to detect SDN malware.

# 2. MOTIVATION

A research group introduces the details of SDN attack cases on their website [13]. Not only present they the known attack cases but also discover some new SDN vulnerabilities. They publish those found attack cases on their web page as a vulnerability genome project, and we notice that most of the attack cases need the installation of a malicious application, which means that an SDN application will play a major role in compromising SDN. This trend implies that it is necessary to analyze the SDN applications before running in a static manner (i.e., static analysis of an SDN application) to defeat these threats.

Static analysis of a program is not a new idea and widely used in the other domains such as a mobile malware analysis platform. For instance, there are several previous studies for the static analysis of Android applications [16] [1]. They apply several practical methodologies such as behavior graph extraction, API dependency graph analysis, to analyze the functions of Android applications. When analyzing SDN applications statically, the methods used in those studies could be employed.

However, a new static analysis method is required to examine activities of SDN applications, because the operational behavior of an SDN application are quite different from that of a smartphone application. For example, most SDN applications operate with an event-driven processing (triggered by network devices), and thus they commonly have specific event handling logic such as an RPC call, callback, or notification. In addition, most SDN controllers provide dedicated APIs (e.g., Java interfaces, listener management APIs) to applications to support such series of event handling process. Although these SDN-specific features are essential in analyzing the application's behavior and inspecting its operations, no previous research has employed such features as the analysis method so far. To the best of our knowledge, our work is the first effort to analyze the SDN applications in a static manner. *Note that we mainly analyze SDN applications operated with the OpenFlow protocol [4], because the protocol is the de-facto standard protocol in SDN environments.*

## 2.1 Motivating Example

Here, we provide an example case of SDN attack scenario that modifies flow rule without administrator notice. Figure 1 shows a scenario of flow rule modification attack. (1) Malware iterates all of OpenFlow switches and gets the instance of a target switch. (2) It iterates all flow entries of the switch and gets the instance of a particular flow entry (a flow between the host A and B in Figure 1). (3) The malware sends a flow rule modification message to the switch with a drop action. (4) Finally, a network connection (between host A and B) is no longer allowed which means the target link is disconnected.

# 3. CRITICAL API ARRANGEMENT

A malicious SDN application (we also call it SDN malware, and we interchangeably use these two terms) can uses some APIs provided by the SDN controller to achieve its malicious goal, and thus we can infer that the usage of the APIs is malicious. For instance, `removeFlow()` could be used to remove a flow entry on a particular node for malicious purposes. `getBundles()` and `uninstall()` could also be associated with the application killing vulnerability.

However, unfortunately, we cannot directly determine (or detect) a malicious SDN application just by inspecting such APIs because benign applications can also call such APIs legitimately, but we can only consider that an application having such API calls is suspicious. For example, `removeFlow()` above mentioned is used in the innocent applications

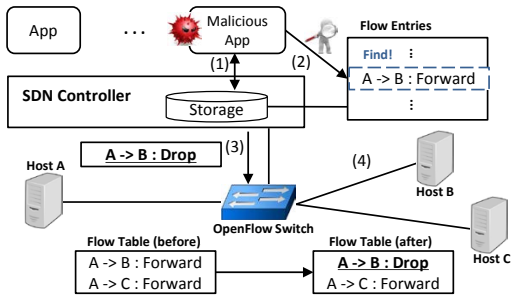



**Figure 1: A scenario of flow rule modification attack**

**Table 1: A portion of critical APIs of OpenDaylight**

| Class / Interface name | Method name |
|---|---|
| IFlowProgrammerService | modifyFlow() |
| IFlowProgrammerService | removeFlow() |
| IStatisticsManager | getFlows() |
| ServiceRegistration | unregister() |
| Bundle | uninstall() |
| java.lang.System | exit() |

such as *ForwardingRulesManager*, which is one of the default applications of OpenDaylight. Because of this issue, we first focus on the analysis methodology for the network application's behavior, instead of detecting SDN malware (it remains as the next step of our work).

To analyze the SDN application's behavior (especially, suspicious activities), first of all, we have researched the previous known SDN vulnerabilities [14] [7] [12] [15] [13]. At the same time, we have manually inspected the core (default) applications of popular SDN controllers. By doing so, we construct the critical API lists, and a portion of such critical APIs of OpenDaylight is shown in Table 1. We deploy these critical APIs and their call sequences to our framework as a analysis method. In other words, SHIELD finds out critical flows of the SDN application based on the critical API call sequences.

# 4. DESIGN AND IMPLEMENTATION

Here, we present how we design SHIELD in detail. SHIELD consists of three main components: **Source Code Tracer**, **Control Flow Analyzer** and **Result Manager**, and it provides a storage service (i.e., SHIELD DB) to load/store the data from each component (shown in Figure 2).

## 4.1 Source Code Tracer

When an application is given to be analyzed, this component will start its analysis (first entry point of SHIELD). It traces and parses the source code of an application from all possible entry points (e.g., it can be an event handler) to each end of the program, and it automatically builds the control-flow graph of the application. To inspect the control-flow of an application, we leverage the newest version of Soot [10], which is an open source Java program static analysis tool. We do not allow omitted statements by considering all possible branches as targets of inspection.

**Irrelevant API Discoverer.** To provide the concise view of the analysis result, we need to find an efficient way to analyze source code. In this context, we try to focus on some relevant functions instead of tracking all functions. For example, invoking some logging functions (e.g., *java.util.printf*) have been disregarded in our analysis. In other words, this subcomponent takes irrelevant APIs from the predefined API list (i.e., irrelevant APIs) to optimize search space, and prevents SHIELD from tracing unhelpful APIs

## 4.2 Control Flow Analyzer

**API Call Extractor.** It takes all nodes from the CFG and parses code by each node. It sorts API call instructions out from all nodes based on *invoke* bytecode (which is a similar to the call instruction in x86 assembly).

In an SDN environment, various information (e.g., return type, parameters, name) of API should be provided for users to analyze the behavior of a target application accurately. For instance, the match actions of a flow rule are delivered as the parameters of API to install/modify a new flow rule. Actually, a *Drop* instance is set as a parameter of `modifyFlow()` to disconnect a target link. To accommodate such needs (considering all features of API to analyze), SHIELD provides the package of API calls that consist of name, return type and parameters of API call.

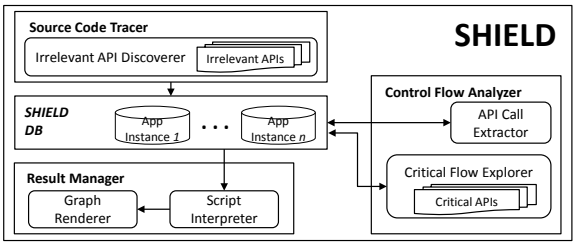



**Figure 2: An architecture of SHIELD**

**Critical Flow Explorer.** It finds the critical flows of a target application based on the predefined critical API table, which is made of APIs specified in Section 3. If any API calls of a target application found in such table, it is marked as a critical flow and stored into the DB with a critical flow indicator. Result Manager uses these indicators to draw a behavior graph of a target application in later.

To describe the critical flow of a target application, we apply the concept of behavior graph, introduced in [9], to our system. In behavior graph, each node indicates a name of an API and each edge represents a relation between its return values and parameters for another API. Through inspecting behavior graphs, we can figure out where the values of parameters come from, and SHIELD provides the critical flows of a target application.

## 4.3 Result Manager

**Script Interpreter.** We use graph description language DOT [5] to automatically draw the CFG and behavior graph of a target application as image format (PNG file). This interpreter translates the control-flows and behaviors of an application into the DOT scripts, and those are used in the next component (Graph Renderer) to draw graphs.

**Graph Renderer.** This component generates the graph images (to the PNG file) by rendering the DOT documents. We use Graphviz [2], which is a package of open source tools for drawing graphs specified in DOT language scripts. To accommodate large-scale images (for the better view), we provide a method to an administrator to select directly an entry function. If so, SHIELD starts from the first node of the entry function. Otherwise, it starts from a default entry point (top of CFG). Through such method, an administrator can selectively obtain a part of CFG in his interest.

# 5. EVALUATION

We evaluate the effectiveness of SHIELD with 33 benign and malicious applications, which are from well-known open source SDN controllers, and they are summarized in Table 2. We analyze 5 default applications[1] (e.g., *ForwardingRulesManager, HostTracker*) and 13 malicious applica-

[1]We consider those default applications are benign.

**Table 2: An overview of use cases**

| | Benign | Malicious |
|---|---|---|
| OpenDaylight Helium SR3 | 5 Apps | 13 Apps |
| Floodlight v1.1 | 7 Apps | 8 Apps |

tions (e.g., *Flow Rule Modification, Service Unregistration*) of OpenDaylight. We also analyze 7 default applications (e.g., *LinkDiscoveryManager, TopologyManager*) and 8 malicious applications (e.g., *Internal Storage Abuse, Event Listener Unsubscription*) of Floodlight. Now, we present the critical-flows of only a few applications of OpenDaylight because of the page limits, and we discuss the meaningful insights of our analysis results.

## 5.1 Use Case Analysis

**ForwardingRulesManager.** Figure 3 shows a part of critical flows of *ForwardingRulesManager*. In this critical flow, i) it uses several methods such as `entrySet(), getNode(), getInstall(), getFlow()` to get an instance of a flow entry from a specific node, and ii) it calls `removeFlow()` with the instance obtained in the previous step to remove a flow entry from the target node.

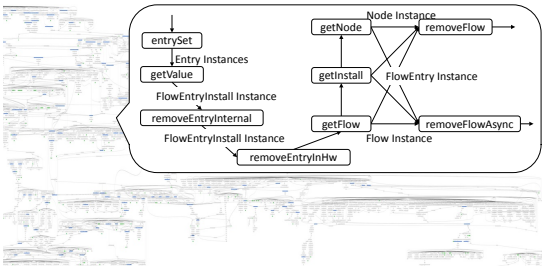



**Figure 3: Benign case: A critical flow of ForwardingRulesManager**

**HostTracker.** Figure 4 presents a host migration part of critical flows of *HostTracker* when a host is attached. i) If the host already exists in a current topology, it only migrates the host with its IP address. If not, ii) it uses both IP and MAC address for host migration.

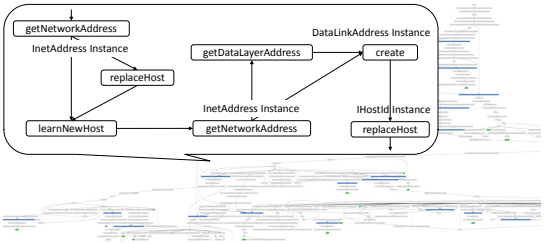



**Figure 4: Benign case: A critical flow of HostTracker**

**Flow Rule Modification Attack.** Figure 5 shows a critical flow of *Flow Rule Modification Attack*. i) It accesses to an internal storage of an SDN controller and obtains each handler for all nodes and flows. ii) It creates a new flow entry with an original match field instance and a new action field instance with a *Drop*. Finally, iii) it calls `modifyFlow()` to replace an old (original) flow entry on a target node to the new flow entry.

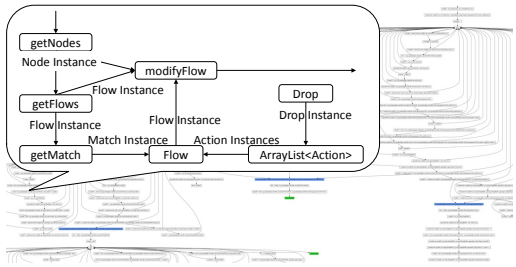



**Figure 5: Malicious case: A critical flow of Flow Rule Modification attack**

**Service Unregistration Attack.** Figure 6 denotes a critical flow of *Service Unregistration Attack*. In this case, i) it accesses to an internal storage of an SDN controller and take an instance of a dependency manager, and ii) it obtains component instances using an API provided by the dependency manager. Through these component instances, iii) it takes instances of service registration and dependency, and then, it iv-i) unregisters a target service using `unregister()` with the service registration instance and iv-ii) removes a dependency of a target service through `remove()`.

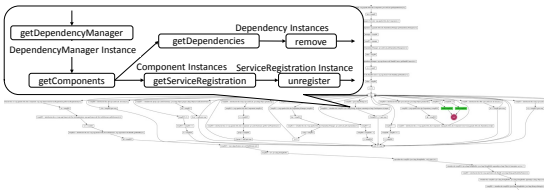



**Figure 6: Malicious case: A critical flow of Service Unregistration attack**

### 5.1.1 Discussion

From the analysis results, we find that there are notable differences between the critical API calls of the benign and malicious applications. While the benign applications commonly use the critical APIs to get a single instance, the malicious applications frequently exploit the critical APIs to get all required instances (e.g., `getNodes()`, `getFlows()`, `getComponents()`, `getDependencies()`). They leverage such APIs to achieve their malicious goals for all instances or a specific instance. The reason that this difference appears is relevant to the SDN-specific feature of event handling process. Most SDN events are triggered from a data plane (actually, a specific node) and delivered to a control plane. Therefore, benign applications hardly use the instances of all nodes at once because they aim to handle a specific node.

Another difference is in that APIs used in the malicious applications are rarely used in the benign applications. For instance, the malicious application shown in Figure 6 uses `unregister()` and `remove()` to remove the listeners of other application. Although those APIs are necessary to exchange some information across applications, they are hardly called by the benign applications in general. Furthermore, when the malicious applications misuse them, it is possible that the serious security problems occur such as application killing. Therefore, the usage of such critical APIs would be suspicious, and some restriction methods against them are required.

### 5.1.2 Critical API for Malware Only

Some critical APIs such as `uninstall()`, `removeNodeAllProps` are used in the malicious applications, whereas the default applications of its controller never use them. Namely, such critical APIs (malicious only) are more useful for malicious applications to achieve their malicious goal than any other benign function although those are supported by SDN controllers to meet the needs of innocent functions.

We have found 24 and 8 such APIs of OpenDaylight and Floodlight from the results of our evaluation. The usage of these APIs give us the valuable insights about the malicious activities of SDN applications because we can consider such APIs as the unique feature of SDN malware. Besides, we can use these APIs to analyze the correlation between the malicious activities and a target application's behavior by allocating the higher weight for these APIs than the others, and this could be effective to detect the SDN malware.

### 5.1.3 Performance Measurement

We measure the time to analyze each use case. We repeat five times for each case and take the average value of the measured times. The test is done on a single virtual machine of 2 dual-core processors and 4GB memory.

In the case of OpenDaylight, it takes 27.85 and 5.94 seconds on average to analyze the benign and malicious applications respectively. In the case of Floodlight, it takes 45.26 and 18.07 seconds on average for the benign and malicious applications. We found that SHIELD can inspect an application within dozens of seconds, and it does not influence the performance of controller because our framework is not supposed to operate with a running SDN controller, inspects an application before running on its controller. Besides, we use just a low-performance virtual machine for the test, and we believe that the analysis performance of SHIELD could be improved with further optimizations in near future.

## 5.2 Definition of Malicious Behavior

From the results of our use case analysis, we have identified 10 malicious behaviors as 4 categories from their target positions: Control Channel (**CC**), Intra-Controller (**IC**), Other Apps (**OA**) and System Flow (**SF**). Table 3 describes such malicious behaviors. The behaviors of Control Channel represent malicious functions using a communication channel between a control plane and a data plane. For example, SDN malware can send the OpenFlow message such as *FlowMod* to data plane to implement its malicious functions. Unlike this, the Intra-Controller behaviors aim for changing the internal resource of SDN controller, and the rewriting of internal storage data is a typical example of such behaviors. The behaviors of Other Apps and System Flow display the malicious functions aimed for other applications running on its controller and the control flow of its controller. We introduce each malicious behavior as a graph, and each node of the graph represents generalized functions, and each edge signifies a dependence on its two nodes.

Malicious behaviors of SDN applications have distinct differences with benign functions. For instance, while **[CC-1]** removes all flow entries on a target switch at once, no benign applications have this function although some benign applications rarely delete a single flow entry on a switch. The **[CC-3]** is another case to install new flow entries until its flow table floods, and this function is only for the malicious purpose. In addition, the behaviors of **[OA]** explain the malicious functions that directly influence other applications running on its controller (e.g., application killing) while innocent applications do not have such functions in general. Of course, OpenDaylight provides similar functions as Command-Line Interfaces (CLIs), but these CLI functions support a network administrator for his/her manual works not for an operation of program code level.

The behaviors involve some SDN-specific characteristics as well as the differences between benign and malicious functions. For instance, **[SF-1]** is a normal function of an application termination in general but in SDN, it is malicious enough because if an application running on SDN controller terminates itself, the controller terminates at the same time [15]. **[OA-3]** is also a simple (maybe innocent) function modifying some properties of its operating system in other domains. However, in SDN, modifying the list of event listener alone can make the effectiveness of other application killings because SDN applications operate with an event driven processing. In other words, if malware deletes listeners of core applications directly concerned packet forward or security functions (e.g., forwarding, firewall), this behavior can incur some serious problems on its network operations such as link disconnection.

# 6. FUTURE WORK

Our ultimate goal is proposing an effective and efficient malware detection system for SDN environment. Especially, to detect well-known malware, a signature-based detection methodology is useful and widely used in other areas (e.g., anti-virus software and network intrusion detection system). In this paper, we dissect benign and malicious SDN applications to understand their behaviors (mostly malicious behaviors). By investigating the correlations between those discovered malicious behaviors and some SDN applications (e.g., Jaccard coefficient), we can make a more clear decision whether an SDN application is malicious or not.

However, the malicious behaviors described in this paper are not enough to cover all (or most) malicious behaviors of SDN malware, and thus we still need to collect and analyze much more SDN malware. To do this, we are actively gathering SDN malware samples as many as possible, and at the same time, we are trying to find feasible vulnerabilities of SDN. As the future work, we plan to present a first detection system for SDN malware.

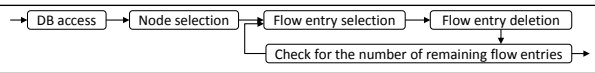
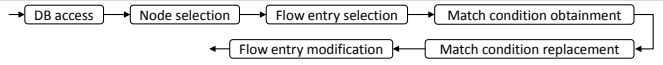
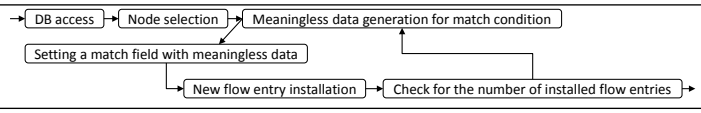
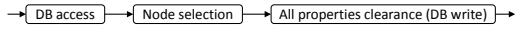
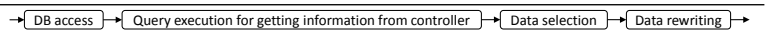
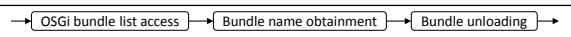
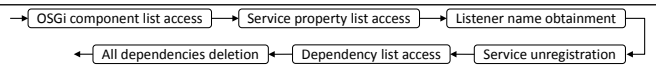
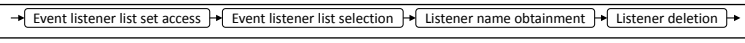
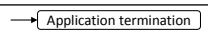
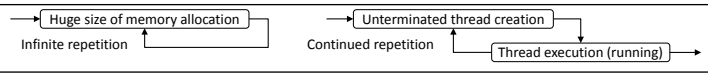
# 7. RELATED WORK

DroidRanger [16] applies permission based behavioral footprinting scheme to the method of malware detection. Drebin [1] infers the behavior pattern of malware by using not only permissions but also API calls and network addresses. These two types of research use some Android-specific features for static analysis, and thus, such approaches cannot directly deploy to SDN applications. In other words, using permission as analysis method is not suitable to SDN environment. Permission is a useful feature for classifying Android malware, not SDN malware because NOS does not provide any permissions yet.

# 8. CONCLUSION

We have presented SHIELD, a new automated framework for static analysis of SDN applications. SHIELD provides the CFG and critical flows of SDN application as an analysis

**Table 3: Malicious behaviors of SDN**

| Target Location | Vulnerability | Behavior |
|---|---|---|
| Control Channel | **[CC-1]** Flow table flushing |   |
| | **[CC-2]** Switch firmware abuse |   |
| | **[CC-3]** Flow rule flooding |   |
| Intra Controller | **[IC-1]** Internal storage abuse |   |
| | **[IC-2]** Internal storage abuse |   |
| Other Apps | **[OA-1]** Application eviction |   |
| | **[OA-2]** Service unregistration |   |
| | **[OA-3]** Listener unsubscription |   |
| System Flow | **[SF-1]** System termination |   |
| | **[SF-2]** Resource exhaustion |   |

result and we carefully consider the characteristics of SDN to increase the precision of analysis. With SHIELD, we analyze 33 real world applications running on OpenDaylight and Floodlight. The analysis results of such applications give us the meaningful insights into the activities of SDN applications. Based on these insights, we identify several malicious behaviors of SDN applications, and such behaviors can be used to detect SDN malware.

# 9. ACKNOWLEDGMENTS

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. B190-15-2012, Global SDN/NFV Open-Source Software Core Module/Function Development).

# 10. REFERENCES

[1] D. Arp, M. Spreitzenbarth, M. H bner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

[2] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull. Graphviz-open source graph drawing tools. In *Graph Drawing*, pages 483–484, 2002.

[3] FloodLight. http://floodlight.openflowhub.org/.

[4] O. N. Foundation. Openflow switch specification 1.5.0. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf.

[5] E. Gansner, E. Koutsofios, and S. North. Drawing graphs with dot. Technical report, AT&T Research. http://www.graphviz.org/Documentation/dotguide.pdf, 2006.

[6] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 15–26, 2013.

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

[8] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 3–14, 2013.

[9] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-y. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *USENIX security symposium*, pages 351–366, 2009.

[10] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, 2011.

[11] J. Medved, R. Varga, A. Tkacik, and K. Gray. Opendaylight: Towards a model-driven sdn controller architecture. In *2014 IEEE 15th International Symposium on*, pages 1–6. IEEE, 2014.

[12] P. Porras, S. Cheung, M. Fong, K. Skinner, and V. Yegneswaran. Securing the software-defined network control layer. In *NDSS*, 2015.

[13] SDNSecurity.org. http://SDNSecurity.org/.

[14] S. Shin and G. Gu. Attacking software-defined networks: A first feasibility study. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 165–166, 2013.

[15] 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 *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 78–89, 2014.

[16] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, 2012.