2018 IEEE 26th International Conference on Network Protocols

INDAGO: A New Framework For Detecting Malicious SDN Applications

Chanhee Lee, Changhoon Yoon, Seungwon Shin, and Sang Kil Cha
KAIST
{mitzvah, chyoon87, claude, sangkilc}@kaist.ac.kr

***Abstract*—Software-Defined Networking (SDN) controllers not only provide centralized control of SDNs, but also implement open and programmable APIs to ultimately establish an open network environment, where anyone can develop and deliver useful SDN applications. In such an environment, malicious SDN applications can be easily developed and distributed by untrusted entities and can even possess full control of SDNs. Thus, the security threat of malicious SDN applications must be taken seriously. In this paper, we propose a novel system, called Indago, which statically analyzes SDN applications to model their behavioral profiles, and finally, it automatically detects malicious SDN applications with a machine learning approach. We implement a prototype system and evaluate its effectiveness with real world SDN applications and malware. Our evaluation results show that the system can detect most known SDN malware with a high detection rate and low error rates.**

I. Introduction

Software-Defined Networking (SDN) has now become a mature networking technology. Industries are eagerly seeking the adoption of it [1], [2] for efficient network operations. IT companies such as HP [3], Cisco [4], and NEC [5] have already released SDN-enabled network devices and products for commercial purposes.

In contrast to the traditional networks, SDN offers a network environment where anyone can easily develop and deploy SDN applications to the networks. Specifically, the control plane (a.k.a. SDN controller) provides an extensible environment to manage network devices where the SDN controller can instantiate a network service by simply deploying an SDN application. For example, a network administrator can easily enable cost-based routing by installing an appropriate SDN application in the SDN controller.

The flexible nature of the SDN controller not only allows versatility in network operations but also signifies the ease of malicious activity. Prior research has confirmed that SDN controllers and applications are feasible attack vectors: an adversary can take control over SDNs by exploiting them. Some pioneering researchers have proposed various solutions (e.g., [6]–[10]) to fortify the security of the control plane.

The current best practice for protecting the control plane relies on a *reactive* defense such as permission-based security enforcement [10]. For example, Open Network Operating System (ONOS) [11] employs a security extension [9] that monitors every execution of SDN applications and reactively blocks the execution when it calls an unprivileged API. However, such mechanisms suffer from two major challenges. First, instrumenting the execution of SDN applications introduces significant performance overhead [9], [12]. Second, they allow the SDN applications to be deployed prior to being vetted. Thus, any failures in discovering malicious behavior can directly put the entire network at risk.

In this paper, we propose the first *proactive* defense mechanism that leverages a *static analysis* on SDN applications. While previous studies focus on the system-wide defenses, we analyze the behavior of SDN applications themselves and detect malicious behavior by checking their semantics. Thus, our technique helps in vetting malicious SDN applications prior to deployment. Our approach is motivated by the following research question: *Is it possible to detect malicious SDN applications without actually deploying (or running) them?* If possible, we would avoid any additional performance overheads and potential risks of detection failures introduced by existing approaches

To answer the research question, we present Indago, a framework for detecting malicious SDN applications and implement a prototype system for the evaluation. Indago is a fully automated system based on a static analysis technique, which significantly reduces operational expenses. Malware analysis and detection is often used in other domains, such as Linux or Android systems [13]–[15]. However, existing approaches cannot be directly applied to detecting SDN malware, because SDN applications introduce completely new behavioral patterns compared to malware in the other domains. For example, a legitimate SDN application that implements an L2 forwarding function may leverage an API to install flow rules to control the network behavior; and at the same time, a malicious SDN application may also abuse the same API to corrupt the network behavior. That is, there is no clear distinction between malicious and benign behaviors of SDN applications, nor directly applicable methods, e.g., there is no system calls in SDN.

Indago tackles this challenge by extracting a behavioral profile of an SDN application and representing the profile in a specialized data structure that we call *Security-Sensitive Behavior Graph* (SSBG). At a high level, Indago leverages some domain-specific knowledge along with a classic control- and data-flow analysis to build an SSBG for an SDN application. We then extract semantic features from the SSBG and use them to detect malicious SDN applications.

To evaluate our approach, we first collected as many SDN applications as possible from real world projects [11], [16]–

978-1-5386-6043-0/18/$31.00 ©2018 IEEE
DOI 10.1109/ICNP.2018.00031

IEEE computer society

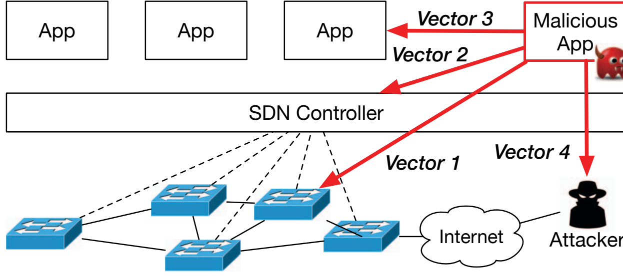

Fig. 1. Threat model and attack vectors.

[20]. We then ran Indago on each of the applications to construct SSBGs. With the SSBGs, Indago successfully identified malicious SDN applications in a fully automated fashion on a total of 73 SDN applications with two clustering methods: reference cluster and sample tagging. Indago achieved high detection rates of 96.42% and 100% with low false-positive rates of 9.67% and 10.53% for each method, respectively.

Our contributions are summarized as follows:

- We present a malware detection approach for SDN. To the best of our knowledge, we are the first in statically detecting SDN malware. Unlike existing work, our technique proactively prevents the spread of malicious SDN applications.
- We introduce a novel data structure, called Security-Sensitive Behavior Graph (SSBG), which encapsulates the semantic behavior of an SDN application.
- We design and implement a fully automated system that effectively detects SDN malware. In our dataset, the system detected SDN malware with high accuracy (around 96% detection rate).
- We summarize the common behaviors of malicious SDN applications observed by our system, which could be potentially used to detect unknown SDN malware.

II. Motivation

Currently, most SDN control platforms (e.g., ONOS [11] and OpenDaylight [16]) provide a rich set of northbound APIs[1] to help any developers easily write their SDN applications. While these open APIs offer a convenient environment for SDN operators, it also creates new opportunities for adversaries. For example, a malicious SDN application could make a network loop or a black hole via the northbound API which updates flow entries and can even control the whole network and shut down its controller. Previous studies (e.g., BlackHat USA 2016 [20], Flow Wars [19], SDN Rootkits [21] and TopoGuard [22]) have introduced several attack scenarios from a malicious SDN application, and they are considered as a real problem in SDNs.

In this study, we summarize known attack cases in four attack vectors, each of which can be exploited by malicious SDN applicaions as presented in Figure 1. We note that our system proactively protects against all these attack vectors by preventing the installation of such malicious applications.

Vector 1. Interference in Data Plane. Malware can control the data plane by manipulating OpenFlow messages. For example, a malicious application can sniff network packets and even reroute legitimate network traffic by putting a forged flow entry on a network device.

Vector 2. Intrusion into SDN Controllers. Malware can interfere in network operations through direct access to the resources of SDN controllers by exploiting controller APIs. For instance, a malicious application can reorder a network service chain, poison the network topology, or fabricate the statistics of network traffic.

Vector 3. Attack on SDN Applications. Malware can kill or run other SDN applications. In particular, deactivating security applications such as firewall and IDS allows an attacker to bypass existing defenses.

Vector 4. Critical Information Leakage. Malware can leak critical information of a network to adversaries. For example, attackers can obtain controller configurations, security options, network status, and flow tables, which can be used to conduct further attacks.

III. System Overview

Indago takes a set of SDN applications as input and outputs a report containing a set of malicious applications identified by our analysis. Figure 2 illustrates the overall architecture of Indago. At a high level, Indago works in two steps: (i) it analyzes SDN applications and constructs their behavior graphs; and (ii) it clusters SDN applications to discriminate malicious ones.

A. Analyzing Behavior

As the first step, Indago builds a Super Control-Flow Graph (SCFG) [23] by parsing the source code of an SDN application. It then performs a context- and flow-sensitive analysis on the SCFG in order to extract a set of API call sequences from the application. The key insight is that the order of API calls can abstract away the execution semantics of an SDN application. Such an abstraction can be intuitively explained since the control plane of SDN is mostly managed through API calls.

However, analyzing the entire set of API calls of an SCFG may overestimate the semantics of an application because the resulting API call sequences can simply contain plenty of APIs that are not relevant to malicious activities. In order to focus on API calls that are security-sensitive, we develop a novel representation of program semantics, called *Security-Sensitive Behavior Graph* (SSBG) (see §IV). Roughly speaking, an SSBG is a graph where every vertex is a call-site of a security-sensitive API that we define in §IV-A

B. Detecting Malicious SDN Applications

After constructing SSBGs from a set of input programs, Indago performs a cluster analysis [24] to detect malicious

[1]The northbound API is an abstraction layer that enables SDN applications at the top of the SDN stack to manipulate lower-level network components.

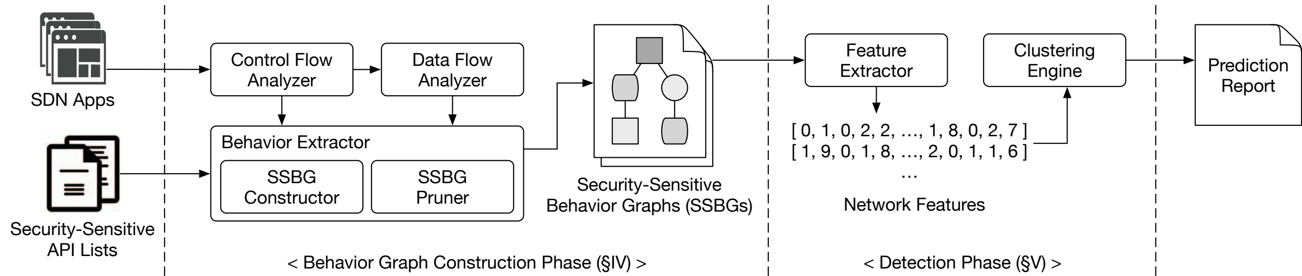

Fig. 2. Indago architecture.

SDN applications. We use semantic features extracted from each of the SSBGs with the help of a data-flow analysis on variables that are related to the security-sensitive APIs. Indago runs k-means clustering algorithm [25] on the derived SSBGs to group semantically relevant applications into clusters. The final decision is made based on the prior knowledge about the dataset: we know exactly which cluster, and which SSBG in the cluster is malicious (see §V).

IV. Security-Sensitive Behavior Graph Construction

In this section, we start by defining what are security-sensitive APIs. We then introduce a novel data structure called Security-Sensitive Behavior Graph (SSBG), which represents the behavioral semantics of an SDN application. Finally, we describe an algorithm to build an SSBG from a given SDN application.

A. Security-Sensitive APIs

Indago statically analyzes API call flows to detect the malicious behavior of an application. Thus, it is desirable to analyze only APIs that can affect the semantic behavior of an SDN application. We note there are some critical SDN APIs that can directly manipulate network configurations, which, in turn, can be abused by malware. In SDN, such APIs are called northbound APIs: SDN applications invoke northbound APIs to interact with SDN controllers. There is an analogy between northbound APIs of SDN and system calls of operating systems: both northbound APIs and system calls are used to access privileged system resources.

As an example, both the `removeFlowRule()` function of ONOS and the `modifyFlow()` function of OpenDaylight allow SDN applications to directly manipulate flow tables in a network switch. These types of APIs are potentially powerful and sensitive enough to directly corrupt network behavior. Also, the `uninstall()` function of OpenDaylight allows an SDN application to arbitrarily terminate legitimate network services, or uninstall other applications. Thus, it can immediately manipulate not only the network behavior but also the status of an SDN controller. Furthermore, the threats from such security-sensitive APIs have also been demonstrated in previous studies [6], [18], [20], [22], [26], [27]. The presented attacks indeed exploit the northbound APIs to corrupt the network/controller behavior.

Since not every northbound API is security-critical, we focus only on a subset of northbound APIs that are capable of manipulating critical SDN assets such as controller, Flow, and Packet. In this paper, we say that a northbound API is *security-sensitive* if it can manipulate the critical assets defined in Table I. To construct the table, we manually identified northbound APIs that are security-sensitive from three popular open-source SDN controllers: ONOS [11], OpenDaylight [16], and Floodlight [17]. We found that there are 440, 489, and 450 security-sensitive APIs on such controllers, respectively. Table I summarizes the list of critical SDN assets and the example of security-sensitive APIs.

B. Security-Sensitive Behavior Graph

A Security-Sensitive Behavior Graph (SSBG) represents the semantics of a program with a set of API calls. The idea of describing program semantics with API calls is not new: previous studies [28], [29] use a data structure called *behavior graph*, which models data dependencies between system calls captured from an execution trace.

Unfortunately, existing approaches on detecting malware with behavior graphs suffer from two major challenges. First, since behavior graphs are generated from an execution, it does not represent the whole semantics of a program: there are API calls that cannot be observed in a single execution, and thus, we may not be able to see any malicious behavior in the execution even though the program is malicious. Second, data dependencies between API calls in a behavior graph are simply based on their parameter and return values. For example, an edge is introduced in a behavior graph ($A \rightarrow B$) when a system call A returns a value 42, and the same value 42 is used as a parameter in a system call B. This may produce false edges between system calls, since the use of a certain value may result from various sources.

These problems motivate our design of SSBG. Unlike traditional behavior graphs where only an individual path is considered at a time, SSBGs are generated from a static data-flow analysis. That is, an SSBG naturally captures data-flow semantics from multiple paths. Furthermore, every edge in an SSBG is created only when there exists a data dependency between two API calls. Thus, it is less likely to have false edges compared to the traditional ones.

TABLE I
THE NUMBER OF SECURITY-SENSITIVE APIS FOR EACH SDN ASSET ON POPULAR SDN CONTROLLERS. THE NUMBER IS MARKED '-' WHEN THE CONTROLLER DOES NOT PROVIDE THE CORRESPONDING ASSET.

Assets	Related Resources	Examples of Security-Sensitive APIs	The Number of APIs		
			ONOS	OpenDaylight	Floodlight
Application	configuration, life cycle, OSGi	`getBundles()`, `uninstall()`, `unregister()`	13	15	-
Controller	property, distributed instance	`getLocalNode()`, `getClusteredControllers()`	44	52	24
Device	node in data plane	`getDpid()`, `removeNodeAllProps()`	46	71	88
Flow	rule, flow table	`installRule()`, `removeFlow()`, `getFlowEntry()`	107	108	15
Host	host in data plane	`getAllHosts()`, `getHostAddress()`, `replaceHost()`	15	57	-
Intent	ONOS intent	`ingressPoint()`, `appId()`, `fingerprint()`	45	-	-
Link	link in data plane	`getDeviceLinks()`, `addLink()`, `updateLink()`	21	-	28
OpenFlow	OpenFlow message	`buildPacketOut()`, `buildFlowModify()`	-	-	113
Packet	control packet	`sendARPReply()`, `getPayload()`, `setPayload()`	64	64	92
Routing	routing decision	`setupForwardingPaths()`, `pushRoute()`	68	11	30
Topology	network topology	`currentTopology()`, `getPaths()`	17	85	60
User	controller user	`getUserRoles()`, `getGrantedAuthorities()`	-	26	-
Total			440	489	450

We now formally define what an SSBG is. Let Σ_α be the universe of all possible northbound API calls. A northbound API $\alpha \in \Sigma_\alpha$ is a function of k arguments, $\alpha : a_1 \times \cdots \times a_k \to r$, where a_i is the type of the i-th argument and r is the type of the return variable. We denote the set of security-sensitive APIs (defined in §IV-A) by Σ_S, where $\Sigma_S \subset \Sigma_\alpha$. The Security-Sensitive Behavior Graph (SSBG) is a directed graph obtained from a program, where each node is an API function $\alpha \in \Sigma_\alpha$, and each edge from a node α_i to α_j indicates that α_i can call α_j during the execution of the program. Each node in the graph is labeled with a set of use-def chains C. A use-def chain maps a parameter of a node to its *use* variables [23]. We define a function γ that takes in a node α_j and an index i, and returns the corresponding use-def chain for the i-th parameter of α_j. Formally, an SSBG is a directed graph $G = (V, E, \gamma)$, where:

- V is the set of nodes, each of which represents a northbound API call $\alpha \in \Sigma_\alpha$.
- E is the set of edges ($E \subseteq V \times V$), each of which indicates a call from a node to another.
- $\gamma : V \times \mathbb{N} \to C$ is a function that associates i-th variable in a function call α with a set of use-def chains C, where $i \in \mathbb{N}$ and $\alpha \in V$.

C. The Algorithm

Indago builds an SSBG in two steps: (i) the SSBG Constructor module creates an SSBG for a given application; and (ii) the SSBG Pruner module refines the SSBG to effectively analyze the behavior of the program. In this section, we describe each step of Indago in detail.

1) SSBG Constructor: Indago builds an SSBG for a given application as it traverses a Super Control-Flow Graph (SCFG) [23] of the program. Specifically, it finds all the call sites of northbound APIs as it walks the SCFG. For every call site found, Indago performs the following three steps. First, it adds two nodes—one node for the caller API and the other for the target API—to the SSBG if they do not already exist in the graph. Indago then connects the two nodes with a directed edge from the caller to the callee. Finally, Indago labels each parameter of the callee node with a use-def chain. Algorithm 1 illustrates how the *SSBG Constructor* module builds an SSBG from an SCFG.

Algorithm 1 SSBG Constructor.

```
1: //  g_scfg: an SCFG
2: //  v_scfg: a pointer to the current node of g_scfg
3: //  g_ssbg: an SSBG
4: //  v_ssbg: a pointer to the current node of g_ssbg
5: procedure BUILDSSBG(g_scfg, v_scfg, g_ssbg, v_ssbg)
6:     for all direct successor v_succ of v_scfg do
7:         if IsNotVisited(v_succ) then
8:             if HasNorthboundAPICall(v_succ) then
9:                 C ← getUseDefChains(v_succ)
10:                s ← getAPIName(v_ssbg)
11:                d ← getAPIName(v_succ)
12:                g_ssbg ← updateSSBG(g_ssbg, s, d, C)
13:                g_ssbg ← BuildSSBG(g_scfg, v_succ, g_ssbg, v_succ)
14:            else
15:                g_ssbg ← BuildSSBG(g_scfg, v_succ, g_ssbg, v_ssbg)
16:    return g_ssbg
```

Algorithm 2 SSBG Pruner.

```
1: //  g_ssbg: an SSBG
2: //  v_ssbg: a pointer to the current node of g_ssbg
3: procedure CANREACHSENSITIVE(g_ssbg, v_ssbg)
4:     if IsSecSensitive(v_ssbg) then return True
5:     else
6:         for all direct successor v_succ of v_ssbg do
7:             if CanReachSensitive(g_ssbg, v_succ) then
8:                 return True
9:     return False
10: //  g_ssbg: an SSBG
11: //  v_ssbg: a pointer to the current node of g_ssbg
12: procedure PRUNE(g_ssbg, v_ssbg)
13:     for all direct successor v_succ of v_ssbg do
14:         if CanReachSensitive(g_ssbg, v_succ) then
15:             g_ssbg ← Prune(g_ssbg, v_succ)
16:         else
17:             g_ssbg ← RemoveNode(g_ssbg, v_succ)
18:     return g_ssbg
```

The `BuildSSBG()` function takes in four arguments as input: an SCFG of a program (g_{scfg}), a pointer to a node in g_{scfg} (v_{scfg}), an SSBG (g_{ssbg}), and a pointer to a node in g_{ssbg}

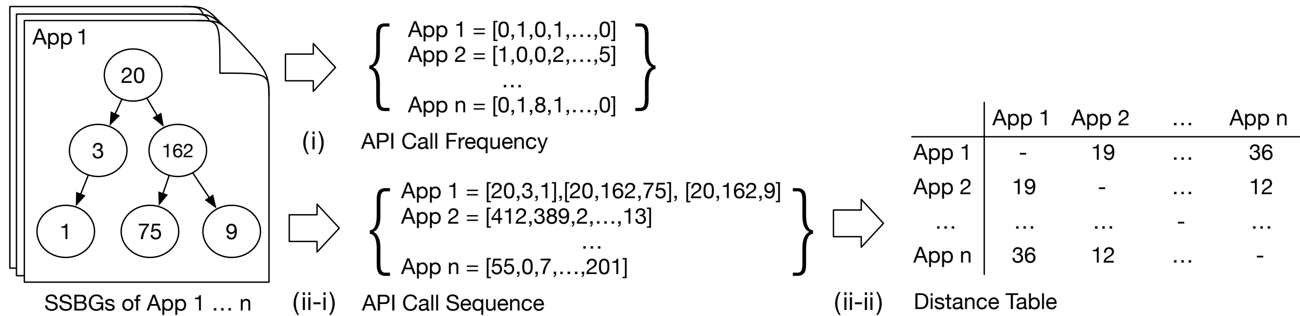

Fig. 3. SSBG-based feature vector extraction: (i) the frequency, and (ii) the sequence of security-sensitive API calls.

(v_{ssbg}). The function initially takes an entry point (e.g., main function) for v_{scfg}, an empty SSBG for g_{ssbg}, and a NULL pointer for v_{ssbg}. The function returns the resulting SSBG as output by recursively traversing the nodes in g_{scfg}. Indago may call this function iteratively with a list of entry points (v_{scfg}).

For every direct successor v_{succ} of the current node in the v_{scfg} (Line 6), and the `BuildSSBG()` function updates g_{ssbg} when the following two conditions hold: (i) the successor is a newly visited node (Line 7); and (ii) the successor ends with a call statement that jumps to a northbound API (Line 8). The first condition guarantees that our algorithm terminates, and the second condition is to satisfy the definition of the SSBG. The `getUseDefChains()` function in Line 9 returns a set of use-def chains for every parameter of the target API (v_{succ}) via a traditional intra-procedural backward data-flow analysis, called liveness analysis, and the set is used to update g_{ssbg} in the `updateSSBG()` function. The `getAPIName()` takes in a basic block of an SCFG and returns the corresponding API name. We use this function to construct nodes for g_{ssbg}. The `updateSSBG()` function updates g_{ssbg} by adding two nodes s and d (if not exist) as well as an edge from the current node to the call target (v_{succ}). When adding the successor v_{succ}, `updateSSBG()` annotates v_{succ} with the use-def chains C, which are used as one of the features for a cluster analysis in §V-A. This process recursively runs until we have no more successors to explore.

2) SSBG Pruner: Indago post-processes the SSBG generated from the first step (§IV-C1) in order to reduce the size of it. The key insight is that having only a set of security-sensitive APIs in an SSBG is enough for describing malicious semantics of a program. We refine the SSBG by removing the nodes that cannot reach any of the security-sensitive APIs we defined in §IV-A through a chain of function calls.

Algorithm 2 describes the refinement procedure. The `Prune()` function traverses all the successors of a given SSBG g_{ssbg} in a depth-first manner. For every successor it encounters, the `CanReachSensitive()` function in Line 14 checks if any of the successors of v_{succ} including v_{succ} itself is security sensitive (Line 3-9). If so, we keep traversing the graph. Otherwise, we remove the node from g_{ssbg} (Line 17). The final result is a reduced SSBG that only contains nodes that are security-relevant.

V. Malware Detection

After building the Security-Sensitive Behavior Graph (SSBG) of an SDN application, Indago conducts a final analysis to investigate whether the application is malicious or not. This analysis consists of two steps: (i) feature selection; extracts discriminative features from an SSBG and (ii) cluster analysis; performs the clustering to makes the last decision.

A. Feature Selection

Indago takes three features from an SSBG; (i) the frequency of security-sensitive API calls, (ii) the sequence of security-sensitive API calls, and (iii) northbound interactiveness.

1) Frequency of Security-Sensitive API Calls: Detecting malware with the frequency of critical function calls (e.g., system calls) is not a new idea. However, in the case of an SDN environment, it is a quite challenging issue, because we need to figure out which APIs are critical or sensitive for detecting malware, which has not been studied so far. Moreover, the target resources of SDN APIs are quite different from those of legacy APIs from other domains (e.g., SDN APIs mostly handle network resources).

In this regard, we first carefully inspect most APIs of several open-source SDN controllers and build security-sensitive API lists to determine which are critical ones (see §IV-A). Based on the lists, we figure out a vertex of an SSBG which has a security-sensitive API call through a similarity-based string comparison technique, Longest Common Subsequence (LCS) [30].

As illustrated in Figure 3 (i), Indago takes in a set of SSBGs, and traverses every node of each SSBG to compute how many security-sensitive API calls are invoked. When computing the number of API calls, we consider the semantics of each call to aggregate semantically-related API calls to bring efficiency to a malware classification. For example, both `installRule()` and `removeFlow()` of ONOS belong to a *Flow* asset. Thus, we combine each number of API calls belong to the *Flow* asset to obtain the total number of *Flow*-sensitive API calls (The assets are listed in the previous section IV, Table I).

2) Sequence of Security-Sensitive API Calls: Additionally, we consider the sequence of security-sensitive API calls as one of the features to detect malicious SDN applications. Figure 3 (ii-i) and (ii-ii) show how Indago organizes such feature from the call sequences.

We assign a unique unsigned integer value, *API ID* to each API not only to determine the API call sequences of an SDN application but also to reduce computational overheads occurred by a string comparison method for quit long file names. Once all the APIs have been mapped to each API ID, Indago discovers security-sensitive API call sequences of an application from its SSBG.

However, such lists of integers (call sequences) do not reflect the dissimilarity between benign and malicious applications. For example, when a malicious application has an API call pattern similar to that of other benign application, then it makes hard to differentiate a malicious application (with this intuitive feature only) from benign ones. They also cannot be directly used as a feature for a cluster analysis because each of them has a different length of entries about the number of its API calls.

To address these challenges, we apply a data comparison technique. Particularly, we use a correlation information in each application, which shows how API call sequences differ from others. As shown in Figure 3 (ii-ii), we construct a distance table with an n-by-n matrix, where n is the number of applications, and each cell in the table indicates the Levenshtein distance [31] between security-sensitive API call sequences of two applications.

In addition, a single SDN application could have various API call sequences due to branch instructions in a program. For example, a routing application may use an *'if'* statement to either flood a packet or install a new flow entry to a network device for each *packet_in* message, and since both operations are security-sensitive, there exist two different call sequences in an SSBG.

To handle this issue, we also measure distances between each sequence extracted from an application and every other application. Then, we take the average value of the distances as a final distance value of an application.

3) Northbound Interactiveness: The third feature that we leverage to detect SDN malware is the northbound interactiveness. Since common legitimate SDN applications aim to provide useful network services, they actively interact with an SDN controller to make meaningful networking decisions via various northbound interfaces. In contrast, malicious ones try to corrupt networks or a controller itself, and thus, they rarely interact or exchange meaningful information with a controller. Based on this intuition, we define such interaction between SDN applications and a controller as the notion of *northbound interactiveness* and use it to distinguish malicious SDN applications from benign ones.

To figure out the northbound interactiveness of SDN applications, we examine how often each application exchanges *meaningful* information with a controller. This raises the following two questions: *how can we track down the information exchanges?* and *how can we determine if the information exchange is meaningful or not?*

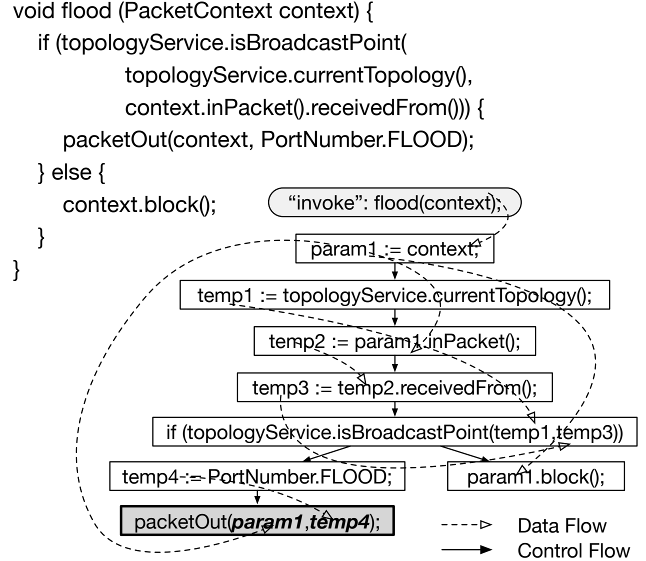

Fig. 4. An example of northbound interactiveness extraction for Reactive Forwarding application of ONOS. We extract the Northbound interactiveness feature by performing data-flow analysis against `packetOut()` and its variables.

To begin with, we track down each parameter of northbound API calls that have been initialized in a controller by referencing its use-def chain. When the parameters supplied to a northbound API call (made by an application) are declared and initialized within the controller, not an application, we consider that an application exchanges information with a controller (i.e., makes app-controller information exchange).

Also, since not all of these exchanges are necessarily *meaningful*, directly using the number of the exchanges as a clustering feature may affect the detection accuracy. Thus, we leverage an SSBG to eliminate negligible API calls. To be specific, we quantify the northbound interactiveness by only counting the number of security-sensitive API calls from an SSBG.

We now describe how we compute the northbound interactiveness with an example shown in Figure 4. On the top side, there is the code snippet of *Reactive Forwarding* application running on ONOS controller and its control-flow is shown on the bottom side with data-flow expressions, i.e., use-def chains. For this example, let us consider `packetOut()` as a security-sensitive API and we have performed a data-flow analysis against its two parameters, *param1* and *temp4*. We backtrack the use-def chains from an API call node using the variables to a node defining the variables. By doing so, we can discover where the variables are defined and whether those are outside of the caller method or not. In the example, the application receives data from its controller (i.e., as a parameter of `flood()`), and that is indicated by *param1*, which is a meaningful data (i.e., packet context) and used as

a variable of an API call node. Indago counts the number of such variables and API calls like this example and uses that number as one of the features, *northbound interactiveness* for a malware clustering.

B. Cluster Analysis

In the case of SDN environments, since it is difficult to collect sufficient application samples from public domains to build a training model for a supervised learning approach, we have no choice but to evaluate Indago with a limited dataset available (we discuss this issue in §VII later). Thus, we decide to use an unsupervised learning method (i.e., a k-means clustering algorithm [25]) to partition SDN applications into benign and malicious clusters instead of other classification methods.

To finally determine whether an SDN application is malicious or not, we employ the following two approaches: (i) reference cluster and (ii) sample tagging.

For the reference clustering approach, we compare our clustering results with an existing reference cluster model. However, since there is no previous clustering result (i.e., note that our work is the first trial to detect malicious SDN applications), we first need to build a reference model to evaluate an Indago prototype. For this, we build a reference model that consists of a set of randomly selected samples from our collection (details about the collection are depicted in the evaluation part, §VI). Based on the reference model, we regard that an SDN application is malicious when the application is included or close to a reference cluster for malicious applications.

Besides the above reference clustering approach, we employ another method, *sample tagging*, to evaluate our system in various ways. To be specific, we conduct clustering with a known dataset including a set of randomly sampled applications (around 20% of whole samples), each of which is attached with a tag indicating either malicious or benign. At the end of the cluster analysis, we conclude that any application in a cluster having more malicious tags than benign ones is malicious.

VI. Evaluation

We have implemented a Indago prototype and evaluated it on the collection of SDN applications running on the most popular open-source SDN controllers. In this section, we describe the collection followed by several experiments we performed.

A. Implementation and DataSet

We have built our prototype in approximately 1.1K lines of Java code on top of two different open-source projects; Soot [32] and Weka [33]. We leverage Soot to perform a control-flow analysis (CFA) as well as a data-flow analysis (DFA) on SDN applications. To carry out a cluster analysis (i.e., k-means clustering), Indago uses Weka, a popular machine learning software.

TABLE II
THE # OF APPS WE COLLECTED FROM PUBLIC DOMAINS.

	Benign	Malicious	**Total**
ONOS 1.3	16	13	29
OpenDaylight Helium SR3	11	14	25
Floodlight 1.1	11	8	19
Total	38	35	73

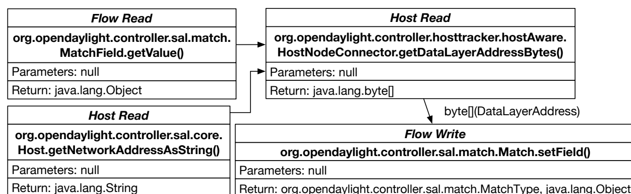

Fig. 5. An SSBG of Firmware Misuse malware running on an OpenDaylight controller. This malware manipulates the match field of a flow entry installed in a target switch. Use-def chains are omitted for a concise view.

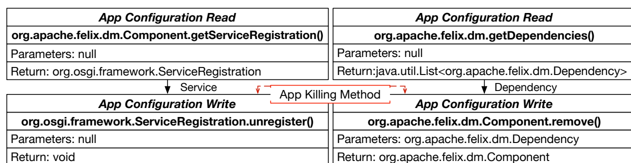

Fig. 6. An SSBG of Service Unregistration malware running on an OpenDaylight controller. This malware modifies a configuration of its controller to kill other applications. Use-def chains are omitted for a concise view.

TABLE III
CLUSTERING RESULT WITH A REFERENCE CLUSTER MODEL.

	True Positive	True Negative	False Positive	False Negative
ONOS	13	14	2	0
OpenDaylight	14	10	1	0
Floodlight	8	10	1	0
Total	35	34	4	0

To evaluate the effectiveness of Indago, we have attempted to collect as many SDN applications as possible. In total, we have gathered 73 (38 benign and 35 malicious) SDN applications that are compatible with three popular open-source SDN controllers; ONOS [11], OpenDaylight [16] and Floodlight [17]. All those SDN controllers provide default applications such as *Routing* and *Forwarding*, and we consider them as benign applications. Besides, malicious applications are collected from several SDN security-related projects; SDNSecurity.org [18], Flow Wars [19] TopoGuard [22] and DELTA [27]. Table II summarizes our collection.

B. Example of SSBG

We introduce SSBGs of two malicious SDN applications to explain the capability of a SSBG representation. Flow Wars [19] gives full details of such applications.

1) Firmware Misuse: Commonly, an SDN-enabled network device includes two types of flow table to handle network

TABLE IV
THE NUMBER OF APPLICATIONS PARTITIONED INTO EACH CLUSTER WITH A SAMPLE TAGGING METHOD. THE '-' INDICATES THAT FLOODLIGHT PRODUCED ONLY FOUR CLUSTERS IN OUR EXPERIMENTS. WE MARK THE NUMBERS BELONGING TO A MALICIOUS CLUSTER WITH BOLD TYPE.

	ONOS				OpenDaylight				Floodlight			
	Untagged		Tagged		Untagged		Tagged		Untagged		Tagged	
	Benign	Malicious	Benign	Malicious	Benign	Malicious	Benign	Malicious	Benign	Malicious	Benign	Malicious
Cluster A	2	0	0	0	2	0	0	0	2	0	0	0
Cluster B	2	0	1	0	1	0	1	0	**1**	**6**	**0**	**2**
Cluster C	**1**	**10**	**0**	**2**	2	0	0	0	5	0	2	0
Cluster D	1	0	0	0	**1**	**11**	**0**	**3**	1	0	0	0
Cluster E	7	1	2	0	3	0	1	0	-	-	-	-

TABLE V
CLUSTERING RESULT WITH A SAMPLE TAGGING METHOD.

	True Positive	True Negative	False Positive	False Negative
ONOS	10	12	1	1
OpenDaylight	11	8	1	0
Floodlight	6	8	1	0
Total	27	28	3	1

flows; (i) hardware table and (ii) software table. Although the software table is relatively flexible and handles much more entries than the hardware table, its performance is much worse than the hardware table. This malware exploits such a situation, and it tries to move most flow rules to the software table from the hardware table, and then the performance of handling network flows will be remarkably bad. To conduct the attack (the overall scenario is presented in Figure 5), a malicious application reads a match field of a flow rule stored in a flow table (*Flow Read*) and rewrites the match field (*Flow Write*) to move the flow rule to a software table. As explained, all critical operations of this malware are clearly presented in an SSBG.

2) Service Unregistration: This malicious application aims to kill other applications running on its controller. As shown in Figure 6, the malware unregisters the event listener of a target application via `unregister()`, and then with `remove()`, it deletes all dependencies related to a target application. To be precise, the malware does not completely uninstall a target application; rather it throws a victim out of the list of event listeners. That said, a victim can no longer interact with a controller and that is almost equivalent to completely uninstalling a target application.

C. Detection Effectiveness

We now describe about the results of our experiments to verify the detection accuracy of Indago. Recall from §V-B, the Indago prototype takes two different clustering approaches to detect malicious SDN applications; *reference cluster* and *sample tagging*.

In the reference cluster approach, we use the *5-fold cross validation*. That is, we randomly split a dataset into five disjoint subsets and iteratively perform the cluster analysis against each subset. For each iteration, Indago builds reference clusters with four subsets, and remaining single subset is used for detection. Meanwhile, for the sample tagging approach, we have randomly chosen 20% of the dataset and have tagged each of them as benign or malicious. Once all the applications are classified, the applications that belong to the clusters having more malicious tags than benign ones are considered as malicious ones.

During the experiments, Indago has detected all malicious SDN applications completely (100% detection rate) with 10.53% false positive rate in the reference cluster approach; while with sample tagging method, has detected 27 out of 28 malicious SDN applications (96.42%) with three false positives (9.67%).

Table III describes the results of our experiments with a reference-cluster-based detection method. In total, Indago has detected all malicious applications in our dataset with four false positives. Table IV and V show the clustering results with a sample-tagging-based detection method. For ONOS applications, *Cluster C* is classified as malicious, and Indago has detected ten malicious ONOS applications (true positives) with a false positive. In the case of OpenDaylight applications, *Cluster D* turns out to be a malicious cluster and, all the malicious applications are detected with a false positive. Indago also has detected all of the malicious Floodlight applications completely with a false positive and the *Cluster B* is discovered as a malicious cluster.

For the experiments, the system fails to detect only one malware (a false negative) and this is because the malware has quite simple logic compare to other samples. Also, from the careful inspection of program source code, we have found that the false positives (benign applications, actually) have quite similar behaviors to those of other malicious ones. What this means is that an adversary probably has an opportunity to evade the proposed detection technique by writing more sophisticated malware. We address this evasion issue and its mitigation methods in §VII.

D. Performance Measurement

We have measured the end-to-end performance of Indago and the measurements were performed on a single Linux machine of Intel Xeon E3-1231 v3 with 16 GB of memory. We have ran the prototype on our dataset for five times and have took the average value of execution time for code analysis. As shown in Table VI, Indago has successfully analyzed each application of our collection within 10 seconds and this highlights that the proposed system has practicality to apply

TABLE VI
ANALYSIS TIME (SECOND) FOR USE CASES.

	min	max	avg	std
ONOS Benign Apps	0.64	3.55	2.10	0.88
ONOS Malicious Apps	0.32	2.36	1.54	0.96
OpenDaylight Benign Apps	1.05	3.22	2.01	0.68
OpenDaylight Malicious Apps	0.32	2.62	1.70	1.03
Floodlight Benign Apps	6.02	7.76	6.94	0.58
Floodlight Malicious Apps	0.35	6.99	2.82	3.33

to a real-world system. We also note that our system does not influence the performance of an SDN controller because it *statically* inspects SDN applications prior to run them on a controller.

E. Insights

Several security-sensitive APIs used in malicious applications are *rarely* used in benign applications. For instance, malicious SDN applications commonly invoke `unregister()` and `remove()` to unregister other applications from the list of event listeners. Although such APIs exchanging some information across two applications are necessary to conduct network functions for the SDN environment, they are likely to be used by malicious applications *NOT* by benign applications in general. Also, when a malicious application misuse them, it is possible that the serious security problems occur such as an application killing.

Moreover, we found that 20 and 24 APIs of ONOS and OpenDaylight, respectively, are *never* used in benign applications. Thus, we can naturally call such APIs as a *malicious only API*. For example, `getBundleId()` and `uninstall()` are used for malicious purposes (e.g., to kill other applications [20]), whereas none of the default applications does not use these APIs. Namely, such security-sensitive APIs (malicious only) are more useful for malicious applications to achieve their malicious goals than any other benign functions although those are supported by SDN controllers to meet the needs of innocent functions.

The usage of such APIs gives us valuable insights into the malicious behaviors of SDN applications. We can consider it as the unique features of malicious SDN applications in the future, and restriction mechanisms against the usage of such APIs, meanwhile, are required for secure SDN environments.

VII. DISCUSSION

A. Small Sample Set

We have tested our system with 73 SDN applications including both benign and malicious ones, and this number may not be enough for clearly verifying the effectiveness of the proposed system. Specifically, compared with the evaluation cases of other malware detection systems for other platforms (e.g., Android malware detection), our testing sample size is relatively small.

However, we argue that the characteristics of an SDN application are quite different from that of an application of other domains, where a lot of applications are designed for personal uses and are easily accessible on a vast online market such as Google Play Store. In contrast, SDN applications are commonly used for managing diverse network devices (enterprise-level), and thus we cannot easily capture those samples in public.

Nevertheless, note that our collection includes most (nearly all) SDN application samples in public domains, provided by most well-known open-source SDN projects. Moreover, since they are commonly used in real world [3]–[5], we believe that our evaluation results are sufficiently valuable. Also, Indago presents the high accuracy of SDN malware detection even with (relatively) a small number of samples, which implies that it has possibilities for producing better results with more samples.

B. Limitations of Static Analysis

Static analysis methods that we have used in this work have fundamental limitations. For instance, Indago cannot inspect the code generated at runtime. Java employs a wild card keyword *'?'* to support the concept of polymorphism and thus, diverse Java applications commonly use such keyword in their program to reference objects at run-time. When a variable is annotated with such wild card keyword, it means the variable can has one of the upper/lower bounded types depending on the runtime context. To overcome this limitation, we have manually linked such keyword with a base type which is one of the candidates and this leads Indago to run with no errors in our experiments.

Also, our implementation does not handle implicit call sequences that are caused by callback functions. In other words, when a function A registers a function pointer for B as a callback, we could not add an edge from A to B in our SSBG. However, we found that none of the SDN applications in our dataset use a northbound API as a callback.

In addition to a static analysis method, several dynamic analysis techniques [34] can certainly resolve the limitations, and we leave it as future work to combine static and dynamic analysis approaches to detect malicious SDN applications.

C. Evasion Technique and Mitigation

Since statistical methods have fundamental limitations [35], [36] an adversary could evade a detection system which leveraged the kind of machine learning approaches by producing a sophisticated malware. Obviously, classification methods coming from statistics involve error rates; false positives and negatives, and our system is no an exception to such errors.

By manipulating the limitation, an adversary can make a detection system report her malicious application as a benign one (produce a false positive). In other words, by writing a malicious application highly similar to a benign one, the application is classified as a benign one, even though it is malicious in fact.

Also, a malware author could position an applications arbitrarily far from any cluster, or could increase the cost of a classification process remarkably by adding enormous erratic

behaviors on her application (e.g., a bunch of dummy API calls).

To mitigate such evasion techniques, we can use additional features for a cluster analysis. For example, weighted features can represent the importance of application's behavior more accurately [37]. What this means is that we can decrease the error ratio of a cluster analysis by assigning weights to different nodes of an SSBG, depending on their importance factor. We also can pattern the access of network resources of an application with a data flow analysis, and use it as an additional feature because the data access model can reveal the malicious behavior of an SDN application.

Thus, we believe that a plentiful feature set could make an adversary hard to evade the proposed detection system.

VIII. Related Work

A. SDN Security

There are several solutions to make SDN, especially its control plane, secure, Rosemary [6] leverages access control to provide robust, secure SDN controller and SE-Floodlight [7] applies security-enforcement kernel layer to a network OS. SDNShield [10] and Security-Mode ONOS [9] are permission control systems to enforce only the minimum required privileges to SDN applications.

NICE [12] uncovers bugs like a duplicated packet transmission in SDN applications with a symbolic execution, while we enforce a behavior analysis on SDN applications and detect malicious ones. Our previous study [38] has presented an analysis framework for SDN applications capturing the behavior patterns of open-source SDN applications, but it could not distinguish malicious applications from benign applications.

The previous studies provide practical approaches to make SDN secure, but none of them provides a way of detecting malicious SDN applications beforehand. We take an approach theoretically differs from those works. To improve the security of an SDN environment, the previous studies mainly focused on the SDN control plane itself and have redesigned an SDN architecture. On the contrary, we change our viewpoint to SDN applications themselves. In other words, Indago prevents the installation of malicious SDN applications by detecting the applications before running on its controller *without any changes* in the control plane. That said, we believe our work is complementary to them.

B. Malware Analysis and Detection

Static analysis methods are often used in various previous studies for a detecting malware. Christodorescu et al. [39] and Kruegel, et al. [40] use the semantics and structure of the binary code to detect malware, and tracing system call dependency with a symbolic execution approach is used to detect malware [29], [41]. Cha et al. [42] use a specialized form of bloom wilter, termed feed-forward bloom filter, to reduce both time and space overhead of static malware detection. Nedim et al. [36] statically detect malicious PDF files by using their hierarchical document structure.

Behavior analysis is also commonly used for detecting malware, and previous studies [28], [29] present the behavior graph of a program with its use of system calls. They use a heuristic method (i.e., an *actual value* of a system call handle) to represent data dependencies between system calls, whereas we use the data type of an API parameter.

Also, malware classification through a program analysis with some machine learning approaches are introduced in Android platform, briskly [37], [43]–[45]. MaMaDroid [45] brings the concept of Markov chain to discover the sequence of API calls performed by an application, while we use distance vectors to handle a multiple sequence issue.

These all previous studies have detected malware effectively, and our work is technically inspired by them. However, none of them can directly be applied to SDN environment because the SDN operating environment is quite different from legacy ones. To be specific, these could not be achieved by applying existing studies directly because of the inability of inspecting SDN-specific behaviors. In this paper, we propose a novel method to inspect the behavior of an SDN application and denote it as a Security-Sensitive Behavior Graph (SSBG).

IX. Conclusion

A malicious SDN application can compromise the entire SDN control plane and then it can do nearly every network functions without any restriction. To protect the SDN control plane proactively, we proposed a new framework, called Indago, which statically analyzes SDN applications and describes their behavioral patterns in order to automatically detect malicious SDN applications. The proposed system draws the Security-Sensitive Behavior Graphs (SSBGs) of SDN applications and extracts meaningful features from the SSBGs, which are used to conduct a machine learning algorithm (e.g., cluster analysis) to detect malicious SDN applications. We have evaluated a prototype system on our dataset all collected from public domains and have demonstrated that the proposed system achieves high detection accuracy with low false positive rates.

Acknowledgment

This work was partly supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No.2016-0-00102, Building a Platform for Automated Reverse Engineering and Vulnerability Detection with Binary Code Analysis). This work was also supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2018-0-00254, SDN security technology development).

References

[1] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hőlzle, S. Stuart, and A. Vahdat, "B4: Experience with a Globally-Deployed Software Defined WAN," in *Proceedings of the ACM SIGCOMM Conference on SIGCOMM*, vol. 43, no. 4, 2013, pp. 3–14.

[2] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving High Utilization with Software-Driven WAN," in *Proceedings of the ACM SIGCOMM Conference on SIGCOMM*, vol. 43, no. 4, 2013, pp. 15–26.

[3] HP, *SDN Products*, http://h17007.www1.hp.com/cz/en/networking/solutions/technology/sdn/portfolio.aspx\#.V0KupLiLQUE.

[4] Cisco, *Open SDN Controller*, http://www.cisco.com/c/en/us/products/cloud-systems-management/open-sdn-controller/index.html.

[5] NEC, *SDN Solutions*, http://www.projectfloodlight.org/floodlight/.

[6] 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 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 78–89.

[7] P. Porras, S. Cheung, M. Fong, K. Skinner, and V. Yegneswaran, "Securing the Software-Defined Network Control Layer," in *Proceedings of the Network and Distributed System Security Symposium*, 2015.

[8] B. Chandrasekaran and T. Benson, "Tolerating SDN Application Failures with LegoSDN," in *Proceedings of the ACM Workshop on Hot Topics in Networks*. ACM, 2014, p. 22.

[9] 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 *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC)*. IEEE, 2017, pp. 461–473.

[10] X. Wen, B. Yang, Y. Chen, C. Hu, Y. Wang, B. Liu, and X. Chen, "SDNShield: Reconciliating Configurable Application Permissions for SDN App Markets," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2016, pp. 121–132.

[11] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "ONOS: Towards an Open, Distributed SDN OS," in *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. ACM, 2014, pp. 1–6.

[12] M. Canini, D. Venzano, P. Peresini, D. Kostic, J. Rexford *et al.*, "A NICE Way to Test OpenFlow Applications," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, vol. 12, no. 2012, 2012, pp. 127–140.

[13] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, Behavior-Based Malware Clustering," in *Proceedings of the Network and Distributed System Security Symposium*, 2009.

[14] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer, "Behavior-based Spyware Detection," in *Proceedings of the USENIX security symposium*, vol. 6, 2006.

[15] C. Willems, T. Holz, and F. Freiling, "Toward Automated Dynamic Malware Analysis Using CWSandbox," *Proceedings of the IEEE Symposium on Security and Privacy*, no. 2, pp. 32–39, 2007.

[16] J. Medved, R. Varga, A. Tkacik, and K. Gray, "OpenDaylight: Towards a Model-Driven SDN Controller Architecture," in *Proceedings of the IEEE International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*. IEEE, 2014, pp. 1–6.

[17] FloodlightProject, *Open SDN Controller*, http://www.projectfloodlight.org/floodlight/.

[18] SDNSecurity.org, http://SDNSecurity.org/.

[19] C. Yoon, S. Lee, H. Kang, T. Park, S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Flow Wars: Systemizing the Attack Surface and Defenses in Software-Defined Networks," *IEEE/ACM Transactions on Networking*, vol. 25, no. 6, pp. 3514–3530, 2017.

[20] Blackhat2016, *Attacking SDN infrastructure: are we ready for the next-gen networking?*, https://www.blackhat.com/us-16/briefings.html#attacking-sdn-infrastructure-are-we-ready-for-the-next-gen-networking.

[21] C. Röpke and T. Holz, "SDN Rootkits: Subverting Network Operating Systems of Software-Defined Networks," in *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. Springer, 2015, pp. 339–356.

[22] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures," in *Proceedings of the Network and Distributed System Security Symposium*, 2015.

[23] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison Wesley.

[24] L. Kaufman and P. J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley & Sons, 2009, vol. 344.

[25] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An Efficient k-Means Clustering Algorithm: Analysis and Implementation," *Proceedings of the IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 7, pp. 881–892, 2002.

[26] S. Shin and G. Gu, "Attacking Software-Defined Networks: A First Feasibility Study," in *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. ACM, 2013, pp. 165–166.

[27] S. Lee, C. Yoon, C. Lee, S. Shin, V. Yegneswaran, and P. A. Porras, "DELTA: A Security Assessment Framework for Software-Defined Networks," in *Proceedings of the Network and Distributed System Security Symposium*, 2017.

[28] M. Christodorescu, S. Jha, and C. Kruegel, "Mining Specifications of Malicious Behavior," in *Proceedings of the India Software Engineering Conference*. ACM, 2008, pp. 5–14.

[29] 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 *Proceedings of the USENIX security symposium*, 2009, pp. 351–366.

[30] L. Bergroth, H. Hakonen, and T. Raita, "A Survey of Longest Common Subsequence Algorithms," in *Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE)*. IEEE, 2000, pp. 39–48.

[31] T. Kohonen and P. Somervuo, "Self-Organizing Maps of Symbol Strings," *Neurocomputing*, vol. 21, no. 1, pp. 19–30, 1998.

[32] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The Soot framework for Java program Analysis: a Retrospective," in *Proceedings of the Cetus Users and Compiler Infrastructure Workshop (CETUS)*, 2011.

[33] Weka, *Data Mining Software in Java*, http://www.cs.waikato.ac.nz/ml/weka/.

[34] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A Survey on Automated Dynamic Malware-Analysis Techniques and Tools," *ACM Computing Surveys (CSUR)*, vol. 44, no. 2, p. 6, 2012.

[35] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*. IEEE, 2007, pp. 421–430.

[36] N. Šrndic and P. Laskov, "Detection of Malicious PDF Files Based on Hierarchical Document Structure," in *Proceedings of the Network and Distributed System Security Symposium*, 2013.

[37] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1105–1116.

[38] C. Lee and S. Shin, "SHIELD: An Automated Framework for Static Analysis of SDN Applications," in *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*. ACM, 2016, pp. 29–34.

[39] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-Aware Malware Detection," in *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 2005, pp. 32–46.

[40] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic Worm Detection using Structural Information of Executables," in *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*. Springer, 2005, pp. 207–226.

[41] C. Kruegel, W. Robertson, and G. Vigna, "Detecting Kernel-Level Rootkits through Binary analysis," in *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC)*. IEEE, 2004, pp. 91–100.

[42] S. K. Cha, I. Moraru, J. Jang, J. Truelove, D. Brumley, and D. G. Andersen, "SplitScreen: Enabling Efficient, Distributed Malware Detection," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, 2010, pp. 377–390.

[43] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket," in *Proceedings of the Network and Distributed System Security Symposium*, vol. 14, 2014, pp. 23–26.

[44] 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 *Proceedings of the Network and Distributed System Security Symposium*, vol. 25, no. 4, 2012, pp. 50–52.

[45] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, "MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models," in *Proceedings of the Network and Distributed Systems Security Symposium*, 2017.