Unexpected Data Dependency Creation and Chaining: A New Attack to SDN

Feng Xiao, Jinquan Zhang, Jianwei Huang†, Guofei Gu†, Dinghao Wu, Peng Liu
The Pennsylvania State University
† SUCCESS Lab, Texas A&M University

***Abstract*—Software-Defined Networking (SDN) is an emerging network architecture that provides programmable networking through a logically centralized controller. As SDN becomes more prominent, its security vulnerabilities become more evident than ever. Serving as the "brain" of a software-defined network, how the control plane (of the network) is exposed to external inputs (i.e., data plane messages) is directly correlated with how secure the network is. Fortunately, due to some unique SDN design choices (e.g., control plane and data plane separation), attackers often struggle to find a reachable path to those vulnerable logic hidden deeply within the control plane.**

In this paper, we demonstrate that it is possible for a weak adversary who only controls a commodity network device (host or switch) to attack previously unreachable control plane components by maliciously increasing reachability in the control plane. We introduce D^2C^2 (data dependency creation and chaining) attack, which leverages some widely-used SDN protocol features (e.g., custom fields) to create and chain unexpected data dependencies in order to achieve greater reachability. We have developed a novel tool, SVHunter, which can effectively identify D^2C^2 vulnerabilities. Till now we have evaluated SVHunter on three mainstream open-source SDN controllers (i.e., ONOS, Floodlight, and Opendaylight) as well as one security-enhanced controller (i.e., SE-Floodlight). SVHunter detects 18 previously unknown vulnerabilities, all of which can be exploited remotely to launch serious attacks such as executing arbitrary commands, exfiltrating confidential files, and crashing SDN services.

I. Introduction

In contrast to traditional computer networks, where switches are a "melting pot" of control plane and data plane, Software-Defined Networks (SDNs) keep control plane and data plane separated. While the data plane (of a network) still stays inside switches, the control plane is detached from the data plane and moved to a dedicated server called an *SDN controller*. This separation enables flexible and dynamic network functionalities, makes troubleshooting easier, and leads to the development of an open network programming interface that accelerates the growth of network applications.

As SDN becomes more prominent, the security vulnerabilities become more evident than ever. Serving as the core of SDN networks, the security of SDN control plane [37], [42], [39], [16], [19] receives the most attention from security researchers. In particular, the data-to-control plane attacks, which enable adversaries remotely attack the control plane, are found to have severe attack effects. Most data-to-control plane attacks involve two stages: (i) inject malicious network events into the SDN control channel via protocol interactions; (ii) exploit vulnerable control plane designs/logic with malicious network events. For example, ConGuard [42] generates asynchronous network events in unexpected schedules to exploit vulnerable asynchronous logic of SDN control plane. The "reachability" (i.e., the set of execution paths triggered by the network events) in the control plane decides which logic can be abused. However, the SDN design principles such as control plane and data plane separation greatly limit such reachabilities. As a result, attackers located in the data plane usually target at one or two control plane logic that closely relates to the malicious network events. Increasing such reachability to attack more logic in the control plane may have tremendous promise. Unfortunately, it is difficult to do so due to the following two unique SDN design choices:

- First, the decoupled SDN control plane (i.e., software controller) and the data plane (i.e., network devices) only communicate with each other via pre-defined protocols (e.g., OpenFlow, NetConf). As a result, attackers (usually located in the data plane) can only input data in restricted and pre-defined forms into the control plane.
- Second, only a few components (i.e., message-handling components) in the controller directly handle protocol messages from the data plane. Hence, even though an attacker can inject malicious data into these message-handling components, it is still very difficult to attack other components (the components that run important network services) in the controller unless a very special data dependency (i.e., the data dependency which can directly send malicious protocol messages to the target sensitive method) exists between the target component and message-handling components.

Because of these SDN design choices, attackers often struggle to find a reachable path to those vulnerable logic hidden deeply within the SDN control plane. In this work, we propose a new attack, D^2C^2 (data dependency creation and chaining), which effectively breaks the security guards brought by the two SDN design choices mentioned above. The new attack provides an unexpected, seemingly-unlikely way to exploit sensitive methods/APIs hidden in the control plane. By creating malicious data dependencies, D^2C^2 is able to connect previously unreachable sensitive methods to the data plane in order to increase the reachability. The D^2C^2 attack succeeds due to two findings. First, we found that some widely-used SDN protocol features can help an attacker to violate the security property provided by the first SDN design

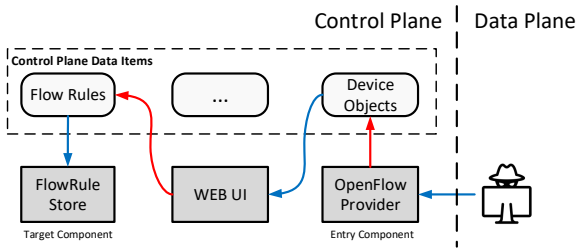


Fig. 1. A real exploit chain of the D^2C^2 attack in ONOS Controller.

choice (control plane and data plane separation). Southbound protocols (e.g., OpenFlow, NetConf) introduce custom fields to enrich the semantics of SDN network states. By abusing this feature, adversaries are now able to input customized malicious data in various forms (e.g., long string or XML) into the control plane through southbound protocol interactions. However, due to the guard brought by the second design choice mentioned above, only the methods in the message-handling components are being exposed to the attackers even though they can abuse the custom fields in the SDN protocols. Hence, we also need to create our own malicious data dependencies in order to increase the reachability. Our second finding is that it is actually possible to create such malicious data dependencies to attack more sensitive methods in the control plane if the attackers firstly exploit a few sensitive methods in those message-handling components by employing a smart strategy (i.e., dependency creation by control plane data poisoning). Such strategies can be illustrated with a real-world exploit chain that we discovered in ONOS [7], one of the most widely used controllers. As shown in Fig. 1, instead of directly introducing a specific attack effect, the attacker first chooses to poison a control plane data item which will be used by another component (WEB UI). With the same strategy, the attacker poisons the data item used by the target component. As a result, when the poisoned data is handled by the target component, the attacker successfully attacks the previously unreachable target method.

The key idea of the D^2C^2 attack is to abuse reachable sensitive methods/APIs with existing data dependencies (e.g., data dependencies created by custom field) to create new data dependencies to abuse previously unreachable sensitive methods/APIs in controllers. To achieve this goal, we designed and implemented a tool, SVHunter, to automatically construct such exploit chains. Constructing a successful exploit chain, however, needs to address the following three challenges:

- First, in order to figure out which data dependencies can be created, how to represent and model the preconditions and postconditions of abusing sensitive methods/APIs?
- Second, how to chain the newly created data dependencies with the existing ones together to construct a D^2C^2 exploit chain?
- Third, how to craft a concrete D^2C^2 attack payload?

The first challenge arises from the fact that it is difficult to decide how much abstraction (generalization) is needed to model the behavior of sensitive methods/APIs. In order to access previously unreachable sensitive methods/APIs, we first need to represent the precondition (the data dependencies needed to abuse the methods/APIs) and postcondition (the data dependencies that can be created after abusing the methods/APIs) of accessing these sensitive methods/APIs. However, it is challenging to find the appropriate model to represent such casualty relationships. If our model is too abstract, the generated representation might be too coarse to describe the correct data dependencies of each method. If we choose a very concrete model to represent the relationships, it is likely that we cannot identify enough information needed by the fine-grained representation with the state-of-the-art program analysis techniques. To address this challenge, we employ a declarative logic language model to represent data dependencies. Unlike an imperative language model that focuses on the details of program state changes, the declarative model provides a flexible representation which expresses the logic of the computation, which is exactly what we are looking for.

The second challenge is about how to analyze and reason the relationships we identified in the previous steps. Since we might identify a large number of sensitive methods/APIs, it is very tedious and error-prone to manually reason and chain their relationships. Hence, we design a reasoning engine to automatically reason the causality relationships of abusing sensitive methods.

The third challenge is daunting because it is complex and time-consuming to craft concrete D^2C^2 attack payloads even for an SDN expert. To craft such a payload, the attacker has to (i) manually emulate the protocol interactions to inject malicious protocol messages, and (ii) fully understand the complex causality relationships within a D^2C^2 chain in order to prepare the proper payload for every sensitive method. To address this challenge, we design an exploit engine to ease the process of synthesizing D^2C^2 exploits.

In summary, the main contributions of this paper are as follows:

- We propose a new D^2C^2 attack against SDN controllers that leverage legitimate protocol interactions to abuse sensitive methods in multiple SDN control plane components. By creating malicious data dependencies, the attack enables a data plane attacker abuse previously inaccessible sensitive methods/APIs in the controller while only controlling a normal network device in the data plane.
- We design SVHunter, a novel tool to pinpoint a wide range of sensitive methods in SDN controllers and create data dependencies to attack these methods. It is a practical tool since it not only leverages program analysis techniques to identify sensitive method usages in the control plane but also semi-automatically constructs exploit chains to introduce various attack effects. We will open source SVHunter at https://github.com/xiaofen9/SVHunter.
- We present a comprehensive evaluation of SVHunter on the mainstream SDN controllers. SVHunter successfully constructed 18 exploit chains to attack previously unknown security risks in the control plane. We have made

TABLE I
CUSTOM FIELDS IN OPENFLOW

Name	Purpose	Length
mfr_desc	Manufacturer description	256
hw_desc	Hardware description	256
serial_num	Serial number	32
dp_desc	Human readable description of datapath	256

responsible disclosure and notified each vendor. By the time of writing, vendors have already patched 9 of them.

II. BACKGROUND

Software-Defined Networking (SDN) is a new network architecture with a decoupled control plane and data plane. Here we introduce some background on SDN: the protocols that bridge the decoupled planes, the software components and internal data in the control plane.

A. Protocols

To bridge the decoupled control plane and data plane, SDN introduces several southbound protocols [27] such as OpenFlow (OF), Open vSwitch Database Management Protocol (OVSDB), and NetConf. Most message fields in these southbound protocols are limited to a finite set of pre-defined values (e.g., 0x00000001~0x000000e0). However, some fields have no constraint and can be customized by the sender. We denote them as custom fields in the subsequent text. Custom fields are used to enrich the semantics of SDN network states. For example, to allow SDN controllers to better support the vendor-dependent features, network devices in the data plane use the custom field of OpenFlow to claim their software/hardware information. Table I demonstrates some default custom fields and their intended purposes in OpenFlow, one of the most important SDN protocols.

B. Control Plane Components

The SDN control plane contains many software components [17], including core services and applications. Core services provide services to other components, while applications use these services to provide functionalities to the network [18]. When a protocol message reaches the control plane, a certain service will parse the message and store it in the controller for further usages [14]. For example, when a new switch connects to the network, a networking event will be sent to the control plane, which contains hardware information about the switch. Then a specific service will update the network states so that other components, such as a topology tracking app, can better understand the network environment with such information.

C. Internal Data

There are mainly two kinds of internal data in the SDN controller, configurational data and runtime data. Important network parameters, such as administrator credentials and the access control list, are stored as configurational data. These data items are usually set by the administrators and stored persistently in the file system or relational databases in the controller. They are critical since the control plane components rely on them to make important decisions such as routing. Runtime data stores the network status information such as network topology, device state, and traffic statistics. The information is mainly collected from the data plane and is usually stored as runtime data structures of the controller.

III. THREAT MODEL

A. Scenario

As the brain of the entire network, it is clear that compromising the controller is much more effective to attackers [42] than just to compromise a single or partial network device(s).[1] In this paper, we do *not* assume that attackers can have direct access to the SDN controller or SDN applications, which can be well protected. In addition, the control channels between the controller and switches as well as the administrative channels between administrators and SDN UI migtht be protected by secure cryptographic protocols like SSL/TLS. In this paper, we consider scenarios where a vulnerable network device (e.g., a switch or a host) exists in the network, and an attacker in the data plane wants to launch D^2C^2 attacks by abusing the device to interact with the controller using legitimate protocol interactions.

We believe that this attack scenario is realistic. First, network switches (including software switches) can be compromised. Actually, many serious vulnerabilities have been found in SDN-enabled switches [3], [2], [36], [26]. Second, in many cases, attackers might not need to compromise network switches; instead, they just compromise normal hosts or virtual machines to launch attacks. When SDN networks are configured as in-band control [12], the control plane and data plane share the same physical links.[2] As a result, normal hosts could interact with the controller as long as the control channel does not enforce SSL/TLS[3].

B. Attack Effects

A data plane attacker who wants to impact the network security will seek to introduce one or several attack effects to the controller. In Table II, we summarize three categories of common attack effects against the controller. (a) Denial of Service. The attack effects in this category can disrupt the availability of a specific network service in the controller. (b) Data leakage. Such attack effects aim at stealing valuable information from the controller (e.g., the network topology). (c) Network manipulation. These attack effects can manipulate arbitrary network elements or alter the controller's knowledge about the network.

We claim that the D^2C^2 attack is able to introduce multiple attack effects which can cover almost every previously

[1] There are existing studies that target the data plane to launch attacks [36], [26], [13]; however, such attacks are not the focus of this paper.

[2] Many real-world networks take this in-band operation approach due to its cost-efficiency [12], [42].

[3] Note that the control channel does not enforce SSL/TLS by default [9], [8], and it can be very complex to manage in real-world deployments especially when there are multiple controllers [32].

TABLE II
COMMON SDN ATTACK EFFECTS AGAINST CONTROLLERS FOR DATA PLANE ATTACKERS

Category	Attack Effect	Examples
Denial-of-Service	Congest SDN architectural bottleneck	A remote attacker generates a large number of events to congest the bottleneck of controller [39].
	Disrupt network services	A remote attacker exploits SDN service logic flaws (e.g., harmful race conditions) [42] to crash core SDN services.
	Install false flow rules	N/A
	Corrupt critical configuration	N/A
Data Leakage	Probe sensitive network information	A remote attacker steals sensitive information (e.g., physical IP) from the controller by violating network policy [42]
	Steal important network configuration	N/A
Network Manipulation	Manipulate network view	A remote attacker alters the controller's knowledge about the network by sending fake LLDP packets [19]
	Install arbitrary flow rules	N/A

achieved attack effect in Table II. In addition, our attack can also introduce previously unachieved attack effects, such as install false/arbitrary flow rules and corrupt/steal critical configuration, listed as "N/A" in Table II.

IV. D^2C^2 ATTACK

In this section, we introduce D^2C^2, a novel attack that employs a data poisoning strategy to abuse previously unreachable sensitive methods in the controller.

The D^2C^2 attack leverages two important insights to abuse sensitive methods/APIs in the controller. The first insight is that some SDN design features, e.g., custom field, can be abused to help attackers send malicious data into the control plane. By abusing custom fields in SDN protocol (e.g., OpenFlow) messages, the attacker is able to manipulate the sensitive methods/APIs in the message-handling components.

Another important new insight of D^2C^2 is that the data dependencies among the many seemingly-separated SDN services/applications can actually be created in unexpected ways (i.e., data item poisoning) to result in the exposure of previously unreachable sensitive methods/APIs to attackers. To take advantage of this, the attacker should realize that his intended attack effect is supposed to result from the execution of a partial order of sensitive methods which belong to multiple services or applications. Hence, to achieve such attack effects in SDN networks, the attacker needs to perform a multi-stage attack which directly or indirectly poisons several control plane data items to finally attack the target component.

A. Example Attack Scenario

We illustrate such an attack by analyzing a real exploit chain of the D^2C^2 attack we mentioned in Section I. The attack effect of this exploit chain is to manipulate arbitrary flow tables in the data plane. To introduce this effect, the exploit chain attacks the core service component FlowRuleManager, which manages flow tables in data plane devices (e.g., network switches). As shown in Fig. IV-A, the exploit chain starts from the data plane, where a poisoned protocol message (e.g., custom field) is sent into the control plane via legitimate protocol interactions. In Phase 1, the core service OpenFlow Provider parses the poisoned protocol message and generates a poisoned network state. In Phase 2, the sensitive method

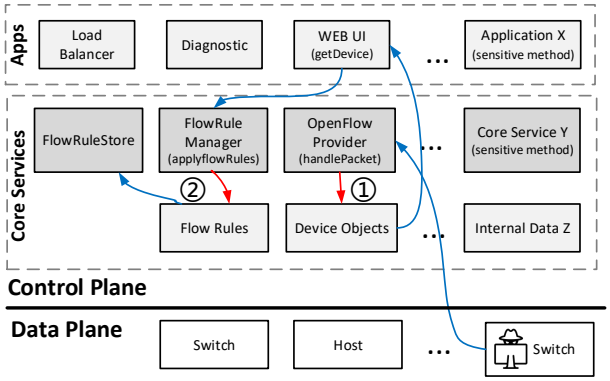


Fig. 2. A real-world exploit chain of the D^2C^2 attack. The exploit chain poisons several control plane data items to create malicious data dependencies (red edges) via existing data dependencies (blue edges).

`getDevice` in the Web UI application is abused by the poisoned network state. By abusing method `getDevice`, the poisoned network state is able to manipulate the Web UI application to launch HTTP requests to access the northbound APIs (i.e., Restful APIs) belonging to the FlowRuleManager. Hence, a malicious data dependency is established from the UI application to the sensitive method `applyflowRules` in the FlowRuleManager. By abusing the newly reachable sensitive method, a malicious flow rule is inserted. In the end, the core service FlowRuleStore handles poisoned flow rules and updates the flow tables in correspond data plane devices. This is a typical exploit chain of the D^2C^2 attack. By employing such data poisoning strategy, the D^2C^2 exploit chain is able to send malicious data to the target sensitive method via existing and newly created data dependencies.

B. Problem Formulation

In this paper, we focus on four categories of internal data items (i.e., protocol messages, network states, databases, configure files) according to our discovered attack surface. Part of these data is stored as a runtime variable in the controller while others are stored persistently in the file system (e.g., databases and configuration files). The D^2C^2 attack poisons these different kinds of internal data by abusing different kinds of sensitive methods/APIs that modify these data items. For example, in one of the cases we found, the File.read() method

is vulnerable and can be abused to poison configuration files. To investigate how control plane components will be affected if one data item is poisoned, we study a list of methods that collect data from the four categories of data items. Most of the methods are critical SDN APIs (e.g., southbound APIs and northbound APIs) and some of them are dangerous Java lib methods (e.g., File.read()). In this paper, we call these methods as data collecting methods. We observe the data collecting methods and sensitive methods have intersections since some data collecting methods are also sensitive methods that can be abused to leak data.

To launch D^2C^2 attacks, we need an exploit chain that consists of several events in which several control plane data items are handled by different sensitive methods. In this paper, we denote such an event as a *poisoning event* since its consequence is to poison the data item in the control plane. However, not all such chains can be regarded as harmful, since only part of them can create exploitable data dependencies between the attacker and the target sensitive method in the SDN control plane. In this paper, our goal is to identify such D^2C^2 chains from numerous seemly exploitable chains.

D^2C^2 chains. There are several requirements that should be met before a chain can be exploitable. (1) Every two neighbor events in such D^2C^2 chain should have the following relationships: (1a) The prior event must be able to create at least one data dependency to the later one. (1b) The prior one does not have to call the later one, which means they can be called separately. (2) The first event and the last event should have additional requirements: (2a) The attacker should be able to trigger the first event from the data plane (e.g., through the custom field). (2b) The last event in the D^2C^2 chain should be abused on a destructive sensitive method which can fulfill the attacker's objective.

The attacker faces a knotty problem when he wants to launch such an attack. That is how to find all D^2C^2 chains from the SDN controller. To address this problem, we introduce our tool, SVHunter, which pinpoints poisoning events in the SDN control plane and generates D^2C^2 chains through backward taint analysis and logic reasoning.

V. Tool Design and Implementation

In this section, we present our tool, SVHunter, for identifying and exploiting the D^2C^2 vulnerabilities in SDN controllers. As shown in Fig. 3, SVHunter comprises three main components: the Tracer pinpoints poisoning events in the controller by utilizing backward data flow tracking. The Reasoning Engine reasons the causality relationships between the identified poisoning events to create and chain data dependencies in order to generate D^2C^2 chains. The Exploit Engine eases the process of synthesizing D^2C^2 exploits.

A. Pinpointing Poisoning Events

In this section, we describe the design of the Tracer, which pinpoints poisoning events in the controller as well as its applications at the Java bytecode level. The Tracer first identifies the usages of sensitive methods in the controller according

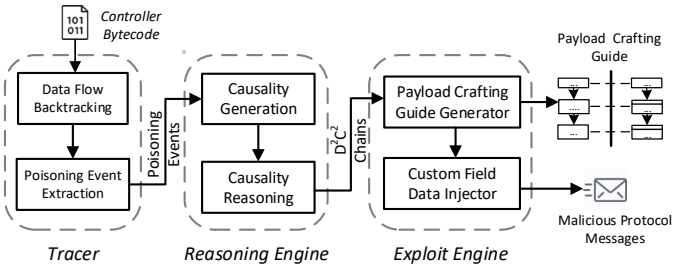


Fig. 3. SVHunter Overview.

to a particular list of method names. Second, it employs static analysis techniques to backward trace data flows from each parameter of each identified sensitive method to its data sources (i.e., data collecting methods). Finally, the Tracer associates sensitive method usages with their corresponding data collecting methods as well as certain context information.

Detecting sensitive method usages. The Tracer detects the usages of sensitive methods with a particular list of method names. We follow two principles to choose sensitive methods. First, according to the essence of the D^2C^2 attack, i.e., data poisoning, we choose from the methods that perform read/write operations on the four categories of control plane internal data mentioned in Section II-C. Second, to introduce destructive attack effects with D^2C^2, we also choose some widely targeted methods (similar to the process of choosing sinks in taint analysis research [41], [33]) that can be abused to introduce destructive effects from both Java library methods (e.g., exec()) and SDN APIs (e.g., firewall switches). Similar to other vulnerability discovery research [28], [11], we made our best effort to collect as many sensitive APIs as possible.[4] Using the list of method names, the Tracer locates the usages of sensitive methods in the controller through keyword matching and marks them as the data sinks of backward tracing.

Backtracking. In this step, the Tracer reversely traces the data flows from each parameter of each located sensitive method. The Tracer is implemented on top of Soot [22]. To backward trace the potentially harmful data flows, the Tracer marks the data collecting methods as data sources and marks the parameters of each located sensitive method as tainted data (data sinks).

To improve tracing efficiency, we optimize the tracing design as follows. (i) Before tracing, the Tracer will first construct a mapping table which records the caller-callee relationships. (ii) During tracing, all the being-traced paths will be saved temporarily so that they can be reused if another sensitive method usage is traced to any of the saved paths.

Identifying poisoning events. After backtracking, the Tracer identifies poisoning events by identifying all the data flows that start from data collecting methods and end at those sensitive methods. Note that if a data flow starting from one data collecting method contains more than one sensitive methods, multiple poisoning events will be identified separately so that

[4] A full list can be found on our project website (https://github.com/xiaofen9/SVHunter). We acknowledge the list might not be complete but can be expanded over time.

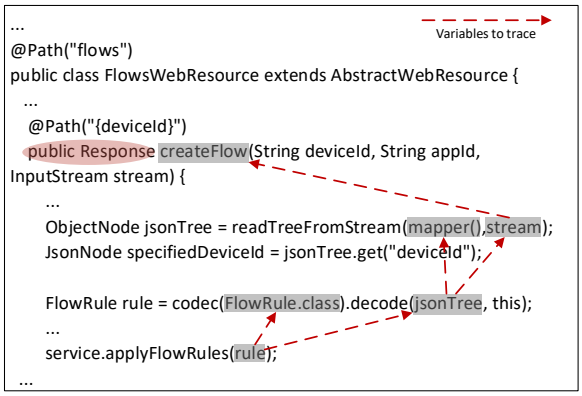


Fig. 4. Illustration of SVHunter's Backtracking (using a real-world example of sensitive method usage).

each poisoning event involves a single sensitive method. For the Tracer to know if a method is a data collecting method or not, it checks the method in the list of data collecting methods.

To illustrate, here we use the same example used in Section IV-A. As shown in Fig. 4, the trace for Phase 2 (poisoning event ②) starts from the sensitive method `applyFlowRules`. To trace its data flows, the Tracer identifies its parameters as the data sinks. In this example, the data sink is variable `rule`. Next, the Tracer traces variable `rule` using the aforementioned approach. Finally, the Tracer traces to method `createFlow` whose type information (red circle) matches one of the data collecting methods, which indicates that the data flow from `createFlow` to `applyFlowRules` should be identified as a poisoning event.

B. Reasoning Event-Triggering Causality

Before we can reason the causality relationships between the poisoning events identified by the Tracer to figure out how they can be chained together, we first need to represent the poisoning events in a unified model. Hence, the Reasoning Engine first generates a unified representation for each poisoning event. This representation describes the causality (i.e., preconditions and postconditions) of each poisoning event. Second, the Reasoning Engine reasons the generated causality representations to decide whether and how two or more poisoning events can be chained together to create malicious (transitive) data dependencies between the data plane and the target method(s) in the control plane.

Modeling poisoning events. To represent different poisoning events in a unified form, we introduce a simple but effective declarative language, Event Reasoning Language (ERL). We implemented the ERL compiler component on top of a popular framework, ANTLR [1]. The basic idea of ERL is to model both the preconditions (i.e., how a poisoning event obtains data items) and the postconditions (i.e., how a sensitive method involved in a poisoning event is abused to poison other data items) of each poisoning event. According to the four aforementioned categories of data items, ERL respectively uses *fs, net_state, proto_msg, db* to denote the data items stored in configuration files, network states, protocol messages from the data plane, and databases. To further describe each specific data item, ERL employs a namespace concept that is very similar to the namespaces in modern languages like C++. A path identifier (*pid*) is assigned to every data item to differentiate data items in the same category. Table III shows examples of supported *pid* formats in ERL. To describe different kinds of operations on data items, ERL employs two commonly used primitives (i.e., read and write) and one SDN-specific primitive (i.e., crash). *read* and *write* mean that the corresponding poisoning event can read and write a data item, respectively. `crash` means that the corresponding event is able to corrupt the format or integrity of a data item. List 1 demonstrates the main ERL grammar. As indicated by the ERL grammar, all poisoning events are represented by a notion called *observations*. Every observation is in the form of *data* → *operation*, which means that a particular operation has been performed on a particular data item. The observations involved in a poisoning event can be divided into two sub-classes, which are used to describe the preconditions and postconditions of the poisoning event, respectively.

TABLE III
EXAMPLES OF PID AND THEIR MEANINGS

Data	Refers to
fs::bin.diagnostics	configure file *diagnostics* under the file dictionary of bin
proto_msg::portstatus.port	protocol field *port* in the PortStatus message

```
observation ::= data → operation
              |  IF observation THEN observation
              |  observation && observation
              |  observation || observation

operation   ::= READ var
              |  WRITE var
              |  CRASH var

data        ::= fs :: pid
              |  net_state :: pid
              |  proto_msg :: pid
              |  db :: pid

pid         ::= letter
              |  !pid
              |  pid.letter
              |  .
```

Listing 1. ERL grammar

We also introduce a language extension to ERL, which is a stage description model that guides the chaining procedure. The model enables users to add customized rules according to their own cases. More specifically, the model enables users to explicitly declare the first event and last events of a D^2C^2 chain with two special labels, respectively. (1) By adding the *@toe-hold* label to a poisoning event, the attacker declares that he can directly manipulate the data item(s) in the event so that the event can be placed at the beginning of the D^2C^2 chain. This label is useful when the attacker controls other internal data items than data plane messages in some cases. For example, compromising an FTP service may lead to the manipulation of some configuration files in the controller). (2)

By adding the *@final* label to an event, the attacker declares that a sensitive method involved in the event can achieve his final attack objective. For example, if he wants to execute arbitrary commands in the controller, an event involving such sensitive methods as `Runtime.exec()` should be labeled as *@final*, since it should be the final event in any D^2C^2 chain that includes the event. The language extension can avoid the generation of meaningless D^2C^2 chains which let a final event be further connected to potentially many other events.

Generating observations. With the unified representation, the Reasoning Engine takes poisoning events as input and generates the corresponding representations using ERL. Each observation consists of two sub-sentences. The first sub-sentence describes what data items are already poisoned when the corresponding poisoning event happens. The second sub-sentence describes what data items can be further poisoned once the sensitive method involved in the event is abused.

The Reasoning Engine employs a heuristic method to generate the two sub-sentences. For the first sub-sentence, the Reasoning Engine extracts the concrete *data* description from the running context of the identified data collecting method in the event. For the second sub-sentence, since it is deterministic in terms of which data items can be abused by each sensitive method, the Reasoning Engine is able to directly decide which data items can be accessed for every sensitive method in a heuristic manner. It is worth noting that in some cases the pid for the second sub-sentence could be ".", which means the sensitive method can affect all data items in that category (e.g., the abusing of `File.write()` can lead to an arbitrary file write).

Using the same example exploit chain used in Section IV, the poisoning event shown in Fig. 4 collects a data item in the category of *net_state*. In this case, the Reasoning Engine extracts information from `@Path` at both class level and method level, which corresponds to the actual URL when accessing the data item via Restful API. As a result, the Reasoning Engine decides that the *data* should be *net_state*::*flows.deviceid*. Finally, the first sub-sentence for the event should be *net_state*::*flows.deviceid* → *read*.

Observation reasoning. The Reasoning Engine takes observations as input to generate the event chain graph. It first generates nodes for every observation and then tries to connect them to generate the graph according to the causality relationships between the poisoning events.

Algorithm 1 outlines our process for constructing the event chain graph with given observations. It takes as input all the observations as a set S and produces the corresponding graph denoted by (N, E), where every node $n_i \in N$ corresponds to an observation $s \in S$, and directed edge $e_{i,j} \in E$ denotes that node j can be triggered with the postconditions in node i. The algorithm can be divided into two phases. In the first phase, it generates a node for each observation. Every node can be denoted as (P, C), where set P denotes the set of preconditions for this observation and set C denotes the set of postconditions.

In the second phase, the algorithm reasons the causality relationships between every two nodes. The Reasoning Engine will add an edge $e_{i,j}$ between node i, denoted as (P_i, C_i), and node j, denoted as (P_j, C_j), if node i can meet the preconditions of node j. Since $e_{i,j}$ means that node i can poison the data items used by node j, the Reasoning Engine will examine both the *operation* and *data* of P_j and C_i. First, *data* in P_j should be a subset of *data* in C_i. Second, the operation of C_i should be *write* or *crash*, which can affect the data items in node j. It is worth noting that the Reasoning Engine also follows two rules introduced by the language extension. First, it will stop expanding a path once it connects a node labeled as a *final* event. Second, the engine will always start reasoning from the nodes with label *toe-hole*, which can be directly triggered by the attacker.

Algorithm 1 Event chain graph Generation

Require:
S = a set of observations;
Ensure:
$ECG = (N, E)$ where N is a set of nodes and E is a set of edges.
$N \leftarrow \{\}, E \leftarrow \{\}$
for all observation $s \in S$ **do**
  $P \leftarrow$ *get_preconditions*(s)
  $C \leftarrow$ *get_postconditions*(s)
  $N \leftarrow N \cup \{(P, C)\}$
end for
for all Node $(P_i, C_i) \in N$ **do**
  for all Node $(P_j, C_j) \in N$ and $(P_i, C_i) \neq (P_j, C_j)$ **do**
    if !*isFinalEvent*((P_i, C_i)) **then**
      for all $p_m \in P_j$ and $c_n \in C_i$ **do**
        if *satisfy*(c_n, p_m) **then**
          $E \leftarrow E \cup \{(e_{i,j})\}$
          break
        end if
      end for
    end if
  end for
end for

Node a
if proto_msg :: OFFeatureReply.mfrDesc → read var#53
then net_state :: root → write var#53

Node b
if net_state :: root.flows.deviceid → read var#24
then net_state :: flowRules → write var#24

The two observations shown above correspond to two observations in the aforementioned example exploit chain. The Reasoning Engine will first generate two nodes (i.e., node a denoted as (P_a, C_a) and node b denoted as (P_b, C_b)), respectively. Then the Reasoning Engine uses Algorithm 1 to reason the causality relationships between the two nodes. It is not difficult to see that C_a and P_b perform operations on the same data item in the category of *net_state*. Also, P_b reads the data item after it is written by C_a. Hence, the Reasoning Engine adds an edge $e_{a,b}$ between node a and node b. In some cases, there might be several paths in the event chain graph, but SVHunter will only highlight the paths with *toe-hold* nodes, whose data can be directly controlled by the attacker.

C. Generating D^2C^2 Payloads

Even with the help of event chain graphs, verifying D^2C^2 vulnerabilities can still be challenging and time-consuming.

Hence, SVHunter provides an Exploit Engine in order to make D^2C^2 exploitation more automated.

Custom Field Data Injector. The data injector is part of the Exploit Engine. Leveraging SVHunter's payload-crafting guide generation capability, which will be presented shortly in this section, users can gain concrete understanding about which (kinds of) data items should be included in the attack payload. However, knowing the content of the attack payload does not mean that users also know *how* to use the content to generate a poisoning event. Without orchestrating a specific set of malicious protocol interactions with the target controller, no poisoning event can be successfully generated. In order to gain this orchestrating capability, users should be familiar with SDN protocol specifications, in order to locate the custom field, and implementation of specific SDN protocols (e.g., Open vSwitch), in order to inject the content of a payload into a custom field.

To answer this "how" question and help users overcome the orchestrating difficulties, we built the Custom Field Data Injector, which can automatically generate specific toe-hold poisoning events through protocol message manipulation (Note that the type of message is decided by the *data* field of the first sub-sentence in the corresponding poisoning event). In particular, the Data Injector does two things: (i) it dynamically injects payloads into target custom fields; (ii) it automatically orchestrates the needed protocol interactions with the SDN controller and triggers the controller to process the target custom fields.

For the first task, the Data Injector hooks the protocol handling functions for each protocol implementation (e.g., Open vSwitch) so that the Data Injector can dynamically modify the desired custom fields when these protocol implementations generate the corresponding protocol messages.

For the second task, the Data Injector first simulates a legitimate network device in order to let the controller handle its messages. Then, it generates particular network events in order to trigger different protocol interactions between the Data Injector and the controller. For example, the Data Injector will connect a new switch to the control plane in order to generate the "switch join" protocol messages.

Currently, the Custom Field Data Injector supports two widely-used protocols in SDN (i.e., OpenFlow and NetConf). The set of custom fields supported by our Exploit Engine in each protocol can be found in Table VIII in the Appendix.

Payload Crafting Guide Generation. To exploit a D^2C^2 chain, the attacker needs to input attack payloads into the Custom Field Data Injector to satisfy the constraints associated with the new attack path (i.e., exploit chain) which is created through malicious data dependency creating and chaining. However, for SVHunter, it is difficult to automatically satisfy all the constraints and generate the final exploit. This is because solving the constraints associated with data dependency creating requires not only general purpose path constraint solvers [15], [43] but also an expert system which incorporates both SDN domain knowledge and hacking skills, which is quite beyond the scope of this paper.

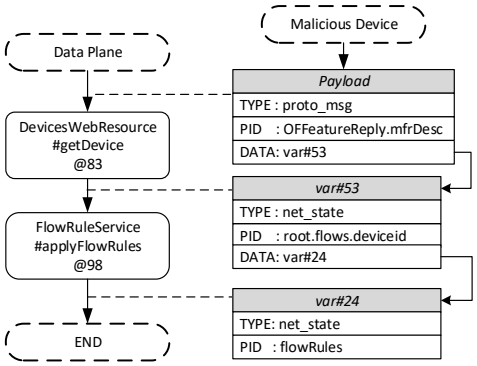


Fig. 5. An example payload crafting guide generated by SVHunter. *Since sensitive method `getDevice` has no caller (It is implicitly called by controller framework via reflection), SVHunter directly shows the method itself when representing its usage.

As a result, SVHunter chooses to address this difficulty in a semi-automatic manner. To help users craft D^2C^2 payloads, we built the Payload Crafting Guide generator. This generator will construct a payload crafting guide for every generated event chain graph. With the guide, users will be able to quickly locate the vulnerable code snippets and understand how data items are processed along the whole exploit chain. As shown in Fig. 5, the guide will indicate to users two types of critical information. (1) The boxes on the left of the figure display the code locations where sensitive methods are being abused (in the format of Class#SensitiveMethod@Line). (2) The boxes on the right provide a detailed description of every poisoned data item in each step (Note that the description is extracted from the *data* part of the corresponding observation).

We again use the aforementioned example exploit chain to further illustrate the generated payload crafting guide. As shown in Fig. 5, node *a* is converted to the box which is connected to the "Data Plane" node. This box indicates that `getDevice` is the first sensitive method to be abused, and it is located in class `DevicesWebResource` at line 83. Similarly, node *b* is converted to the box which is connected to the "END" node. The box records the code location of the second sensitive method (i.e., `applyFlowRules`) in the exploit chain. Moreover, the boxes on the right side of the figure illustrate how poisoned data propagates. For example, a malicious payload in the form of *proto_msg* is sent from the "malicious device". By abusing the sensitive methods in nodes *a* and *b*, the payload further propagates to `var#53` and `var#24`, which are respectively stored in the *root.flows.deviceid* date item and the *flowRules* data item. Note that both data items are in the category of *net_state*.

VI. Evaluation

In this section, we present our evaluation results of SVHunter. Our evaluation is focused on using SVHunter to construct exploit chains against three open-source SDN controllers (i.e., ONOS, Floodlight, and Opendaylight). The three controllers are currently the most widely used controllers in both academia and industry. In addition, to understand

whether security-enhanced SDN controllers are immune to the D^2C^2 attack or not, we also use SVHunter to exploit SE-Floodlight, which is a security-enhanced variant of Floodlight. SVHunter consists of more than 11K LoC (Java+Python) in total. More specifically, the Tracer has 3,449 LOC, the Reasoning Engine has 3,847 LOC, and the Exploit Engine has 3,854 LOC. We are releasing SVHunter as an open-source tool at https://github.com/xiaofen9/SVHunter in the hope that it will be useful for future SDN security research. We run SVHunter on a machine running Ubuntu 16.04 LTS with a dual-core 2.4 GHz CPU and 16 GB memory.

Conceptually speaking, we seek to answer the following evaluation questions:

- Is "Data Dependency Creation and Chaining" a pre-existing vulnerability in widely-used mainstream SDN controllers? If so, how serious is the vulnerability?
- Can SVHunter effectively exploit the "Data Dependency Creation and Chaining" vulnerability and construct previously-unknown exploit chains in different kinds of SDN controllers?
- In which ways do the discovered vulnerabilities and explain chains enlarge the attack surface of SDN?

A. Identifying Poisoning Events

Table IV presents our results of poisoning-event pinpointing from ONOS v1.13.1, Floodlight v1.2, OpenDaylight v0.4.1, and SE-Floodlight, respectively. SVHunter detected 74 poisoning events among 8,664 tracked information flows in ONOS using 98 seconds. 38 poisoning events and 19 poisoning events were identified in Floodlight and SE-Floodlight respectively. 17 poisoning events among 5,150 tracked information flows were identified in OpenDaylight using 46 seconds. Compared with Floodlight and Opendaylight, we analyzed more applications (i.e. 164) in ONOS. This is because the ONOS project maintains both the controller and a large number of applications while the other two projects mainly maintain the corresponding controllers. Hence, we analyzed all the official applications in ONOS while in Floodlight and OpenDaylight we only analyzed a handful of applications which are necessary for SDN networks to function. For the same reason, SVHunter identified more poisoning events in ONOS than in Floodlight and OpenDaylight.

Regarding the "quality" of the detected poisoning events, on one hand, we note that it is very difficult if not impossible to find concrete and justifiable per-event evaluation criteria. On the other hand, we note that the quality of the detected events can actually be indirectly evaluated through the quality of the D^2C^2 chains constructed by the Reasoning Engine, which we will evaluate in the following subsections.

B. Reasoning Results

The effectiveness of SVHunter is affected by two main factors. (a) Since the Reasoning Engine uses heuristics, the generated observations are imperfect in some cases. (b) The Tracer might fail to recognize the semantics of some usages

TABLE IV
DETECTION RESULTS OF POISONING EVENTS.

Controller	Version	LoC	Time	Event Detection Results			
				#NTC	*#TIF*	*#SMU*	*#PE*
ONOS	1.13.1	673985	98s	164	8664	536	74
Floodlight	1.2	60090	19s	31	992	163	38
OpenDaylight	0.4.1	326479	46s	5	5150	406	17
SE-Floodlight	Beta7	N/A	21s	N/A	1256	108	19

#: the number of
TIF: traced data flows; SMU: sensitive method usages;
PE: poisoned events; NTC: tested components.

of sensitive methods (e.g., some poisoning events could have an unknown calling precondition).

Due to the two factors, it is not guaranteed that every D^2C^2 chain generated by the Reasoning Engine is exploitable. Therefore, this part of the evaluation will not only focus on whether SVHunter can generate previously unknown exploitable D^2C^2 chains, but also on how likely SVHunter falsely generates *unexploitable* D^2C^2 chains. We denote such unexploitable D^2C^2 chains as a false positive. In order to distinguish exploitable D^2C^2 chains from unexploitable ones, we first built an SDN testbed with Mininet 2.3 [6], which is an SDN network emulator. Then, we ran the four controllers on the testbed and tried to run each of the generated D^2C^2 chains via modifying the payloads generated by the Exploit Engine. Note that all the toe-hold poisoning events involved can be automatically generated by the Exploit Engine.

Table V summarizes our reasoning results. The first column records the number of D^2C^2 chains generated by SVHunter (i.e. Constructed Chains). The second column records the number of chains that are found to be exploitable (i.e. Exploitable Chains) for each controller. We manually examined all the exploitable chains generated by SVHunter and found that some exploitable chains actually exploit the same sensitive method(s) in the same component (e.g., in Floodlight four D^2C^2 chains exploit the same sensitive API), although they use different parameter values when calling the sensitive method(s). It is clear that the adversary will pay attention to this special kind of "distinct but redundant" redundancy. Based on this finding, the third column records the number of non-redundant chains, i.e., the number of exploitable chains after this special kind of redundancy is removed.

The results show that SVHunter identified 58 D^2C^2 chains in ONOS, and 48 of them are exploitable. In Floodlight, we found 13 out of 19 identified D^2C^2 chains can be exploited. In OpenDaylight, 2 D^2C^2 chains are identified by SVHunter and both can be exploited. In SE-Floodlight, SVHunter identified 5 D^2C^2 chains and 4 of them can be exploited. Since SE-Floodlight is not open source, we directly used the heuristics obtained from Floodlight to test SE-Floodlight (Note that SE-Floodlight is adapted from Floodlight). After comparison, we found that the result of SE-Floodlight and the result of Floodlight are almost the same: only one non-redundant chain identified from Floodlight is not detected in SE-Floodlight. The missing non-redundant chain results from the fact that SE-Floodlight is based on an older version of Floodlight (v0.87)

TABLE V
REASONING RESULTS FOR THE FOUR CONTROLLERS.

Controller	Constructed Chains	Exploitable Chains	Non-redundant Chains
ONOS	58	48	11
Floodlight	19	13	3
OpenDaylight	2	2	2
SE-Floodlight	5	4	2

and the sensitive method abused by the missing chain doesn't appear in the old version.

In fact, it is not a surprise that SE-Floodlight is vulnerable to the D^2C^2 attack: existing security-enhanced controllers only focus on SDN application resilience and permission management rather than the abuse of sensitive methods in the control plane. Moreover, according to the fact that the detection results from SE-Floodlight and Floodlight are only slightly different from each other, we can also posit that the heuristics obtained from the three mainstream controllers are also applicable to their variants. Since it is a common practice for industry and academia to develop their own controllers [5], [4] by adapting one of the three mainstream controllers (i.e., Floodlight, ONOS, and Opendaylight), SVHunter and the heuristics used by SVHunter should be very useful for many if not most vendors and researchers to test their own controllers before releasing. We also observed that SVHunter identified more exploit chains in ONOS than in the other three controllers. We posit that this is due to two reasons: (i) As we have discussed in Section VI-A, SVHunter identified more poisoning events in ONOS than in others. Because of this, a larger number of chains are constructed from a larger number of candidate events. (ii) In ONOS, the identified (A3) event(s) is able to poison arbitrary data items in the category of *net_state*, which leads to greater reachability than other events.

Table VI demonstrates the 18 non-redundant D^2C^2 chains identified by SVHunter (Note that all the involved poisoning events are listed in Table IX in Appendix A). We noticed that the D^2C^2 attack substantially increases the reachability mentioned in Section 1: With the data dependencies created by the 21 poisoning events, 12 previously unreachable sensitive methods from 13 control plane components were successfully abused. We also noticed there are five identified chains only consisting of one poisoning event. However, we posit that they are still important. Since D^2C^2 chains aim at increasing the reachability in the control plane, the significance of a chain should be reflected in how previously unreachable methods/components are reached instead of its length. Although each of the five chains only consists of a single event, every single event in the five chains successfully manipulates the parameters of critical sensitive methods. According to the results above, despite the fact that we cannot guarantee that the heuristics (e.g., sensitive methods and data collecting methods) used by SVHunter are complete and exhaustive, we posit that these heuristics are essential and effective in D^2C^2 vulnerability analysis. Otherwise, it is unlikely that SVHunter successfully found 18 previously-unknown D^2C^2 chains.

Although the results are very encouraging, we also found that in some special cases SVHunter doesn't construct 100% correct D^2C^2 chain that could be exploited. The false positives of SVHunter mainly result from certain semantic checks ignored by SVHunter. For example, when we verified one of D^2C^2 chains identified from ONOS, we found that one of the poisoning events located in an IP address converting component cannot be triggered. This is because the parameter `srcIp` of the sensitive method `parseInt` to be abused cannot be manipulated since its format is strictly checked. However, SVHunter still "concludes" that its preceding events are sufficient for poisoning this parameter.

C. Impact Analysis of Identified D^2C^2 Chains

As shown in Table VI, we identified 18 non-redundant D^2C^2 chains. To fix these vulnerabilities, we have made responsible disclosure and notified the vendors of each vulnerable controller. They reacted immediately and so far 9 of them have been fixed and assigned a CVE number. In this section, we conduct an impact analysis on these D^2C^2 chains according to the three types of attack effects listed in Table II.

Network Manipulation. We found that 6 D^2C^2 chains (i.e., DC-1, DC-2, DC-3, DC-4, DC-12 and DC-17) can generate serious network manipulation effects (e.g., arbitrary command execution or installation of arbitrary flow rules). In ONOS, we identified 4 chains. First, DC-1, DC-2, and DC-3 abuse the sensitive method `Runtime.getRuntime().exec()` in different components. The common attack effect of the three chains is to have the controller execute arbitrary system commands. Second, DC-4 abuses a sensitive method in northbound APIs (i.e., `applyFlowRules()`) to install specific malicious flow rules. In Floodlight, we found that one similar D^2C^2 chain (i.e., DC-12) can install malicious flow rules by abusing sensitive methods in northbound APIs. In SE-Floodlight, we also identified DC-17 and DC-18, which is respectively identical to DC-12 and DC-13 in Floodlight.

Data Leakage. We found that 12 D^2C^2 chains (i.e., DC-1, DC-2, DC-3, DC-5, DC-6, DC-7, DC-8, DC-12, DC-13, DC-15, DC-17 and DC-18) can leak sensitive information (e.g., network topology, flow rules, and network traffic) from the control plane. In ONOS, we found 7 D^2C^2 chains. Since DC-1, DC-2 and DC-3 are able to execute commands in the controller, they are able to read network states through command execution. In addition, we found that DC-5 and DC-6 can access *net_state* data items by abusing northbound APIs. Finally, we found that DC-7 and DC-8 can let information in configuration files leak out. Their last events are in the NETCONF application and the Driver service, respectively. They abuse XML parser methods such as `Javax.xml.parsers.DocumentBuilder.parse()`. In Floodlight, DC-12 and DC-13 were found to be able to abuse the northbound APIs to access *net_state* so as to acquire sensitive network information such as network topologies and flow rules. Similarly, we found that in SE-Floodlight DC-17 and DC-18 are generating the same kind of attack effect. In OpenDaylight, we found that DC-15 can be leveraged by the

TABLE VI
D^2C^2 CHAINS (NON-REDUNDANT) CONSTRUCTED BY SVHUNTER.

Controller	Chain#	Target Component	Event Chain	Description	Attack Effects			Disclosure
					#1	#2	#3	
ONOS	1	Diagnostic	(A3) → (A5) → (A7)	Execute arbitrary commands	✔	✔	✔	Fixed (CVE-2017-1000078, CVE-2018-1999020)
	2	Diagnostic	(A3) → (A6) → (A7)	Execute arbitrary commands	✔	✔	✔	Fixed (CVE-2017-1000078, CVE-2018-1999020)
	3	YangLiveCompiler	(A3) → (A9)	Execute arbitrary commands	✔	✔	✔	Fixed (CVE-2019-13624)
	4	FlowRuleManager	(A3) → (A10)	Modify specific network states	✔			Reported
	5	WEB UI	(A1) → (A8) ((A10))	Read arbitrary network states		✔		Fixed (CVE-2018-1000614)
	6	WEB UI	(A2) → (A8) ((A10))	Read arbitrary network states		✔		Fixed (CVE-2018-1000616)
	7	NETCONF	(A1)	Read arbitrary configuration files		✔		Fixed (CVE-2018-1000614)
	8	XMLPARSER	(A2)	Read arbitrary configuration files		✔		Fixed (CVE-2018-1000616)
	9	OVSDB	(A4)	Disrupt specific network service			✔	Fixed (CVE-2018-1000615)
	10	YangLiveCompiler	(A3) → (A5)	Corrupt arbitrary configuration files			✔	Fixed (CVE-2018-1999020)
	11	Core	(A3) → (A6)	Corrupt arbitrary configuration files			✔	Fixed (CVE-2018-1999020)
Floodlight	12	StaticEntryPusher	(B1) → (B3)	Write specific network states	✔	✔		Fixed (CVE-2018-1000163)
	13	Web GUI	(B1) → (B4)	Read specific network states		✔		Fixed(CVE-2018-1000163)
	14	Forwarding	(B2)	Disrupt specific network service			✔	Fixed (CVE-2018-1000617)
OpenDaylight	15	ODL-SDNi	(C1) → (C3)	Read specific network states		✔		Fixed (CVE-2018-1132)
	16	VPNService	(C2)	Disrupt specific network service			✔	Fixed
SE-Floodlight	17	StaticEntryPusher	(D1) → (D3)	Write specific network states	✔	✔		Reported
	18	Web GUI	(D1) → (D2)	Read specific network states		✔		Reported

1#: Network Manipulation **2#**: Data Leakage **3#**: Denial of Service
Researchers from Fraunhofer AISEC also discovered CVE-2017-1000078 and they reported it earlier than us.

attacker to access the database of the ODL-SDNi application and obtain certain network device information.

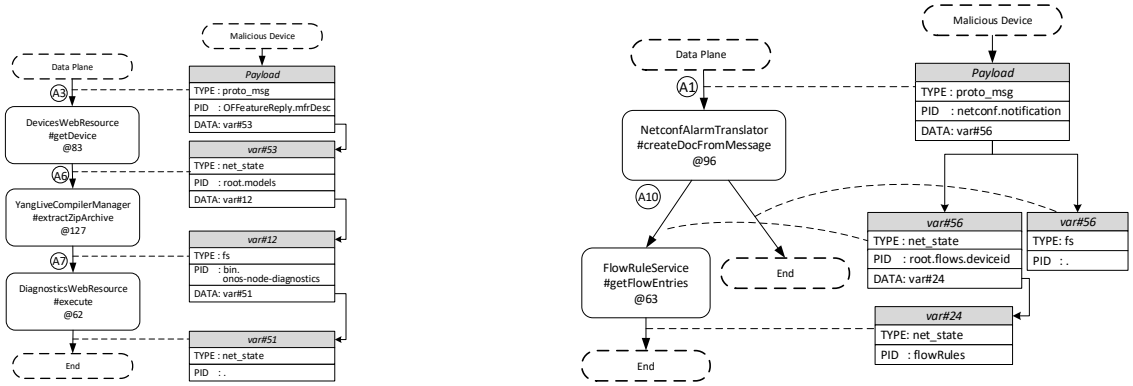
Denial of Service. We found that 8 D^2C^2 chains (i.e., DC-1, DC-2, DC-3, DC-9, DC-10, DC-11, DC-14 and DC-16) can hurt the availability of the controllers. In ONOS, we found 6 D^2C^2 chains. DC-1, DC-2 and DC-3 can terminate the controller through command execution. DC-9 incurs a `NumberFormatException` exception into the OVSDB component, which leads to disruption of the legitimate protocol interactions involved in the corresponding service. By abusing the sensitive method `FileOutputStream`, DC-10 and DC-11 can affect the functionalities of a component by corrupting its configuration files. In Floodlight, we found that DC-14 can crash the forwarding component by incurring exceptions during protocol handling. In OpenDaylight, DC-16 crashes the `vpnservice` component by incurring exceptions into its protocol parsing method(s).

Remark. Based on the impact analysis results, we claim that the D^2C^2 attack indeed achieves greater reachability in the control plane, which leads to a much larger SDN attack surface. This claim is supported by two main insights. (i) By triggering creation and chaining of unexpected data dependencies in a creative way, the D^2C^2 attack effectively attacks many components hidden deep in the control plane. According to our results, 13 previously unreachable control plane components are attacked via 21 newly created data dependencies. Without taking the newly created data dependencies into consideration, some of these exposed components (e.g., Diagnostic) in fact originally do not have any direct or indirect data dependencies with the data plane. (ii) The D^2C^2 attack achieves significant attack effects by abusing these exposed components. By "significant", we mean the following indicators. First, the D^2C^2 attack is able to cause almost every attack effect in previous data-to-control plane attacks. The only exception is the data-to-control plane saturation attack. We cannot achieve this particular attack effect because the D^2C^2 attack is not volumetric attack. Second, some of the attack effects previously caused by local control-plane-only attacks [37], [16], such as injection of manipulated flow rules, can also be caused by the D^2C^2 attack. Third, the D^2C^2 attack causes some completely new kinds of attack effects, including the execution of arbitrary commands in the controller and exfiltration/corruption of arbitrary configuration files.

D. Case Studies

In this section, we illustrate the severity of the D^2C^2 attack through two representative attack examples.

Arbitrary command execution in the control plane. With DC-2, an attacker in the data plane is able to execute arbitrary commands in the controller. Fig. 6(a) shows the exploit chain's event chain graph (left half) and payload crafting guide (right half). As shown in the left half, the target is the diagnostics component which holds the sensitive method `Runtime.getRuntime().exec()`. SVHunter identified a poisoning event (i.e. event (A7)) which takes input data from a local configuration file. To trigger this event, SVHunter identified another poisoning event (i.e. event (A6)) which can poison a category *fs* data item. This event is identified from the Yang component which creates new files with the sensitive method `Files.write()`. However, to control the content of the new files and trigger this event, we need a particular network state to be poisoned. This precondition is met by



(a) Remote command execution in ONOS.

(b) Steal sensitive network information in ONOS.

Fig. 6. Two D^2C^2 attack examples (We manually adjust the placement of some nodes to make the graphs look neat).

event (A3). In (A3), method `getDevice` is abused so that it can poison the data items in the *net_state* by accessing the corresponding Restful API. As shown in the figure, (A3) can be directly triggered by a custom field in a legitimate protocol message from the data plane. Hence, the attacker can exploit the whole D^2C^2 chain by sending a single malicious protocol message. Although there are several events and data items involved, crafting the payload is actually quite straightforward with the help of SVHunter. As shown in the right half of Fig. 6(a), the D^2C^2 chain can be exploited by sending a `OFFeatureReply.mfrDesc` protocol message to the controller while letting the message have a malicious payload crafted by using the guide shown in the figure. Regarding the attack effect, an attacker located in the data plane can leverage this D^2C^2 chain to execute arbitrary commands in the controller and even get a reverse system shell from it.

Stealing sensitive network information. The attack effect of chain DC-5 is to steal important network information (e.g., configurations, network topologies and routing policies). As shown in Figure 6(b), the first poisoning event of this D^2C^2 chain is event (A1), which happens in the core service NetConf. Since a method of this component (i.e. `createDocFromMessage()`) calls a sensitive method that can be abused to launch XML external entity attacks (XXE) [10] to handle certain data items in the category of *proto_msg* (i.e., custom field). As shown in the figure, there are two potential consequences of this poisoning event. First, it can directly read configurations in the local file system through the XXE attack. Second, it can also be leveraged to launch a HTTP request to access data through Restful API, which is denoted as event (A10). Event (A10) involves the sensitive method `getFlowEntries` which is used by ONOS to read flow rules. As shown in the right half of Fig. 6(b), an attacker only needs to craft the corresponding XXE payload into the notification message and then send it to the controller from a data plane device in order to launch this attack. Regarding the attack effect, a data plane attacker can leverage this D^2C^2 chain to steal either configuration (e.g., user credentials) or network state information (e.g., flow rules, network topology).

VII. Discussion

A. Countermeasures

Security Checks. To launch the D^2C^2 attack, an attacker has to find several exploitable sensitive methods. One way to defeat the D^2C^2 attack is to add security checks for the arguments of each sensitive method. For example, in Fig. 8(a), the attacker abused `extractZipArchive` in (A6) with a malicious input to write an arbitrary file into an arbitrary directory. By checking the arguments, the attacker will no longer be able to exploit this chain. Several chains we reported to the vendors have been patched with security checks.

Mitigating Malicious Dependency Creation. The D^2C^2 attack employs a data dependency creation strategy to increase the reachability. Therefore another potential mitigation method is to detect and mitigate the malicious data dependency creation. We can dynamically monitor the data flow of the entire controller to detect such malicious dependency creation. However, global monitoring may incur significant overhead to the controller. As future work, we are considering to design a custom algorithm to more efficiently detect abnormal data dependencies from massive legitimate data dependencies.

Sanitizing Protocol Interaction. The first step of all chains is to inject malicious payloads into the protocol messages and send them to the controllers. So another potential mitigation is to sanitize malicious protocol messages. We can sanitize the format and value range of every custom field to mitigate the malicious payload injection. Moreover, custom fields in the SDN protocol specifications should be carefully inspected and re-defined to make such attacks less possible.

B. Limitations

First, like many other static program analysis tools, SVHunter may trace a number of redundant information flows or miss some flows. To reduce false positives and false negatives, we can combine SVHunter with dynamic program analysis techniques. Second, since we do not perform fine-grained analysis of program semantics (SVHunter only performs keyword-based filtering on basic security checks in the SDN controllers), it is possible for SVHunter to construct an

inaccurate exploit chain whose exploitation might be impeded by some specific check functions. A potential method to increase accuracy is to combine SVHunter with, for example, symbolic execution [38], [31].

Also, SVHunter relies on a domain-specific language ERL to model the behavior of sensitive methods and APIs. Hence, it requires some manual effort to identify new sensitive methods or add some new tracing rules if the user wants to support more SDN controllers. The sensitive methods/APIs belong to different controllers usually have an overlap. As a result, users don't have to replace the whole list of sensitive methods to scale to a new controller.

VIII. Related Work

Security Vulnerabilities of SDN. Recently, researchers have discovered many security issues in SDN. Existing SDN attack research can be generally classified into two categories: attacks launching from the data plane and attacks launching from the control plane. The first category of research [34], [39], [20], [19], [35], [42] demonstrated that it is possible to introduce serious security and reliability issues to SDN networks by controlling a data plane device, e.g., switches or hosts. For example, ConGuard [42] found that a malicious network device/host can remotely exploit harmful race conditions in the control plane to introduce several different attack effects. Different from the previous work, we discovered a new type of control plane vulnerabilities which can be remotely exploited by creating malicious data dependencies.

In the second category of research [16], [37], a malicious (but underprivileged) control plane application (app) might introduce significant risks to the control plane regardless of the security policies that have been enforced into SDN application management. For instance, ProvSDN [37] discovered and mitigated the CAP (Cross-App Poisoning) attack, which is a powerful attack that can bypass SDN role-based access control to poison the control plane integrity with malicious SDN applications. Different from the attacks in the second categories that need a malicious app to be installed in the control plane, the D^2C^2 attack can be launched remotely from the data plane. More importantly, the D^2C^2 attack creates new data dependencies in an unexpected way to establish new attack paths towards the sensitive methods hidden deep in the control plane.

Security Enhancements in SDN. To mitigate potential vulnerabilities and attacks in SDN, researchers also developed new security applications/enhancements. For example, SE-FloodLight [29] achieved the detection and reconciliation of conflicting flow rules from different control applications by introducing a security enforcement kernel. However, as we have demonstrated, the security-enhanced controllers are not able to mitigate the D^2C^2 attack due to their inabilities to detect the creations of malicious data dependencies. Researchers also developed tools to identify malicious applications or behaviors in the control plane. INDAGO [24] introduced a static analysis framework to detect malicious SDN applications by extracting and classifying semantic features in these applications. SHIELD [23] also leveraged a static analysis approach to categorize several malicious behaviors of SDN applications. Since D^2C^2 attacks do not require any malicious applications to be installed, these enhancements are not suitable for the D^2C^2 attack detection.

SDN Testing and Auditing. Researchers also developed tools to help detect potential SDN bugs/vulnerabilities. One popular approach is to leverage fuzzing techniques to facilitate SDN bug discovery [25], [21], [30]. For example, BEADS [21] proposed a protocol fuzzer for SDN networks that identifies potential risks in the protocol handling logic within the control channel. However, all of them took a black box approach which is incapable of performing fine-grained data flow analysis in order to detect the critical creations of potentially malicious data dependencies needed by the D^2C^2 attack. Another widely used methodology is data/control flow analysis. Many existing studies [40], [24], [37], [23] also leveraged static program analysis techniques (e.g., taint analysis) to pinpoint potential vulnerable data flow or control flows. For example, CAP attacks [37] leveraged data flow analysis to study the data sharing relationships between different control plane applications. It is noteworthy that, while SVHunter employs a similar methodology to perform data flow analysis like previous work [37], [40], it possesses a different analysis goal, which is to identify existing data dependencies that are vulnerable to the unique data poisoning strategy of the D^2C^2 attack.

IX. Conclusion

In this work, we approach the vulnerability analysis problem of SDN networks from a new angle. We present a new attack that leverage legitimate protocol interactions to abuse sensitive methods in multiple SDN control plane components. The significance of this work is indicated by two critical indicators. The first indicator is a new discovery: a new kind of attack is discovered. With the new attack, attackers can achieve greater control plane reachability, which results in a much larger SDN attack surface. The enlarged attack surface leads to the discovery of 18 zero-day SDN vulnerabilities, all of which can be exploited remotely to introduce serious attack effects to the control plane. The second indicator is SVHunter, a one-of-a-kind tool which can effectively identify the newly discovered D^2C^2 vulnerabilities and construct the corresponding exploit chains. The tool combines data flow backtracking, an event reasoning language used to formally specify the preconditions and postconditions of data dependency chaining events, and automated causality reasoning.

Acknowledgement

We would like to thank our paper shepherd David Choffnes and the anonymous reviewers, for their insightful feedback that helped shape the final version of this paper. This work was supported in part by ARO W911NF-13-1-0421 (MURI), W911NF-15-1-0576, ONR N00014-16-1-2265, N00014-16-1-2912, N00014-17-1-2894, NSF CNS-1814679, CNS-1652790, 1617985, 1642129, 1700544, and 1740791.

REFERENCES

[1] *ANTLR tool*, http://www.antlr.org.

[2] *CVE-2016-2074: Open vSwitch Buffer Overflow*, https://nvd.nist.gov/vuln/detail/CVE-2016-2074.

[3] *CVE-2017-3881: Cisco Catalyst Remote Code Execution*, https://nvd.nist.gov/vuln/detail/CVE-2017-3881.

[4] *HPE VAN SDN Controller*, https://h17007.www1.hpe.com/ie/en/\networking/solutions/technology/sdn.

[5] *Huawei Agile Controller*, https://e.huawei.com/us/products/enterprise-networking/sdn-controller.

[6] "Mininet: Rapid prototyping for software defined networks," http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/.

[7] "Onos controller platform," https://onosproject.org/.

[8] *OpenDaylight OpenFlow Plugin: TLS Support.*, https://wiki.opendaylight.org/view/OpenDaylight_OpenFlow_Plugin:_TLS_Support.

[9] *OpenFlow and REST API Security Configuration.*, https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/5636115/OpenFlow+and+REST+API+Security+Configuration.

[10] *XML External Entity attack (XXE)*, https://en.wikipedia.org/wiki/XML_external_entity_attack.

[11] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan, "{NAVEX}: Precise and scalable exploit generation for dynamic web applications," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 377–392.

[12] W. Braun and M. Menth, "Software-defined networking using openflow: Protocols, applications and architectural design choices," *Future Internet*, vol. 6, no. 2, pp. 302–336, 2014.

[13] J. Cao, Q. Li, R. Xie, K. Sun, G. Gu, M. Xu, and Y. Yang, "The crosspath attack: Disrupting the sdn control channel via shared links," in *Proceedings of The 28th USENIX Security Symposium (Security'19)*, August 2019.

[14] B. Chandrasekaran and T. Benson, "Tolerating sdn application failures with legosdn," in *Proceedings of the 13th ACM workshop on hot topics in networks*. ACM, 2014, p. 22.

[15] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[16] V. H. Dixit, A. Doupé, Y. Shoshitaishvili, Z. Zhao, and G.-J. Ahn, "Aim-sdn: Attacking information mismanagement in sdn-datastores," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 664–676.

[17] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: an intellectual history of programmable networks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 87–98, 2014.

[18] E. Haleplidis, K. Pentikousis, S. Denazis, J. H. Salim, D. Meyer, and O. Koufopavlou, "Software-defined networking (sdn): Layers and architecture terminology," Tech. Rep., 2015.

[19] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: New attacks and countermeasures," in *Proceedings of the 22th Annual Network and Distributed System Security Symposium (NDSS)*, February 2015.

[20] S. Jero, W. Koch, R. Skowyra, H. Okhravi, C. Nita-Rotaru, and D. Bigelow, "Identifier binding attacks and defenses in software-defined networks," in *Proceeding of the 24th USENIX Security Symposium (USENIX Security)*, August 2017.

[21] S. Jero, X. Bu, C. Nita-Rotaru, H. Okhravi, R. Skowyra, and S. Fahmy, "Beads: automated attack discovery in openflow-based sdn systems," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2017, pp. 311–333.

[22] P. Lam, E. Bodden, O. Lhotak, and L. Hendren, "The soot framework for java program analysis: a retrospective," in *CETUS 2011*.

[23] C. Lee and S. Shin, "Shield: an automated framework for static analysis of sdn applications," in *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*. ACM, 2016, pp. 29–34.

[24] C. Lee, C. Yoon, S. Shin, and S. K. Cha, "Indago: A new framework for detecting malicious sdn applications," in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. IEEE, 2018, pp. 220–230.

[25] S. Lee, C. Yoon, C. Lee, S. Shin, V. Yegneswaran, and P. Porras, "Delta: A security assessment framework for software-defined networks," in *Proceedings of The 2017 Network and Distributed System Security Symposium (NDSS)*, February 2017.

[26] S. Liu, M. K. Reiter, and V. Sekar, "Flow reconnaissance via timing attacks on sdn switches," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 196–206.

[27] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[28] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software." in *NDSS*, vol. 5. Citeseer, 2005, pp. 3–4.

[29] P. Porras, S. Cheung, M. Fong, K. Skinner, and V. Yegneswaran, "Securing the Software-Defined Network Control Layer," in *NDSS'15*, 2015.

[30] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock *et al.*, "Troubleshooting blackbox sdn control software with minimal causal sequences," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 395–406, 2015.

[31] K. Sen and G. Agha, "Cute and jcute: Concolic unit testing and explicit path model-checking tools," in *International Conference on Computer Aided Verification*. Springer, 2006, pp. 419–423.

[32] S. Sezer, S. Scott-Hayward, P. K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao, "Are we ready for sdn? implementation challenges for software-defined networks," *IEEE Communications Magazine*, vol. 51, no. 7, pp. 36–43, 2013.

[33] U. Shankar, K. Talwar, J. S. Foster, and D. A. Wagner, "Detecting format string vulnerabilities with type qualifiers." in *USENIX Security Symposium*, 2001, pp. 201–220.

[34] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Avant-guard: Scalable and vigilant switch flow management in software-defined networks," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, November 2013.

[35] R. Skowyra, L. Xu, G. Gu, V. Dedhia, T. Hobson, H. Okhravi, and J. Landry, "Effective topology tampering attacks and defenses in software-defined networks," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 374–385.

[36] K. Thimmaraju, B. Shastry, T. Fiebig, F. Hetzelt, J.-P. Seifert, A. Feldmann, and S. Schmid, "Taking control of sdn-based cloud systems via the data plane," in *Proceedings of the Symposium on SDN Research*. ACM, 2018, p. 1.

[37] B. E. Ujcich, S. Jero, A. Edmundson, Q. Wang, R. Skowyra, J. Landry, A. Bates, W. H. Sanders, C. Nita-Rotaru, and H. Okhravi, "Cross-app poisoning in software-defined networking," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 648–663.

[38] W. Visser, C. S. Psreanu, and S. Khurshid, "Test input generation with java pathfinder," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 97–107, 2004.

[39] H. Wang, L. Xu, and G. Gu, "Floodguard: A dos attack prevention extension in software-defined networks," in *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2015.

[40] H. Wang, G. Yang, P. Chinprutthiwong, L. Xu, Y. Zhang, and G. Gu, "Towards fine-grained network security forensics and diagnosis in the sdn era," in *Proc. of the 25th ACM Conference on Computer and Communications Security (CCS'18)*, October 2018.

[41] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Security and privacy (SP), 2010 IEEE symposium on*. IEEE, 2010, pp. 497–512.

[42] L. Xu, J. Huang, S. Hong, J. Zhang, and G. Gu, "Attacking the brain: Races in the sdn control plane," in *Proceedings of The 26th USENIX Security Symposium (Usenix Security)*, August 2017.

[43] Y. Zheng, X. Zhang, and V. Ganesh, "Z3-str: A z3-based string solver for web application analysis," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 114–124.

APPENDIX

TABLE VII
OBSERVATIONS OF IDENTIFIED POISONING EVENTS

Controller	Event #	Observation
ONOS	A1	if proto_msg::NetconfAlarmProvider→read var#56 then (fs::.→read var#56) || (net_state::root→read var#56)
	A2	if proto_msg::NetConfControllerConfig→read var#1 then (fs::.→read var#1) || (net_state::root→read var#1)
	A3	if proto_msg::OFFeatureReply.mfrDesc→read var#53 then net_state::root→write var#53
	A4	if proto_msg::VersionNum→read var#68 then net_state::ovsdb→crash var#68
	A5	if net_state::root.applications.upload→write var#9 then fs::.→write var#49
	A6	if net_state::root.models→write var#12 then fs::.→write var#12
	A7	if fs::bin.onos-node-diagnostics→read var#51 then net_state::.→write var#51
	A8	if net_state::root.topo→read var#73 then net_state::topology→read var#73
	A9	if fs::model.jar→read var#64 then net_state::.→write var#64
	A10	if net_state::root.flows.deviceid→read var#24 then net_state::flowRules→write var#24
Floodlight	B1	if proto_msg::OFFeatureReply.mfrDesc→read var#1 then net_state::.→write var#1
	B2	if proto_msg::OFFeatureReply.SwitchDescription→read var#2 then net_state::Forwarding→crash var#2
	B3	if net_state::wm.staticflowentrypusher.json→write var#11 then net_state::flow→write var#11
	B4	if net_state::wm.device.all.json→read var#17 then net_state::device→read var#17
OpenDaylight	C1	if proto_msg::getPortName→read var#5 then net_state::PortID→write var#5
	C2	if proto_msg::LLDP→read var#7 then net_state::AlivenessProtocolHandlerLLDP→crash var#7
	C3	if net_state::PortID→read var#8 then db::.→crash var#8
SE-Floodlight	D1	if proto_msg::OFFeaturesReply→read var#7 then net_state::root.wm.core.switch.switchId.statType.json→write var#7
	D2	if net_state::root.wm.core.switch.switchId.statType.json→write var#9 then net_state::.→write var#9
	D3	if net_state::wm.staticflowentrypusher.json→write var#11 then net_state::flow→write var#11

TABLE VIII
CUSTOM FIELDS SUPPORTED BY EXPLOIT ENGINE

Protocol Name	Message Name	Field Name
OpenFlow	MultipartRes	mfr_desc
	MultipartRes	sw_desc
	MultipartRes	serial_num
	MultipartRes	dp_desc
	MultipartRes	hw_desc
	FeatureRes::phy_port	name
NetConf	Notification	event