Towards Automated Verification of Active Cyber Defense Strategies on Software Defined Networks

Mohammed Noraden Alsaleh and Ehab Al-Shaer
Department of Software and Information Systems
University of North Carolina at Charlotte
Charlotte, NC, USA
{malsaleh, ealshaer}@uncc.edu

ABSTRACT

Active Cyber Defense (ACD) reconfigures cyber systems (networks and hosts) in timely manner in order to automatically respond to cyber incidents and mitigate potential risks or attacks. However, to launch a successful cyber defense, ACD strategies need to be proven effective in neutralizing the threats and enforceable under the current state and capabilities of the network. In this paper, we present a bounded model checking framework based on SMT to verify that the network can support the given ACD strategies accurately and safely without jeopardizing cyber mission invariants. We abstract the ACD strategies as sets of serializable reconfigurations and provide user interfaces to define cyber mission invariants as reachability, security, and QoS properties. We then verify the satisfaction of these invariants under the given strategies. We implemented this system on OpenFlow-based Software Defined Networks and we evaluated the time complexity for verifying ACD strategies on OpenFlow networks of over two thousand nodes and thousands of rules.

Keywords

Active cyber defense, configuration, verification, OpenFlow, Software Defined Networks, Bounded model checking

1. INTRODUCTION

Passive cyber defense is not sufficient to address increasingly fast and sophisticated attacks [11, 10]. According to DARPA's active cyber defense (ACD) program, a successful ACD requires synchronized, real-time capabilities to discover, define, analyze and mitigate cyber threats and vulnerabilities [1]. Thus, the scope of ACD spans a wide range of activities related to network monitoring and management in order to monitor networks, detect attacks, and safely mitigate them.

In order to mitigate cyber threats, multiple network reconfigurations are often required. For example, if a DDoS attack causes a certain critical link to be flooded, the ACD strategy may dictate to migrate the affected services to another secure server. However, migrating services is not an atomic task as it may require multiple changes in the network to be executed in a certain order. For a successful ACD, we believe that this sequence of reconfigurations should satisfy three important properties: *consistency*, *enforceability*, and *effectiveness*. The *Consistency* property is satisfied if the consecutive or potentially concurrent reconfigurations do not contradict each other. The *Enforceability* means that the network is capable of handling the reconfigurations dictated by the ACD strategies without introducing any violations for the network mission invariants due to misconfigurations or unforeseen lack of resources. The *Effectiveness* means that the strategy should achieve the expected outcome by disrupting or neutralizing the targeted threat.

In this work, we focus on the *enforceability* property and we defer the other two properties for the future work. As depicted in Figure 1, we present the design and implementation of a bounded model checking framework that verifies the enforceability of ACD strategies in OpenFlow-based SDNs with respect to given mission invariants. If the ACD strategy is found not enforceable, we provide the means to diagnose the potential violations that may result from invalid ordering of the strategy reconfigurations. If the violation is not related to the order, a counter example will be returned to help understanding how the ACD strategies or the network configuration can be changed to make the strategies enforceable. Specifically, our contribution is as follows:

- We provide a formal specification for the ACD strategies as a set of reconfiguration. This set will be used to generate a series of configuration states (i.e., snapshots), on which the network mission invariants will be verified.
- We provide a flexible high level language to specify the network mission invariants as reachability, security, and QoS requirements. The language allows users to define any combination of QoS parameters at run time and supports linear and non-linear arithmetic operations to set thresholds on the QoS parameters.
- We model the complete pipeline processing of OpenFlow switches to capture the packet transformations between OpenFlow switches and between flow tables inside a single OpenFlow switch. Our model considers the data rate configurations of OpenFlow queues, among other performance parameters, to detect QoS violations.

To model the complete network configuration in addition to the mission invariants and ACD specification, we implemented ACDChecker, a special-purpose model checker for

SafeConfig'16, October 24 2016, Vienna, Austria
© 2016 ACM. ISBN 978-1-4503-4566-8/16/10. . . $15.00
DOI: http://dx.doi.org/10.1145/2994475.2994482

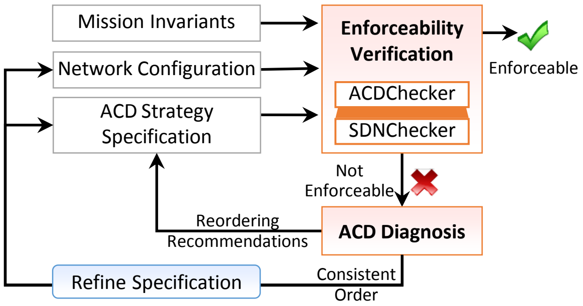

Figure 1: ACD Enforceability Verification.

ACD strategies, on top of SDNChecker, a general-purpose verification tool for OpenFlow-based SDNs. While SDNChecker models the data plane configuration and allows for the verification of properties expressed using the standard LTL language, ACDChecker is customized for the verification of ACD strategies. It translates both the mission and ACD specifications to the appropriate LTL expressions that are fed to SDNChecker. The SDNChecker and, inherently, ACDChecker are implemented using bounded model checking approach based on Satisfiability Modulo Theories (SMT). OpenFlow Software defined networks provide central configuration management, which makes it more suited for ACD than traditional networks as the network controller can update the configuration of multiple switches in response to attack events and incidents. OpenFlow allows multiple actions to be executed on the same flow. The temporal relations between these actions is crucial for the correctness of mission invariants. Model checking can efficiently capture this temporal dependency by exploring all relevant paths. Bounded model checking does not require exponential space or manual manipulation as the case of variable ordering in BDD-based verification [5, 13]. Moreover, using SMT provides a flexible model that supports advanced decision procedures for linear and non-linear arithmetic and difference logic beyond bit level operations, which are needed to consider quantitative invariants, such as QoS requirements.

The rest of the paper is organized as follows. In Sections 2, we present the technical details to build the general-purpose SDNChecker based on OpenFlow data plane configuration. Section 3 presents our model of ACD strategies and how we verify their enforceability. In Section 4, we present the performance evaluation. Finally, the related work and conclusions are presented in Section 5 and Section 6, respectively.

2. NETWORK CONFIGURATION MODEL

In this section, we present SDNChecker, a bounded model checker that encodes the entire data plane of an OpenFlow-based SDN and verifies properties expressed using the generic LTL language. We show how we build the transition system based on the OpenFlow data plane configuration and how we encoded it using SMT in order to verify the properties.

2.1 An Overview of OpenFlow Configuration

An OpenFlow-based network consists of a set of OpenFlow-enabled switches controlled by a central controller. The controller provides interfaces to deploy network management applications and dynamically update the flow tables of the OpenFlow switches. OpenFlow switches operate based on a flow-based rule sets. Each flow rule consists of six components: the match fields, priority, counters, instructions, timeouts, and cookie [14]. In this work, we model the components that directly affect the packet processing, which include the match fields, the priority, and the instructions set. The match fields represent filters on the flow headers that determine the set of the flows matching the flow entry. They include the Ethernet, IP, and TCP/UDP headers in addition to the VLAN, MPLS, and PBB tags. The priority determines the matching precedence of the flow rules. The instructions set contains any of the instructions {*Apply-Actions, Clear-Actions, Write-Actions* or *Goto-Table*}.

Location Variables	
loc	The unique switch ID
tab	The number of the flow table in a switch
Match Fields Variables ($\mathbb{F}$)	
IN_PORT	Ingress port: A physical or logical port.
ETH_DST	Ethernet destination MAC address.
ETH_SRC	Ethernet source MAC address.
ETH_TYPE	Ethernet type of the OF packet payload.
VLAN_ID	VLAN-ID from 802.1Q header.
VLAN_PCP	VLAN-PCP from 802.1Q header.
IP_PROTO	IP protocol number.
IP_SRC	IPv4 source address.
IP_DST	IPv4 destination address.
SRC_PORT	Source port number.
DST_PORT	Destination port number.
IP_DSCP	Diff Serv Code Point (DSCP).
IP_ECN	ECN bits of the IP header.
Action Set Variables ($\mathbb{A}$)	
AS_POP_V	Pop the outer-most VLAN header.
AS_PUSH_V	Push a new VLAN header.
*AS_SET_*<Var>	Set the value of *Var* ($Var \in \mathbb{F}$).
AS_SET_QUEUE	Set the output queue to a specific value.
AS_OUT	Forward the packet to a specific port.

Table 1: State Variables.

We model the hosts as basic OF switches with one port and flow table. A host's flow table contains only one rule that redirects all the outgoing packets to the host's gateway.

2.2 Transition System

The transition systems consists of states, that are defined over a set of system variables, connected by transitions, which capture the changes in system's variables based on the actions of the system and its environment.

States. In our model, the state of the network is determined by the unique flows that can be transferred through the network and their possible locations. The flows are represented by the flow match fields. In addition, our model provides the ability to incorporate special purpose variables for particular mission invariants, such as QoS guarantees. Therefore, the state of the network is modeled by four groups of variables: the location variables, the match fields variables $\mathbb{F}$, the Action Set variables $\mathbb{A}$, and the special purpose variables $\mathbb{L}$. Formally, the state is encoded by the following characteristic function:

$$\sigma : \boldsymbol{loc} \times \boldsymbol{tab} \times \mathbb{F} \times \mathbb{A} \times \mathbb{L} \rightarrow \{true, false\} \qquad (1)$$

Table 1 shows a list of the location, match fields, and action set variables along with their meanings. The Action Set contains a set of actions carried between the flow tables during the pipeline processing. Since the instructions of flow entries can modify the Action Set by inserting or removing actions, we need to keep track of its content during the transitions. The actions in the Action Set are executed when the instruction set of the matching flow entry does not contain a

Goto instruction. Based on OpenFlow specification, the Action Set contains a maximum of one action of each type. We define a variable for each possible action in the Action Set. If the value of that variable is *null*, the associated action is not part of the Action Set.

Transitions. Transitions are built based on packet transformations across flow tables. We add a transition if we encounter an *Output* action (a transition to a new OF switch) or a *Goto* instruction (a transition to a new table in the same OF switch). Multiple transitions may be associated with the same flow rule. Let us consider the rule r_i that belongs to the table t in the OF switch s. Let $\mathbb{R}_i$ be the set of values specified in the rule r_i for the match fields indexed by the variables' names (i.e., $\mathbb{R}_i[\mathcal{F}]$ is the value of the field $\mathcal{F}$ in the rule r_i). The transitions of the rule r_i are calculated as follows:

$$S_i = (\boldsymbol{loc} = s) \wedge (\boldsymbol{tab} = t) \wedge \textstyle\bigwedge_{\mathcal{F} \in \mathbb{F}} (\boldsymbol{\mathcal{F}} = \mathbb{R}_i[\mathcal{F}]) \wedge \neg \boldsymbol{S_{-i}} \quad (2)$$

$$S'_{i,a} = \textstyle\bigvee_{k \in \mathbb{O}} \left[\begin{array}{l} (\boldsymbol{loc'} = k_{tar}) \wedge (\boldsymbol{tab'} = 0) \\ \wedge \displaystyle\bigwedge_{\mathcal{F} \in \mathbb{F}} (\boldsymbol{\mathcal{F}'} = \mathbb{R}'_i[\mathcal{F}]) \end{array} \right] \wedge \textstyle\bigwedge_{\mathcal{A} \in \mathbb{A}} (\boldsymbol{\mathcal{A}'} = 0) \quad (3)$$

$$\begin{aligned} S'_{i,g} = {} & (\boldsymbol{loc'} = loc) \wedge (\boldsymbol{tab'} = new\_table) \wedge \\ & \bigwedge_{\mathcal{F} \in \mathbb{F}} (\boldsymbol{\mathcal{F}'} = \mathbb{R}'_i[\mathcal{F}]) \wedge \bigwedge_{\mathcal{A} \in \mathbb{A}} (\boldsymbol{\mathcal{A}'} = \mathbb{R}'_i[\mathcal{A}]) \end{aligned} \quad (4)$$

$$\begin{aligned} S'_{i,as} = {} & \mathbb{R}'_i[AS\_OUT] \rightarrow [(\boldsymbol{loc'} = \mathbb{R}'_i[AS\_OUT]) \wedge \\ & (\boldsymbol{tab'} = 0) \wedge \bigwedge_{\mathcal{F} \in \mathbb{F}} Exp[\mathcal{F}] \wedge \bigwedge_{\mathcal{A} \in \mathbb{A}} (\boldsymbol{\mathcal{A}'} = 0)] \end{aligned} \quad (5)$$

In equation 2, we calculate the flow space S_i of the rule (i.e. all the flows that match the values in the rule r_i and do not match another rule with higher priority). The expression S_{-i} captures the flow space for all the rules that have higher priority in the same flow table.

The flow space calculated in Equation 2 encodes the current state of the transitions associated with the rule r_i. To encode the next state(s), we use the primed variable $\mathcal{F}'$ to represent the next state variable of the field $\mathcal{F}$. We also define the set $\mathbb{R}'_i$ that keeps the values of the next state variables during the execution of the rule's instructions list. Initially, the next state variables have the same values as the current state ones (i.e., $\forall_{\mathcal{F} \in \mathbb{F}} : \mathbb{R}'[\mathcal{F}] = \mathcal{F}$). There are three sources of transitions in a flow entry. (1) The *Output* action(s) in the *Apply-Actions* instruction's actions list. (2) The *Goto* instruction, which transfers packets processing from one flow table to another in the same OpenFlow switch. (3) The *Output* action in the packet's Action Set. If the instructions list of a flow entry does not contain a *Goto* instruction, the actions in the packet's Action Set are executed, where the *Output* action is executed last. If no *Output* action exists in the Action Set, the packet is dropped. The next states of the three cases are encoded in equations 3, 4, and 5, respectively. The set $\mathbb{O}$ of Equation 3 includes the *Output* actions in the *Apply-Actions* instruction's actions list, where k_{tar} is the target switch of the *Output* action at index k. Note that in equations 3 and 5, the packet is transferred to another switch; hence, the Action Set's variables ($\mathbb{A}$) need to be cleared. The Exp set in Equation 5 captures the effects of *Set* actions in the Action Set. For example, the final value of the next state variable *IP_SRC'* depends on the value of the Action Set variable *AS_SET_IP_SRC'*. $Exp[IP\_SRC]$ captures the conditional statement shown in Equation 6 that encodes the value of the variable *IP_SRC'* depending on the value of *AS_SET_IP_SRC'*. The ITE operator stands for IF-THEN-ELSE.

$$\begin{aligned} & ITE(\mathbb{R}'_i[AS\_SET\_IP\_SRC], \\ & IP\_SRC' = \mathbb{R}'_i[AS\_SET\_IP\_SRC], IP\_SRC' = \mathbb{R}'_i[IP\_SRC]) \end{aligned} \quad (6)$$

The complete transition relation of the rule r_i based on Equations 2 to 5 is represented as

$$T(r_i) = S_i \wedge (S'_{i,a} \vee S'_{i,g} \vee S'_{i,as}) \quad (7)$$

Global Transition Relation. The global transition relation is the disjunction of the transition relations of all the rules. For the network N that has $\mathbb{S}$ switches, the global transition relation $T_g(N)$ is calculated as

$$Tg(N) = \left[\bigvee_{s \in \mathbb{S}} \bigvee_{t=1}^{len(s)} \bigvee_{i=1}^{len(t)} T(r_{s,t,i}) \right] \wedge \bigwedge_{\mathcal{L} \in \mathbb{L}} f(\mathcal{L}, \mathcal{L}') \quad (8)$$

Where $len(s)$ is the number of flow tables in the switch s, $len(t)$ is the number of rules in the flow table t and $r_{s,t,i}$ is the rule i in the flow table t that belongs to the switch s. $T(r_{s,t,i})$ is calculated according to Equation 7. $\mathbb{L}$ is the set of special-purpose variables and $f(\mathcal{L}, \mathcal{L}')$ is the transition of variable $\mathcal{L}$.

2.3 SMT-based Bounded Model Checking

Encoding the model as an SMT-based satisfaction formula is done by unfolding the model for k steps starting from the initial states S_0. Let $V = \mathbb{F} \cup \mathbb{A} \cup \mathbb{L} \cup \{loc, tab\}$ be the set of variables that represent a state in the system, and let V_i denote the set of variables that represent the state i. We use $I(V_i)$ to represent a relation in terms of the variables V_i. The global transition relation computed in Equation 8 is denoted in terms of the current and next state variables as $T(V_i, V_j)$ for a transition from state i to state j. The network properties are encoded as a relation in terms of the variables $\{V_0, V_1 \dots V_k\}$. The formula representing the unfolded system M_k can be represented as:

$$M_k = I(V_0) \wedge \bigwedge_{i=1}^{k} T(V_{i-1}, V_i) \wedge I(V_0, V_1, \dots V_k) \quad (9)$$

The transition relation is unfolded starting from the base transition relation computed in Equation 8. Given the base transition $T(V, V')$ and a bound k, we unfold the transition relation as follows. (1) Define the variables set V_0. (2) Initialize i to 1, and for all $i \leq k$, perform the following three steps. (3) Define the new variables set V_i. (4) Construct a new formula $T(V_{i-1}, V_i)$ by replacing all variables of V and V' with the corresponding variables from V_{i-1} and V_i. (5) Add the new transition formula to the model as an assertion.

The network properties need to be translated to SMT expressions as well. We use the standard LTL specification language as generic means to specify system properties. A property in LTL can contain the temporal connectives: *next* (**X**), *eventually* (**F**), *global* (**G**), *until* (**U**) and *release* (**R**) operators. Let Ψ_i be a constraint in the property q expressed in terms of the i^{th} state variables (V_i); we denote the SMT expression of a property q at point i on a path given a bound of k as $[q]_i^k$. Based on this notation, the property encoding as SMT formula can be recursively defined as shown in Figure 2. The translation is based on the standard semantics of LTL operators except for the $\boldsymbol{G}$ operator. To encode the $\boldsymbol{G}$ operator, we forward the packets that are dropped or have reached their destinations to a designated node called *sink*.

$$
\begin{array}{rcl}
[q]_i^k & \Leftrightarrow & \Psi_i \\
[\neg q]_i^k & \Leftrightarrow & \neg [q]_i^k \\
[q_1 \wedge q_2]_i^k & \Leftrightarrow & [q_1]_i^k \wedge [q_2]_i^k \\
[q_1 \vee q_2]_i^k & \Leftrightarrow & [q_1]_i^k \vee [q_2]_i^k \\
[\boldsymbol{X}\ q]_i^k & \Leftrightarrow & [q]_{i+1}^k \text{ if } i < k \text{ , and } \bot \text{ otherwise} \\
[\boldsymbol{F}\ q]_i^k & \Leftrightarrow & \bigvee_{j \in [i,k]} \ [q]_j^k \\
[\boldsymbol{G}\ q]_i^k & \Leftrightarrow & [q\ \boldsymbol{U}\ (\boldsymbol{loc} = sink)]_i^k \\
[q_1\ \boldsymbol{U}\ q_2]_i^k & \Leftrightarrow & \bigvee_{j \in [i,k]} \left([q_2]_j^k \wedge \bigwedge_{l \in [i,j)} \ [q_1]_l^k \right) \\
[q_1\ \boldsymbol{R}\ q_2]_i^k & \Leftrightarrow & \bigwedge_{j \in [i,k]} \left([q_2]_j^k \vee \bigvee_{l \in [i,j)} \ [q_1]_l^k \right)
\end{array}
$$

Figure 2: Property Encoding.

If a packet reaches the *sink* node then the path is terminated. The property $[\boldsymbol{G}\ q]$ is simplified to $[q\ \boldsymbol{U}\ (\boldsymbol{loc} = sink)]$ in our model, which means that q holds until the path is terminated.

After unfolding the transition relation and the LTL properties, the complete SMT formula M_k that is calculated based on Equation 9 is fed to Z3 SMT solver, which determines if the formula is satisfiable or not.

3. VERIFYING ACD STRATEGIES

The generation of the ACD strategies in response to attacks and incidents is out of the scope of this work. However, we demonstrate in this section how to formalize given ACD strategies and translate them along with the mission requirements to LTL properties that can be fed to SDNChecker.

3.1 ACD Strategy Specification

We abstract an ACD strategy as a set of reconfigurations that may be applied sequentially or in parallel. The basic building block of ACD strategies is called Cyber Command.

Cyber Command We define the cyber command $\mathcal{C}$, as the triplet $\langle t, o, \alpha \rangle$, which specifies that the action α is taken on the object o by the actuator t.

In Figure 3, we present the syntax of a simple specification language to define ACD strategies. The language allows the users to specify multiple cyber commands and combine them using sequential and/or parallel operators to compose a single ACD strategy. The operators $\|$ and $\gg$ are used to denote parallel and sequential composition, respectively. In each cyber command, the actuator is the device that performs the action. For example, OpenFlow switches perform OF actions, such as *Set*, on the traffic flows; the switches are the actuators in this case. The objects can be traffic flows or other networking components such as the hosts, services, switches, and their ports. Each object may have a number of attributes that can uniquely identify it. The attributes of traffic flows are their header information, while other entities like hosts and services may be identified by their IP addresses/port numbers. We define two groups of actions that may be selected based on the object. If the object is a traffic flow, the action belongs to any of the actions supported by the OpenFlow switches, such as *forward* and *Set*. If the object is another entity in the network such as hosts, services, or virtual machines, the action will be selected from the management actions group, denoted by *Mgnt Actions* in Figure 3. Examples of management actions include enabling/disabling ports in OF switches, setting the data rate of a queue, or migrating virtual machines and services.

$$
\begin{array}{rcl}
\textit{Actuator}\ \mathcal{L} & ::= & <\text{A device in the network}> \\
\textit{OF Action}\ \alpha_{of} & ::= & \textit{forward} \mid \textit{block} \mid \textit{Set} \mid ... \\
\textit{Mgnt Action}\ \alpha_{mg} & ::= & \textit{enable} \mid \textit{disable} \mid \textit{migrate} \mid ... \\
\textit{Object}\ \mathcal{O} & ::= & \textit{switch} \mid \textit{host} \mid \textit{VM} \\
\textit{Cyber Command}\ \mathcal{C} & ::= & \langle \mathcal{L}, \textit{flow}(\{\textit{Attrib}\}), \alpha_{of}(\{\textit{Arg}\}) \rangle \mid \\
 & & \langle \mathcal{L}, \mathcal{O}(\{\textit{Attrib}\}), \alpha_{mg}(\{\textit{Arg}\}) \rangle \\
\textit{ACD Strategy}\ \mathcal{S} & ::= & \mathcal{C} \mid \mathcal{S} \parallel \mathcal{S} \mid \mathcal{S} \gg \mathcal{S}
\end{array}
$$

Figure 3: ACD Specification Language.

Example. Let's consider an ACD strategy that migrates a virtual machine, whose address and universal ID are VM_{add} and VM_{web}, to the server *Srv* in response to flooding a critical link. On the network configuration level, the migration will not be successful until the flow tables at the switch SW_{border} are updated to reroute the appropriate traffic to the new location's gateway SW_{gw} avoiding the flooded link. In addition, we need to update the access control at SW_{gw} to allow the traffic to reach the VM.

$$
\begin{aligned}
\mathcal{S}_{mig} = \ & \langle SW_{gw}, flow(dst = VM_{add}), forward \rangle \gg \\
& [\langle Srv, VM(UUID = VM_{web}), migrate \rangle \parallel \\
& \langle SW_{border}, flow(dst = VM_{add}), forward(SW_{gw}) \rangle]
\end{aligned}
$$

In this strategy, we define three cyber commands. First, the access control at the gateway switch is updated, then the traffic rerouting and VM migration are executed concurrently. Note that we used an object called *VM*, which has an attribute called *UUID*.

3.2 Mission Invariants Specification

We provide the means to specify network mission invariants as reachability between network services and QoS requirements in terms of QoS parameters, such as bandwidth, delays, and jitter. The network data plane can directly affect these parameters as it determines which heterogeneous switches, ports, and queues the traffic passes through.

The language shown in Figure 4 can be used to specify the mission invariants. The construct *CanReach* is used to specify a reachability requirement with QoS constraints between a pair of locations in the network. A location in the network is specified by an IP address and a port number with the ability to use wild cards to specify multiple addresses or port numbers. The *QoS Constraints* are composed of a set of conditions on the aggregate values of the QoS parameters. The aggregate functions *max*, *min*, *sum*, and *avg* calculates the maximum, the minimum, the summation, and the average values of the parameter ρ in the path between the specified locations.

3.3 ACD Enforceability Verification

The enforceability of an ACD strategy is satisfied if the network mission invariants are ensured at any configuration state during and after the execution of the strategy. Recall that the strategy may consist of multiple cyber commands. Starting from the initial configuration of the system, S_0, we generate new configuration states, $\{S_1, S_2, ..., S_k\}$, that result from executing the k-commands ACD strategy, and

$$\begin{array}{rl}
\textit{QoS Param } \rho ::= & \textit{BW} \mid \textit{D\_RATE} \mid \textit{DELAY} \mid \ldots \\
\textit{Operator } \bowtie ::= & > \mid < \mid \geq \mid \leq \mid == \\
\textit{Term } \Phi ::= & \mathbb{Z}^{+} \mid \mathit{max}(\rho) \mid \mathit{min}(\rho) \mid \mathit{sum}(\rho) \mid \mathit{avg}(\rho) \\
\textit{QoS Constraint } \Psi ::= & \Phi \bowtie \Phi \mid \Psi \wedge \Psi \mid \Psi \vee \Psi \\
\textit{Location } \mathcal{L} ::= & < \text{Location in the network (ip:port)} > \\
\textit{Invariant } R ::= & \text{CanReach}(\mathcal{L},\ \mathcal{L},\ \Psi) \mid R \vee R \mid R \wedge R
\end{array}$$

Figure 4: Mission Invariants Specification Language. $\mathbb{Z}^{+}$ is the set of positive integers.

we verify the mission invariants at all the new configuration states. To clarify this point, an ACD strategy that consists of n sequential cyber commands is enforceable if:

$$S_k \models V \text{ for all } k \leq n$$

where V is the set of mission invariants. However, generating the correct sequence of configuration states is not always straight forward because cyber commands may be combined using the parallel composition operator. Although the ACD specification may specify that two commands are executed in parallel, the parallel notion cannot be enforced due to the lack of synchronization between the actuators in the network (i.e., there is no guarantee that the two parallel commands will start and end at exactly the same moments).

To handle the parallel composition, we do not generate only one deterministic sequence of configuration states. Instead, we generate all possible order permutations for the parallel cyber commands. For example, for the strategy $G = c_1 \parallel c_2$, we generate two sequences. In one sequence, c_1 is executed before c_2 and in the second, c_2 is executed before c_1. We verify the mission invariants at all possible permutations. The ACD is considered enforceable if the mission invariants are satisfied regardless of the execution order of the parallel commands.

3.4 Verifying Mission Invariants

The verification of the mission invariants is repeated at each potential configuration state of the system that results from executing the cyber commands in an ACD strategy. In this section, we show the steps we follow to translate mission invariants to LTL expressions and verify them at a single configuration state (i.e., snapshot).

- The QoS parameters, such as bandwidth and data rate, are encoded as special purpose variables. We provide a special keyword, *map*, to map the values of these parameters based on other state variables, such as *loc*. The *map* keyword allows users to integrate as much variables as required at run time without the need to rebuild the framework. We assume that every device/port in the network will have its own values for the QoS parameters.
- The aggregate functions, such as *sum* and *max*, are also encoded as special purpose variables utilizing the arithmetic theory in SMT.
- Since the QoS parameters and aggregate functions are defined as special purpose variables, the constraints are encoded directly into SMT by replacing the QoS parameters defined in the mission invariants with the corresponding special purpose variables. We then write the invariant using the $\boldsymbol{F}$, eventually, LTL operator with constraints on the QoS variables.

Example. The following shows an example for an invariant that requires the traffic from a particular server s to the data center dc not to pass through a port which has a data rate less than the threshold τ. First, we define the DR parameter as a *map* between the OpenFlow queue ID and the data rate of that particular queue. This map will add a variable in our model whose value is dependent on the queue ID. Next, we write an LTL expression with a constraint in terms of the new variable DR and the aggregate function min that is satisfied if the data rate in all the paths between s and dc does not drop below τ.

Mission Invariant: *CanReach(s, dc,* $min(DR) \geq \tau$*)*

map($\boldsymbol{queue}, \boldsymbol{DR}$) {{1, 512}, {2, 1024} . . . }
$P = (loc == s) \wedge (IP\_DST == dc) \wedge \boldsymbol{F}\,[(min(DR) < \tau) \wedge (loc == dc))]$

Since the mission invariants are translated to LTL expression, they are compatible with the generic SDNChecker. We run the invariants against the configuration and retrieve whether they are satisfied or not.

4. IMPLEMENTATION AND EVALUATION

We implemented a tool using C#.NET that automatically reads the complete data plane of an OpenFlow network and provides a GUI for the user to specify the ACD strategy and the mission invariants. This tool uses the Z3 .NET API to compose the proper SMT expressions and generate an SMT file that is fed to the Z3 SMT solver. We ran all experiments on a standard PC with 3.4 GHz Intel Core i7 CPU and 16 GB of RAM.

To evaluate the performance and the scalability of our framework, we measure the time required to solve the generated SMT assertions with respect to multiple parameters, such as the network size, the sizes of flow tables, the number of special purpose variables, and the complexity of the mission invariants. The measures reported in this section are for one snapshot of the network configuration. The verification of an ACD strategy may require repeating this verification multiple times based on the number and the order of the ACD strategy commands.

4.1 Real Network Case Study

In this case study, we evaluate the performance of our framework on the Stanford backbone network, a mid-size enterprise network whose entire configuration has been made public for researchers [16]. The network consists of 14 zones connected to two backbone routers via ten layer-2 switches with a total of 240 hosts and a total of 3840 flow rules.

We generated and verified 100 mission invariants with a quality of service constraints on the number of hops between random pairs of hosts in the network. The source and destination in each pair were selected from different zones. For both the satisfied and not satisfied requirements, the measured time ranged between 6 and 18 seconds with a mean of 14.34 seconds.

4.2 Scalability Evaluation

To evaluate the scalability of our bounded model checking approach, we generated synthetic networks with based on the tree topology, where the leafs are hosts and the inner nodes are OF switches. In all the generated networks, the core switches do not constitute more than 15% of the total nodes in the network. We then built the SMT assertions for each instance and collected the time required by Z3 SMT solver to solve the assertions.

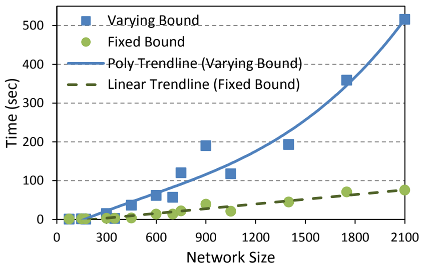

Figure 5: The impact of network size.

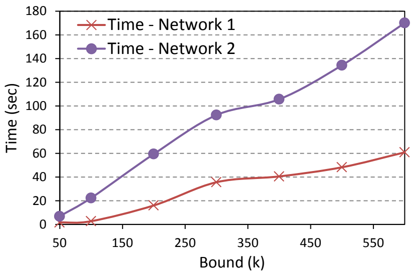

Figure 6: The impact of the bound.

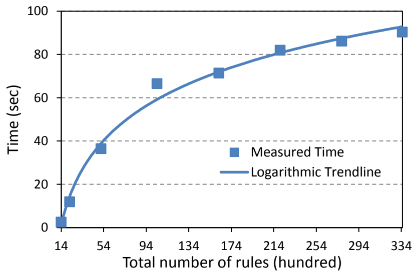

Figure 7: The impact of the table size.

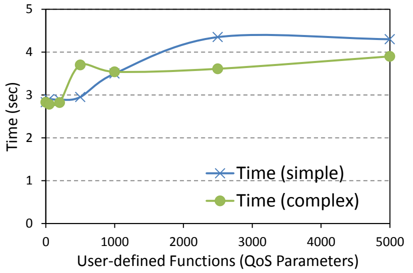

Figure 8: The impact of QoS params.

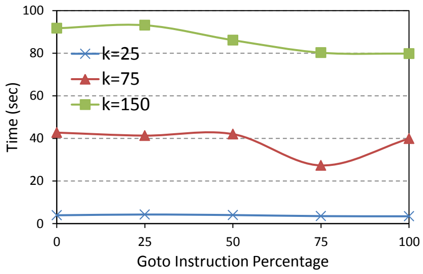

Figure 9: The impact of GoTo.

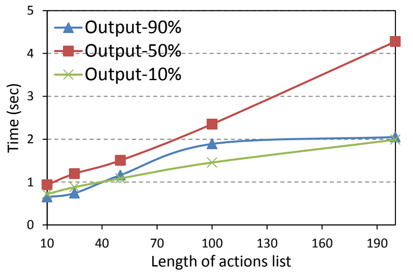

Figure 10: The impact of actions list.

The impact of network size. In this experiment, we study the impact of the overall number of network nodes. We generated a number of networks whose sizes varied from 75 to 2100 nodes. For each of them, we report the average time of verifying 20 random reachability invariants. We conducted the experiment under two different settings of the bound k. In one setting, the bound k varies based on the number of flow tables. In the other, it is set to a constant value regardless of the network size. Each switch in these experiments contains up to two flow tables with an average length of 50 flow rules. Figure 5 shows the results of this experiment. We can see that in the case of fixed bound, the time is linear with respect to the network size. However, in the case of varying bound, the performance is affected by both: the network size and the bound and it is best described by a quadratic polynomial.

The impact of the flow table size. In this experiment, we generated various networks with the same number of nodes (300 nodes), but with varying number of rules per flow table. All the rules have the same number of instructions and length of actions lists. As reported in Figure 7, the total number of rules ranged from 1400 to 33500 rules. The results show that the growth in time stabilizes after a threshold. We believe this behavior is due to the compact representation in Z3 that merges similar expressions together.

The impact of the bound k. The bound determines the number of steps to consider in the bounded model. For each step, a new set of variables and a replica of the transition relation is added. To study the impact of the bound value, we generated two networks: *Network 1* that consists of 300 nodes and *Network 2* that consists of 600 nodes. Each switch in the network has up to 2 tables and an average of 50 flow rules per table. For each network, we ran our experiment multiple times, selecting a bound between 50 and 600. Figure 6 shows that in both networks the time increases linearly with respect to the bound. However, in *Network 2* the increase is faster, which is expected due to its larger size.

The impact of Goto instruction. The *Goto* instructions cause the transitions across flow tables inside a single switch. In Figure 9, the *Goto* instruction percentage represents the ratio of the number of rules that have a *Goto* instruction to the total number of rules. We ran the experiment for different values of the bound k. All the experiments where conducted on a network of 300 nodes with an average of 50 rules per flow table. Despite that more *Goto* instructions implies more transitions, the time is inversely proportional to the *Goto* instructions percentage. We believe this is due to the Action Set encoding shown in Equation 5 that is added to the model only if the rule does not contain a *Goto* instruction. The growth in time follows the same trend for different bounds (linearly decreasing with *Goto* percentage).

The impact of actions list structure. In this experiment, we study the impact of the actions list of *Apply-Actions* instruction. We generated three groups of configurations that have different actions distributions in the actions list. In the first group, 90% of actions are *Output* actions, and the rest are *Set* actions. In the second and the third groups, the *Output* actions constitute 50% and 10% of the list, respectively. All the networks in this experiment consist of 300 nodes with up to two flow tables that have an average of 50 rules per table. Figure 10 shows that the time increases linearly with the number of actions in all the groups. We were expecting that the required time is proportional to the percentage of *Output* actions. However, interestingly, it was the highest when 50% of actions are *Output*. Since the number of output ports is relatively small (<10), many of the consecutive *Output* actions will be similar, which results in similar expressions that can be merged together. When the *Output* and *Set* actions are alternating, every *Output* action will result in a completely different transition that increases the time requirements.

The impact of QoS parameters. We conducted an experiment to study the impact of the number of QoS parameters, which are encoded as special purpose variables. We ran our framework against a network of 300 nodes with a varying number of QoS parameters that ranged from zero to 5000. Moreover, we defined two types of invariants, namely *simple* and *complex*. In the *simple* invariant we used the *max* aggregate function, while we used the *sum* function in the *complex*. Figure 8 reports the time for both types. We can see that the number of QoS parameters has no effect on the time requirements.

5. RELATED WORK

The verification of network invariants has attracted a significant body of research in both enterprise and software defined networks. FlowChecker [2] encodes the OpenFlow flow tables using Binary Decision Diagrams (BDD) to verify security properties. VeriFlow [9] proposes to slice the OF network into equivalence classes to efficiently check for reachability violations. FLOVER [15] is a model checker that checks the OpenFlow configuration for security violations using Yices SMT solver. NetPlumber [7] is a real time policy checking tool that utilizes a dependency graph between flow entries to incrementally check for loops, black holes, and reachability properties. FlowGuard [6] examines dynamic flow updates to detect firewall policy violations and it provides violation resolution approaches. Although these works can check the compliance of OpenFlow network updates with specific invariants, Their applications are limited to reachability or related analyses in [8, 7, 15] and to firewall policy verification in [6].

ConfigChecker [3] and Anteater [12] are two model checking frameworks that allow the specification of system properties using temporal logics. They both employ similar configuration abstraction as our framework, but they are targeting traditional networks configuration and they use binary analysis platforms (BDD and SAT), which make it hard to verify properties with arithmetic constraints. SecGuru [4] is another tool that is based on the bit-vectors theory in Z3 solver for checking network invariants.

While these works provide multiple platforms to verify the end-to-end reachability in enterprise and software defined networks, they do not focus on ACD verification. Even the real-time verification tools, they verify invariants against one update at a time and they do not consider multi-step strategies. They also have very limited support for QoS requirements with the exception of NetPlumber that provides path length constraints. We provide the ability to verify complete ACD strategies and we utilize the arithmetic theory in SMT to verify quantitative QoS invariants.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented the first step towards ACD strategies verification using SMT-based bounded model checking approach for OpenFlow-based SDNs. We presented a formal model to encode ACD strategies along with the mission invariants as a set of assertions that can be checked for satisfiability using state of the art SMT solvers. The verification reveals any shortcoming in the network configuration that can render the ACD strategies unsuccessful. We believe that the time requirement is moderate considering the network size. In the next steps, we will extend our verification to verify the effectiveness of ACD in addition to the enforceability. We will also employ hierarchical verification techniques by dividing the network and invariants into groups and investigate optimization and expression simplification techniques to enhance the performance of our framework.

7. REFERENCES

[1] Active cyber defense (acd). http://www.darpa.mil/program/active-cyber-defense.

[2] E. Al-Shaer and S. Al-Haj. Flowchecker: Configuration analysis and verification of federated openflow infrastructures. In *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*, pages 37–44. ACM, 2010.

[3] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. Elbadawi. Network configuration in a box: Towards end-to-end verification of network reachability and security. In *ICNP*, pages 123–132, 2009.

[4] N. Bjorner and K. Jayaraman. Network verification: Calculus and solvers. In *Science and Technology Conference (Modern Networking Technologies)(MoNeTeC), 2014 International*, pages 1–4. IEEE, 2014.

[5] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[6] H. Hu, W. Han, G.-J. Ahn, and Z. Zhao. Flowguard: Building robust firewalls for software-defined networks. 2014.

[7] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *NSDI*, pages 99–111, 2013.

[8] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *NSDI*, pages 113–126, 2012.

[9] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. Veriflow: Verifying network-wide invariants in real time. *ACM SIGCOMM Computer Communication Review*, 42(4):467–472, 2012.

[10] I. lachow. Active cyber defense a framework for policymakers. 2013.

[11] R. M. Lee. The sliding scale of cyber security. 2015.

[12] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King. Debugging the data plane with anteater. *ACM SIGCOMM Computer Communication Review*, 41(4):290–301, 2011.

[13] F. Merz, S. Falke, and C. Sinz. Llbmc: Bounded model checking of c and c++ programs using a compiler ir. In *Verified Software: Theories, Tools, Experiments*, pages 146–161. Springer, 2012.

[14] ONF. Openflow switch specification, version 1.4.0 (wire protocol 0x05). https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf, October 2013.

[15] S. Son, S. Shin, V. Yegneswaran, P. Porras, and G. Gu. Model checking invariant security properties in openflow. In *Communications (ICC), 2013 IEEE International Conference on*, pages 1974–1979, June 2013.

[16] J. H. Zeng and P. Kazemian. Mini-Stanford Backbone). https://reproducingnetworkresearch.wordpress.com/2012/07/11/atpg/.