Verifying the Enforcement and Effectiveness of Network Lateral Movement Resistance Techniques

Mohammed Noraden Alsaleh, Ehab Al-Shaer and Qi Duan
Department of Software and Information Systems,
University of North Carolina at Charlotte, Charlotte, NC, U.S.A.

Keywords: Resistance, Cyber Attacks, Resilience, Configuration, Model Checking.

Abstract: As the sophistication of cyber-attacks is ever increasing, cyber breaches become inevitable and their consequences are often highly damaging. Isolation and diversity are key techniques of cyber resilience for creating built-in resistance in cyber networks against the lateral movement of multi-step Advanced Persistent Threats (APTs) and epidemic attacks. However, the key unaddressed challenges are (1) how to ensure that specific isolation and diversity configurations are sufficient to prevent the lateral movement of attacks and (2) how to verify that such configurations are enforced safely despite the complex inter-dependency between cyber components. In this paper, we address these challenges by developing formal models and properties to verify the effectiveness and enforceability of proactive cyber resistance techniques. We present a bounded model checking approach based on satisfiability Modulo theories (SMT) for OpenFlow software defined networks (SDNs). We verify that given resistance techniques are enforced in a way that does not violate the cyber mission requirements and we evaluate the configuration resistance based on user-defined resistance properties.

1 INTRODUCTION

The recent incidents of data breaches, such as the attacks on SnapChat, Yahoo, the SWIFT system, and others reported in the recent Verizon Data Breach Investigation Reports (Verizon, 2016; Verizon, 2017) emphasize the important fact that cyber breaches have become inevitable. This requires high assurance of resilient proactive defense techniques to increase the resistance against the propagation of cyber attacks and significantly limit their impacts.

Cyber resilience is the capability that allows the cyber to defend against active attacks (*i.e.,* when the attackers are partially or completely successful) (Goldman, 2010; Melin et al., 2013). Although different definitions exist in literature for cyber resilience, they are all centered around three key strategies: resistance, deterrence, and recovery. In this work, we focus on the cyber resistance and we define it as "*the capability of the network configuration to increase the required time, effort, skill set, resources and knowledge for active attackers in order to achieve their goal.*" Specifically, we study two key techniques of cyber resistance, namely isolation and diversity, that can potentially impede the lateral movement of cyber attacks. Lateral movement is a key step in the kill-chains of cyber attacks where attackers compromise the most vulnerable systems and move through the network to reach their ultimate targets. Isolation aims at segregating the network components such that critical assets are less susceptible to threats from vulnerable sources (Rahman and Al-Shaer, 2013; Goldman et al., 2011; Sahinoglu, 2006). Thus, ensuring that critical portions of the network continue to function even if others are compromised. Diversity aims at changing the attack surface as the attacker progresses through the attack kill-chain by using heterogeneous variants of software and hardware components (Larsen et al., 2015; Miu et al., 2005; Yang et al., 2008), in an effort to increase the time and resources that she requires to identify and exploit unforeseen cyber vulnerabilities.

Network resistance is not only about whether an attacker can reach and exploit a particular asset, but it is more about the effort she has to spend in order to accomplish that. Resistance techniques are implemented to maximize this effort without violating the network mission. However, in order to ensure cyber resistance, two properties must be satisfied: (1) the desired isolation and diversity configurations are implemented correctly and (2) they are effective against the lateral movement of specific cyber attacks. Through

Alsaleh, M., Al-Shaer, E. and Duan, Q.
Verifying the Enforcement and Effectiveness of Network Lateral Movement Resistance Techniques.
DOI: 10.5220/0006868902460257
In *Proceedings of the 15th International Joint Conference on e-Business and Telecommunications (ICETE 2018) - Volume 2: SECRYPT*, pages 246-257
ISBN: 978-989-758-319-3

the rest of this paper, we refer to these two properties by the *Enforcement* and *Effectiveness*, respectively. We believe that violating the enforcement and effectiveness properties can impose substantial risks on both the security and the performance of the network. On the one hand, resistance techniques are realized through one or more layers of network elements, including the configuration of hosts, the applications hosted by them, and the network infrastructure, which performs a variety of traffic transformations, such as filtering, inspection, tunneling, encryption, and authentication. This makes ensuring correct and consistent resistance configuration a challenge, especially in complex, large-scale, and densely connected cyber networks. A single mistake may jeopardize the mission of the network by granting unauthorized access to critical network assets, denying legitimate access to network services, exposing confidential information by inappropriately encrypting or decrypting network traffic, misplacing software components which renders them nonfunctional, etc. On the other hand, violating the effectiveness property implies the presence of security configuration weaknesses and mandates more resistive configuration. Hence, techniques and tools are required to automatically verify the enforcement and effectiveness of resistance configuration.

In this paper, we present a model checking framework to verify the enforcement and effectiveness of resistance configuration. Specifically, our contribution in this paper is twofold. First, we develop a scalable Linear Temporal Logic (LTL)-based bounded model checker that models the entire data plane of OpenFlow-based software defined networks along with the hosts configuration and verifies its resistance against cyber attacks. To model the lateral movement, our model checker encodes the indirect reachability through stepping stones or intermediate hubs. To the best of our knowledge, this is the first network configuration model checking framework that goes beyond end-to-end reachability and provides the ability to define and verify properties that span an entire attack path with all its intermediate hubs. In addition, our model can be extended to integrate different types of security middle-boxes and host configurations to accommodate variant isolation and diversity strategies. Second, we model two of the network resistance techniques: isolation and diversity, and verify their enforcement and effectiveness. We provide a high level language to facilitate the specification of resistance techniques and demonstrate the capabilities of our model checker. Our model checker is not limited to isolation and diversity and any other resistance technique can be verified as long as it can be represented in single or multiple LTL expressions.

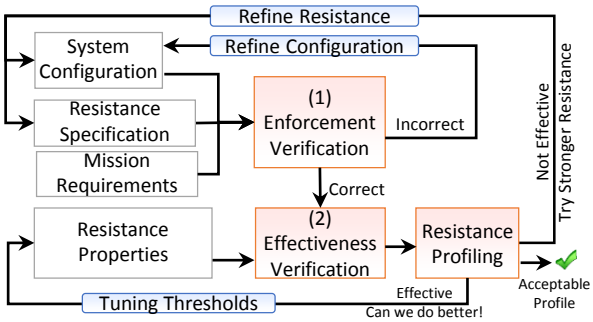

Figure 1: Design Philosophy.

The scope of this work encompasses the network data plane and host configuration abstracted by the software variants running on top of it. Hence, we do not model the complete application-layer configurations such as application level access control. We realize that some lateral movement techniques require higher level actions, such as privilege escalation, but it also depends on the weaknesses that might exist in the network data plane and the basic host configuration. For example, to exploit a remote access tool, the attacker's machine should be able to communicate with the remote victim, which is controlled by the isolation countermeasures in the network. Our objective in this work is to discover and address such weaknesses.

We designed our framework to support the two-phase verification that is depicted in Figure 1. In the first phase, we verify the enforcement of the resistance specifications, which is a composition of isolation and diversity configurations. If the enforcement property is not satisfied, a counter-example will be generated to refine the hosts and network data plane configuration. Otherwise, the model will further verify the effectiveness property with respect to user-defined resistance properties. If this is not satisfied, a counter-example will be generated to show the weaknesses of the resistance techniques and refine them accordingly. Multiple resistance properties can be composed to measure the effectiveness of cyber resistance against different attack capabilities. Thus, resistance configuration can be iteratively tuned until a satisfactory resistance profile is found. We implemented our bounded model checker using Satisfiability Modulo Theories (SMT). The temporal relation between consecutive packet transformations, performed by OpenFlow switches and security middle-boxes in the network, is crucial for the enforcement and effectiveness of the resistance configurations. Model checking can efficiently capture such temporal dependency by exploring all relevant paths. Moreover, bounded model checking does not require exponential space or manual manipulation of variables as the case in BDD-

based verification (Clarke et al., 2001). Using SMT allows our framework to support advanced decision procedures for linear and non-linear arithmetic and difference logic, which are needed to consider mission requirements and resistance thresholds.

The rest of the paper is organized as follows. In Section 2, we present our technical details for a resistance-oriented bounded model checker. In Sections 3 and 4, we present the models required to verify the resistance *enforcement* and *effectiveness* properties. In Section 5, we report the performance evaluation. Finally, the related work and conclusions are presented in Sections 6 and 7, respectively.

2 OPENFLOW BOUNDED MODEL CHECKING

An OpenFlow-based SDN consists of a set of OpenFlow-enabled switches controlled by a central controller. The controller provides interfaces to deploy network management applications and dynamically updates the flow tables in the 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 cookies. We focus on the components that directly affect the packet processing, which include the match fields, the priority, and the instructions set. The match fields are filters that determine the set of flows that match the flow rule and 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 rule. The instructions set contains one or more of the instructions {*Apply-Actions, Clear-Actions, Write-Actions*, or *Goto-Table*} according to the OpenFlow specification document 1.4 (ONF, 2013). Thus, a rule in an OpenFlow table is a set of conditions on the OpenFlow match fields and a set of instructions that are performed on flows matching the conditions.

In this section, we show how we model the network data plane as a transition system for bounded model checking. We also present the unique features of our model checker that makes it suitable for resistance verification. The complete formalization and low-level details of building the exact SMT assertions based on the OpenFlow configuration are available at (Alsaleh and Al-Shaer, 2016).

2.1 Transition System

The state of the network is determined by the unique flows (represented by the match fields) that can be transferred through the network and their possible locations. The flows are represented by the flow match fields. The state in our model is encoded by the following characteristic function:

$$\sigma : loc \times tbl \times \mathbb{F} \times \mathbb{A} \times \mathbb{L} \times st \times stns \rightarrow \{true, false\}$$

The variables *loc* and *tbl* encode the unique switch ID and the index of the flow table within the OpenFlow switch, respectively. The set $\mathbb{F}$ includes the match fields' variables. The Action Set variables $\mathbb{A}$ contains a variable for each action that is carried between the flow tables during the pipeline processing inside an OpenFlow switch. This set of variables are required to capture the flow transformations inside a single switch. $\mathbb{L}$ is an extensible set of special purpose variables, referred to by *app-specific* variables, that are integrated in the model only for particular applications. The variables *st* and *stns* are defined to capture the indirect reachability between network hosts, where *st* represents the current step within a multi-step path and *stns* is a list of the visited stepping stones in the path.

Transitions between states are realized through packet transformation and forwarding across OpenFlow switches or across flow tables in the same switch. We add a transition to the global transition relation 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). According to OpenFlow specification, multiple transitions may be associated with the same flow entry.

2.2 Resistance Verification Model

We emphasize here the features provided in our model that are required for the resistance verification.

Modeling Stepping Stones. Using stepping stones or intermediate attack hubs is a technique that is widely used in attacks to bridge the connection between the attacker and her targets (Nicol and Mallapura, 2014; Shullich et al., 2011). Proactive resistance techniques can break the chain of stepping stones by deploying multiple heterogeneous resistance countermeasures in potential attack paths.

Instead of discarding the packets once they reach their direct destinations, we transform and forward them back to the network. Incorporating the variables *st* and *stns* in the state characteristic function allows us to track potential attack steps and the list of stepping stones that has been visited in an attack path. We modify the hosts behavior so that they will echo the packets back to the network once they are received and the list of previously visited stepping stones is used to prevent loops.

Modeling Middle-boxes. OpenFlow switches support the basic forwarding and filtering functions, but they do not support other important functions, such as encryption, inspection, or authentication. This limitation is addressed by integrating legacy middle-boxes, such as VPN gateways, proxies, and intrusion detection and prevention systems and configuring the OpenFLow switches to forward the traffic to the appropriate middle-boxes according to high level policy (Qazi et al., 2013). In our work, these middle-boxes provide the expected isolation between the network assets and threat sources. Hence, it is essential to model their complete configuration along with the OpenFlow data plane. We provide a unified model that captures both the data plane of the OpenFlow switches and the complete configuration of selected middle-boxes. We model the configuration of a middle-box as an ordered rule set. Each rule consists of a set of conditions on the match fields and an action that belongs to a predefined set that might include {*stateful-inspect, deep-inspect, AH, ESP, tunnel*}. This set may be extended based on the supported middle-boxes in a network.

App-specific Variables. These variables are integrated in the model for specific purposes such as defining QoS parameters or aggregate functions at run time without the need to recompile the framework. We provide special keywords, *def* and *map*, to define new app-specific variables and use them to verify resistance properties. The *def* declares a variable that will be encoded as an integer. The *map* is used to assign values to the new variable based on the other variables of the same state. For example, the app-specific variable shown below defines a new variable called *Thr* whose value is mapped from the *loc* variable. In this example, the value of the new variable is 50 if *loc=1*, 30 if *loc=2*, and so on.

$$def\ Thr = map(loc)\{\{1,\ 50\},\ \{2,\ 30\},\ \dots\}$$

All the state variables including the location, match fields ($\mathbb{F}$), action set ($\mathbb{A}$), and other app-specific variables ($\mathbb{L}$) can be used in the *map* construct. Once defined, users can use the new variables directly in writing the mission or resistance properties.

2.3 Specification Language

We use the standard LTL specification language as a generic means to specify various properties. Any property that can be expressed using LTL can be verified using our framework. However, as will be shown later, we provide high level specification language to describe the resistance requirements and properties and we translate them automatically to LTL and, later, to SMT constraints. A property in LTL can contain any of the standard LTL temporal connectives: *next* (**X**), *eventually* (**F**), *global* (**G**), *until* (**U**), and *release* (**R**). In addition, multiple temporal constraints can be combined using the logical operators *AND* ($\wedge$), *OR* ($\vee$), and *NOT* ($\neg$).

Mission Requirements Specification Syntax	
QoS Param ρ ::=	$BW \mid D\_RATE \mid DELAY \mid \dots$
Operator $\bowtie$::=	$> \mid < \mid \geq \mid \leq \mid ==$
predicate Φ ::=	$\rho \mid \mathbb{Z}^+ \mid max(\rho) \mid min(\rho) \mid sum(\rho) \mid avg(\rho)$
QoSCond Ψ ::=	$\Phi \bowtie \Phi \mid \Psi \wedge \Psi \mid \Psi \vee \Psi$
Location $\mathcal{L}$ ::=	$<$ (ip:port) $>$
Sec. Comp. $\mathcal{M}$ ::=	$C \mid I \mid A \mid *$
Requirement R ::=	$CanReach(\mathcal{L},\ \mathcal{L},\ \Psi) \mid Protect(\mathcal{L}, \mathcal{M}) \mid R \vee R \mid R \wedge R$
Resistance Techniques Specification Syntax	
Pattern $\mathcal{P}$ ::=	$\mathcal{E}_i \mid \mathcal{E}_d$
Isolation $\mathcal{E}_i$ ::=	$I_{cm} \mid \mathcal{E}_i \wedge \mathcal{E}_i \mid \mathcal{E}_i \vee \mathcal{E}_i$
Diversity $\mathcal{E}_d$ ::=	$\mathcal{D} \mid \mathcal{E}_d \wedge \mathcal{E}_d \mid \mathcal{E}_d \vee \mathcal{E}_d$
CMeasure I_{cm} ::=	$FW \mid VPN \mid PROXY \mid IDS \mid \dots$
Div. Param $\mathcal{D}$ ::=	$OS \mid VERSION \mid SERVICE \mid \dots$
Res. Tech. $\mathcal{S}$ ::=	$ResistSpec(\mathcal{L}, \mathcal{L}, \mathbb{Z}^+, \mathcal{P})$
Resistance Properties Specification Syntax	
Vector $\mathcal{V}$ ::=	$<$ software/hardware variants $>$
Property $\mathcal{P}_{res}$ ::=	$CanResist(\{\mathcal{L}\}, \{\mathcal{V}\}, \{\mathcal{M}\}, \{I_{cm}\})$

Figure 2: Specification Language Syntax. $\mathbb{Z}^+$ is the set of positive integers.

3 RESISTANCE ENFORCEMENT VERIFICATION

The network data plane satisfies the resistance enforcement property if it meets two conditions: (1) it correctly implements desired resistance techniques (i.e., isolation and diversity patterns) and (2) implementing the desired techniques does not introduce violations to the network mission requirements. In this section, we formally define the resistance techniques and mission requirements and show how we verify them using our bounded model checker.

3.1 Resistance Techniques and Network Mission Specification

Resistance techniques enforce specific attack resistance and deterrence measures, such as isolation,

through inspection or encryption, and diversifying the network parameters in all paths from threat sources to system assets. In this work, we consider two major resistance techniques, isolation and diversity, and we define them formally as follows.

Resistance Techniques Specification. We define a resistance technique as the tuple $(\mathcal{S}, \mathcal{D}, \mathcal{P}, \mathcal{F})$, where:

- $\mathcal{S}$ is a set of locations in the network that represent potential insider or outsider threat sources.
- $\mathcal{D}$ is a set of locations inside the network that represent critical assets.
- $\mathcal{P}$ is the isolation or diversity pattern. For isolation techniques, $\mathcal{P}$ is expressed as a logical expression in terms of the countermeasures deployed in the network, such as VPN gateways, proxies, and intrusion detection/prevention systems. For diversity techniques, $\mathcal{P}$ is expressed in terms of the hosts attributes, such as vendor, platform, applications, and their version.
- $\mathcal{F}$ is the frequency that determines in how many steps of the attack steps the isolation pattern, $\mathcal{P}$, should hold in any attack path.

Intuitively, an isolation technique specified based on this definition means that the specified pattern should be met in all direct and indirect paths between the specified threat sources and critical assets for at least $\mathcal{F}$ number of times. Similarly, a diversity technique determines how many times, in an attack path, the attacker should encounter hosts that have different values for the diversity parameters specified in the the pattern $\mathcal{P}$. The frequency $\mathcal{F}$ makes our definition suitable to specify existing diversity requirements and metrics, such as the d2-diversity metric (Zhang et al., 2016), which typically specify the least number of distinct components that must be encountered in potential attack paths.

To simplify the specification of the resistance requirements, we provide a unified language to express the resistance techniques in terms of the construct *ResistSpec*, which can define both the diversity and isolation techniques. In Figure 2, we show the syntax of the *ResistSpec* construct, which takes four arguments. The first two represent the set of threat sources and critical destinations. The third argument represents the frequency. The fourth argument is a Boolean expression for the isolation or the diversity pattern. For example, the isolation technique that *"if a path is possible from students labs to the records servers, at least two rounds of inspection or header authentication (AH) must exist"* can be represented as:

$$\textit{ResistSpec(Labs, Records, 2, (inspect} \vee \textit{AH))}$$

Mission Requirements Specification. The mission of cyber networks is not limited to binary configuration decisions that specify whether hosts are connected to each other or not. In addition to basic connectivity, we identify two types of requirements that may be crucial for successful mission: (1) to guarantee particular performance and QoS requirements and (2) to preserve the confidentiality, availability, and the integrity of particular critical assets in the system. The performance and QoS requirements are normally described in the Service Level Agreement (SLA) and translated to the QoS parameters of the networking infrastructure in terms of bandwidth, delays, jitter, etc. The network data plane affects these parameters as it determines which switches, ports, and queues the traffic passes through, which may have different performance metrics. Therefore, end-to-end analysis is required in order to ensure that the global network configuration can support the required QoS.

To formally specify the mission requirements, we provide the language shown in Figure 2. According to this syntax, a mission requirement, *R*, can be specified in terms of the two constructs *CanReach* and *Protect*. *CanReach* is used to specify the QoS requirements between a pair of locations in the network. The *QoS Constraints* are composed of a set of conditions on the aggregate values of the network infrastructure 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 each possible path between the specified locations. For example, let the set *C* be a set of clients in an organization that are using a particular service *s*. And let *P* be a pool of servers that are running that service. Let us assume that the mission of this organization requires that: (1) *"each client should be able to reach at least one server with a data rate greater than or equal to* τ_{dr}*"* and (2) *"the response from any server to any client should not be delayed more that* τ_{dl}*"*. These requirements can be represented using our language as follows:

$$\bigwedge_{c \in C} \bigvee_{p \in P} \textit{CanReach}(c, p : s, \textit{min}(D_{rate}) \geq \tau_{dr})$$
$$\bigwedge_{p \in P} \bigwedge_{c \in C} \textit{CanReach}(p : s, c, \textit{sum}(\textit{DELAY}) \leq \tau_{dl})$$

The construct *Protect* can be used to mark the critical assets in the network, those that if compromised, the system mission will be in jeopardy. We further provide the option to specify the fine-grain security component (i.e., *Confidentiality, Integrity, Availability*) of particular locations in the network, that is critical to the mission.

3.2 Verifying Resistance Specifications

To verify the correct implementation of resistance techniques, we translate their specification to LTL properties and verify them using our bounded model checker. This cannot be possible without considering the indirect reachability. We show in the following how the resistance techniques specification is translated to LTL expressions.

- Isolation Countermeasures. The isolation techniques specification depends on the set of countermeasures that may be deployed in the network. We define an app-specific variable that associates each location in the network to its resistance functionality (i.e., whether it is a filter, a VPN gateway, a packet inspection box, a proxy, etc). The values of those variables are determined based on the *loc* variables using the *map* construct. Then, we define a marking variable for each type of countermeasures. The marking variables capture which countermeasures a packet passes through between one particular source to a destination. At each transition, the value of each marking variable is set to true if the flow passes through the corresponding countermeasure. Otherwise, its value is copied from the previous state as is.
- Diversity Attributes. For each diversity attribute that is part of the resistance technique, we define a new variable in the model and we use the *map* construct to assign its value based on the locations in the network.
- Resistance Pattern. At this point, we have defined the isolation marking variables in addition to the diversity variables. The resistance pattern can be expressed as a logical expression in terms of the marking and diversity variables.
- Frequency. We define an app-specific variable ($step$) that works as a counter. The value of this counter is incremented only if the resistance pattern is satisfied at the end of each attack step.
- Finally, the isolation specification is expressed as an LTL property that expresses a counter example for the desired isolation specification (i.e., it verifies if the attacker can indirectly reach the destination with inappropriate frequency of the specified resistance pattern). The absence of counter examples proves the correct deployment of the specified resistance techniques.

In Table 1, we show examples for isolation and diversity techniques specifications along with the translated LTL. In the isolation example, we define the variable *cm* that maps each location in the network to its type (i.e., host, filter, IDS, etc.). Then we define the marking variables cm_ftr, cm_vpn, and cm_dnp. We define these variables using *if-then-else* statement (? :). For example, the statement $(cm\_vpn = (cm == vpn)?\, true : cm\_vpn^{-1})$ will set the value of cm_vpn to true if the system reaches a state at which a packet is at a location whose type is vpn. Otherwise, the previous value is carried along. We use the notation v^{-1} to represent the previous value of the variable v. The same goes for the other marking variables. Then, we defined the variable *iso*, which counts the steps at which the specified isolation pattern is satisfied, based on the marking variables. Note that this counter is incremented at each stepping stone (i.e., when $step \neq step^{-1}$). The final LTL expression P_{iso} defines a counter example for the desired specification utilizing the state's basic and app-specific variables. We follow the same procedure for the diversity example except that the counter div is incremented only if the current stepping stone diversity attributes are different from the previous one. The expression P_{div} represent a counter example for the desired diversity specification.

Table 1: Resistance Techniques Examples.

Isolation Technique Specification
ResistSpec(dmz, c_db, 2,$(vpn \vee (filter \wedge d\text{-}inspect))$
.....
$def\ cm =$ map (loc) {{1, filter}, {2, vpn}, ...}
$def\ cm\_ftr = (cm == filter)?true : cm\_ftr^{-1}$
$def\ cm\_vpn = (cm == vpn)?true : cm\_vpn^{-1}$
$def\ cm\_dnp = (cm == d\text{-}inspect)?true : cm\_dnp^{-1}$
$def\ iso = ((step \neq step^{-1}) \wedge (cm\_vpn \vee (cm\_ftr \wedge cm\_dnp)))\ ?iso^{-1} + 1 : iso^{-1}$
$P_{iso} = (loc == dmz) \wedge F\,[(loc == c\_db) \wedge \neg(iso > 2)]$
Diversity fTechnique Specification
ResistSpec(i_net, c_db, 3,os)
.....
$def\ os =$ map (loc) {{1, win7}, ...}
$def\ div = ((step \neq step^{-1}) \wedge (os \neq os^{-1}))\ ?\ div^{-1} + 1 : div^{-1}$
$P_{div} = (loc == i\_net) \wedge F\,[(loc == c\_db) \wedge \neg(div \geq 3))]$

3.3 Verifying Reachability and QoS Requirements

In this section, we show the steps we follow to translate the mission requirements to LTL properties and verify them using our model checker. The translation of the *Protect* is deferred to the following section because it depends on the attack description.

The reason that we need to verify the network mission requirements is that misconfigurations in the

Table 2: Examples of *CanReach* Requirements.

Property	LTL Property Expression
1. CanReach(s, d, _)	$P_1 = (loc == s) \wedge (IP\_SRC == s) \wedge (IP\_DST == d) \wedge F[(loc == d) \wedge (IP\_DST == d)]$
2. CanReach(s, dc, min(DR) $\geq \tau$*)*	*def* $DR =$ map (*queue*) $\{\{1, 512\}, \{2, 1024\} \dots\}$ $P_2 = (loc == s) \wedge (IP\_DST == dc) \wedge F\,[(min(DR) < \tau) \wedge (loc == dc))]$

isolation and diversity techniques may have negative impacts on the network mission requirements. Incorrect placement of software and hardware variants may cause reachability and/or performance violations because some variants may require specific configuration in the network infrastructure level to make them accessible or they may not be capable of providing the required performance. Similarly, if an isolation technique is implemented incorrectly, it may introduce reachability, performance, or security violations by blocking some legit traffic flow, forwarding traffic flows through inappropriate ports or queues that are incapable of providing the required QoS, or granting undesired access to critical assets in the network. As a part of verifying the enforcement of resistance techniques, we ensure that all network mission requirements are satisfied under the deployed techniques.

The *CanReach* construct can specify basic reachability requirements without any constraints on the QoS as appears in the first example of Table 2. The corresponding LTL expression that is shown in the table is satisfied if packets that are initially located at location s will be located at the destination d at any future state.

The *CanReach* requirements can also specify constraints on the QoS between sources and destinations in the network. In order for these requirements to be satisfied, the source should be able to reach the destination and the QoS, in terms of the QoS parameters, should meet particular thresholds. In this case, we follow the following steps:

- The QoS parameters such as bandwidth, data rate, and delay are defined as app-specific variables and encoded using the *map* construct. We assume that every device/port in the network will have a value for each quality of service parameter.
- If any aggregate functions are used, they will be encoded as app-specific variables and their values are computed accumulatively at each transition. The use of SMT allows us to compute their commutative values utilizing the basic arithmetic and conditional operators.
- Since the QoS parameters and the aggregate values are defined as app-specific variables, the QoS constraints are encoded directly into SMT by replacing the QoS parameters defined in the constraints with the corresponding app-specific variables.

The second property in Table 2 shows an example of *CanReach* requirements with QoS constraints. In this example, the mission requires that the traffic from a particular server s to the data center dc does not pass through a port which has a data rate less than a particular threshold τ. Such requirement is crucial in several solutions that have proposed techniques for task scheduling in MapReduce architectures (Qin et al., 2014). Since this requirement depends only on one QoS parameter, data rate, we define the *DR* parameter and assign it is value as a *map* between the OpenFlow queue ID and the data rate of that particular queue. Next, we express the requirement as an LTL expression that contains 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 the source and destination does not drop below the given threshold.

4 RESISTANCE EFFECTIVENESS VERIFICATION

The effectiveness of resistance techniques is evaluated with respect to specific attack propagation models. We model in this section the resistance properties, which define attacks with certain capabilities and we show how to verify the effectiveness of the resistance configuration against them.

4.1 Network Resistance Properties

We study the resistance of the network configuration within the context of multi-step attacks, in which the attacker compromises multiple hosts (stepping stones) sequentially in order to reach his target. This behavior is commonly used in two major classes of cyber attacks: worms and Advanced Persistent Threats (APTs). We characterize cyber attacks based on the following attributes:

- Attack Target. The attack target is reaching and compromising critical assets in the network in order to gather sensitive information or gain access to private computer systems.
- Impact. The impact specifies which security component(s) (i.e., Confidentiality, Integrity, and/or

Availability) the attack is targeting.

- Attack Vector. The attacker in our model possesses some capabilities to exploit specific software or hardware components in order to propagate and compromise network resources.
- Propagation. If the attacker can reach a vulnerable host whose vulnerabilities belong to the attack vector, that host becomes a stepping stone and the attack can automatically propagate to its neighbors.
- Countermeasures. Each attack is associated with a set of countermeasures that are capable of preventing it. An attack is successful if there is at least one attack path in which the attacker reaches his target without encountering any of the specified countermeasures.

The resistance properties evaluate the ability of the implemented resistance techniques to resist a given attack with certain capabilities. Hence, a resistance property describes an attack based on the attack model discussed above as follows:

Resistance Properties We define a resistance property as the tuple $(O, \mathcal{V}, \mathcal{M}, \mathcal{C})$, where:

- O is a set of untrusted locations that are assumed to be compromised and they constitute the potential origins of the attack.
- $\mathcal{V}$ is the attack vector, a set of software/hardware variants that the attacker is capable of compromising.
- $\mathcal{M} \subseteq \{C, I, A\}$ is the impact vector.
- $\mathcal{C}$ is a set of countermeasures that can prevent and/or divert the attack if encountered in an attack path.

The network configuration satisfies a resistance property if it can prevent the specified attack from propagating and compromising its target. We consider any critical asset in the network, that is defined as part of the mission requirements using the *Protect* construct, as a potential target of the attack.

Based on this definition, we can see that there is a direct relation between the implemented isolation and diversity patterns and the satisfaction of the resistance properties. The selection of software and hardware variants in each location in the network through diversity techniques determines whether or not the attacker encounters variants that she is capable of compromising (i.e., those variants that belong to the attack vector $\mathcal{V}$). On the other hand, the optimal deployment of isolation techniques can guarantee that attackers encounter appropriate attack countermeasures ((i.e., those that belong to set $\mathcal{C}$) in any potential path to the critical assets of the network.

To specify resistance properties, we provide the high level construct, *CanResist*, whose syntax is shown in Figure 2. For example, let us assume that preserving the integrity and confidentiality of two database servers, db_1 and db_2, is specified as part of the mission requirements using the requirement $Protect(\{db_1, db_2\}, \{I, C\})$, then the following represent a valid resistance property:

CanResist({dmz, i_net}, {RH-6.0}, {I, C}, ids)

This property specifies that threat sources may exist in the Internet (*i_net*) and the *dmz*, the attacker is capable of exploiting Red Hat 6.0 (*RH-6.0*), the attacker is targeting the integrity and confidentiality of its victims, and IDS is an effective countermeasure against this attack profile.

4.2 Verifying Resistance Properties

In this section, we demonstrate how we translate the resistance properties to LTL and verify them using our bounded model checker. Modeling indirect reachability and potential attack propagation makes the verification of such properties straight forward. Resistance properties are specified using the construct $CanResist(O, \mathcal{V}, \mathcal{M}, \mathcal{C})$. To verify that the resistance configuration can prevent the specified attack from reaching the critical locations in the network, we identify the set of critical locations defined using the *Protect* construct as part of the mission requirements. Specifically, we follow the following procedure to translate resistance properties to LTL expressions.

- First, we define an app-specific variable, *cm*, and assign its value using the *map* construct to identify the type of each location in the network (i.e., whether it is a host, filter, ids, etc.).
- Second, we define another app-specific variable, *v*, that identifies the software/hardware profile of each location in the network (i.e., the installed software and hardware variants). Similar to the *cm* variables, it is mapped to appropriate values using the *map* construct based on the *loc* variable.
- Third, We compile a set of hosts in the network ($\mathbb{W}$) that are declared critical for the network mission and they are potential targets of the specified attack. Recall that the *Protect* construct not only specifies the location, but also the critical security component (i.e., confidentiality, integrity, and/or availability). If the security component specified in the *Protect* construct is the wild card ($*$) or it belongs to the impact vector of the specified attack, we add the location to the set of critical hosts $\mathbb{W}$.

Formally,

$$\forall\, Protect(l,m) : (l \in \mathbb{W}) \iff (m = *) \vee (m \in \mathcal{M})$$

- Finally, we compile the LTL expression that searches for violations of the desired resistance property utilizing the ***Until*** temporal operator as follows:

$$\bigvee_{o \in O} \left[\begin{array}{l} (loc = o) \wedge \\ \left(\begin{array}{l} ((cm \notin \mathcal{C}) \wedge ((cm = host) \rightarrow v \in \mathcal{V})) \\ U(loc \in \mathbb{W}) \end{array} \right) \end{array} \right]$$

This LTL property is satisfied if there is a path from any location that belongs to the untrusted set (O) to any of the critical locations ($\mathbb{W}$), such that all the intermediate nodes are either vulnerable hosts or middle-boxes that do not belong to the set of countermeasures ($\mathcal{C}$) that are effective against the specified attack.

5 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 mission requirements, the resistance techniques specification, and the resistance properties. This tool uses the Z3 .NET API to compose the proper SMT expressions and generate an SMT file that contains the complete problem encoded as SMT assertions. We then feed this file 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 satisfaction problem with respect to multiple parameters, such as the network size, the sizes of flow tables, the number of app-specific variables, and the complexity of the mission requirements and isolation specifications.

5.1 Extensive Scalability Evaluation

The scalability of model checking tools depends on the size of the problem in terms of the state space and the number of transitions. In our case, the size of the problem depends on multiple network parameters, such as the network size and the length of flow tables. Due to the lack of real large-scale networks configurations, we synthesized a number of network instances with given parameters based on 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 generate the constraints satisfaction problem and solve it using the Z3 SMT solver. We record the time and memory required to solve the problem. As presented in Sections 3 and 4, both the resistance specifications and properties are translated into some form of reachability queries with varying number and structure of app-specific variables. Hence, the following evaluation applies on each of the mission requirements, the resistance specifications, and the properties.

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 and space of verifying 20 reachability properties between random pairs of hosts. 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. It was set to a constant value in the other setting 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 3 shows the results of this experiment. We can see that in the case of fixed bound, the time and space requirements are 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 with respect to network size.

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 flow table sizes measured by the number of rules. All the rules have the same structure (i.e. the same number of instructions and the same length of actions lists). As reported in Figure 4, the total number of rules ranged from 1.4 to 33.5 thousand rules. The results show that the growth in time and space tends to stabilize after a threshold (i.e. increasing the flow table size does not significantly increase the requirements). We believe this behavior is due to the compact representation of assertions 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 different bound each time. The

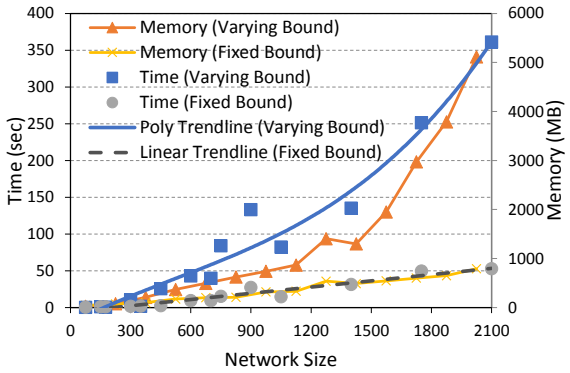

Figure 3: The impact of network size.

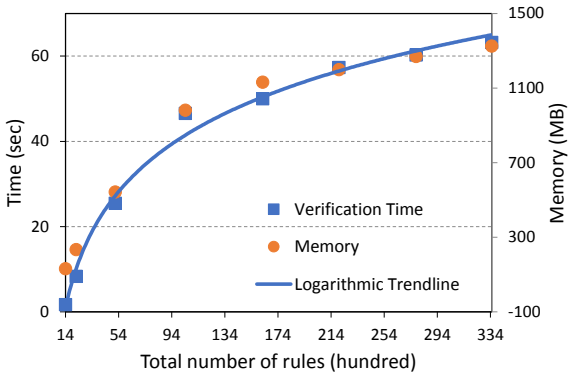

Figure 4: The impact of the table size.

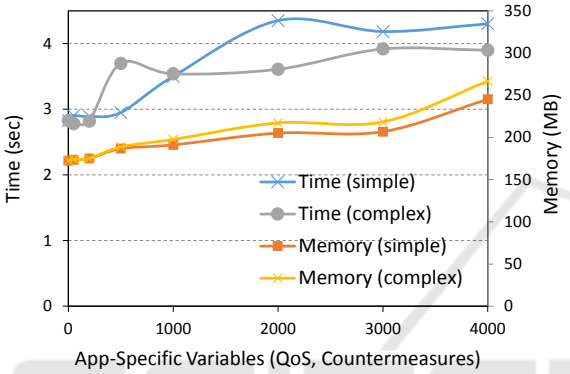

Figure 5: The impact of app-specific variables.

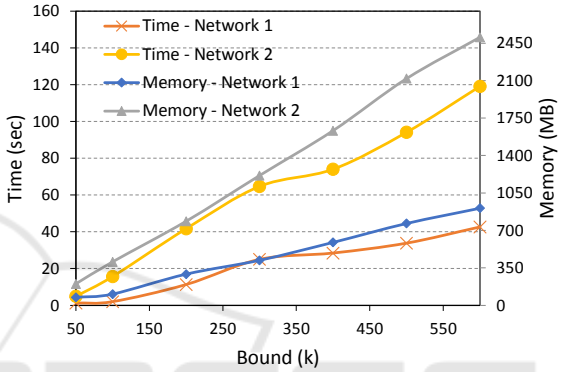

Figure 6: The impact of the bound.

bounds ranged from 50 to 600. Figure 6 shows that in both networks the time and space requirements increase linearly with respect to the bound.

The Impact of App-specific Variables. We conducted an experiment to study the impact of app-specific variables, which depends on the QoS parameters, diversity attributes, and the number of countermeasures. For this purpose, we ran our framework against a network of 300 nodes with a varying number of app-specific variables that ranged from zero to 5000. Moreover, we defined two types of app-specific variables, namely *simple* and *complex*. The *simple* variables were defined as additive operations (e.g., *sum* of delays over a path), while the *complex* includes non-linear multiplication. Figure 5 reports the time and space for both types. We can see that the number of app-specific variables of both types has a very minor impact on the time and space requirements. They remain almost constant even with a large number of app-specific variables.

5.2 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 (Zeng and Kazemian,). The network consists of 14 zones connected to two backbone routers via ten layer-2 switches with a total of 240 hosts in the 14 zones and a total of 3840 flow rules distributed over 16 switches.

We ran more than 50 reachability requirements with quality of service constraints on the number of hops between random pairs of hosts in the network. The source and destination of 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. This case study shows the applicability of our verification framework for verifying real requirements on real networks. We believe that the running time is acceptable as our verification is conducted offline and does not interfere with the live operation of the network.

6 RELATED WORK

The verification of network invariants has attracted a significant body of research in both enterprise and software defined networks. In this literature review, we focus on the data plane verification tools for OpenFlow-based Software Defined Networks. FlowChecker (Al-Shaer and Al-Haj, 2010) and HSA (Kazemian et al., 2012) are offline configuration analysis tools. FlowChecker encodes the OpenFlow flow tables using Binary Decision Diagrams (BDD) and uses model checking to verify security properties. HSA verifies tha data plane correctness by modeling the network as a geometric model to discover reachability violations, forwarding loops, and traffic isolation. VeriFlow (Khurshid et al., 2012), FLOVER (Son et al., 2013), NetPlumber (Kazemian et al., 2013), and FlowGuard (Hu et al., 2014) are real time policy verification tools. VeriFlow proposes to slice the OF network into equivalence classes to efficiently check for reachability violations. FLOVER is a model checker that checks the OpenFlow configuration for security violations using Yices SMT solver. NetPlumber is a real time policy checking tool based on HSA that utilizes a dependency graph between flow entries to incrementally check for loops, black holes, and reachability properties. FlowGuard examines dynamic flow updates to detect firewall policy violations. FlowGuard also 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 (Kazemian et al., 2012; Kazemian et al., 2013; Son et al., 2013) and to firewall policy verification in (Hu et al., 2014).

ConfigChecker (Al-Shaer et al., 2009) (BDD-based model checker) and Anteater (Mai et al., 2011) (SAT-based model checker) 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 (Bjorner and Jayaraman, 2014) is another tool that is based on the bit-vectors theory in Z3 solver for checking network invariants. Since SecGuru is also built using Z3, there may be a potential for extending it to apply bit-vector arithmetic. However, this has not been demonstrated in the applications presented in (Bjorner and Jayaraman, 2014).

The aforementioned related works provide multiple platforms to verify the end-to-end reachability in enterprise and software defined networks, but they do not focus on resistance verification. None of them provides the ability to express properties for indirect reachability and reason about multi-step attack paths. They also have very limited support for QoS requirements verification, which is required to ensure the network mission integrity. In this bounded model checking framework, we utilize the arithmetic theory in SMT to verify end-to-end reachability with QoS credentials and we model the indirect reachability between network hosts in order to verify the resistance specifications and their effectiveness against multi-step attacks.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we propose a model checking approach to model the complete configuration of networks and verify that (i) given mission requirements and resistance techniques specifications are properly enforced in the network and (ii) they are effective in resisting certain attacks. We also demonstrated how to achieve this using our model checker utilizing the linear and nonlinear arithmetic theories of SMT solvers. Our evaluation shows that we can verify the effectiveness of resistance measures against worms/APT attacks propagation. Although the performance evaluation reveals that the framework needs relatively large time to verify resiliency requirements for large networks, the verification is conducted offline. Thus, it will not affect the live operation of the network. We believe that the result are comparable to other tools that have been presented recently for SDN configuration verification. We plan to extend our framework by integrating more middle-boxes that support advanced functions and modeling other categories of cyber attacks. We are also investigating the feasibility of using hierarchical verification techniques by dividing the network and the properties into groups and investigate optimization and expression simplification techniques to enhance the performance of our framework.

ACKNOWLEDGEMENTS

This research was supported in part by the National Security Agency and Army Research Office. Any opinions, conclusions, or recommendations stated in this material are those of the authors and do not necessarily reflect the views of the funding sources.

REFERENCES

Al-Shaer, E. and Al-Haj, S. (2010). 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.

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

Alsaleh, M. N. and Al-Shaer, E. (2016). Towards automated verification of active cyber defense strategies on software defined networks. In *Proceedings of the 2016 ACM Workshop on Automated Decision Making for Active Cyber Defense*, SafeConfig '16, pages 23–29, New York, NY, USA. ACM.

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

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

Goldman, H. (2010). Building secure, resilient architectures for cyber mission assurance. *The MITRE Corporation*.

Goldman, H., McQuaid, R., and Picciotto, J. (2011). Cyber resilience for mission assurance. In *Technologies for Homeland Security (HST), 2011 IEEE International Conference on*, pages 236–241. IEEE.

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

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

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

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

Larsen, P., Brunthaler, S., Davi, L., Sadeghi, A.-R., and Franz, M. (2015). Automated software diversity. *Synthesis Lectures on Information Security, Privacy, & Trust*, 10(2):1–88.

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

Melin, A., Ferragut, E., Laska, J., Fugate, D., and Kisner, R. (2013). A mathematical framework for the analysis of cyber-resilient control systems. In *Resilient Control Systems (ISRCS), 2013 6th International Symposium on*, pages 13–18.

Miu, A., Balakrishnan, H., and Koksal, C. E. (2005). Improving loss resilience with multi-radio diversity in wireless networks. In *Proceedings of the 11th Annual International Conference on Mobile Computing and Networking*, MobiCom '05, pages 16–30, New York, NY, USA. ACM.

Nicol, D. M. and Mallapura, V. (2014). Modeling and analysis of stepping stone attacks. In *Proceedings of the 2014 Winter Simulation Conference*, WSC '14, pages 3036–3047, Piscataway, NJ, USA. IEEE Press.

ONF (2013). 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.

Qazi, Z. A., Tu, C.-C., Chiang, L., Miao, R., Sekar, V., and Yu, M. (2013). Simple-fying middlebox policy enforcement using sdn. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 27–38. ACM.

Qin, P., Dai, B., Huang, B., and Xu, G. (2014). Bandwidth-aware scheduling with sdn in hadoop: A new trend for big data. *arXiv preprint arXiv:1403.2800*.

Rahman, M. A. and Al-Shaer, E. (2013). A formal framework for network security design synthesis. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pages 560–570. IEEE.

Sahinoglu, M. (2006). Quantitative risk assessment for dependent vulnerabilities. In *Reliability and Maintainability Symposium, 2006. RAMS '06. Annual*, pages 82–85.

Shullich, R., Chu, J., Ji, P., and Chen, W. (2011). A survey of research in stepping-stone detection. *International Journal of Electronic Commerce Studies*, 2(2):103–126.

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

Verizon (2016). 2016 data breach investigations report. http://www.verizonenterprise.com/resources/reports/rp_DBIR_2016_Report_en_xg.pdf.

Verizon (2017). 2017 data breach investigations report. http://www.verizonenterprise.com/resources/reports/rp_DBIR_2017_Report_en_xg.pdf.

Yang, Y., Zhu, S., and Cao, G. (2008). Improving sensor network immunity under worm attacks: A software diversity approach. In *Proceedings of the 9th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, MobiHoc '08, pages 149–158, New York, NY, USA. ACM.

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

Zhang, M., Wang, L., Jajodia, S., Singhal, A., and Albanese, M. (2016). Network diversity: a security metric for evaluating the resilience of networks against zero-day attacks. *IEEE Transactions on Information Forensics and Security*, 11(5):1071–1086.