S3: A DFW-based *S*calable *S*ecurity *S*tate Analysis Framework for Large-Scale Data Center Networks

Abdulhakim Sabur, Ankur Chowdhary, and Dijiang Huang
{asabur, achaud16, dijiang}@asu.edu
Arizona State University

Myong Kang, Anya Kim, and Alexander Velazquez
{myong.kang, anya.kim, alexander.velazquez}@nrl.navy.mil
US Naval Research Lab

Abstract

With an average network size approaching 8000 servers, datacenter networks need scalable security-state monitoring solutions. Using Attack Graph (AG) to identify possible attack paths and network risks is a common approach. However, existing AG generation approaches suffer from the state-space explosion issue. The size of AG increases exponentially as the number of services and vulnerabilities increases. To address this issue, we propose a network segmentation-based scalable security state management framework, called *S3*, which applies a *divide-and-conquer* approach to create multiple small-scale AGs (i.e., sub-AGs) by partitioning a large network into manageable smaller *segments*, and then merge them to establish the entire AG for the whole system. S3 utilizes SDN-based *distributed firewall* (DFW) for managing service reachability among different network segments. Therefore, it avoids reconstructing the entire system-level AG due to the dependencies among vulnerabilities.

Our experimental analysis shows that S3 (i) reduces AG generation and analysis complexity by *reducing* AG's density compared to existing AG-based solutions; (ii) utilizes SDN-based DFW to provide a granular security management framework, by incorporating security policies at the level of individual *hosts* and *segments*. In effect, S3 helps in limiting targeted slow and low attacks involving lateral movement.

1 Introduction

With the surge in cloud infrastructure and new technology such as containers and server-less applications, the attack surface has increased significantly. In order to have a better understanding of the security situation of a system, scalable and effective *attack representation methods (ARMs)* are required. The security administrator can use information derived from ARMs and apply the appropriate countermeasure on the identified critical path [6].

Graphical security states management and analysis solutions, e.g., *Attack Graphs* (AGs), is such a tool to fulfill the purpose of security state analysis. AGs are defined as a data structure, used to model all possible critical attack paths and vulnerabilities of a system, which an adversary can exploit in order to achieve his/her attacking goals. However, generating and analyzing AG in a security system requires a significant generation and analysis overhead, an issue addressed in this paper, that has not been effectively addressed by existing solutions. The AG generation and attack path searching when performing AG-based attack scenario analysis can be an NP-hard problem as noted by Durkota *et. al.* [12], which depends on the density of a given AG. In a large network system, AGs are often incomprehensible to a user due to its complex interdependence among vulnerabilities. The identification of information regarding vulnerability dependencies becomes increasingly difficult as the number of services and vulnerabilities are increasing in the network system. Amman *et. al.* [1] proposed an AG generation approach with the scalability of the order $O(N^6)$. MulVAL [27] reduces the AG generation and analysis complexity from $O(N^6)$ to $O(N^2) - O(N^3)$, where N is the number of network hosts[1]. Nevertheless, the generation of AG by using MulVAL still takes an order of minutes for a few hundreds of hosts.

The second problem *S3* framework considers in this work is *lateral movement*. This sophisticated attack is conducted by highly expert individuals or organizations, which allow for multiple exploits and movement from one system node to another. AG can be used to identify the critical paths that can be exploited by an expert attacker. AG-based security analysis can identify the dependencies between services in a network and minimize the security issues that will allow an adversary to compromise critical services by lateral movement along *east-west* network using exhaustive trials method [8] over different attack paths. Moreover, AG has been widely used in identifying the least expensive countermeasure solution [6].

[1]MulVAL ignored the scenarios when multiple vulnerabilities exist on the same host. In current virtualized environments, vulnerabilities can be interdependent within a given host and thus contribute to the complexity of AG generation and analysis. In this paper, we consider N is the measurement of the number of given vulnerabilities.

Thus, to identify and detect lateral movement, a well-modeled security analysis tool is needed. S3 framework focuses on the scalability of AG, which in return can be used to identify critical systems and help in the detecting complex attack scenarios.

In this paper, we propose to leverage SDN-based Distributed Firewall *(DFW)* [13] rules to partition the large-scale network into small network segments in order to efficiently compute an AG in a data-centric network. We utilize the SDN controller to inspect network events and obtain a global view of the network and service reachability. SDN controller interacts with distributed firewall's *Security Policy Database* (SPD) at the application plane to obtain an updated list of *security policies*. The data-plane is based on OpenFlow switches. SDN controller installs DFW rules on connected switches at data-plane layer to limit traffic across different logical segments. We apply micro-segmentation [3,21] in order to divide the large data center into small segments based on granular security policies. This helps in restricting the lateral movements along *east-west* communication paths among network segments.

In general, S3 framework follows a *divide-and-conquer* approach to generates AG for each segment (*sub-AG*) after obtaining vulnerability and reachability information from vulnerability analysis tools. Finally, the result of divide-and-conquer approach (sub-AGs) are merged based on the $|DFW|$ rules to obtain the system AG, or we call it the Composite Attack Graph *(CAG)*. The key contributions of S3 are as follows:

- Our proposed S3-based micro-segmentation algorithm is able to generate a scalable AG by utilizing DFW rules. The algorithm complexity We achieved is $O((\frac{N}{K})^2) + O(Klog(K))$, where N is the total number of vulnerabilities and K is the number of established segments. The AG generation time achieved using S3 is much faster compared to state-of-the-art AG generation tools, i.e., S3 takes ~ 20 *sec* for generating AG for a network with services over 6000 services, as shown in Section 6, compared to over 1 hour taken by MulVAL [27].

- We propose a granular *divide-and-conquer* based security state management approach for large data center network *micro-segmentation*, which utilizes DFW to significantly reduce the AG density and number of attack paths. Consequently, our approach is able to generate a scalable AG by leveraging SDN capabilities and utilizing DFW to obtain real-time security state policies.

The rest of the paper is organized as follows, Section 2 provides a literature review of existing AG generation methods and DFW frameworks, along with their limitations. We discuss the threat model, S3 *architecture* and the details about *AG formalism* in the Section 3. We discuss the SDN-based S3 system and *API architecture* in Section 4. The description of S3-based *micro-segmentation* approach, graph segmentation *algorithm*, optimal number of micro-segments, and a proof of scalability of *micro-segmentation* method has been provided in the Section 5. The evaluation of AG *scalability*, SDN *controller overhead*, and experimental details on *optimal number* of *micro-segments* is discussed in Section 6. In Section 7, we discuss related issues such as cycles in directed graphs, alternative segmentation heuristics, and possible security policy conflicts that can be induced by the *micro-segmentation*. Finally we conclude the research paper in Section 8, and provide details on the future work.

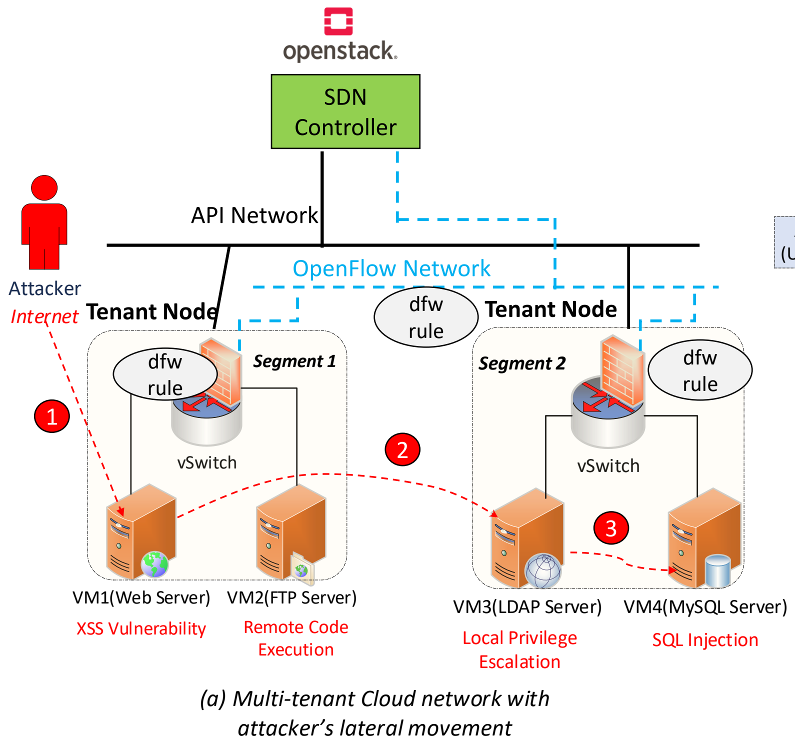
2 Related Work

The Scalability of Attack Graphs: Generation of scalable attack graphs has been a popular area of research. Amman *et al.* [1] presented a scalable solution in comparison with prior modules [32] by assuming *monotonicity*. This assumption allowed them to achieve scalability of $O(N^6)$ [19]. To mitigate the state space explosion problem, most of the existing solutions try to *reduce the dependency* among *vulnerabilities* by using logical representation [27]. Hong *et. al.* [18] apply a *hierarchical strategy* to reduce the computing and analysis complexity of constructing and using AGs by *grouping and dividing* the connectivity of the system into hierarchical architecture. The performance time is, however, $\sim 50s$ for 50 services, whereas our framework generates scalable attack graph of similar scale in *2.2s*.

Kaynar and Sivrikaya [23] proposed a framework for distributed AG generation that utilizes a shared memory approach. The graph generation time is of order *2-3 minutes* for *450 hosts*, which *cannot be used for real-time security analysis*. Chowdhary *et. al.* [5], use distributed hypergraph partitioning for Attack Graph generation. The research work has however not considered application of DFW for further optimizing the size of Attack Graph. Cao *et. al.* [2] proposed an approach to compute AG in parallel. The division is based on the privileges inside the hosts. The experimental analysis in this work shows that the required generation time for ~ 500 hosts is $\sim$ *20 sec*, while in our work it takes only $\sim$ *6.5 sec*.

Mjihil *et. al.* [25] used a parallel graph decomposition approach. The evaluation in this research work tests the effect of the number of vulnerabilities on the AG generation time, which is not reliable since it does not explain how the number of vulnerabilities is related to each service in the system as we did in this paper. The research work tested a maximum of 50 vulnerabilities in which they obtaind an AG in ~ 10 sec, while in our work we obtain an AG in ~ 2 sec.

Distributed Firewall: Some researchers have addressed DFW in SDN [20] [30] [28]. Yet, they only consider stateless firewall which does not leverage the full advantage of both SDN and DFW. *VMware* has proposed a distributed firewall for their *NSX model*, by using a central object that manages the distributed firewall's policies [26]. Unfortunately, this ar-



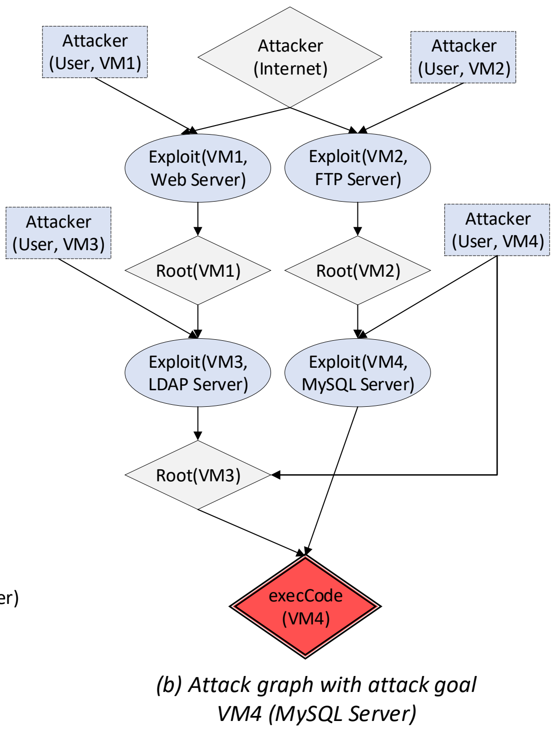


Figure 1: Representation of vulnerability information and corresponding attack graph in a multi-tenant data-center network. Distributed Firewall (DFW) can be implemented at each tenant/segment.

chitecture is only applicable to the NSX model and cannot be adopted to OpenFlow standards because *NSX* comprises of both *stateful* and *stateless* components. The firewall rules of the host machines are also controlled by the NSX manager, whereas the OpenFlow protocol based SDN framework implements a *stateless firewall*. There are no existing research works, which attempt to control the security state explosion in attack representation methods using better security policy design and management framework. In S3 framework, we utilize design principles based on VMWare NSX architecture, and the programming flexibility afforded by the SDN, in order to *limit the connections* between different logically separated regions of a data-center.

3 Background

3.1 Threat Model

In this section, we describe threat model, AG model, and AG scalability challenge in order to motivate the need for a scalable attack representation framework and allow the reader to have a comprehensive understanding of AG scalability issue.

To explain the attack graph more clearly, we show Figure 1 (a), which illustrates a simple multi-tenant data-center based on *Openstack* framework. The network in this example consists of 4 virtual machines (VM1-4), but this general model can be applied to large scale cloud networks. The OpenStack management framework can be used by the security administrator in order to insert DFW policies or monitor the status of each VM present on tenant nodes.

The attacker is located on the *Internet*. The *attack model* here is vulnerability exploitation to achieve privilege escalation. There are several attack paths the attacker may take to achieve their *goal*, which is to ex-filtrate data from the *MySQL* server, by obtaining *root privileges* on *VM4*, i.e., *execCode(VM4)*.

Lateral Movement: The scope of security enforcement offered by a traditional firewall is limited to *north-south* traffic, i.e., firewall serves as a sentry between trusted and untrusted networks. Once the attacker has managed to breach the security restrictions at the network edge, they can laterally move inside the network (east-west traffic) and exploiting key resources virtually unchecked. Centralized firewalls do not protect networks from *multi-stage* attacks using *lateral movement*. Since everyone on the internal networks is trusted and the traffic within these trusted networks is not rigorously inspected by the traditional firewall based defense mechanisms.

The volume of *east-west* traffic in the datacenter environment is around 76%, as compared to north-south traffic - 17% [7]. As shown in Figure 1 (a), if the attacker can compromise Web Service on VM1 in Step (1) of the multi-stage attack, they can use this attack as a pivot for compromising VM3, and VM4 in steps (2) and (3). The lateral movement is hard to detect and prevent using traditional security architecture since in most cases its intended purpose is to defend the network system against outsider adversaries.

3.2 Attack Graphs and Scalability Challenge

Data-center networks are scaling up at a fast pace. For example, *Amazon Web Services (AWS)* has on average between 50,000 servers, to 80,000 servers, according to [22]. An efficient security analysis of such data-center is expected to be scalable and granular. Hence, there is a need for scalable security analysis, and AG serves this purpose to module the critical paths in the system.

In this paper, our research uses the *exploit dependency graph* [4], since it directly models dependencies between the vulnerabilities in a computer networked system. Also, all services for application-based on networked-based attacks are related in this graph and it shows what are the pre-requisites (pre-conditions) and post-conditions for those attacks. *Nodes* in such an AG are not the network states, but rather, they are *vulnerabilities*. AG can be formally defined as follows:

Definition 1. *(Attack Graph (AG)) An attack graph is represented as a graph $G = \{V, E\}$, where V is the set of nodes and E is the set of edges of the graph G, where*

1. *$V = N_C \cup N_D \cup N_R$, where N_C denotes the set of conjunctive or exploit nodes (pre-condition), N_D is a set of disjunctive nodes or result of an exploit (post-condition), and N_R is the set of a starting nodes of an attack graph, i.e. root nodes.*

2. *$E = E_{pre} \cup E_{post}$ are sets of directed edges, such that $e \in E_{pre} \subseteq N_D \times N_C$, i.e., N_C must be satisfied to obtain N_D. An edge $e \in E_{post} \subseteq N_C \times N_D$ means that condition N_C leads to the consequence N_D.*

An example of *vulnerability* information, network *service* information, and *Host Access Control List* (HACL) represented in *datalog* format is shown below:

```
vulExists (ipaddr, cve-id, service)
networkServiceInfo(ipaddr, service, prot, port)
hacl(srcip, dstip, prot, port)
```

Attack graph uses HACL tuples to model network and firewall configurations, in which it uses a general rule to test and specify reachability information (i.e., any host can access any host using any port and protocol).

Attack Graph Scalability Challenge: As can be seen in the Figure 1 (b), a network consisting from 4 hosts resulted in a graph of 13 nodes. Large data-center networks have thousands of services, servers, and VMs. The expected AG size of such system is huge due to representing network state using a conditional or combination of conditional and exploit representation of security situation, which will lead to huge number of nodes and edges in the AG [24, 34]. Therefore, an efficient and scalable methodology is needed to help the administrator in representing and analyzing the security situation in the system.

4 System Model and Architecture

4.1 S3 System Architecture

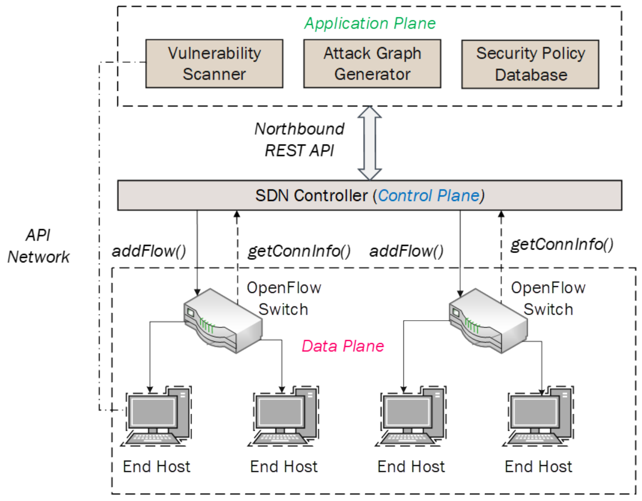


Figure 2: S3 System Architecture and operating layers

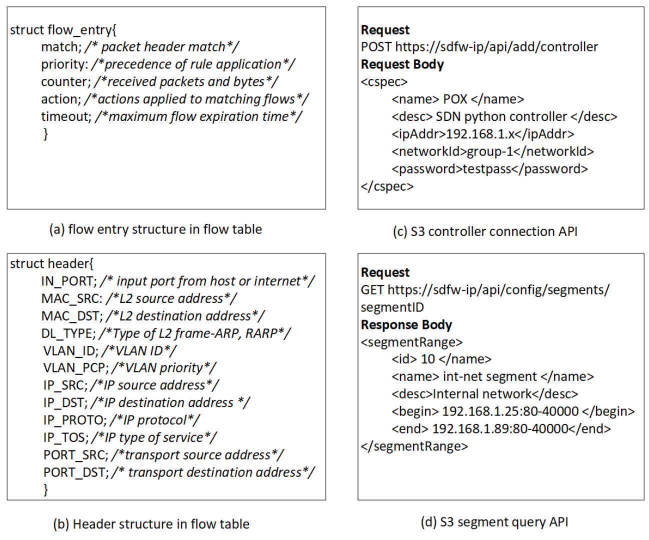
We consider the cloud infrastructure shown in Figure 2 as the architecture for framework S3, where the networking infrastructure is based on *Software Defined Networking* (SDN) solutions. SDN is an emerging technology aiming to enhance the current networking protocols by separating control-plane from data-plane. The *Application Plane* comprises of *vulnerability scanner* (Nessus) which collects vulnerability information from each network host. The vulnerability scanner, on the other hand, interacts with the individual hosts at *data plane* using *API network.*

The *Security Policy Database* (SPD), as shown in Figure 2, creates security policies to define micro-segmentation policies. These policies dictate how segments are created. The traffic between segments is regulated using security policies defined by SPD. The SPD interacts with SDN controller using *northbound REST APIs* to update *security policy* information.

The *Attack Graph Generator* module, which is a wrapper program we developed for the generation of *sub-AGs* at the level of each segment. This module interacts with a vulnerability scanner and SPD to create segments using an algorithm, and finally, utilize *merge* algorithm to merge segments into fully connected AG.

S3 framework utilizes OpenFlow *southbound APIs* to provide flexible and programmable micro-segmentation architecture. The SDN framework utilizes OpenFlow protocol to communicate with switches at the *Control Plane* level. As shown in the Figure 2, the SDN controller (control plane) collects connection information from software switches situated at the *data plane* using *getConnInfo()* API. The security policies are implemented on each switch using *addFlow()* API as shown in the communication channel between controller and switch - Figure 2.

Each *OpenFlow* switch has *flow tables*, which are used to store incoming/outgoing flow rules based on packet header match. The rules are stored in the *Ternary Content Addressable Memory* (TCAM) format.



```
struct flow_entry{
        match; /* packet header match*/
        priority: /*precedence of rule application*/
        counter; /*received packets and bytes*/
        action; /*actions applied to matching flows*/
        timeout; /*maximum flow expiration time*/
        }
```

(a) flow entry structure in flow table

```
Request
POST https://sdfw-ip/api/add/controller
Request Body
<cspec>
        <name> POX </name>
        <desc> SDN python controller </desc>
        <ipAddr>192.168.1.x</ipAddr>
        <networkId>group-1</networkId>
        <password>testpass</password>
</cspec>
```

(c) S3 controller connection API

```
struct header{
        IN_PORT; /* input port from host or internet*/
        MAC_SRC: /*L2 source address*/
        MAC_DST; /*L2 destination address*/
        DL_TYPE; /*Type of L2 frame-ARP, RARP*/
        VLAN_ID; /*VLAN ID*/
        VLAN_PCP; /*VLAN priority*/
        IP_SRC; /*IP source address*/
        IP_DST; /*IP destination address */
        IP_PROTO; /*IP protocol*/
        IP_TOS; /*IP type of service*/
        PORT_SRC; /*transport source address*/
        PORT_DST; /* transport destination address*/
        }
```

(b) Header structure in flow table

```
Request
GET https://sdfw-ip/api/config/segments/
segmentID
Response Body
<segmentRange>
        <id> 10 </name>
        <name> int-net segment </name>
        <desc>Internal network</desc>
        <begin> 192.168.1.25:80-40000 </begin>
        <end> 192.168.1.89:80-40000</end>
</segmentRange>
```

(d) S3 segment query API

Figure 3: S3 data structures utilzed by control plane software (a), (b) and application plane REST API used by network admin (c), (d).

Flow Table consists of other important fields besides *match* and *action* fields. Each flow entry also has *priority*, *counter*, and timeout fields, as shown in the Figure 3 (a). The *header* structure - Figure 3 (b) of each flow is used for matching traffic against incoming traffic. The *REST API* at the application plane help in management of the distributed control plane. For instance, if a new controller needs to be added to the control plane, the *POST API* - Figure 3 (c) is used by controller in order to announce intent to *join* DFW. The *application plane* checks the vulnerability, network topology, and reachability information periodically in order to update the network segments. Information about each segment can be obtained using *GET API* as shown in Figure 3 (d). The network segment generated by *S3* framework in this case *int-net segment*, with segment ID *10* consists of services present on ports *80-40000* on all machines in range *192.168.1.25-89*.

5 DFW and Micro-Segmentation

Micro-segmentation based on SDN-framework is a method of creating secure zones in data centers and cloud deployments to isolate workloads from one another and secure them individually to make network security more granular.

We utilize *distributed firewall (DFW)* functionality based on micro-segmentation architecture to control the reachability across different network segments, and in effect, reducing the AG scalability. By leveraging the $|DFW|$ rules, as shown in Figure 1 (a), we can control the access between *Web Server (VM1)* and *FTP Server (VM2)* in *Segment 1*. Similarly, a granular security policy can enforced to control the connection requests from *LDAP Server (VM3)* to *SQL Server (VM4)*.

Table 1: Example of Network topology vulnerabilities and connectivity information.

Segment	VM	Service	Vulnerability	Attack Path
Segment 1	VM1	WebServer	Cross-Site Scripting	Internet- VM1 port 80
	VM2	FTPServer	Remote Code Execution	VM1 - VM2 port 25
Segment 2	VM3	LDAP Server	Local Buffer Overflow	Segment 1 via DFW
	VM4	SQL Server	SQL Injection	VM3 - VM4 port 3306

When the *DFW* is absent, the attacker can reach to any virtual machine from the internet and exploit it since it has a vulnerability as shown in Table 1. The resulted AG for this topology is shown in Figure 1 (b). The provided Figure is just a simple example to illustrate multi-stage attacks that can be launched in a cloud network. In reality, the graph will have more nodes and edges to illustrate connectivity information.

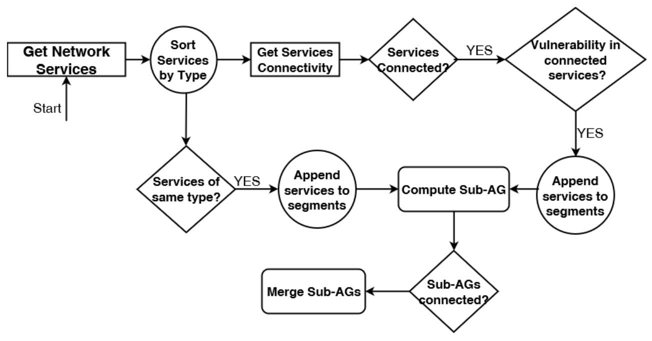


Figure 4: Flow Diagram of S3 Segmentation and AG Generation.

5.1 Implementation

Segmentation can also be achieved by considering security requirements. The requirements, however, are not analyzed and do not take into account the existing vulnerabilities and the connectivity between those vulnerabilities (vulnerability dependencies). Therefore, the goal of S3 is to provide a segmentation solution to enhance system security and ensure fine-grained security rules between vulnerable and mission-critical services. S3 utilizes the system administrator's knowledge about the existing vulnerabilities residing in the system and provides a methodology to build *logical segments* containing services based on their vulnerability and reachability information. The flow diagram of S3 is presented in Figure 4. The

Algorithm 1 Segmentation and Scalable AG Generation Algorithm

```
1:  procedure SEGMENT ESTABLISHMENT
2:  V ← {1,n}                                   ▷ list of vulnerabilities
3:  R ← {1,m}                                   ▷ distributed firewall rules
4:  S ← {1, l}                                  ▷ network services
5:      for all service ∈ S, vuln ∈ V, rule ∈ R do
6:          sort(service, type)
7:          if connected ($service_i$, $service_j$, $rule_k$) & vuln ∈ (service$_i$, $service_j$) then
8:              update (segment, $service_i$, $service_j$)
9:          else if type ($service_i$) == type ($service_j$) then
10:             update (segment, $service_i$, $service_j$)
11:     end if
12: end for
13: procedure ATTACK GRAPH GENERATION
14:     for all Segments do
15:         Compute sub-AG for $segment_i$
16:     end for
17:     for all subAGs do
18:         if connected ($subAG_i$, $subAG_j$, $rule_k$) then
19:             merge ($subAG_i$, $subAG_j$)
20:         end if
21:     end for
```

network services are first analyzed and sorted according to the service type and connectivity between the services based on the $|DFW|$ rules. If the services are of the same type, then we append them to segments. If the services are connected by $|DFW|$ rule, and there is a vulnerability in the connected services, then we also append the services into different separate segments. We separated the segments because we want to keep services of same type in one segment and separated from other ones having different type. After that, we compute the sub-AG of all the segments and check for connectivity between the sub-AG based on the $|DFW|$ as well. Finally, the sub-AG from individual segments are merged and *Composite Attack Graph* (CAG) is created. The graph is *frequently updated*, based on changes in network and vulnerability configurations, which are *monitored* by *SDN controller*. Next, we present an Algorithm 1, which explains more on how the segmentation is achieved for the of building scalable CAG.

5.1.1 Algorithm Analysis

S3 segmentation and AG generation approach is shown in Algorithm 1. The algorithm first starts by obtaining all vulnerabilities, running services, and $|DFW|$ rules in the system *lines 2-4*. Next, the algorithm span over all of these gathered information and sorts the services by their type as shown in *line 5*. After that, the algorithm checks for services that are of the same type and put them in the same logical segment. Also, if there are two services connected to each other by a $|DFW|$ rule such that there is a vulnerability in $service_i$ allowing the attacker to exploit a vulnerability in $service_j$, then these two services are aggregated in the same segment. To avoid redundancy, if the services have been already added to an existing segment, the algorithm continues. After sorting and aggregating services, vulnerabilities, and $|DFW|$ rules, we now have a number of segments that contain services of similar type, or vulnerable services connected by $|DFW|$ rule. The next step is to compute an AG for each of these segments to check and the critical paths inside each of these segments - *lines 13-15*. Next, the sub-AG needs to be merged together to allow the system administrator to see the relationship between these segments and how an adversary can move from one segment to another one to achieve their final goal. The merge procedure, as shown in *line 16-19* is based on the connectivity of the separated segments.

5.1.2 Complexity Analysis

The sort operation over services in algorithm 1 can be performed using quick sort algorithm [9] which has average complexity of $O(Slog(S))$, where S is the number of services. Computing the sub-AG is a *linear* time operation as the computation is being performed in parallel with the help of S3 SDN controller. The merge operation requires searching among the segments and their connectivity, thus the search can be done using *divide and conquer* approach which has the complexity of $O(Klog(K))$ [9], where K is the number of segments. As a result, the complexity of computing the global-view of AG is based on the total number of vulnerabilities in the system N, divided by the number of segments K, plus the complexity for the merge operations, a total of $O((\frac{N}{K})^2)$ + $O(Klog(K))$ + $O(Slog(S)) \sim O((\frac{N}{K})^2)$ when $N >> K$ and $N >> S$.

To make the full picture more clear about the resulting AG, we present the following abstraction representation definitions of AG:

Definition 2. Composite Attack Graph *(CAG) is a tuple CAG={S, E, N}.*

- *S denotes the set of all segments. Each segment has a sub attack graph (sub-AG), i.e., sub-AG$_1$, sub-AG$_2$ ∈ S. If there is a connectivity ($|DFW|$ rule) from one service in segment s to service present in segment s', $s \neq s'$, where this connection is the one required to exploit a vulnerability present on service in s', then this information (post and pre-condition, vulnerability and connectivity) are appended and concatenated in a segment.*

- *N denotes the nodes the set of all nodes present in the CAG. A node in an individual segment* s *can be denoted by N^s. The nodes can be conjunct nodes N^s_C, disjunct node N^s_D or root node N^s_R. A link from segment s to segment s' indicates reachability to a vulnerability in the* *target segment s', or a $|DFW|$ rule that exists based on the Service Level Agreement (SLA). In other words, this link is the result from exploiting segment s which we call post-condition that is needed for the attacker to reach and exploit s'. Hence the post-condition from s becomes a pre-condition s'.*

- *$E \subseteq E^S_{pre} \cup E^S_{post}$ is the edges present across all segments. If an edge from segment (sub-AG) s creates a post-condition in segment (sub-AG) s', we denote the post-condition using edge $E^{s'}_{post} = N^s_C \times N^{s'}_D$.*

5.2 Optimal Micro-Segmentation and Validation

In a large cloud network, when creating segmentation, the natural question is to identify the *quality* of segmentation. The network administrator needs to identify the *optimal number* (K) of segments and the basis of segmentation. We use a heuristic approach based on service similarity, vulnerability weight, and DFW rules, in order to validate segment quality and optimal number (K). We define segmentation properties used in S3 framework as follows.

1. **Segment Compactness:** Evaluates how closely the services in the same segment are related to each other. For instance, all *Web Server (http/https)* services can be put in the same segment. We represent this using variable s_{com}.

2. **Separation:** The separation can be decided based on the number of $|DFW|$ rules currently present between two segments. The higher the number, the higher the distance will be between the segments. Variable s_d is used for representing separation measure.

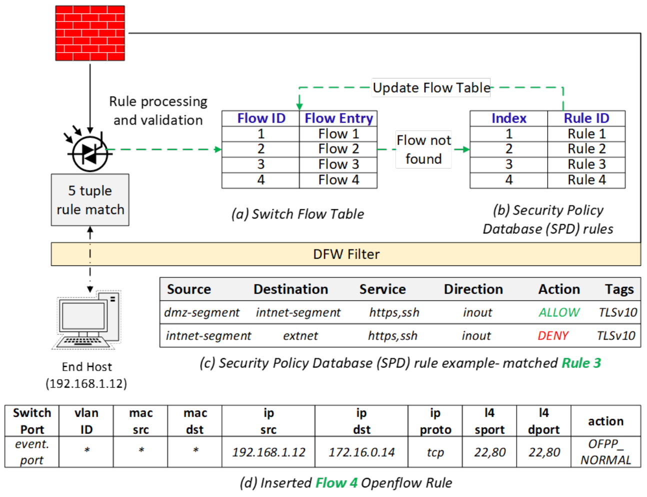
3. **Connectivity:** The connectivity depends on dependencies between vulnerabilities in the same segment as described in Definition 2. The connectivity is denoted using variable s_{con}.

The segmentation procedure aims at finding segmentation with high separation across segments, and high connectivity between nodes in the same segment. We utilize *Segmentation Index* (SI), an indexing measure based on *Dunn Index* [11], often used in *K-Means* clustering [15] approach in order to validate the quality of our micro-segmentation. Using the variables enumerated above, we define segmentation index as,

$$SI = \frac{\alpha \times \{s_{com} + s_{con}\}}{\beta \times s_d}, \alpha, \beta \in (0, 1], \tag{1}$$

where α and β are indexing parameters, their values are chosen based on the administrator's need. This indexing equation can help the system administrator to decide the optimal number of segments based on the desired requirements. If $\alpha > \beta$, it means the SI places higher weight on connectivity between services in the same segment. On the other hand if $\alpha < \beta$, the SI places higher weight on $|DFW|$ rules between the segments. High segmentation index value indicates optimal number of segments, since this equation is derived from *Dunn Index*, which also aims at finding the maximum distance between clusters [11]. We show in the Evaluation section 6.4 through experimental results how SI has been used for identifying optimal number of segments.

5.3 DFW Dynamic Traffic Match and Flow Update


Update Flow Table
Rule processing and validation
Flow not found
5 tuple rule match
(a) Switch Flow Table
(b) Security Policy Database (SPD) rules
DFW Filter
End Host (192.168.1.12)
(c) Security Policy Database (SPD) rule example- matched Rule 3
(d) Inserted Flow 4 Openflow Rule

Flow ID	Flow Entry
1	Flow 1
2	Flow 2
3	Flow 3
4	Flow 4

Index	Rule ID
1	Rule 1
2	Rule 2
3	Rule 3
4	Rule 4

Source	Destination	Service	Direction	Action	Tags
dmz-segment	intnet-segment	https,ssh	inout	ALLOW	TLSv10
intnet-segment	extnet	https,ssh	inout	DENY	TLSv10

Switch Port	vlan ID	mac src	mac dst	ip src	ip dst	ip proto	l4 sport	l4 dport	action
event. port	*	*	*	192.168.1.12	172.16.0.14	tcp	22,80	22,80	OFPP_ NORMAL

Figure 5: Distributed Firewall (DFW) security policy rule match and flow table update.

The DFW utilizes OpenFlow and REST API network to match the traffic based on five tuples, i.e., {srcip, dstip, sport, dstport, protocol}. We use the example Figure 5, to illustrate DFW traffic match and rule update operations.

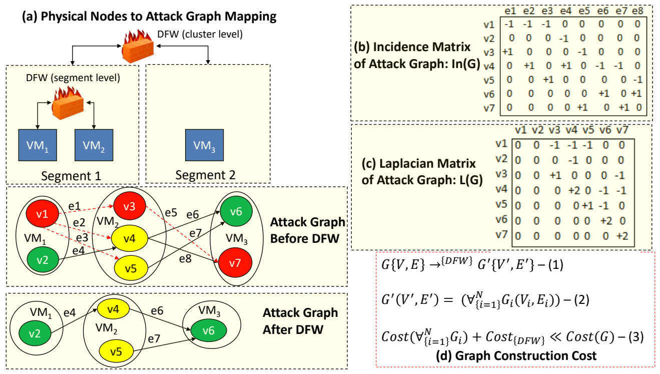
- **Step 1:** The *end-host* (192.168.1.12) from *intranet-segment*, attempts to send *http* traffic to port *80* and *ssh* traffic to port *22* of host (172.16.0.14) situated in another segment *dmz-segment*.

- **Step 2:** Initially, when flow table is checked using *table_lookup*, there is no rule present for the matching traffic rule. The flow table only has rules with *Flow ID {1-3}* - Figure 5 (a). The packet is sent to the controller using *action=OFPP_CONTROLLER*.

- **Step 3:** The controller checks the security policies defined by the *Security Policy Database* (SPD) rules present in application plane, using northbound REST API. The traffic pattern matches the *Rule ID {3}* - Figure 5 (b), (c). The action defined in the SPD for this traffic is *ALLOW*.

- **Step 4:** The flow table is updated with new OpenFlow rule - *Flow ID {4}*. The fields corresponding to layer 3,4 are updated and layer 2 fields are wildcarded - Figure 5 (d). Thus communication is enabled between two hosts. If there is no match for the traffic, in either flow table or SPD, the traffic is discarded based on whitelisting policy.

5.4 Scalable Attack Graph Generation Cost Analysis


(a) Physical Nodes to Attack Graph Mapping
DFW (cluster level)
DFW (segment level)
VM1
VM2
VM3
Segment 1
Segment 2
v1
v2
v3
v4
v5
v6
v7
e1
e2
e3
e4
e5
e6
e7
e8
Attack Graph Before DFW
Attack Graph After DFW
(b) Incidence Matrix of Attack Graph: In(G)
(c) Laplacian Matrix of Attack Graph: L(G)
(d) Graph Construction Cost

	e1	e2	e3	e4	e5	e6	e7	e8
v1	-1	-1	-1	0	0	0	0	0
v2	0	0	0	-1	0	0	0	0
v3	+1	0	0	0	-1	0	0	0
v4	0	+1	0	+1	0	-1	-1	0
v5	0	0	+1	0	0	0	0	-1
v6	0	0	0	0	0	+1	0	+1
v7	0	0	0	0	+1	0	+1	0

	v1	v2	v3	v4	v5	v6	v7
v1	0	0	-1	-1	-1	0	0
v2	0	0	0	-1	0	0	0
v3	0	0	+1	0	0	0	-1
v4	0	0	0	+2	0	-1	-1
v5	0	0	0	0	+1	-1	0
v6	0	0	0	0	0	+2	0
v7	0	0	0	0	0	0	+2

$G\{V,E\} \rightarrow^{\{DFW\}} G'\{V',E'\} - (1)$

$G'(V',E') = (\forall_{\{i=1\}}^{N} G_i(V_i,E_i)) - (2)$

$Cost(\forall_{\{i=1\}}^{N} G_i) + Cost_{\{DFW\}} \ll Cost(G) - (3)$

Figure 6: Distributed Firewall-based Multi-Level AG Generation.

We consider the mapping between the physical network and virtual network shown in Figure 6 (a). The physical topology consists of two segments, i.e., *Segment 1* and *Segment 2*, with $VM_1, VM_2 \in Segment1$ and $VM_3 \in Segment2$. Each VM consists of a number of services such as apache2, mysql, etc. The connectivity relation between the VMs are used to determine the AG for the entire network. For instance, if the firewall rules are defined between VMs and segments in a coarse grained manner, the AG will be huge in size as shown in *Before DFW* case in the above Figure 6. The traffic across each segment according to whitelisting policy might be limited, whereas, if we enforce white-listing policy at segment and service (SSH, MySQL) level as shown in Figure 5 (c), the amount traffic can be scale of attack graph generated can be finite for security analysis. Once the DFW is enforced at different levels of network, i.e., at the granularity of per-VM, per-segment, or entire network, we obtain a sparse AG, as shown in *After DFW* in Figure 6 (a). We define the *Incidence* and *Laplacian* matrices for the attack graph G below,

Definition 3. *Incidence Matrix The incidence matrix In(G) of graph G{V,E} is a $|V| \times |E|$ matrix, as shown in the Figure 6 (b), with one row for each node and one column for each edge. For each edge $e(i,j) \in E$, column entry e of In(G) is zero, except for i^{th} and j^{th} entries, which are +1 and -1, respectively (if there is an edge from i to j, the value is +1, whereas it is -1 if there is an edge from j to i in the graph, the value is zero if there is no edge e(i,j)).*

Definition 4. *Laplacian Matrix The Laplacian matrix L(G) of graph G{V,E} as shown in the Figure 6 (c), is a $|V| \times |V|$ symmetric matrix, with one row and column for each node. It is defined by*

- *L(G) (i,i): is the degree of node I (number of incident edges).*
- *L(G) (i,j): -1 if $i \neq j$ and there is an edge (i,j).*
- *L(G) (i,j): 0 otherwise.*

The application of DFW at different levels of the physical and logical network increases graph sparsity. The aggregated graph has reduced state space compared to the original AG.

5.5 Sparse Graph Connectivity using DFW

The incidence graph In(G) and laplacian graph L(G) have the following properties.

- L(G) is *symmetric*, i.e., *eigenvalues* of L(G) are real and its *eigenvectors* are *real* and *orthogonal*. For example, let $e = [1,...,1]^T$ be a *column vector*. Then $L(G) \times e = 0$.
- Matrices are independent of signs chosen for each column of In(G), $In(G) \times In(G)^T = L(G)$.
- Let $L(G) \times v = \lambda \times v$ and $\lambda \neq 0$, where v is eigenvector and λ is eigenvalue of L(G),

$$\lambda = ||In(G)^T - v||^2/||v||^2$$
$$\lambda = \frac{\sum_{e(i,j)\in E}(v(i)-v(j))^2}{\sum_i v(i)^2} \qquad (2)$$

- Eigenvalues of L(G) are non-negative, i.e., $0 = \lambda_1 \leq \lambda_2 ... \leq \lambda_n$.
- The number of connected components of G is equal to number of λ_i equal to 0. In particular, $\lambda_2 \neq 0$ *if & only if* G is connected.

Using the properties defined above, we check the *algebraic connectivity* of two graphs G and G', which can be compared to check the density reduction. The graph $G'\{V',E'\}$ obtained in the case of *After DFW* scenario, is composed on sub attack graphs (sub-AGs), $G_1, G_2 ..., G_n$, i.e., $G'\{V',E'\} = \cup_{i=1}^{N} G_i$, as shown in Figure 6 (d). Since $G'\{V',E'\}$ is obtained from $G\{V,E\}$ after collapsing vertices and edges at different layers using a multi-level DFW, it naturally follows that G' is a subgraph of G, i.e., $G' \subseteq G$. We utilize an important corollary from spectral bisection algorithm [33] and the properties of laplacian matrix discussed in this subsection to derive the equation $\lambda_2(L(G')) \leq \lambda_2(L(G))$.

Result: $G'\{V',E'\} \subseteq G\{V,E\} \rightarrow \lambda_2(L(G')) \leq \lambda_2(L(G))$, i.e., on application of DFW, the algebraic connectivity, and in effect, density of the AG reduces. Thus, our approach, helps in creating *scalable AGs* (CAG) in a multi-tenant cloud network.

Cost Analysis: *Upped bound* on the cost can be obtained by considering that graph G{V,E} is *fully connected*, in which case, the *micro-segmentation* will not be able to achieve noticeable benefit. The cost of generating the full AG in the absence of DFW, Cost(G), is much higher than in the case of using DFW. The goal of *micro-segmentation*, however, is to ensure that the graph is sparsely connected based on white-listing approach.

Consequently, $Cost(G') = \forall_{i=1}^{N} Cost(G_i) + Cost(DFW)$ - Figure 6 (d) and $Cost(G') << Cost(G)$ since the effort for generation of graphs $G_1, .., G_i$ is computed in parallel with the help of SDN controller. The only additional effort $Cost(DFW)$ is needed for checking DFW rules, and maintaining synchronization between different DFW agents present on individual *segments*.

6 Performance Evaluation

In order to evaluate and measure the performance of our proposed approach. First, we created the system is shown in Figure 2, which is a *OpenStack* [31] based system that is running SDN controller and a number of virtual machines (VMs) connected to OpenFlow switches. Our evaluation consists of evaluating the scalability of AG when the number of vulnerabilities increases, in which S3 proved to have a reduced number of nodes and edges compared to not using S3. Our second experimental evaluation is to measure the AG generation time when the number of services increase, taking into account the generated number of segments as shown in Table 2. Moreover, since S3 is utilizing SDN computing capabilities, we conducted experiments to check how much overhead our algorithm and AG module consume from the SDN controller, which turned out not exceed 11% from the overall SDN bandwidth and an optimal number of segments in a scalable AG.

6.1 Attack Graph scalability Evaluation

As we mentioned in the introduction, the number of vulnerabilities have a direct influence on the AG solubility due to the overhead of managing and analyzing all the security state those vulnerabilities cause. To show how scalable S3 is, we simulated a system with a different number of vulnerabilities as shown in Figure 7, the vulnerabilities in the OpenStack based cloud system. The Figure 7 emphasizes on the relationship between the number of vulnerabilities and the size of the resulted AG in terms of nodes and edges, where the x-axis shows the total number of vulnerabilities in the entire system, and the y-axis shows the number of nodes and edges in the AG. The black and blue lines show the number of nodes and edges in the AG before using our approach (which is equivalent to MulVAL's [27] approach), respectively. The red and

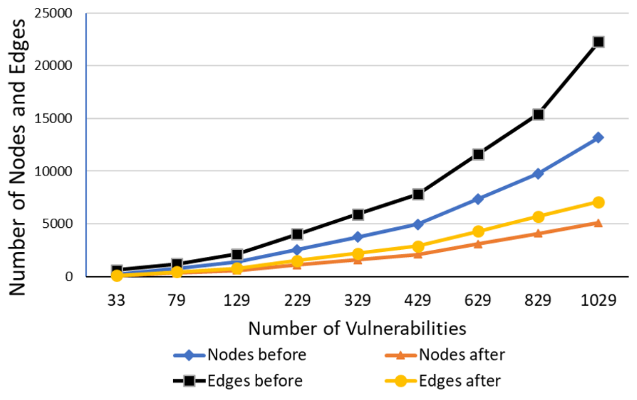


Figure 7: Comparison for the number of Nodes and Edges before and after Using S3. Marked reduction in density achieved using DFW

yellow lines show the number of nodes and edges in the AG after using our approach, respectively. The total number of nodes and edges before using S3 when the system has over 1000 vulnerabilities is about *13k* nodes and *22k* edges. This is due to the absence of $|DFW|$ rules affecting the reachability between the individual components in the system. After using S3, where the exact reachability information that is being enforced by the $|DFW|$ is stated, the number of nodes drop to about *5k* and the number of edges is *7k*, respectively - Figure 7. This is a significant reduction compared to an AG without any $|DFW|$ rules.

6.2 Attack Graph Generation Time and density Reduction Evaluation

It is crucial to generating AG in a timely manner. We created several test cases to test the time required to generate the AG when we have a different number of segments, and a different number of services in each of those segments. We first started to measure the generation time of AG in a system that contains 50-100 services, we inserted a mixture of vulnerabilities in the hosts such that we obtain the provided number of segments shown in Table 2. Thus, in the first experiment, we are testing how much time is needed to generate an AG for a system having 50-100 services with a various number of vulnerabilities on those services that resulted in 5 segments. Moreover, we are measuring the graph density of the resulted AG using the following formula [10]:

$$Density = \frac{|E|}{|V|(|V|-1)}, \tag{3}$$

where $|E|$ is the total number of edges for the AG, and $|V|$ is the total number of nodes or vertices in the AG. Our evaluation and approach show scalable AG generation. For instance, in the last case in Table 2 where the system has 300-500 services

Table 2: Sub-AG generation time, graph density, and the number of nodes and edges for each sub-AG when increasing the number of services.

# Services	**50-100 Services**				**100-200 Services**				**200-300 Services**				**300-500 Services**			
#Segments	5	10	15	20	5	10	15	20	5	10	15	20	5	10	15	20
Time (sec)	2.22	3.88	5.925	8.22	2.386	4.93	7.2112	10.229	3.56	7.15	10.6	13.96	6.46	11.05	15.91	19.7
# Edges	6552	12186	18990	27450	14400	28494	40698	52956	18819	44100	63918	88065	34242	65922	93117	128580
# Nodes	5829	10842	16895	24420	12805	25338	36191	47092	18101	39210	57951	79668	32533	60698	85623	116304
Density	19.3E-05	10.4E-05	6.6E-05	4.6E-05	8.8E-05	4.44E-05	3.1E-05	2.4E-05	5.7E-05	2.9E-05	1.9E-05	1.4E-05	3.2E-05	1.8E-05	1.3E-05	9.5E-06

and it was divided based on the vulnerabilities in the system into 20 segments, the AG generation time is about 20 seconds, which is rational time for such a large system. In Table 3, we show the average time for AG generation and the standard deviation for the 5, 10, 15, and 20 segments cases respectively.

To prove the effectiveness of our DFW-based segmentation approach, S3, we conducted additional experiments to examine the generation time by not considering the segmentation and using a *Firewall* (centralized one); and segmentation by DFW. Table 4 shows the AG generation time with and without segmentation, for the specified number of hosts where the vulnerabilities are simulated to give the shown number of segments. The results when using $|DFW|$ are significantly better than when not using segmentation and using a centralized firewall. This is due to the absence of east/west traffic among running services, which did not specify reachability information between running services. Hence, AG is computing centrally and resulted in a magnificent time.

Table 3: Mean and standard deviation for the AG generation time for the displayed number of segments in Table 2.

#Segments	5	10	15	20
Mean time	3.66	6.75	9.91	13.03
standard deviation	1.96	3.17	4.46	5.04

Table 4: sub-AG scalability generation time by using DFW, and both with and without Segmentation

# Services	**# Segments**	Generation Time without Segmentation (sec)	Generation time using Segmentation (sec)
750	5	3.51	0.872
1450	10	17.344	2.083
2490	15	4980	4.468
3360	20	6720	10.027

6.3 SDN Controller Overhead

Since S3 is based on an SDN-managed data-center network, evaluating the overhead of computing AG using SDN controller is necessary to ensure that the AG generation does not overwhelm the SDN controller, since this may result in service disruption for end users. Specifically, our proposed algorithm 1 *line 15* relies on the SDN controller to compute the sub-AGs for all the obtained segments. Hence, we measure the effect of this operation to the SDN controller bandwidth.

To do this, we used the first case in Table 2, where the system has 50-100 services ($\sim$ 4000 vulnerabilities) running and emulated the scenario. We utilized network throughput measurement tool *iperf* to assess end-to-end bandwidth. Figure 8 shows a comparison of the SDN controller bandwidth overhead before the AG computation takes place and during computation. The evaluation results are an average of *three runs*. The network throughput for a network with 5 segments was around $\sim$11.3 Gbps without micro-segmentation. On incorporating micro-segmentation, the throughput decreases to 9.95 Gbps. Similarly, for the case with 10, 15 and 20 network segments, the throughput drops slightly, as expected.

This drop can be explained as the overhead induced by AG generation in each network segment, and the computation required to merge individual segments into full AG. The worst-case throughput impact on SDN controller was $\sim$10% (20 segment case). This experiment shows that on an average the scalable AG generation process will not impact the SDN controller's performance in a large data-center network.

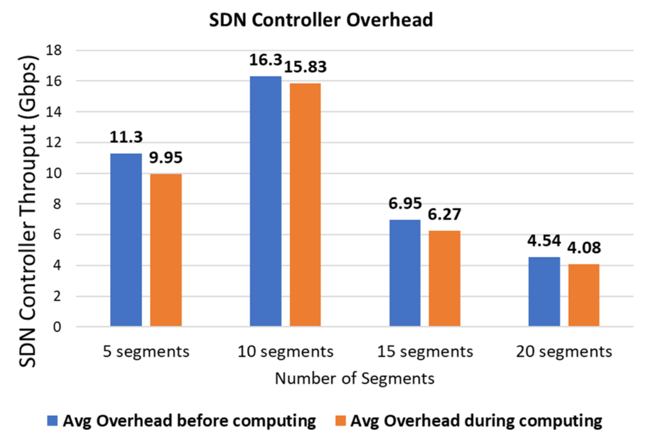


Figure 8: Evaluation of SDN Controller overhead when before computing segmented AG and during computation shows limited overhead

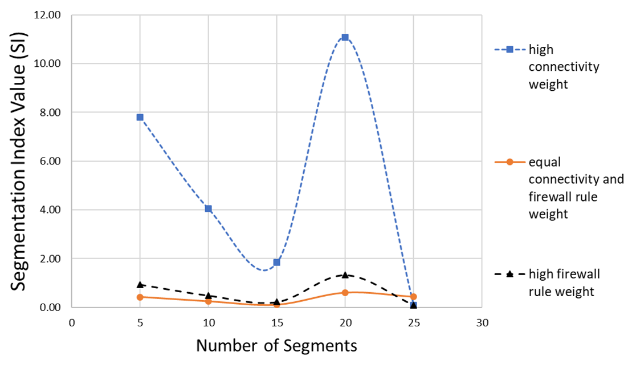


Figure 9: Optimal Number of Segments Evaluation shows 20 segments to have high Segmentation Index (SI)

6.4 Optimal Number of Segment Experiments

We conducted a simulation experiment to identify an optimal number of segments in a large network with 50 services and 50 vulnerabilities. We varied the number of segments from 5 to 25, with an increasing number of DFW rules (3 in case of 5 segments, 17 in case of 25 segments), induced by the increase in the number of segments.

In section 5.2, we showed and discussed a heuristic approach to evaluate what is the optimal number of segmentation based on the derived equation for *Segmentation Index* (SI) 1. The equation showed, depending on the system administrator requirement, how to obtain the optimum number of segments, *i.e.* whether connectivity is more important or $|DFW|$ rule. The Figure 9 shows our experiments, where the blue line indicates the segments have *high connectivity*, i.e. $s_{con} + s_{com} > s_d$ ($\alpha = 0.877, \beta = 0.105$) between services in one segment, the orange line indicates *higher separation* between the segments based on the number of $|DFW|$ *rules*, i.e., $s_{con} + s_{com} < s_d$ ($\alpha = 0.4, \beta = 0.877$). Finally, the black line shows an *equal weight* for *connectivity* and number of $|DFW|$ *rules*, i.e, $s_{con} + s_{com} = s_d$ ($\alpha = 0.5, \beta = 0.5$).

For a small number of segments (5), the connectivity influences the segmentation index (SI=8.0 high connectivity, SI<1 when DFW rules are dominant). This is because of the high degree of intra-segment traffic. As the number of segments, increase (15) the connectivity becomes much less of a factor and drops drastically (SI=2.0 for 15 segments).

From Figure 9, it is shown that the *optimal number of segments turned out to be 20* since the SI > 10 for high connectivity and SI also increases steadily for cases where firewall rules weight is high. This can be explained by the fact that dependencies between vulnerabilities in each segment are reduced, using traffic regulation provided by DFW. Finally, increasing the number of segments more (25) turned out to have a low SI value for all 3 lines, which indicates segments are disconnected from each other.

7 Discussion

Cycle Detection: The dependencies between services in a network can cause cycles in the directed AG. Homer *et. al.* [17] discussed the problems of cycles that can limit the scalability of AGs. The research work takes about *150 ms* for cycle detection over a network with *10 hosts* and *46 vulnerabilities*. S3 utilizes the network connectivity and vulnerability dependency information to detect any cycles present in the final directed AG. We use parallel *nested Depth First Search* [16] over each *sub-AG* in order to identify the cycles present within each segment. The algorithm scales linearly with the number of vulnerabilities present in each network segment. We omit details on *cycle detection* in the paper for the sake of brevity.

Segment Validation and Segmentation Heuristics: We utilized a *Segmentation Index* based sub-AG (segment), that has a validation heuristic approach. The algorithm provides information about the appropriate size of each segment, such that not only the complexity concerns for AG generation are addressed, but also each segment is highly *cohesive* (has the same type of services and vulnerabilities). This will help in the application of security patches to the entire segment. There are other segmentation heuristics, classified under *graph clustering* algorithms, e.g., *k-spanning tree*, which creates k-groups of non-overlapping vertices, *shared nearest neighbor* (SNN) graph. We plan to compare the optimal segmentation heuristic discussed in Section 5.2 with other state-of-the-art graph segmentation heuristics in future work.

Policy Conflicts and SLA Impact: It is important taking into account the *Service Level Agreement (SLA)* that states the relationship between a service provider and client. This *SLA* will have an impact on the $|DFW|$ rule that the system administrator will enforce to create segments and isolate vulnerable services from protected ones. After applying segmentation and deriving a new $|DFW|$ rules, a conflict might exist between the SLA and the $|DFW|$ rules. In effect, the $|DFW|$ rule might cause a service disruption for users. Security policy conflict [14,29] handling, however, is another area of research that will be considered as a part of future work.

8 Conclusion

Attack graph scalability and granular security enforcement are key problems in data-centric networks today. We provide a SDN-based *micro-segmentation* approach using *S3* framework for addressing these issues. S3 enforces granular security policies in the data-center network to deal with threats such as *lateral movement* of the attack. S3 reduces the number of security states in the network by reducing attack graph density and generation time as evident from section 6.2. The *micro-segmentation* approach is capable of establishing and generating a scalable AG for a large network - Section 6.1. Moreover, the impact on the SDN controller because of *miro-segmentation* is limited, as proved from the experimental analysis in Section 6.3. We also identified optimal number of segments using *Segmentation Index* (SI) method, which can ensure high-quality (cohesive) segments with fine-grained access control policies across segments - Section 5.2. The current research work doesn't identify the security policy conflicts that can be induced by co-dependency between *micro-segmentation* policies. Additionally, we have not compared our segmentation method with a diverse set of graph segmentation/clustering heuristics. In the future, we plan to address these limitations.

Acknowledgment

All authors are thankful for research grants from Naval Research Lab N00173-15-G017, N0017319-1-G002 and National Science Foundation US DGE-1723440, OAC-1642031, SaTC-1528099. Special thank to Jim Kirby from NRL for the valuable feedback on the paper. Also, Abdulhakim Sabur is a scholarship recipient from Taibah University through Saudi Arabian Cultural Mission (SACM).

References

[1] Paul Ammann, Duminda Wijesekera, and Saket Kaushik. Scalable, graph-based network vulnerability analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 217–224. ACM, 2002.

[2] Ningyuan Cao, Kun Lv, and Changzhen Hu. An attack graph generation method based on parallel computing. In *International Conference on Science of Cyber Security*, pages 34–48. Springer, 2018.

[3] Ramaswamy Chandramouli and Ramaswamy Chandramouli. Secure virtual network configuration for virtual machine (vm) protection. *NIST Special Publication*, 800:125B, 2016.

[4] Ioannis Chochliouros, Anastasia S Spiliopoulou, and Stergios P Chochliouros. Methods for dependability and security analysis of large networks. In *Encyclopedia of Multimedia Technology and Networking, Second Edition*, pages 921–929. IGI Global, 2009.

[5] Ankur Chowdhary, Sandeep Pisharody, and Dijiang Huang. Sdn based scalable mtd solution in cloud network. In *Proceedings of the 2016 ACM Workshop on Moving Target Defense*, pages 27–36. ACM, 2016.

[6] Chun-Jen Chung, Pankaj Khatkar, Tianyi Xing, Jeongkeun Lee, and Dijiang Huang. Nice: Network intrusion detection and countermeasure selection in virtual network systems. *IEEE transactions on dependable and secure computing*, 10(4):198–211, 2013.

[7] Cisco. Trends in Data Center Security. url = https://blogs.cisco.com/security/trends-in-data-center-security-part-1-traffic-trends, May 2014. Online; accessed 20 Dec 2018.

[8] Eric Cole. Detect, contain, and control cyberthreats. *SANS Institute, June*, 2015.

[9] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. Introduction to algorithms mit press. *Cambridge, MA*, page 819, 2001.

[10] Reinhard Diestel. *Graduate texts in mathematics*. Springer-Verlag New York, Incorporated, 2000.

[11] Joseph C Dunn. Well-separated clusters and optimal fuzzy partitions. *Journal of cybernetics*, 4(1):95–104, 1974.

[12] Karel Durkota, Viliam Lisỳ, Branislav Bosanskỳ, and Christopher Kiekintveld. Optimal network security hardening using attack graph games. In *IJCAI*, pages 526–532, 2015.

[13] Massimo Ferrari. Release: Vmware nsx 6.1. 2014.

[14] Hazem Hamed and Ehab Al-Shaer. Taxonomy of conflicts in network security policies. *IEEE Communications Magazine*, 44(3):134–141, 2006.

[15] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[16] Gerard J Holzmann, Doron A Peled, and Mihalis Yannakakis. On nested depth first search. *The Spin Verification System*, 32:81–89, 1996.

[17] John Homer, Xinming Ou, and David Schmidt. A sound and practical approach to quantifying security risk in enterprise networks. *Kansas State University Technical Report*, pages 1–15, 2009.

[18] Jin B Hong and Dong Seong Kim. Performance analysis of scalable attack representation models. In *IFIP International Information Security Conference*, pages 330–343. Springer, 2013.

[19] Jin B Hong, Dong Seong Kim, Chun-Jen Chung, and Dijiang Huang. A survey on the usability and practical applications of graphical security models. *Computer Science Review*, 26:1–16, 2017.

[20] Hongxin Hu, Wonkyu Han, Gail-Joon Ahn, and Ziming Zhao. Flowguard: building robust firewalls for software-defined networks. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 97–102. ACM, 2014.

[21] Dijiang Huang, Ankur Chowdhary, and Sandeep Pisharody. *Software-Defined Networking and Security: From Theory to Practice*. CRC Press, 2018.

[22] Pierr Johnson. With The Public Clouds Of Amazon, Microsoft And Google, Big Data Is The Proverbial Big Deal. url=https://www.forbes.com/sites/johnsonpierr/2017/06/15/with-the-public-clouds-of-amazon-microsoft-and-google-big-data-is-the-proverbial-big-deal/409452d02ac3, 2017. Online; accessed 3 Mar 2019.

[23] Kerem Kaynar and Fikret Sivrikaya. Distributed attack graph generation. *IEEE Transactions on Dependable and Secure Computing*, 13(5):519–532, 2016.

[24] Peter Mell and Richard Harang. Minimizing attack graph data structures.

[25] Oussama Mjihil, Dijiang Huang, and Abdelkrim Haqiq. Improving attack graph scalability for the cloud through sdn-based decomposition and parallel processing. In *International Symposium on Ubiquitous Networking*, pages 193–205. Springer, 2017.

[26] Nilesh Mojidra. Stateful vs. Stateless Firewalls. url=https://www.cybrary.it/0p3n/stateful-vs-stateless-firewalls/, 2016. Online; accessed 20 Sep 2018.

[27] Xinming Ou, Sudhakar Govindavajhala, and Andrew W Appel. Mulval: A logic-based network security analyzer. In *USENIX Security Symposium*, pages 8–8. Baltimore, MD, 2005.

[28] Justin Gregory V Pena and William Emmanuel Yu. Development of a distributed firewall using software defined networking technology. In *Information Science and Technology (ICIST), 2014 4th IEEE International Conference on*, pages 449–452. IEEE, 2014.

[29] Sandeep Pisharody, Janakarajan Natarajan, Ankur Chowdhary, Abdullah Alshalan, and Dijiang Huang. Brew: A security policy analysis framework for distributed sdn-based cloud environments. *IEEE Transactions on Dependable and Secure Computing*, 2017.

[30] Dhaval Satasiya, Rupal Raviya, and Hiresh Kumar. Enhanced sdn security using firewall in a distributed scenario. In *Advanced Communication Control and Computing Technologies (ICACCCT), 2016 International Conference on*, pages 588–592. IEEE, 2016.

[31] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. Openstack: toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3):38–42, 2012.

[32] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette M Wing. Automated generation and analysis of attack graphs. In *null*, page 273. IEEE, 2002.

[33] Horst D Simon. Partitioning of unstructured problems for parallel processing. *Computing systems in engineering*, 2(2):135–148, 1991.

[34] Su Zhang, Xinming Ou, and John Homer. Effective network vulnerability assessment through model abstraction. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 17–34. Springer, 2011.