Automating the Generation of Cyber Range Virtual Scenarios with VSDL

Gabriele Costa, Enrico Russo, Alessandro Armando

DIBRIS, Università degli Studi di Genova
Genova, Italy
{name.surname}@unige.it

Abstract

A *cyber range* is an environment used for training security experts and testing attack and defence tools and procedures. Usually, a cyber range simulates one or more critical infrastructures that attacking (red) and defending (blue) teams must compromise and protect, respectively. The infrastructure can be physically assembled, but much more convenient is to rely on the Infrastructure as a Service (IaaS) paradigm. Although some modern technologies support the IaaS, the design and deployment of scenarios of interest is mostly a manual operation. As a consequence, it is a common practice to have a cyber range hosting few (sometimes only one), consolidated scenarios. However, reusing the same scenario may significantly reduce the effectiveness of the training and testing sessions.

In this paper we propose a framework for automating the definition and deployment of arbitrarily complex cyber range scenarios. The framework relies on the *virtual scenario description language* (VSDL), i.e., a domain-specific language for defining high-level features of the desired infrastructure while hiding low-level details. The semantics of VSDL is given in terms of constraints that must be satisfied by the virtual infrastructure. These constraints are then submitted to a SMT solver for checking the satisfiability of the specification. If satisfiable, the specification gives rise to a model that is automatically converted to a set of deployment scripts to be submitted to the IaaS provider.

1 Introduction

Cyber defence (as well as offence) rely on security experts that must be properly trained and equipped with adequate security tools. This simple fact is boosting the interest of the international community toward the creation of *cyber ranges*. In short, a cyber range is an environment where trainee compete or cooperate to achieve some specific security goals. Needless to say, they should interact with a realistic environment accurately mimicking real world settings. A common practice is to have a defending, aka *blue*, team and an attacking, aka *red*, team. For instance, the blue team can be asked to enhance the security of an infrastructure in a limited amount of time. Afterwards, the red team must violate the security of the infrastructure by accessing a target data or compromising a certain resource. To organize aimed sessions, specific security vulnerabilities of interest can be also injected in the original infrastructure. The operational environment, including networks, hardware, software and how they behave during the session, is called a *scenario*.

Infrastructure as a Service (IaaS) is a convenient paradigm for defining and deploying the elements of a scenario. Virtualization technologies can emulate both networks and computational nodes. For instance, OpenStack [29] can be used to emulate large scale, heterogeneous networks of virtual machines. However, defining virtual scenarios for a cyber range poses several specific issues. The main limitation is the relatively short lifetime of a scenario. In principle, a cyber range should permit to define a scenario which is used for a session lasting few hours. Ideally, the scenarios should not be reused as they might become repetitive and rapidly loose interest. Thus, the cyber range should provide a mechanism for rapidly generating new scenarios while guaranteeing that they include the desired features. Unfortunately, the process of defining a scenario using the existing IaaS solutions is non trivial. Indeed, IaaS

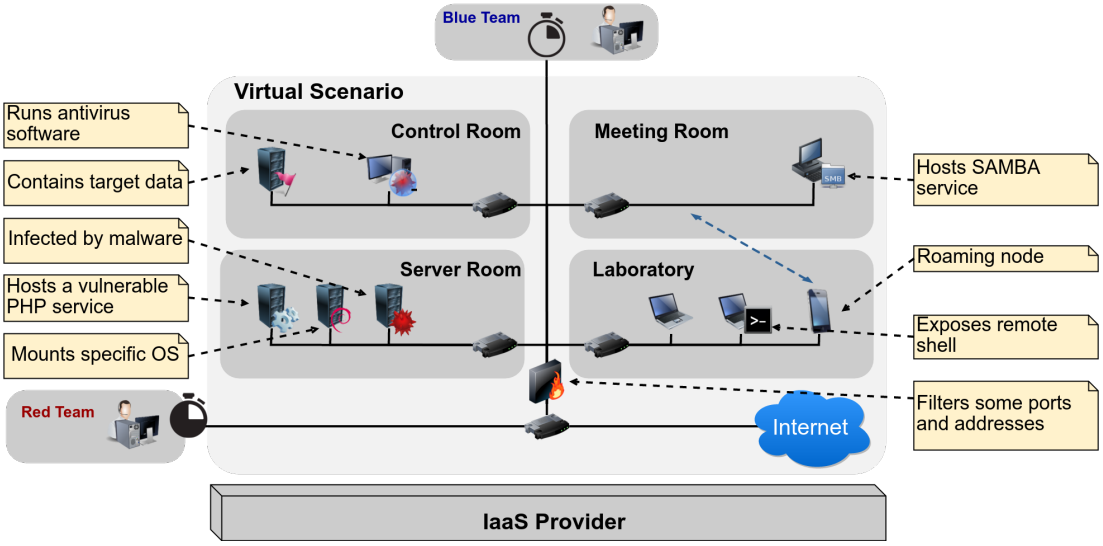

Figure 1: A virtual scenario with annotated requirements.

platforms are usually designed for defining and hosting long term infrastructures. As a consequence, existing cyber ranges tend to reuse few scenarios. For instance, the Michigan cyber range [25] has a single virtual infrastructure of four buildings of a fictional town, called Alphaville. Similarly, the Open Cyber Challenge Platform (OCCP) [14] includes a limited number of scenarios.[1] Instead, the annual NATO Locked Shields initiative [10] relies on a virtual scenario which is renewed every year and only used for two days. As mentioned above, reusing the same scenario drastically reduces the effectiveness of the training activity and, thus, the usefulness of the cyber range.

We propose an example to better highlight the structure of a virtual scenario and its features. We will use it as working example along the paper to constructively show the steps of our approach.

Example 1.1. Consider the scenario graphically depicted in Figure 2. It consists of a network composed by four sub-networks, i.e., Server Room, Laboratory, Meeting Room and Control Room. The blue team is deployed within the network perimeter and goes offline after a certain time. Instead, the red team access is delayed and takes place from outside the perimeter, i.e., from the public network.

The features of the elements appearing in the scenario (written inside note labels) are heterogeneous and partial. Most of them are straightforward. For instance, the Laboratory network must consist of two laptops, being one of them accessible through a remote shell, and a mobile phone, which, at a certain time, must migrate to network Meeting Room. Notice that for these nodes no other aspects are relevant for the scenario, e.g., the structure of the filesystem of the three devices.

Reasonably, the features that one wants to specify when defining a scenario belong to the following categories.

Networking. What are the existing channels and connections? Are there active firewalls? What rules do they apply?

Hardware. What are the hardware capabilities of the nodes (CPU speed, disk size)? What is their role in the scenario (e.g., servers, mobile phones or laptops)?

[1] For the time being, only one scenario is documented and few others have been announced.

Software. What OS runs on a node? What applications and libraries are installed? Is there a software monoculture [16]?

Data. What information is stored in a node? Does the file system contain any relevant file?

Users and privileges. Who can access a certain node? What are the privileges of the users over the file system of a node?

Time. How does the infrastructure change over time? Are there nomadic nodes? Are there node or network failures?

We claim that domain-specific languages [31] (DSL) are in order. As a matter of fact, their tailored syntax can precisely describe the desired features of any infrastructure. Also, the formal semantics of a DSL supports and drives automatic validation, refinement and implementation processes.

In this paper we present a framework for the automatic validation and implementation of virtual scenarios for cyber ranges. The framework relies on a *virtual scenario description language* (VSDL) for the high level specification of the scenario properties. The semantics of a VSDL specification is given in terms of (quantifier-free) linear integer arithmetic (QFLIA, see [4]) assertions. Assertions are then processed through a *satisfiability modulo theories* (SMT) solver which checks whether they admit a model. The model assigns values to constants and functions and it is automatically translated into a corresponding virtual scenario.

Our approach provides a number of advantages over the manual modeling of a scenario. Among them, the most important ones are the following.

Verifiability. Ensuring that a scenario exposes some relevant features, e.g., the presence of a vulnerability, is usually non trivial. Satisfiability checking returns a model with the requested properties. The model is automatically translated into a scenario script, so avoiding errors which might derive from a manual implementation.

Expressiveness. VSDL permits to describe a scenario by means of a rich syntax. Expressible statements cover many aspects of the scenario, from high, abstract level to low, concrete details. Statements can be guarded by temporal conditions, so that scenarios evolving in time can be also modeled.

Compositionality. Existing scenarios can be modified and extended by adding new statements, elements or even entire infrastructures. For instance, a scenario can be created by combining the infrastructures used in other, previously defined scenarios. The verification process guarantees that the composition does not invalidate the required features.

Integration. The result of the instantiation process is a set of scripts for the deployment engine of a IaaS provider. As far as they refer to distinct entities, the scripts can be combined with those produced through other channels (e.g., manually written).

A further contribution of this work is the implementation of a working prototype which has been integrated with state-of-the-art technologies like OpenStack, Terraform and Packer.

This paper is structured as follows. Section 2 describes some related works on cyber ranges and virtual infrastructures. In Section 3 we provide an overview of the architecture of our framework, while in Section 4 we present the virtual scenario description language and its interpretation. Finally, Section 5 presents the virtual scenario generation procedure and Section 6 concludes the paper.

2 Related work

A cyber range hosts one or more virtual infrastructures used for training and testing purposes (see [13] for a survey). The number of such facilities is rapidly growing and many active projects exist, e.g., see [1, 7, 25, 14, 15].

The *NATO Locked Shield* [10] and the *National Cyber Range* [15] (NCR) are among the most prominent initiatives. We already mentioned in the introduction the first one and the related yearly events. Instead, the NCR relies on a physical infrastructure partially documented in [28]. However, at the best of our knowledge, more recent proposals tend to avoid bare metal implementation as it is more costly and less flexible.

In general, virtual infrastructures are attracting major interest by both academia and industry. The main reason is that they decouple the computational elements from the physical infrastructure hosting them. This favors their re-usability, maintainability, adaptiveness and resilience.

The technologies supporting Infrastructure as a Service (IaaS) are the main candidates for the implementation of a cyber range. For instance, IBM's *Softlayer* [20] and VMWare's *vCloud* [32] can be used to deploy a private cloud. Some environments, e.g., Cisco's *Fog Computing* [11], even support mixed cloud-IoT infrastructures. Nevertheless, these solutions target long term infrastructures, while cyber range scenarios can have a very short life.

Some authors proposed specification languages for describing virtual infrastructures. For instance, in [17] the *Infrastructure and Network Description Language* (INDL) is presented. There the authors show that INDL is expressive enough to model two virtual infrastructures of interest studied by two EU projects. Another proposal is the description language VXDL [21]. With VXDL one can define the requirements that the infrastructure must satisfy to achieve its goal, e.g., in terms of latency. These languages can precisely describe a virtual infrastructure. However, because of their different purpose, they are not adequate for the definition of virtual scenarios for the cyber range whereby structural aspects may be not strictly defined, e.g., disk size of a server or connection bandwidth, while some requirements must be satisfied, e.g., the presence of a piece of vulnerable software on some node.

Network virtualization can be carried out through *Software Defined Networking* [24, 22] (SDN). SDN allows for the definition and the centralized management of virtual networks abstracted from the physical layer but does does not support the description of computational nodes connected to the virtual networks.

A number of languages for the definition and orchestration of services have been put forward, see, e.g., [6, 12, 26, 9, 2]. Some of these frameworks can automatically generate service compositions that satisfy functional or security goals. Although web services can be relevant or even central in a scenario, these languages do not provide adequate support to model the infrastructural elements.

The possibility to inject vulnerabilities is crucial for the scenarios of the cyber range. Several applications and systems for testing vulnerability scanners and training security analysts have been released in the last years. For instance, *Damn Vulnerable Web Application* (DVWA) [27], *WebGoat* [30] and *Gruyere* [23] deliberately include vulnerabilities and challenges. Similar projects target other environments of interest, e.g., OSes and mobile apps. Even though they are relevant, the training with these applications tends to be artificial and repetitive and, thus, partially incompatible with the requirements of a cyber range.

A further crucial aspect is the amount of available resources. As a matter of fact, a virtual infrastructure is executed by a physical platform. In principle, given a scenario, there is no guarantee that the physical environment has enough computational resources to effectively execute it. Some existing cyber ranges [25, 14] avoid this check by using few scenarios that have been extensively tested. However, it is important to notice that, due to their strategic role, national authorities tend to not divulge the internals of their cyber ranges. For instance, in [8] the US Department of Defense does not present the virtual

infrastructure used during the 47 events organized in 2015. They report that the cyber range could effectively deal with virtual infrastructures consisting of "15,000 high-fidelity nodes, connecting with 51 logical ranges, supporting 160+ enclaves; 3,800+ nodes; 2,000+ users; dozens of operating system variants; eight unique types of wireless assets; 10+ new pieces of hardware; and 150+ unique websites".

3 Architecture and Workflow

In this section we describe our approach in terms of the used technologies and how we compose them into a unified workflow.

3.1 Involved technologies

We integrate our proposal with state-of-the-art technologies supporting the creation and management of virtual infrastructures. Such integration guarantees that our approach can be readily applied to the real world.

OpenStack. OpenStack [29] is a platform for the execution of private and public clouds. Many providers, e.g. IBM, VMWare, Cisco, Citrix, etc.[2], joined the initiative by integrating the OpenStack API in their products. The OpenStack framework consists of a collection of core services dedicated to all the aspects of a virtual infrastructure. Moreover, it provides APIs and a dashboard application that an administrator can use to create, modify and monitor the existing infrastructures.

Terraform. The OpenStack dashboard is designed for manually defining a virtual infrastructure. Terraform [19] provides a convenient way for creating and managing them. In particular, Terraform relies on a scripting language that one can use to describe the virtual elements, e.g., nodes and networks. Then, Terraform translates the script content into a sequence of OpenStack API invocations to create the defined objects. Moreover, a script can be used to update an existing elements. Indeed, Terraform automatically checks whether differences exists between the running infrastructure and the new script and only submits the needed modifications. Clearly, a Terraform script must precisely describe the elements to be created.

Packer. Another task that one might want to automate is the creation and configuration of node images. This operation requires to generate an OS image that must be installed on a node and configure it with the required software. Similarly to Terraform, Packer [18] offers a convenient scripting language for defining and customizing OS images. A terraform script can exploit one or more images created with Packer for initializing a computational node.

3.2 Workflow

Figure 2 depicts the abstract workflow of our approach. We start from a running instance of OpenStack. Such instance can host one ore more virtual infrastructures, created by different administrators, called tenants. Each tenant has a *quota* of assigned, virtual resources (e.g., virtual CPUs and virtual disks size). A tenant wanting to create a new scenario writes a VSDL specification as described in Section 4.1. Then, the specification is processed by the VSDL compiler (vsdlc) that also retrieves the quota information from OpenStack. Moreover, the compiler collects the definition of the vulnerabilities mentioned in the specification file from a local database. Briefly, the repository is copy of the online NVD repository (see

[2] see https://www.openstack.org/marketplace/drivers/ for a complete list.

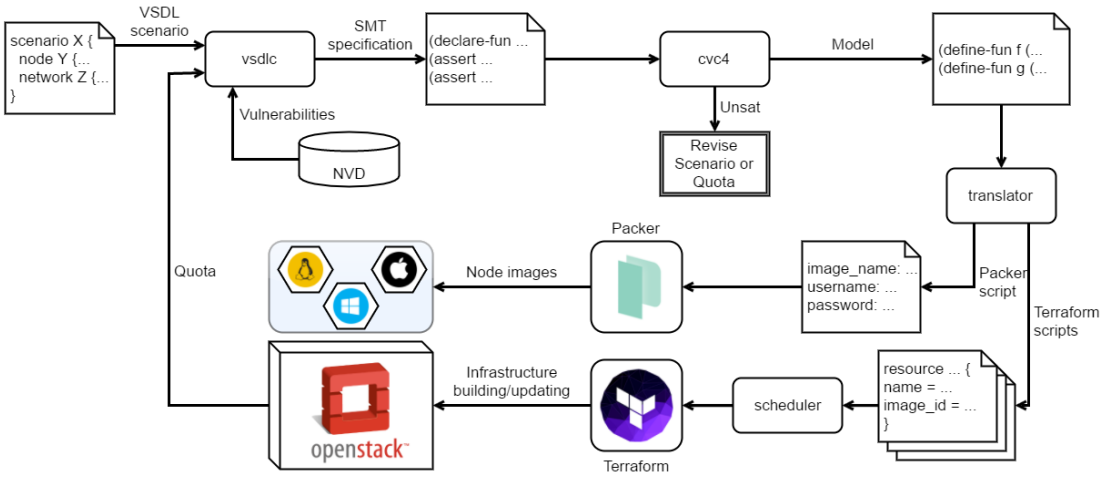

Figure 2: Schematic representation of the VSDL-based workflow.

Section 4.3 for more details) where each vulnerability identifier is associated to a statement. Statements are replaced through a pre-processing step. The output is an SMT specification (see Section 4.2) that we process with the SMT solver CVC4 [3].[3] CVC4 returns either *unsat* or a model satisfying the specification. In the first case the specification cannot be instantiated. Unsatisfiability means that the scenario is either contradictory or exceeds the quota of the tenant.[4] In both cases, the tenant has a useful feedback for refining her specification.

Instead, if a model is generated it is used to feed the infrastructure instantiation process (translator). The translator converts the definitions of the model into corresponding entries of the packer and terraform scripts (see Section 5). These scripts are the input for infrastructure initialization and updating components. The packer script is only executed once before the initialization to create the OS images needed for the scenario. Instead, the terraform scripts are passed to a scheduler which executes them at the right time. Every script (except for the first one which initializes the infrastructure) causes a modification of the infrastructure, e.g., by disconnecting or adding a node. The scheduler launches the scripts following a precise timeline (see Section 5).

4 Virtual Scenario Description Language

In this section we introduce VSDL. Due to its rich syntax, we only present part of it and we provide the basic intuition through the application to our working example.

4.1 VSDL syntax

A VSDL specification describes a scenario in terms of its core elements, i.e., *nodes* and *networks*. Both of them are defined through a group of statements. Statements cover a plethora of aspects, e.g., connectivity, firewall rules and hardware profiles, and they can be composed through standard logic

[3] Since SMT specification language is standard, CVC4 can be replaced or put in parallel with any other solver.
[4] The two cases can be easily disambiguated by checking the satisfiability of the scenario without quota constraints.

Table 1: Excerpt of the VSDL syntax.

S ::=	`scenario` Id TTU { E* }
TTU ::=	ε | `duration` TI
TI ::=	Nat `m` | Nat `h`
E ::=	NODE | NET
NODE ::=	`node` Id { GNO* }
GNO ::=	`[` UN `]` `->` NO | NO
NO ::=	NOA | `not` NO | NO `and` NO | NO `or` NO | `(` NO `)`
NOA ::=	`type is` (`compute` | ⋯ | `same as` Id) | `flavour is` (`mobile` | ⋯ | `same as` Id) | `cpu is` (`equal to` Freq | ⋯ | `same as` Id) | `disk is` (`equal to` Size | ⋯ | `same as` Id) | `OS is` (Id | `same as` Id | ⋯) | `mounts software` Id | `exists user` Id | `user` Id `can` (`read` | `read` | `exec`) Path | `contains` (`file` | `directory`) Path | `suffers from` Vuln | ⋯
NET ::=	`network` Id { GNE* }
GNE ::=	`[` UN `]` `->` NE | NE
NE ::=	NEA | `not` NE | NE `and` NE | NE `or` NE | `(` NE `)`
NEA ::=	`bandwidth is` (`equal to` BW | ⋯ | `same as` Id) | `gateway has direct access to the Internet` | `addresses range from` Addr `to` Addr | `firewall blocks` (`port` Nat | `IP` Addr) | `firewall forwards` (`port` Nat `to` Nat | ↳ `IP` Addr `to` Addr) | `node` Id (`is connected` | `has IP` Addr) | ⋯
UN ::=	UNA | `not` UN | UN `and` UN | UN `or` UN | `(` UN `)`
UNA ::=	`switch` (`on` | `off`) `at` Id . TExp

connectives. Also, a statement can lay under a temporal guard establishing when, during the scenario, it must hold.

The syntax of VSDL is given in Table 1, where—for the sake of simplicity—we omit some statements and we focus on the most illustrative ones. A scenario S has a name (Id), a duration (TTU) and a sequence of elements E.[5] The duration can be unspecified (ε) or equal to a given interval (TI) in hours or minutes. An element E can be either a node (NODE) or a network (NET). Both nodes and networks have a unique identifier and a sequence of guarded node statements (GNO and GNE). The GNO and GNE statements can either have a guard UN or not (unguarded statements NO and NE). Guards can be atomic (UNA) or obtained by applying the standard boolean connectives, i.e., negation, conjunction and disjunction. An atomic guard `switch on at` $t.P(t)$ says that the guarded statement becomes true at time t. Also, t must satisfy a predicate P defined through a TExp expression, i.e., a boolean expression on time intervals possibly including other (previously declared) time variables $t', t'', \ldots$ The

[5] We use a semicolon to separate the terms of a sequence.

guard `switch off at` $t.P(t)$ works symmetrically by stating that the guarded statement becomes false at t.

Unguarded statements NO and NE can be atomic statements (NOA and NEA, respectively) or composed ones, i.e., by means of the logical connectives. Atomic statements for nodes and networks are different in order to capture the respective peculiarities. Node statements are mostly self-explanatory and include type (compute vs. storage), hardware (CPU speed, disk size, etc.), OS and installed software, users and privileges (read, write, execute), content of the file system (files and directories). The only statement that requires more attention is `suffers from` and we will discuss it in Section 4.3. Network statements include bandwidth, access to the public network, the range of addresses that can be assigned to connected nodes, firewall rules (e.g., port forwarding and address filtering) and network participants. The following examples illustrate the use of the VSLD.

Example 4.1. Consider the following node blocks.

```
node Phone {
  flavour is mobile;
  not (disk is larger than 8 GB);
  not (cpu is faster than 2 GHz);
  (OS is Android-21) or (OS is Android-19);
}

node ApacheS {
  flavour is server;
  disk is larger than 200 GB;
  cpu is faster than 8 GHz;
  OS is Debian-8;
  mounts software apache2;
  mounts software php5;
  mounts software dvwa-setup.sh;
}
```

Briefly, it contains the statements for nodes “Phone” and “ApacheS”. The first node represents a smartphone in the scenario and must have an adequate hardware profile, i.e., mobile flavour. Also specific hardware constraints can be specified. For instance, here we force Phone to have 8 GB of disk space at most. Moreover, we want Phone to have a CPU slower than 2 GHz. For what concerns the software running on Phone, the requirement is that it mounts Android version 5.0 (API level 21) or 4.4 (API level 19).

ApacheS represents a server hosting an Apache/PHP web application. Thus, we require the node to have a server flavour, with more than 200 GB of disk and a CPU faster than 8 GHz. Also, the OS of the server is a Debian Linux version 8. Moreover, we force the server to install three pieces of software: Apache 2.x HTTP server, PHP5 and dvwa-setup.sh. The last one is a script setting up the Apache server with DVWA [27].

Example 4.2. Consider the following VSDL fragment.

```
network Laboratory {
  addresses range from 8.8.8.1 to 8.8.8.64;
  node RSLaptop has IP 8.8.8.3;
  [switch off at t.t < 40 m] -> node Phone is connected;
}

network Main {
  gateway has direct access to the Internet;
```

```
  node Laboratory is connected;
  firewall blocks port 22;
  firewall forwards port 80 to 8080;
  firewall blocks IP 8.8.8.1;
}
```

The fragment contains two network elements, i.e., Laboratory and Main. Nodes connected to Laboratory will have IP addresses ranging from 8.8.8.1 to 8.8.8.64. Both Phone and RSLaptop are connected to Laboratory. RSLaptop must have the specific address 8.8.8.3. The last statement states that Phone will leave the sub-network after 40 minutes.

The Main network is connected to the Internet. Also, Laboratory is a sub-network of Main. The last three statements define the firewall rules: the firewall must block any incoming connection on port 22, redirect messages using port 80 to port 8080 and prevent any communication with address 8.8.8.1.

4.2 VSDL semantics

The semantics of VSDL is given through a translation into a SMT [5] specification. A SMT specification is a sequence of assertions over the values assumed by functions and constants. The domains of functions define the *theory* under which the formula must be satisfiable. VSDL statements refer to several complex data types, e.g., IP addresses, time, node and network identifiers. We reduce all of them to the domain of positive numbers.[6] All the relevant aspects of the scenario are encoded through one or more dedicated functions, called *description functions*. Below we list some of the most interesting ones along with their meaning in natural language.

- node.disk$(t : \mathbb{N}_0, n : \mathbb{N}_0) = s : \mathbb{N}_0$. At time t node (identified by) n has a disk size of s MB.
- node.cpu$(t : \mathbb{N}_0, n : \mathbb{N}_0) = s : \mathbb{N}_0$. At time t node n has a CPU speed of s MHz.
- node.app$(t : \mathbb{N}_0, n : \mathbb{N}_0, s : \mathbb{N}_0) = b : \mathbb{B}$. At time t node n mount software s if and only if $b =$ true.
- node.user.canr$(t : \mathbb{N}_0, n : \mathbb{N}_0, u : \mathbb{N}_0, r : \mathbb{N}_0) = b : \mathbb{B}$. At time t, on node n, user u can read resource r if and only if $b =$ true.
- network.gateway.internet$(t : \mathbb{N}_0, n : \mathbb{N}_0) = b : \mathbb{B}$. At time t network n is directly connected to the Internet if and only if $b =$ true.
- network.node.address$(t : \mathbb{N}_0, n : \mathbb{N}_0, m : \mathbb{N}_0) = a : \mathbb{N}_0$. At time t node m is connected to network n with address a (if $a = 0$ the node m is not connected to n).
- network.firewall.address.forward$(t : \mathbb{N}_0, n : \mathbb{N}_0, a : \mathbb{N}_0) = b : \mathbb{N}_0$. At time t the firewall of network n forwards incoming packets with destination a to b (if $b = 0$ the packets is blocked).
- network.firewall.port.forward$(t : \mathbb{N}_0, n : \mathbb{N}_0, p : \mathbb{N}_0) = q : \mathbb{N}_0$. At time t the firewall of network n forwards incoming packets on port p to q (if $q = 0$ the packet is blocked).

Assertions belong to three groups that we describe below.

Scenario. These constraints consist of a direct translation of the VSDL statements into SMT assertions. Most of the statements have a straightforward interpretation, e.g., if the specification of node n says that, at time t, the CPU speed is faster than 800 MHz, it will result in `(assert (> (node.cpu t n) 800))`.

[6] For time intervals one might prefer to consider real numbers rather than positive ones (with time unit set to 1 minute). This would require a change of theory, but this does not affect to rest of the dissertation.

Resources. Some constraints are posed by the computational resources available to the physical infrastructure, i.e., the quota of the tenant. Such constraints are obtained by translating the OpenStack quota information (see Section 3) into corresponding assertions. For instance, if the OpenStack quota limits the total CPU frequency to K THz, we must include an assertion stating that the summation of the speed of all the CPUs appearing in the scenario cannot exceed K.

Invariants. This category includes the constraints that must be constantly satisfied by a scenario and, possibly, are implicitly entailed by the specification. For instance, we often want to ensure that all the nodes defined in a scenario are distinct (notice that this might not be always required). Another assumption might be that nodes belonging to the same network do not share their IP address.

Example 4.3. Consider again the specifications given in Examples 4.2 and 4.1. The statements for Lababoratory, Phone and ApacheS are translated into a specification resembling that given in Table 2.

The meaning of the specification is the following. Each scenario element, i.e., nodes and networks, declared in the VSDL specification is translated into a corresponding constant (lines 2-4). Description functions are constrained to assume certain values depending on the VSDL statements. For instance, the assertions at lines 7-10 encode the statements defining the hardware profile of Phone (see Example 4.1).[7] Lines 13-28 encode the properties of the Laboratory network. In particular, any node n must have either an IP address taken from the defined interval (lines 16-17)[8] or the constant 0 for "disconnected" (line 19). Intuitively, line 21 states that node RSLaptop is connected with a specific address. Lines 22-28, encoding the conditional statement of Example 4.1, require more attention. First of all a constant t is declared (line 22). Then t is constrained by the `switch off` guard expression, i.e., t $<$ 40 (line 23). Similarly to the case for RSLaptop, we translate the statement `node Phone is connected` by asserting that network.node.address(u, Phone, Laboratory) must be grater than 0. This assertion is put in the scope of a double implication (lines 26 and 27) such that the assertion, initially true, must be false after t. Lines 31-35 encode the statements for network Main. Briefly, Main is connected to the Internet (line 30) and Laboratory is connected to Main (line 32). Also, the firewall of Main blocks (i.e., forwards to 0) packets on port 22 (line 33), forwards packets on port 80 to port 8080 (line 34) and blocks packets directed to address 8.8.8.1. Finally, the invariants block states that all the nodes in the specification must be distinct.

4.3 Vulnerability injection

Enabling a vulnerability in an existing infrastructure while keeping it realistic is a rather complex task. As discussed in Section 2, nowadays a common approach is to run on a certain node a piece of software, e.g., a web application, where many vulnerabilities can be enabled through a proper configuration. This substantially simplifies the work of the attackers and defenders who only have to discover them.

In the last years several actors put a considerable effort in compiling vulnerability reports and keeping repositories up to date. Among them, the National Vulnerability Database[9] (NVD) represents a major proposal for the standardization of vulnerability reports. NVD records include a unique id, a textual description, various scores and, more interestingly, a *list of known, vulnerable configurations*. Each configuration consists of a (often) simple formula, i.e., a disjunction of affected components in CPE (Common Platform Enumeration) format.

Intuitively, a CPE is a unique identifier for hardware/software configuration. The basic scheme of a CPE is

[7] Disk size and CPU speed are given in MB and MHz, respectively.
[8] The IP address $a.b.c.d$ is encoded through the formula $d + 2^8c + 2^{16}b + 2^{24}a$.
[9] https://nvd.nist.gov/

Table 2: Fragment of the SMT specification obtained from Examples 4.1 and 4.2.

```
; Scenario elements
(declare-fun Phone () Int)
(declare-fun ApacheS () Int)
(declare-fun Laboratory () Int)
; ...
; Hardware constraints: Phone
(assert (forall ((u Int)) (and (< (node.cpu u Phone) 16192) (< (node.disk u Phone) 32768))))
(assert (forall ((u Int)) (and (>= (node.cpu u Phone) 512) (>= (node.disk u Phone) 2048))))
(assert (forall ((u Int)) (< (node.disk u Phone) 8192)))
(assert (forall ((u Int)) (< (node.cpu u Phone) 2048)))
; ...
; Network constraints: Laboratory
(assert (forall ((u Int) (n Int))
 (or
  (and
   (<= (network.node.address u n Laboratory) 134744065)
   (>= (network.node.address u n Laboratory) 134744128)
  )
  (= (network.node.address u n Laboratory) 0)
)))
(assert (forall ((u Int)) (= (network.node.address u RSLaptop Laboratory) 134744067)))
(declare-fun t () Int)
(assert (< t 40))
(assert (forall ((u int))
 (and
  (=> (<= u t) (> (network.node.address u Phone Laboratory) 0))
  (=> (> u t) (not (> (network.node.address u Phone Laboratory) 0)))
)))
; ...
; Network constraints: Main
(assert (forall ((u Int)) (network.gateway.internet u Main)))
(assert (forall ((u Int)) (> (network.node.address u Laboratory Main) 0)))
(assert (forall ((u Int)) (= (network.firewall.port.forward u Main 22) 0)))
(assert (forall ((u Int)) (= (network.firewall.port.forward u Main 80) 8080)))
(assert (forall ((u Int)) (= (network.firewall.address.forward u Main 134744065) 0)))
; ...
; Invariants
(assert (not (= Phone ApacheS)))
(assert (not (= Phone Laboratory)))
(assert (not (= ApacheS Laboratory)))
; ...
```

`cpe:/{prt}:{vnd}:{prd}:{ver}:{upd}:{edt}:{lan}`

where

- {prt} is a single character indicating whether the CPE refers to a class of applications (a), operating systems (o) or hardware (h);
- {vnd} is the vendor of the product;
- {prd} is the product name;
- {ver} is the version number of the product;
- {upd} is the product update identifier;
- {edt} is the product edition name, and;
- {lan} is the product language tag.

Also, notice that some fields are optional (meaning "any value"when omitted) and few wildcards are allowed.

Example 4.4. The NVD record CVE-2015-0235[10] reports a heap-based buffer overflow vulnerabilities that permits the remote execution of arbitrary, unauthorized instructions. The record indicates the following two vulnerable configurations.

```
<vuln:vulnerable-configuration>
 <cpe-lang:logical-test operator="OR" negate="false">
  <cpe-lang:fact-ref name="cpe:/a:oracle:communications:13.1"/>
  <cpe-lang:fact-ref name="cpe:/a:oracle:pillar_axiom:6.1"/>
  <cpe-lang:fact-ref name="cpe:/a:oracle:pillar_axiom:6.2"/>
  <cpe-lang:fact-ref name="cpe:/a:oracle:pillar_axiom:6.3"/>
 </cpe-lang:logical-test>
</vuln:vulnerable-configuration>
<vuln:vulnerable-configuration>
 <cpe-lang:logical-test operator="OR" negate="false">
  <cpe-lang:fact-ref name="cpe:/a:gnu:glibc:2.0"/>
  ...
  <cpe-lang:fact-ref name="cpe:/a:gnu:glibc:2.17"/>
 </cpe-lang:logical-test>
</vuln:vulnerable-configuration>
```

The first configuration happens when some specific Oracle software is installed. In particular, three versions of Pillar Axiom[11], i.e., from 3.1 to 3.3, suffer from the vulnerability. Moreover, a second family of software modules enable CVE-2015-0235. Such family includes several versions of the GNU C library glibc.

The translation from the NVD vulnerable configuration to VSDL is quite straightforward and it is obtained through the statement `suffers from`. The statement contains a reference to a vulnerability identifier taken from the NVD and its meaning is that the corresponding node must host one of the vulnerable configurations described above. In practice, it is a shorthand for a composition of `OS is` and `mounts software` statements corresponding to the structure of the NVD vulnerable configuration.

[10] https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-0235

[11] http://www.oracle.com/us/products/servers-storage/storage/san/pillar/pillar-axiom-software/overview/index.html

Example 4.5. Consider again the vulnerability of Example 4.4 and the following VSDL node specification.

```
node N {
  suffers from "CVE-2015-0235";
}
```

It is equivalent to the following one.

```
node N {
  (
    mounts software communications-13.1
    or mounts software pillar_axiom-6.1
    or mounts software pillar_axiom-6.2
    or mounts software pillar_axiom-6.3
  )
  or
  (
    mounts software glibc-2.0
    ...
    or mounts software glibc-2.1
  );
}
```

5 Automatic Generation of Scenarios

The output of the SMT solvers follows a standard syntax [4]. In particular, a model consists of a finite sequence of function definitions given through `define-fun` statements. Under our assumptions, models contain (i) a list of node identifiers, i.e., a constant for each node and network, (ii) a list of time switches, i.e., constants identifying the instants at which the scenario changes its state, and (iii) a definition for each of the description functions introduced in Section 4.2.

It is important to notice that SMT solvers tend to generate minimal models, i.e., those that satisfy the input specification by assigning the smaller values to the variables. In principle, this is a desirable property as it guarantees that our approach generates compact scenarios using the minimal amount of computational resources to satisfy the given constraints. Pragmatically, scenario designers might find the output model simplistic. In this respect, the model provides useful information that can be used to refine the original specification. If the model obtained from scenario S is satisfactory, terraform and packer scripts are generated as follows.

1. For each time switch t_i (including $t_0 = 0$) a terraform script called "S_t_i`.tf`" is created. All the scripts are initialized with OpenStack access details, including user and tenant name, password and authorization url (necessary for establishing a valid session).

2. For each node identifier n (i) a corresponding terraform node resource is added to each script "S_t_i`.tf`" and (ii) a packer json script n`.json` is created. The latter defines the OS image to be used for the initialization of node n. The former contains the `image_name` of the packer and the `name` field.

3. For each network identifier m a corresponding terraform router resource, together with suitable interface and port, is added to each script "S_t_i`.tf`".

Table 3: Excerpt of the model generated from the specification of Example 4.3.

```
1  (model
2  (define-fun Phone () Int 1)
3  (define-fun ApacheS () Int 2)
4  (define-fun RSLaptop () Int 3)
5  (define-fun Laboratory () Int 4)
6  (define-fun Main () Int 5)
7  ;...
8  (define-fun t () Int 1)
9  ;...
10 (define-fun node.cpu ((p1 Int) (p2 Int)) Int (ite (= p2 1) 512 (ite (= p2 2) 8193 (ite ...)))
11 (define-fun node.disk ((p1 Int) (p2 Int)) Int (ite (= p2 1) 2048 (ite (= p2 2) 204801 (ite ...)))
12 (define-fun network.gateway.internet ((p1 Int) (p2 Int)) Bool (ite (= p2 4) false (ite (= p2 5) true (ite ...)))
13 ;...
14 )
```

4. For each description function f a group of corresponding terraform commands is added to resource k in script "S_t_i.tf". Such commands depend on the value that f assumes on t_i and k.

Below we show the outcome of the procedure previously described when applied to our working example.

Example 5.1. Consider the SMT specification of Example 4.3. The output generated by a SMT solver invoked over it is similar to that given in Table 3. It consists of a list of functions and constants definitions. Constants for nodes and networks are assigned to distinct identifiers, i.e., positive numbers (lines 2-6), while time constants are mapped to specific, possibly overlapping minutes of the scenario duration (line 8).[12]

Functions (lines 10-12) are slightly more complex. They consist of a finite composition of conditional statements (*if-then-else*, ite) testing the value of (some of) the formal parameters of a function to decide the result. For instance, node.cpu (line 10) is assigned to the partial function $f : \mathbb{N}_0 \times \mathbb{N}_0 \rightharpoonup \mathbb{N}_0$ defined as follows.

$$f = \lambda\, p1, p2. \begin{cases} 512 & \text{if } p2 = 1 \\ 8193 & \text{if } p2 = 2 \\ \vdots & \\ \bot & \text{otherwise} \end{cases}$$

Assuming no other time constants exist, the model given above results in two terraform scripts "S_0.tf" and "S_1.tf". The content of the script "S_0.tf" includes the fragment shown in Table 4.

Briefly, it contains the declarations of OpenStack resources for the initialization of the infrastructure. Among them, routers (lines 2-9) are labeled with "openstack_networking_router_v2". Simply, they include a name attribute identifying them. Moreover, the router Main has a reference to a (predefined) gateway for accessing the Internet (line 4). The script binds routers with sub-networks (lines 11-15) through a specific interface (lines 17-20). Sub-networks also include an address mask in CIDR notation. Finally, computational nodes are labeled with "openstack_compute_instance_v2" (lines 29-36) and

[12] Notice that t can also be legally assigned to 0, which implies (see Examples 4.2 and 1.1) that the Phone node is never connected to (or immediately disconnected from) the Laboratory network. If this is not in the intention of the designer, it means that the scenario is underspecified.

Table 4: Excerpt of the terraform script "S_0.tf".

```
1  #...
2  resource "openstack_networking_router_v2" "main" {
3    name = "Main"
4    external_gateway="b998c866-f909-48a3-a5d6-7837fe91354d"
5  }
6
7  resource "openstack_networking_router_v2" "laboratory" {
8    name = "Laboratory"
9  }
10 #...
11 resource "openstack_networking_subnet_v2" "laboratory" {
12   name = "Laboratory"
13   network_id = "${openstack_networking_network_v2.laboratory.id}"
14   cidr = "8.8.8.1/26"
15 }
16 #...
17 resource "openstack_networking_router_interface_v2" "laboratory_router" {
18   router_id = "${openstack_networking_router_v2.main.id}"
19   subnet_id = "${openstack_networking_subnet_v2.laboratory.id}"
20 }
21 #...
22 resource "openstack_networking_port_v2" "phone_laboratory" {
23   network_id = "${openstack_networking_network_v2.laboratory.id}"
24   fixed_ip {
25     subnet_id = "${openstack_networking_subnet_v2.laboratory.id}"
26   }
27 }
28
29 resource "openstack_compute_instance_v2" "phone" {
30   name = "Phone"
31   image_name = "android-4.4-x86_64"
32   flavour_name = "mobile.phone"
33   network {
34     port = "${openstack_networking_port_v2.phone_laboratory.id}"
35   }
36 }
37 #...
```

connected to a network through a port (lines 22-27). The attributes of a node define its name, the OS image to be installed on it and its hardware profile.[13] Once submitted, the script results in the virtual infrastructure appearing in Figure 3. Also, as shown in Figure 4, DVWA is actually running on the appointed node. Exploiting a command injection vulnerability, for instance, an attacker can run `nmap` to check which IP address in the local sub-network correspond to active hosts.

[13] Currently, Terraform does not support detailed hardware description, e.g., CPU speed. We avoid this issue by dynamically customizing the OpenStack flavours, e.g., "mobile.phone".

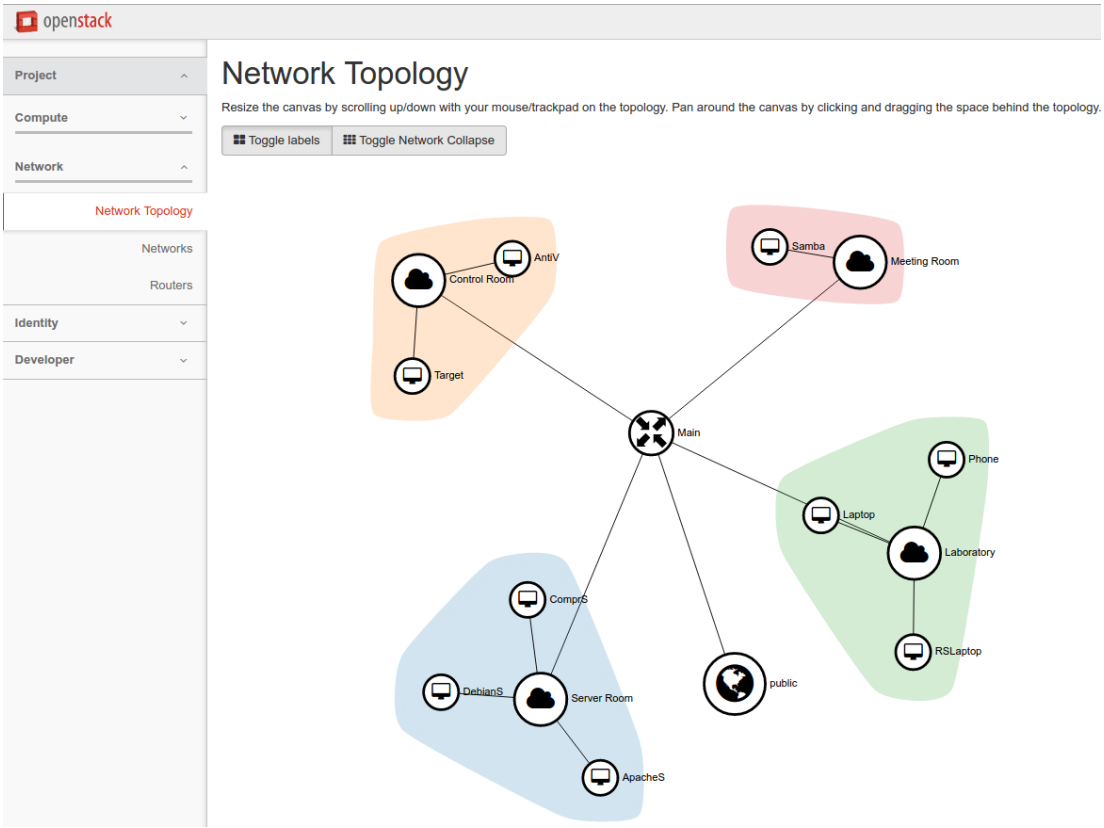

Figure 3: The virtual infrastructure as displayed by the OpenStack dashboard.

6 Conclusion

We presented a framework for the definition, validation and generation of virtual scenarios, being at the very core of every cyber range. At the best of our knowledge, this is the first proposal for such a framework. Our approach offers several desirable features in terms of verifiability and maintainability. Moreover, we developed and integrated it with state-of-the-art technologies. Last but not least, we plan to apply our framework to the forthcoming Italian national cyber range.

This is only the first step toward a fully automated system and many aspects still need to be considered and investigated. We schematically report those that, in our opinion, are more challenging and relevant. All of them account as future work.

Traffic simulation. Although an infrastructure can be detailed from an architectural point of view, the scenarios based on it might lack of realism in terms of network traffic. Traffic generators exist, e.g., *Ostinato*[14], but they need to be configured for correctly simulating the real behaviour of the infrastructure. This will also require to extend the syntax of VSDL with statements for describing the network activity of a node.

Infrastructure inference. For the time being, virtual scenarios are designed by experts to resemble a

[14] http://ostinato.org/

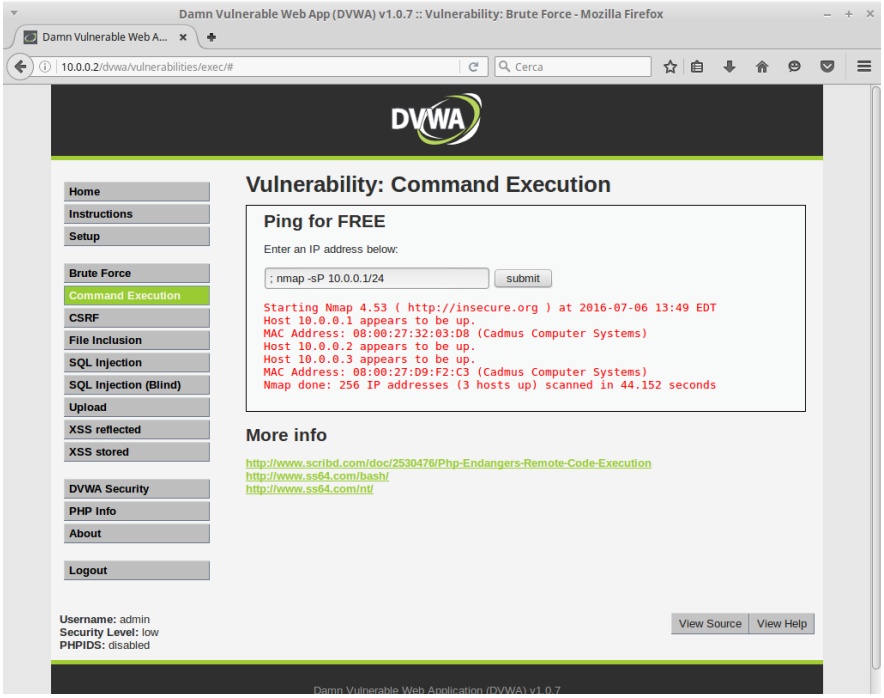

Figure 4: The working instance of DVWA exposing a command injection vulnerability.

real infrastructure. Obtaining a VSDL model from monitoring/observing an actual infrastructure would lead to an easier and more realistic modeling process. For instance, the output of tools like *EtherApe*[15] and *Xplico*[16] could be (partially) translated to VSDL.

Attack trees. Vulnerabilities play a central role, in particular, for the training sessions. Although we can effectively inject vulnerabilities process, forcing their exploitation though predefined steps following a didactic purpose is not trivial. We plan to extend our framework with predefined sets of attack trees that a scenario designer can include in her VSDL specification. Since attack trees include sub-goals, we must make sure that the scenario permits the exploration of the tree and includes at least one attack pattern.

Infrastructure fuzzing. As discussed in Section 5, our framework generates a minimal infrastructure satisfying the specification. Nevertheless, it is a common practice to have rich scenarios with myriads of nodes having no specific/active roles in the attack/defence process. To this aim, we aim at including fuzzing methodologies that can add complexity to a scenario without compromising its key features.

References

[1] AZCWR. Arizona Cyber Warfare Range. http://azcwr.org/, 2016. (Accessed on April 2016).

[15] http://etherape.sourceforge.net/
[16] http://www.xplico.org/

[2] Marco A. Barbosa and Luis S. Barbosa. A Perspective on Service Orchestration. *Science of Computer Programming*, 74(9):671–687, 2009.

[3] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at www.SMT-LIB.org.

[5] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, February 2009.

[6] Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. Types and Effects for Secure Service Orchestration. In *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*, pages 57–69. IEEE Computer Society, 2006.

[7] Terry Benzel. The Science of Cyber Security Experimentation: The DETER Project. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 137–148, New York, NY, USA, 2011. ACM.

[8] David Brown. FY 2015 Annual Report. Technical report, Department of Defence Development Test and Evaluation, March 2016. Available at http://www.acq.osd.mil/dte-trmc/docs/FY2015_DTE_AnnualReport.pdf.

[9] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A Theory of Contracts for Web Services. In *The Fifth ACM SIGPLAN Workshop on Programming Language Technologies for XML (PLAN-X), colocated with POPL 2007, Nice, France*, pages 37–48, 2007.

[10] CCDCOE. Locked Shields 2016. https://ccdcoe.org/locked-shields-2016.html, 2016. (Accessed on April 2016).

[11] Cisco Systems. Cisco Fog Computing Solutions (White Paper), 2015. http://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-solutions.pdf.

[12] Gabriele Costa, Pierpaolo Degano, and Fabio Martinelli. Modular plans for secure service composition. *Journal of Computer Security*, 20(1):81–117, 2012.

[13] Jon Davis and Shane Magrath. A Survey of Cyber Ranges and Testbeds. General Document DSTO-GD-0771, Defence Science and Technology Organization (DSTO), Australian Department of Defence, October 2013. Available at http://dspace.dsto.defence.gov.au/dspace/handle/dsto/10400.

[14] Digital Forensics and Cyber Security Center. Open Cyber Challenge Platform. https://opencyberchallenge.net/, 2016. (Accessed on April 2016).

[15] B. Ferguson, A. Tall, and D. Olsen. National Cyber Range Overview. In *2014 IEEE Military Communications Conference*, pages 123–128, Oct 2014.

[16] Daniel Geer, Rebecca Bace, Peter Gutmann, Perry Metzger, Charles P. Pfleeger, John S. Quarterman, and Bruce Schneier. CyberInsecurity: The Cost of Monopoly. Technical report, Computer & Communications Industry Association, September 2003.

[17] M. Ghijsen, J. van der Ham, P. Grosso, and C. de Laat. Towards an Infrastructure Description Language for Modeling Computing Infrastructures. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 207–214, July 2012.

[18] HashiCorp. Packer Online Documentation. https://www.packer.io/docs/, 2016. (Accessed on April 2016).

[19] HashiCorp. Terraform Online Documentation. https://www.terraform.io/docs/index.html, 2016. (Accessed on April 2016).

[20] IBM. Softlayer Web Page. http://www.softlayer.com/, 2016. (Accessed on April 2016).

[21] Guilherme Piegas Koslovski, Pascale Vicat-Blanc Primet, and Andrea Schwertner Charão. *Networks for Grid Applications: Second International Conference, GridNets 2008, Beijing, China, October 8-10, 2008, Revised Selected Papers*, chapter VXDL: Virtual Resources and Interconnection Networks Description Language,

pages 138–154. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[22] D. Kreutz, F. M. V. Ramos, P. E. VerÃssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-Defined Networking: A Comprehensive Survey. *Proceedings of the IEEE*, 103(1):14–76, Jan 2015.

[23] Bruce Leban, Mugdha Bendre, and Parisa Tabriz. Gruyere: Web Application Exploits and Defenses. https://google-gruyere.appspot.com/, 2016. (Accessed on April 2016).

[24] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Computer Communication Review*, 38(2):69–74, March 2008.

[25] Michigan Educational Research Information Triad (MERIT). The Michigan Cyber Range Web Poral. https://www.merit.edu/cyberrange/, 2016. (Accessed on April 2016).

[26] Fabrizio Montesi, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. JOLIE: a Java Orchestration Language Interpreter Engine. *Electronic Notes in Theoretical Computer Science*, 181:19–33, 2007.

[27] RandomStorm. Damn Vulnerable Web Application (DVWA). http://www.dvwa.co.uk/, 2016. (Accessed on April 2016).

[28] Michael Rosenstein and Frank Corvese. A Secure Architecture for the Range-Level Command and Control System of a National Cyber Range Testbed. In *Presented as part of the 5th Workshop on Cyber Security Experimentation and Test*, Berkeley, CA, 2012. USENIX.

[29] Anuj Sehgal. Running a Cloud Computing Infrastructure with OpenStack. In *6th International Conference on Autonomous Infrastructure, Management and Security*, June 2012. (Tutorial).

[30] WebGoat Team. WebGoat Project. https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project, 2016. (Accessed on April 2016).

[31] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific Languages: An Annotated Bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.

[32] VMWare. vCloud Web Page. http://vcloud.vmware.com/, 2016. (Accessed on April 2016).