Building Next Generation Cyber Ranges with CRACK

Enrico Russo[1], Gabriele Costa[2], Alessandro Armando[1]

Abstract

Cyber Ranges are complex infrastructures hosting exercises of the highest quality that simulate cybersecurity scenarios of real-world complexity. Building the computing infrastructure is only the first step towards the successful execution of the cyber exercises. The design, validation and deployment of scenarios are costly and error-prone activities. As a matter of facts, a misconfiguration in the scenario can compromise the exercise and the training goals. This make the design, development and deployment of live-fire cyber exercises of real-world complexity complexity so expensive that can be afforded only by a limited number of organizations. In this paper we present CRACK, a framework for automating the *(i)* design, *(ii)* model validation, *(iii)* generation and *(iv)* testing of cyber scenarios. We introduce the CRACK SDL, a Scenario Definition Language based on TOSCA, an OASIS standard for the specification and orchestration of virtual infrastructures. CRACK SDL allows for the high level, declarative specification of the components and their interplay. Through a formal encoding of the properties of a SDL specification, CRACK also supports the automatic validation of a scenario against its training objectives. After a successful validation the scenario is automatically deployed in the Cyber Range and automatically tested to check the correspondence between the behavior of the deployed system and its specification. The key functionalities offered by CRACK are presented through a simple, yet representative case study. Experimental results confirm the effectiveness of the proposed approach.

1. Introduction

The complexity and scale of today's cybersecurity threat landscape are straining the defenses of organizations which are thus increasingly being exposed to risks that can disruptively affect their businesses. Threats agents are no longer only endowed with advanced technical skills. Often they now have powerful tools in their arsenal, are well resourced, and may even have in-depth knowledge of the business processes of the organizations they target. Mitigating, let alone countering, these risks is a formidable challenge for cybersecurity operators. This requires the ability to detect early signs of cybersecurity incidents in large, sophisticated ICT infrastructures as well as to swiftly identify the most appropriate countermeasures. In order to achieve the needed level of preparedness, operators must be properly trained by means of live-fire exercises

[1]DIBRIS, University of Genova, IT
[2]SysMA Group, IMT School for Advanced Studies, IT

conducted in (possibly simulated) ICT infrastructures with scale and complexity comparable to those they are asked to protect on a daily basis.

Cyber Ranges are computing platforms that aim at providing a sophisticated training environment for cybersecurity operators. By leveraging virtualization technologies, Cyber Ranges can offer cyber exercises of extreme realism: large scale ICT infrastructures consisting of hundreds of interconnected devices, each supporting the protocol stack, operating system, and applications of choice can be simulated on commercial hardware or by leveraging IaaS providers. Cyber Ranges are being used to routinely carry out large scale cyber exercises. For instance, NATO conducts the Locked Shields cyber exercise [1] on a yearly basis since 2010. In Locked Shields blue teams must defend a given (simulated) ICT infrastructure from attacks mounted by a red team. The simulated ICT infrastructures used in Locked Shields consist of around four thousand virtualized systems and are executed in a dedicated Cyber Range.

Since the need of advanced cybersecurity training is now perceived by a wide audience (e.g., the civil sector), a number of Cyber Range solutions, both commercial (e.g., Cyber Range in a Box [2], Cyberbit Range [3], and NetWars [4]) and open source (e.g., ADLES [5], CyRIS [6], and KYPO [7]), have recently being put forward. These solutions enable the execution and monitoring of sophisticated cyber exercises. Yet, live-fire cyber exercises run against simulated ICT infrastructures of real-world complexity can be afforded only by a limited number of organizations. This is due to the fact that the design, development, and deployment of scenarios of real-world complexity are error-prone, time-consuming activities that require the involvement of highly specialized personnel.

To illustrate, consider the difficulty in installing the necessary software in and configuring each and every client, server, router, gateway in a complex scenario comprising hundreds of such devices. The resulting infrastructure, namely the *theater*, must then turned into an appropriate training scenario by injecting vulnerabilities (e.g., misconfigurations and software bugs) in it, thereby leading to a *training scenario*. This step is fraught of difficulties as the injection of too many (or too easy to discover) vulnerabilities may lead to a scenario that can be trivially solved, whereas the injection of too few (or too difficult to discover) vulnerabilities may lead to a frustrating experience for the participants. The problem is even more challenging if multi-stage attacks must be simulated. Unfortunately, a mistake made during the design of the theater may be discovered only at a later stage, e.g., during the development, the deployment or even the execution of the cyber exercise.

In this paper we present the *Cyber Range Automated Construction Kit (CRACK)*. CRACK supports the (i) design, (ii) automated validation and (iii) automated testing of complex Cyber Range scenarios of real-world complexity.

Design. We define *CRACK SDL*, a Scenario Definition Language based on TOSCA [8], an infrastructure specification language standardized by OASIS. As we will see, specifications expressed in CRACK SDL can be readily composed and reused and this greatly simplifies the design process. Moreover, since CRACK SDL is an extension of TOSCA, its integration with the existing infrastructure design technologies comes with no additional effort.

Validation. The CRACK SDL type system allows for the automatic validation of the scenarios against several design errors, e.g., incorrect hardware/software bindings.

More importantly, we show how a CRACK SDL specification can be translated into a corresponding Datalog specification which can be automatically checked by off-the-shelf Datalog engines. This allows the user to validate the CRACK SDL specifications against the training objectives.

Deployment. We show how a CRACK SDL specification is automatically translated into a sequence of instructions for a virtualization environment. Executing these instructions leads to the fully automated instantiation of all the elements of the scenario. As we will see, this process can take place on any TOSCA-compatible infrastructure virtualization platform.

Testing. We show that validation traces generated by the Datalog engine can be automatically turned into test cases for the scenario. The execution of the test cases checks whether the properties validated on the SDL specification (cf. Validation) are also enjoyed by the scenario at runtime.

As a further contribution, we introduce a case study consisting of two scenarios based on the same infrastructure and sharing the same goal (i.e., data exfiltration). Yet, the two scenarios are affected by distinct vulnerabilities that allow for different training objectives. The case study will be used as a working example through the paper in order to illustrate the functionalities of CRACK as well as the experimental results that confirm the effectiveness of our approach.

This paper is structured as follows. In Section 2 we describe our case study and the working example. In Section 3 we recall some preliminary notions and, in Section 4 we present our scenario definition language. Then we introduce CRACK in Section 5 and we demonstrate and evaluate it in Section 6. In Section 7 we discuss the related work. Finally, in Section 8 we draw the conclusions.

2. Case Study: ACME Corp

In this section, we introduce ACME Corp, a case study based on a fictional ICT infrastructure depicted in Figure 1. It consists of a segmented network where each segment hosts services and devices related to a specific task: *(i) Server* contains the internal services (i.e., not meant to be publicly accessible), *(ii) DMZ* contains the public services (i.e., exposed to the outside world) and *(iii) IoT* connects field devices (i.e., sensors, actuators and controllers). These three networks lay behind a *firewall* protecting the perimeter of the company. The firewall is intentionally left open toward the DMZ to allow remote connections. A domain name server (DNS), called *ns*, translates the symbolic names of the DMZ hosts into their actual IP addresses. The infrastructure is connected to the public Internet through the backbone of the Internet service provider.

The infrastructure of Figure 1 is the stage for the execution of two scenarios. Both of them involve a *blue team* and a *red team*. The blue team has the generic goal to protect the ACME Corp assets (e.g., data and services). The red team has the specific goal to exfiltrate data from the private database residing on *db*. Playing the role of the ACME Corp IT security department, the blue team has full access to the internal network, whereas the red team has only access to the public Internet through a remote *client* machine.

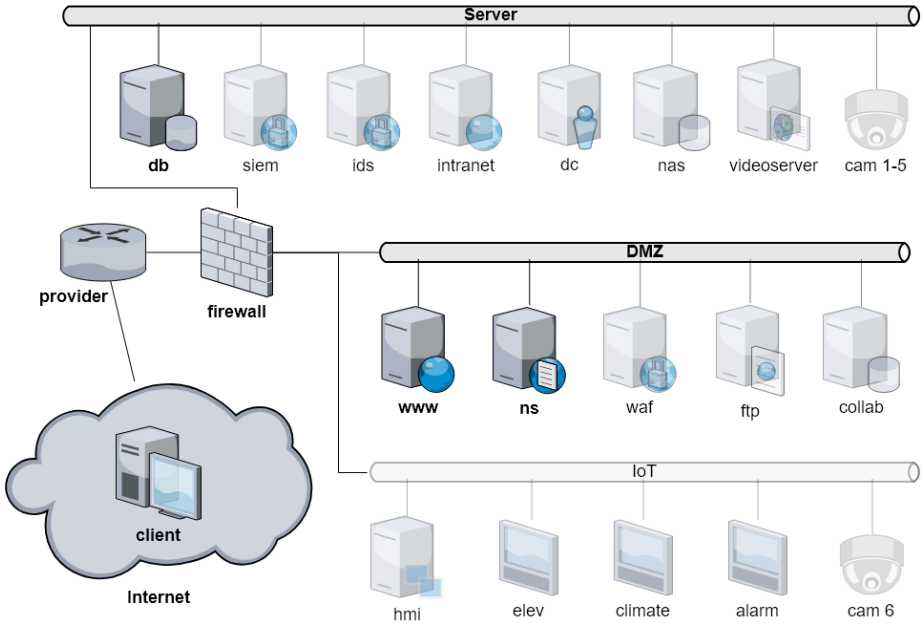


Figure 1: Case study infrastructure.

The two scenarios have different scope and training objectives. We briefly discuss them below.

Scenario 1 – Host Security. In this scenario the red team can achieve its goal by exploiting some security weaknesses in the configuration of the *www* server, including:

1. *www* runs an HTTP server exposing the home directories of the users; this enables a dictionary-based enumeration of the existing accounts.[3]
2. One of the enumerable users of *www* has a weak password, i.e., a password that is subject to brute-force attack.[4]
3. The administrator of *www* is exposed to an *Escalation of Privileges* (EoP) An EoP vulnerability allows the attacker to acquire the privileges of the administrator.

Scenario 2 – Web Security. In this case, the red team can exploit the weaknesses given below.

1. The remote debugging interface of the CMS is active, so allowing for Python commands injection.[5]

[3] https://attack.mitre.org/techniques/T1087/
[4] https://attack.mitre.org/techniques/T1110/
[5] https://www.netscylla.com/blog/2018/10/03/werkzeug-debugger.html

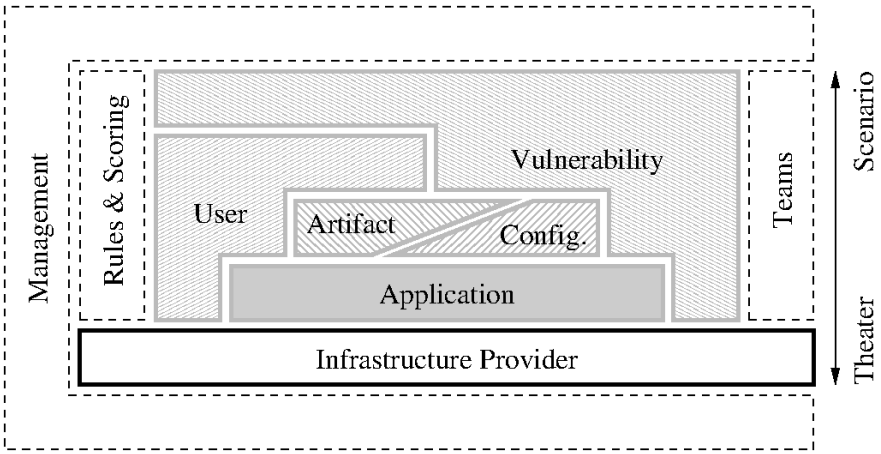


Figure 2: Cyber Range logical scheme.

2. The application server runs with unnecessary administrator privileges,[6] so exposing the database access credentials.

For the sake of presentation, here we only focus on the elements discussed above. In particular, we only consider the part of the infrastructure highlighted in Figure 1. We will use this part of the infrastructure and the first scenario as a working example through the paper. Then, in Section 6.4, we will discuss the effort needed to pass from the first to the second scenario.

3. Preliminaries

In this section, we briefly recall the notions that are relevant for correctly understanding the content of the paper.

3.1. Cyber ranges and training

According to the National Institute of Standards and Technologies Cyber Ranges are "interactive, [. . .] representations of an organization's local network, system, tools, and applications that are connected to a simulated Internet level environment."[7] The goal of a Cyber Range is to "provide a safe, legal environment to gain hands-on cyber skills and a secure environment for product development and security posture testing."[8]

In the spirit of the above definition, in this work, we propose a logical structure of a generic Cyber Range. Such a structure is depicted in Figure 2. The *management* facilities support the planning and execution of the activities conducted within the Cyber

[6] https://cwe.mitre.org/data/definitions/250.html

[7] https://www.nist.gov/sites/default/files/documents/2017/05/23/cyber_ranges_2017.pdf

[8] *ibid.*

Range. This may imply the monitoring of the Cyber Range activities possibly driving their evolution along a given story line.

The hands-on training is carried out within an *infrastructure*, consisting of a pool of (virtualized) networks, computers, and applications hosted by some provider. We call such an infrastructure the *theater*. Intuitively, we consider part of the theater all the elements that are not specific to the training objectives, i.e., any object that is passively or marginally involved in the current exercise. For instance, the routing infrastructure of a layered network architecture, being only responsible for the exchange of messages, is a part of the theater.

On top of the theater, the *scenario* is the collection of all the items that are relevant for the hands-on activity. A typical scenario involves some *applications*, e.g., a remote shell or a Content Management System (CMS). Such applications are customized with *configurations*, e.g., the remote authentication method, and *artifacts*, e.g., an authentication key, that plays a role in the scenario. Similarly, *user* accounts can be included. A user can be related to all of the elements mentioned above, e.g., the administrator of a service (application) has also access to its files (artifacts and configurations). Finally, *vulnerabilities* must be injected to enable the attackers' exploits. Vulnerabilities are a cornerstone of every scenario and they can involve any of the elements discussed above, e.g., a user setting a weak password or an application failing in sanitizing an input.

Beside the scenario, the theater must also provide the gameplay facilities, i.e., the *scoring* and *rule* systems as well as the *team* support. For instance, in a scenario some servers cannot be attacked (engagement rules) or the blue team loses points when a certain service becomes unavailable (service level agreement). Similarly, the teams may be required to operate through some terminals that only provide a limited number of security tools. These elements cannot be developed once for all as they are scenario-dependent. Nevertheless, they can be occasionally reused, e.g., the same scoring system might apply to a certain category of exercises.

Since in this work we deal with the design and verification of the scenarios, we only focus on the part of the elements of Figure 2. In particular, we will reason about the scenario elements (light gray boxes). Moreover, we will discuss the infrastructure provider technologies that support the deployment of theaters and scenarios. Instead, we will skip the presentation of the management and gameplay elements (dashed boxes).

3.2. Infrastructure provisioning

In this section, we briefly recall the two infrastructure provisioning paradigms involved in our proposal.

Infrastructure-as-a-Service (IaaS). IaaS [9] aims at providing a flexible and reconfigurable infrastructure development platform. In particular, an IaaS provider allows for a direct control over machines, operating systems, applications, and networking. By relying on virtualization technologies, IaaS platforms hide the underlying, physical infrastructure (a.k.a. bare-metal).

In a Cyber Range, each theater consists of many different elements including hosts (e.g., servers and desktop clients), software (e.g., operating systems and applications) and network facilities (e.g., routers and firewalls). Conveniently, IaaS providers expose

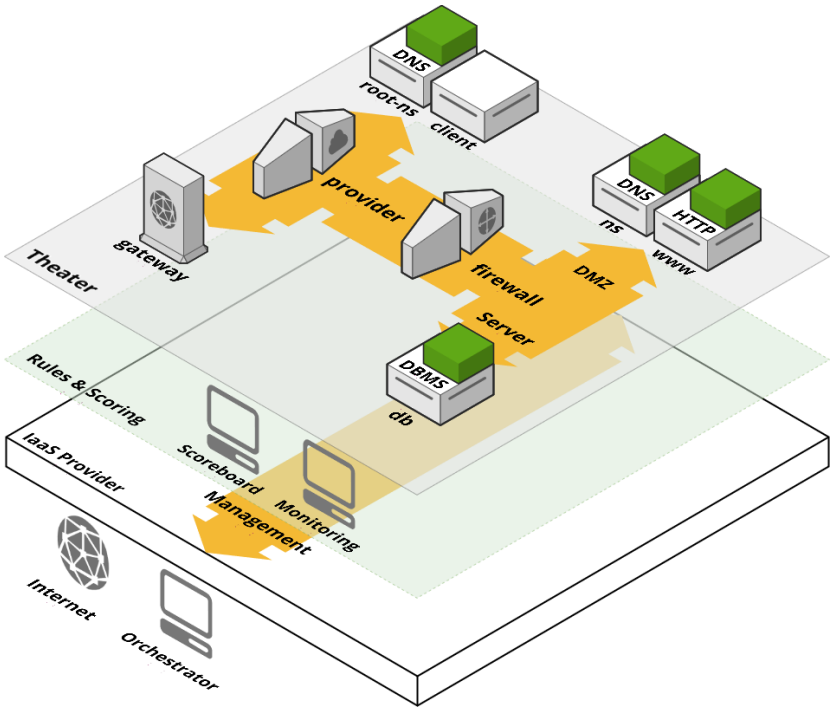


Figure 3: Layered view of the working example theater deployed over an IaaS provider.

APIs for creating, deleting, and reconfiguring these elements. This makes IaaS a suitable paradigm for defining and deploying a Cyber Range theater. In this setting, the building blocks of any theater are virtual machines (for computing and storage) and virtual switches, routers, networks, and network ports (for implementing the network infrastructure). Although some elements may not allow virtualization, a virtual network can also be connected with some physical resources outside the IaaS platform. For instance, an infrastructure can be connected to the Internet through a gateway.

Figure 3 represents the deployment of the theater of our working example (see Section 2) on an IaaS provider. The theater is depicted on the top layer. Intuitively, all the elements of the theater are virtual with the only exception of the Internet which is only partially simulated. The real Internet is accessible through a gateway that directly connects to the external network. Moreover, the IaaS supports for the virtualization of part of the Cyber Range facilities, such as the scoreboard and rule monitoring services. These facilities stay on a different layer as they are not accessible from within the theater. Instead, they operate through a management network that is responsible for the cross-layer connectivity. Such connectivity is necessary to *orchestrate* the theater, i.e., to create and configure the virtual infrastructure.

Infrastructure-as-Code (IaC). On an IaaS provider, instantiating the infrastructure as in Section 2 requires the following operations.

1. Create the virtual networks, e.g., Server and DMZ.

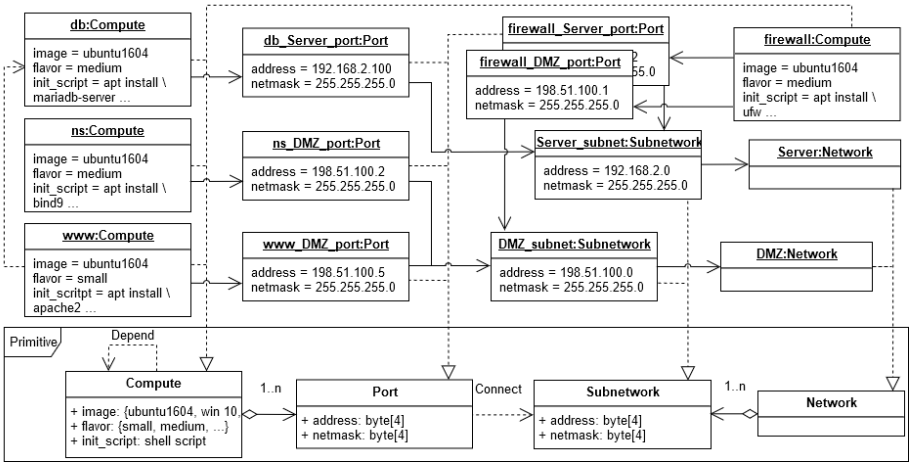


Figure 4: A generic IaC specification for the working example theater.

2. Create all the virtual machines, e.g., db and www.
3. Connect each virtual machine to the proper networks, e.g., db to Server.
4. Install all the operating systems and applications, e.g., DBMS on db.
5. Finalize the infrastructure by adding configurations, artifacts, and users.

All these operations are carried out by submitting the corresponding commands, e.g., via some APIs, to the IaaS provider. Nevertheless, as the complexity of the infrastructure increases, handling these design and deployment operations without a systematic approach quickly becomes cumbersome and error-prone.

In the last years *Infrastructure-as-Code* (IaC) [10] emerged as the main infrastructure design approach. A IaC framework uses a specification language to model the desired infrastructure. A provisioning tool, called *orchestrator*, takes as input the specification and automatically deploys the infrastructure on an IaaS provider. We propose the following example to clarify the structure of a generic IaC specification language.

Figure 4 provides a class diagram representation of the infrastructure appearing in the working example. The box at the bottom, labeled with *Primitive*, contains (some of) the primitive classes defined by a generic IaC provider. These classes abstractly define the building blocks of the infrastructure, e.g., machines and networks.

The *Network* class allows for the creation of virtual networks. Each virtual network is a collection of virtual subnetworks, i.e., the *Subnetwork* class. A virtual subnetwork is labeled with two properties, i.e., *address* and *netmask*, that specify the network address and netmask of the subnetwork.

The *Compute* class represents a generic host, e.g., a virtual machine. An instance of Compute must declare its *image*, i.e., the installed OS, *flavor*, i.e., the hardware profile, and *init_script*, i.e., the instructions to correctly configure the host. There can also be dependencies between Compute objects. For instance, the www server depends on the db server. Typically, the orchestrator is responsible for resolving the existing

```
P(a, b).                          1: P(b, c).
P(b, c).                          2: P(a, b).
Q(A, B) :- P(A, C), P(C, B).      3: Q(a, c) :- P(a, b), P(b, c).
Q(X, c)?                             Q(X = a, c)
```

Figure 5: A Datalog specification with a query (left) and a proof trace (right).

dependencies, e.g., by creating the Compute objects in the right order.

Finally, Compute objects can be connected to one or more subnetworks. This behavior is modeled by the *Port* class that defines a generic network port for connecting to a subnetwork. Each port can also carry a (fixed) IP address specified in the *address* and *netmask* properties. In the diagram, the *db_Server_port* and *www_DMZ_port* are instances of the Port class and connect the db and www server to the two subnetworks with addresses 192.168.2.100 and 198.51.100.5, respectively.

3.3. Datalog

Datalog [11] is a declarative logic programming language which enjoys efficient algorithms for query resolution. A Datalog specification consists of a list of *facts* and *clauses*. A fact $\mathtt{P}(\mathtt{a}_1, \ldots, \mathtt{a}_n)$ states that a predicate P is satisfied by the elements of a tuple $(\mathtt{a}_1, \ldots, \mathtt{a}_n)$. A clause $\mathtt{T} :- \mathtt{T}_1, ..., \mathtt{T}_n$ states that a term T can be inferred from the terms $\mathtt{T}_1, ..., \mathtt{T}_n$, called the *premises* of the clause. Terms are also predicates, but, unlike facts, they can contain *variables*, e.g., A, B, X, ...

A Datalog query T? is evaluated by an engine, i.e., a solver, against a specification to decide whether T is entailed by the given facts and clauses in the specification. When this is the case, the Datalog engine returns the list of facts and clauses, namely the *proof trace*, that have been applied to validate the given query.

To exemplify, consider the Datalog specification on the left of Figure 5. The specification consists of two facts and one clause. Moreover, we append a query at the end of the specification. The query is valid if an assignment to X can be found that satisfies Q(X,c). This is trivially true for the query and a possible proof trace is given on the right of Figure 5. The proof ends by finding an assignment, i.e., X = a, that satisfies the query. To obtain this result, the engine applied the (only) clause in the specification (line 3) by instantiating its variables. The right-hand side of the clause contains the premises that are available in the initial part of the trace (lines 1 and 2).

The decision problem for a Datalog query is P-complete in the size of the Datalog specification [12].[9] This ensures that the validation process scales well even with large specifications.

4. Scenario Definition Language

The design phase aims at generating a suitable blueprint of the scenario, covering all the relevant aspects from the infrastructure description to the objectives of the cyber

[9] The study of the Datalog fragments that admit efficient solvers is an active research field, e.g., see [13].

exercise. Although the infrastructure has an important role, there are other aspects to consider when designing a scenario. In general, IaC is not meant to model these components and must be extended to support them. Our *Scenario Definition Language* (SDL) builds on TOSCA [8], a prominent IaC language, but it introduces several new elements that we describe in this section. Briefly, we carry out three extensions, i.e., (i) we define new, scenario-specific node and relationship types, (ii) we introduce two special properties to support the verification and testing process and (iii) we implement a novel query language based on *access patterns*.

4.1. TOSCA integration

The *Topology and Orchestration Specification for Cloud Applications* (TOSCA) [8] is a YAML-based[10] OASIS standard language for designing the topology and the life-cycle of a cloud application. A TOSCA-enabled IaaS provider must have a suitable TOSCA orchestrator.[11] TOSCA implements the concepts of Figure 4 by means of a rich type system. This will be further discussed in Section 5.2. Briefly, the main constituents of TOSCA are the following.

Node types. They define an infrastructure component, e.g., a server or a network, or a component element, e.g., a software installed on a server. A node type can include *properties*, *attributes*, *capabilities* and *requirements*. Properties represent some static, node-specific feature, e.g., the hostname. Attributes resemble properties, but they are used to store a value that is set by the orchestrator after the instantiation, e.g., think of a dynamically assigned IP address. Requirements and capabilities define what the node needs and (optionally) provides to the others. Requirements and capabilities mainly serve as the joints for the relationships (see below).

Relationship and capability types. They are used to connect nodes and, as it happens for node types, can include properties and attributes, e.g., the credentials for the authenticated service exposed by the node we are connecting. A relationship has a direction, and it connects the requirement of a source node to the capability of a target node. Moreover, each requirement can put a constraint on the types of both the target node and capability. For instance,[12] a WordPress web application requires to connect to (i) a database (ii) endpoint, i.e., a network database. To model this, the WordPress node includes a requirement *database_endpoint*. The *database_endpoint* requirement constrains the type of the target node to be *Database* and the type of the target capability to be *Endpoint*. These two constraints capture (i) and (ii), respectively.

Interfaces. Nodes and relationships may have *interfaces*. An interface defines a custom operation to be invoked by the orchestrator. Two kinds of interfaces exist, i.e., *standard* and *on demand*. A standard interface defines a task related to the life-cycle phases of a node (e.g., create, start, and stop). For instance, one can add a standard, *create* interface to a compute node to ask the orchestrator for installing a certain software

[10] `http://yaml.org`

[11] Existing TOSCA-enabled orchestrators often accept a slightly extended version of the TOSCA standard, i.e., a TOSCA dialect. If not differently stated, the examples in this paper refer to the ARIA TOSCA dialect [14].

[12] See [8] for more details.

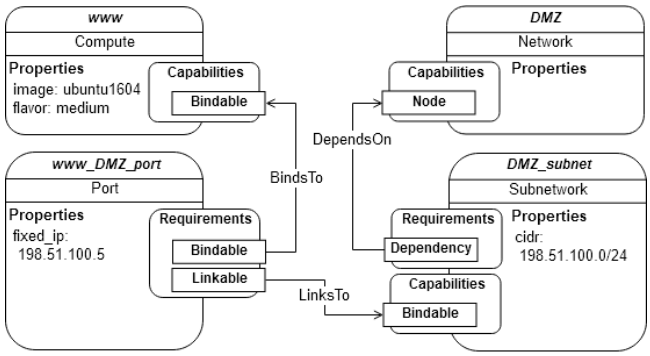


Figure 6: An excerpt from the (TOSCA-style) diagram for the working example.

package when the node is created. Instead, on-demand interfaces introduce new tasks. The orchestrator permits to invoke the tasks through the definition of a new workflow. For instance, the on-demand interface can be used to implement an application-specific logic (see [15, § 7.3.2]).

A *node template* is a specification of a cloud application obtained through the composition of the elements mentioned above. In particular, each element is obtained by instantiating its base, namely *normative*, type. Roughly speaking, the TOSCA normative types provide a set of primitive classes (see Figure 4). Designers can define their own types by extending the normative types. As discussed in [16], the type inheritance enables some well-know mechanisms, e.g., type substitution and reuse, that simplify the design process.

Example 1. Consider the diagram depicted in Figure 6. It is an excerpt of the specification for the working example introduced in Section 2. In particular, it specifies the infrastructure of the web server (www) and the hosting network (DMZ). The www server runs on a virtual machine, an instance of the *Compute* node type. The hardware configuration of www and its operating system image are set by using the *flavor* and *image* properties, respectively. The DMZ network is an instance of the *Network* node type. A DMZ subnetwork, instance of *Subnetwork* node type, is in relationship, *DependsOn*, with the DMZ network and allows to specify its block of IP addresses in the *cidr* property. Moreover, the DMZ subnetwork provides the *Bindable* capability for supplying connections to the DMZ network. The www connectivity is represented by *www_DMZ_port*, an instance of the *Port* node type which also includes the *fixed_ip* property for assigning a fixed address to the connected node. The node *www_DMZ_port* is the source of two relationships, namely *BindsTo* and *LinksTo*. The former connects the *Bindable* requirement to the *Bindable* capability of the Compute node www. The latter connects the *Linkable* requirement to the *Bindable* capability of the *DMZ_subnet* Subnetwork node. □

```
1  www:
2   type: Server
3   properties:
4    image: ubuntu1604
5    flavor: medium
6    requirements:
7     - port: www_DMZ_port
8  www_DMZ_port:
9   type: Port
10  properties:
11   fixed_ip: 198.51.100.5
12  requirements:
13   - network: DMZ
14   - subnet: DMZ_subnet
15 DMZ:
16  type: Network
17 DMZ_subnet:
18  type: Subnet
19  properties:
20   subnet:
21    cidr: 198.51.100.0/24
22  requirements:
23   - network: DMZ
```

Figure 7: An excerpt of the TOSCA specification for the working example.

The syntax of the TOSCA language is YAML-based.[13] Node instances are collections containing (i) the entity type, (ii) a key-value dictionary of properties, and (iii) a list of requirement bindings. A relationship between two nodes exists when a requirement of a source node instance is bound to the name of the target.

Example 2. Consider the YAML specification given in Figure 7. It is the TOSCA encoding of the diagram of Figure 6. Node `www` (line 1) represents the compute entity for the web server. It is an instance of the `aria.openstack.nodes.Server` (line 2) type, i.e., a subtype of `tosca.nodes.Compute` denoting a virtual machine that runs on an OpenStack IaaS. [14] This node contains two properties: the name of the base operating system image (line 4) and the flavor (line 5) of the virtual machine. A port requirement (line 7) permits to establish a relationship with the `Port` node `www_DMZ_port` (line 8). The port assigns a fixed IP address to the virtual machine using the property *fixed_ip* (line 11). Also, the port is related with the `DMZ Network` node (line 13) and `DMZ_subnet Subnet` node (line 14). The IP addressing configuration of `DMZ_subnet` is specified in the `subnet` property (line 20). □

4.2. SDL types

SDL introduces a number of new node types. Their primary purpose is to model the scenario specific aspects and to integrate them in a TOSCA blueprint. Below we introduce the most relevant ones.

System types. The type `System` represents the base class of a generic system of the scenario (e.g., workstations, servers, smartphones). Each System node is associated (through a relationship) with an existing TOSCA `Compute` node. The `Compute` node is the virtual machine where the `System` runs.

[13] `http://yaml.org`

[14] For brevity we may omit namespaces such as `aria.openstack.nodes`.

Subtypes of System are used to model more specific elements. For instance, Firewall represents a System node with firewall functionalities. In particular, it is characterized by a default policy, e.g., allow all the traffic passing through its network interfaces (default: allow). It has the capability Rule that enables a relationship with the Policy nodes. Briefly, a Policy node defines a firewall rule through the specification of a traffic pattern. The pattern is represented by the properties source, destination, protocol, and port. The meaning is that the associated firewall has to block the connections matching the pattern.[15]

A System node can can also be related to some Artifact, Software and User. An Artifact node denotes a file or some other piece of data, e.g., a cryptographic key. A Software node represents a program installed on a the System. It may provide a service endpoint (specifying a port and protocol) if it is network accessible. Also, a relationship with a User defines the running privileges of the software. Finally, a User node models a user of the System through a requirement Host. Its properties include its username, password, and role (denoting its privileges in the System).

All the types introduced above include a capability denoting the fact that they can be involved in some vulnerability (see below). For instance, a software can suffer from a known security flaw, while a user can have a weak password.

Scenario-specific types. The Vulnerability type is perhaps the most interesting element in our context. As a matter of fact, it represents a generic, security vulnerability involved in the scenario. Precisely characterize the notion of vulnerability is not trivial and many definitions exist.[16]

In our context, a vulnerability is any security weakness introduced by some (mis) configuration. As discussed above, vulnerabilities may refer to any system type in the scenario. Moreover, they must specify the configuration procedure, i.e., the steps injecting the vulnerability in the scenario, and the exploit operations, i.e., how the attacker uses the vulnerability. Its properties and relationships with the scenario elements vary with the specific vulnerability. Since vulnerabilities are extremely heterogeneous, we do not put further constraints on their structure.

A Principal represents a subject operating in the scenario. For instance, attackers (red team) and defenders (blue team) are principals. Typically, a Principal has a relationship with some User nodes representing the accounts initially controlled. The objective of a Principal is to achieve its Goals. This is specified through a relationship between the two nodes. A Goal represents a state of the scenario that identifies the winning conditions of the related Principal, e.g., gain access to a certain system. A detailed discussion on the role of Goal nodes is given in Section 5.2.

Relationship types. SDL also introduces new relationship types. The primitive relationship types of TOSCA model the infrastructural dependencies only. For instance, we use HostedOn to connect a SDL System to the TOSCA Compute node hosting

[15] Actual firewall policy languages can be more complex and the definition of rigorous languages is still an open research issue, e.g., see [17].

[16] E.g., see https://csrc.nist.gov/glossary/term/vulnerability.

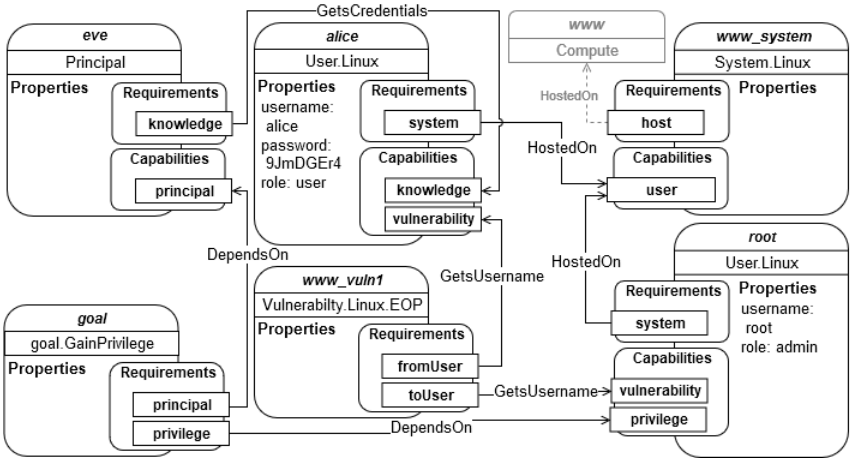


Figure 8: A SDL fragment from the specification for the working example.

it. Scenario-specific dependencies, e.g., the one occurring between a vulnerability and its attack vector, fall outside the scope of TOSCA. In general, one might use the generic `DependsOn` relationship. Yet, this would require to customize each instance of the relationship with its specific deployment logic (see Section 5.3.1). An alternative to `DependsOn` is to define new relationship types that commonly occur in the scenarios. For instance, a relationship `GetsUsername` can connect a `Vulnerability` node to a `User` node. The meaning is that the `Vulnerability` node reads the `username` property of the `User` node and uses it for the configuration of the vulnerability. To exemplify, think of a user enumeration vulnerability where an attacker enumerates the users of a service by testing a dictionary of common usernames. To properly configure it, one has to make sure that the target username belongs to the dictionary.

Example 3. The SDL diagram in Figure 8 extends the TOSCA specification of Example 1 by adding some elements of Scenario 1 (see Section 2). In particular, here we only consider vulnerability 4 and we assume that the red team already knows the credentials to remotely access as an unprivileged user. The `www_system` node represents the Linux system running on the `Compute` node `www`. The `User` node `alice` is hosted on the Linux system. The properties of `alice` define its username, password, role (where `user` stands for a standard, unprivileged user). Similarly, `root` models the administrator of `www_system`. Like `alice`, `root` has a username and a role, i.e., `admin`. However, by not setting the `password` property, we model the

```
1  User:
2   derived_from: Root
3   properties:
4    username:
5     type: string
6     required: yes
7    password:
8     type: string
9    role:
10    type: string
11    default: user
12    constraints:
13     - valid_values:
14       [ admin, user ]
15  capabilities:
16   vulnerability:
17    type: VulnerabilityContainer
18  [...]
19  User.Linux:
20   derived_from: User
21  [...]
22  Vulnerability.Linux.EoP:
23   derived_from:
24    Vulnerability.Linux
25   requirements:
26    - fromUser:
27       capability:
28        PrivilegeProvider
29       relationship:
30        GetsUsername
31       node: User.Linux
32    - toUser:
33       capability:
34        PrivilegeProvider
35       relationship:
36        GetsUsername
37       node: User.Linux
38    [...]
```

Figure 9: An excerpt of SDL node types declaration.

passwordless,[17] default configuration of the Ubuntu Linux distribution (defined by the `www` node in Example 1).

The `www_vuln1` node represents the EoP vulnerability discussed in Section 2. Its type is `Vulnerability.Linux.EoP` and it has two requirements, i.e., `fromUser` and `toUser`. Respectively, they connect to the unprivileged and the privileged users. Such a connection occurs through the `GetsUsername` relationship.

A `Principal` node `eve` represents the attacker (red team). It is related with `alice`, i.e., the account initially controlled by the attacker. Finally, the `goal` node is related to `eve` and it represents the objective of the attacker. It is an instance of the `goal.GainPrivilege` type and it has two requirements, namely `principal` and `privilege`, meaning that the connected `Principal` (namely `eve`) aims at acquiring the privileges of the connected `User` (namely `root`). Notice that this is not the final goal of Scenario 1 (i.e., reading the content of the database). It is an intermediate goal only, that is a sub-goal enabling the final one, that we consider here for the sake of presentation.

In Figure 9 we give the type declaration of the the node types related to the vulnerability introduced in the SDL diagram. The `User` type (lines 1-18) represents a SDL primitive and inherits from the root of all the SDL types, namely `sdl.nodes.Root`. It has three properties: `username`, is a mandatory string identifying the user in the system, `password`, an optional string representing the user's password, and `role` representing the user's privileges. Two roles are modeled in the example, i.e., `admin` and `user` (line 13-14). An unspecified value for the role property implies a user role (line 11). Moreover, `User` has the capability `vulnerability`

[17] `https://help.ubuntu.com/community/RootSudo`

The specific type for Linux users, namely `User.Linux` (lines 19-21), inherits from the `User` type described above. It mainly denotes a class of users we need for constraining nodes requirements (line 31 and line 37).

The type `Vulnerability.Linux.EoP` (lines 20-36) derives from `Vulnerability.Linux`. It contains the `fromUser` and `toUser` requirements described above. They both specify `VulnerabilityProvider` as capability and `linux.User` as type for the target node and use `GetsUsername` as relationship type. □

4.3. Behavior and runtime

All the SDL node and relationship types have two special properties, i.e., `behavior` and `runtime`. Intuitively, `runtime` is associated with *commands*, for instance shell scripts, to be executed. Such commands are mostly used for the testing phase (see Section 5.3.2). Instead, `behavior` properties contain terms used for the validation process (see Section 5.2). Both runtime and behavior properties are given as finite mappings between unique identifiers and commands and terms, respectively. Moreover, we require the two mappings to have exactly the same domain. Differently said, the runtime maps an identifier to a command if and only if the behavior maps the same identifier to a term. These aspects are detailed in Sections 5.2 and 5.3.2.

4.4. Access pattern language

TOSCA natively provides operations, called intrinsic functions, to access the information stored inside a node [15, § 4.3]. For instance, a node n can use `get_property: [r,p]` to read the value assigned to property `p` by the node related to n through the requirement `r`. Notice that intrinsic functions can only walk through a single relationship. This is reasonable for the design of an infrastructure, where each node is related to the others it depends on. However, it makes extremely hard to model complex dependencies such as those introduced by the `behavior` property of vulnerabilities and goals. The motivation is that, for instance, a goal can be related to nodes that are very far in the blueprint.

For this reason we introduce an *access pattern* language to specify structured, path-based queries. An access pattern ρ follows the syntax below.

$$
\begin{array}{lll}
\rho & ::= & \pi\texttt{[P]} \mid \pi\texttt{\{A\}} \\
\pi & ::= & \pi_{nod} \mid \pi_{rel} \mid \pi_{cap} \\
\pi_{nod} & ::= & \texttt{this} \mid \pi_{rel}\texttt{.src} \mid \pi_{cap}\texttt{.node} \\
\pi_{cap} & ::= & \texttt{this} \mid \pi_{nod}\texttt{<-C} \mid \pi_{rel}\texttt{.dst} \\
\pi_{rel} & ::= & \texttt{this} \mid \pi_{nod}\texttt{->R} \mid \pi_{cap}\texttt{.rel}
\end{array}
$$

An interpreter evaluates and replaces an access pattern ρ with a set[18] of values according to the target SDL specification. In particular, π`[P]` amounts to the value of property `P` of the SDL elements *pointed* by π (therefore called a pointer).[19] Similarly,

[18] Since SDL relationships can be many-to-many, the evaluation of an access pattern is not guaranteed to result in a single value.

[19] For brevity, we may omit `[P]` when `P = name`.

π{A} reduces to the value of the attribute A of the elements pointed by π. A pointer π can be of three kinds depending on the class of the SDL elements it refers to, i.e., nodes (π_{nod}), relationships (π_{rel}) or capabilities (π_{cap}).[20] Each element can use this as a self-pointer.[21] A node pointer π_{nod} is also obtained by means of the operator .node applied to (a pointer to) a capability contained by the node. A pointer to a capability C is obtained by applying the operator <− to a node pointer. Moreover, from a relationship pointer π_{rel} one can access the destination capability by means o f the operator .dst. Conversely, a pointer to a relationship can be obtained either (i) from a node pointer π_{nod} by means of the requirement R where the relationship originates (operator −>R) or (ii) from a capability through the operator .rel. For the sake of presentation we also introduce the following abbreviations.

π_{nod}=>R $\equiv$ π_{nod}->R.dst.node $\quad$ π_{nod}<=C $\equiv$ π_{nod}<-C.rel.src

Example 4. Consider the following access patterns defined by node eve (see Figure 8).

```
this[name]   this->knowledge[name]   this=>knowledge[role]
```

The first pattern trivially evaluates to eve. The second one requires to access property name of the relationship originating from the requirement knowledge, i.e., GetsCredentials. Finally, the last access pattern follows the relationship originating from knowledge and points to node alice whose property role is assigned to user. □

5. Introducing CRACK

In this section, we present our framework CRACK. We start in Section 5.1 by providing a general description of its structure. Then we detail the main constituents, i.e., its specification language, the scenario validation and automatic testing process.

5.1. Overview of the approach

Figure 10 depicts the abstract workflow of CRACK. The scenario development workflow starts with the modeling task. During this task, the designer creates a blueprint of the scenario by using SDL (see Section 4). The model is then type checked to detect possible inconsistencies. If it is the case, type errors are returned to the modeling task to be fixed. Otherwise, the process proceeds to the verification task. In the current implementation, CRACK generates a Datalog specification from the model and feeds it to a Datalog engine (see Section 5.2). We stress the fact that our approach can be extended with other verification techniques. To support this operation, CRACK is based on a modular design. If the Datalog verification fails, the unreachable goals are returned to the design process, otherwise a proof trace for each goal is generated and

[20] Notice that requirements cannot declare properties in TOSCA.
[21] We omit it when clear from the context.

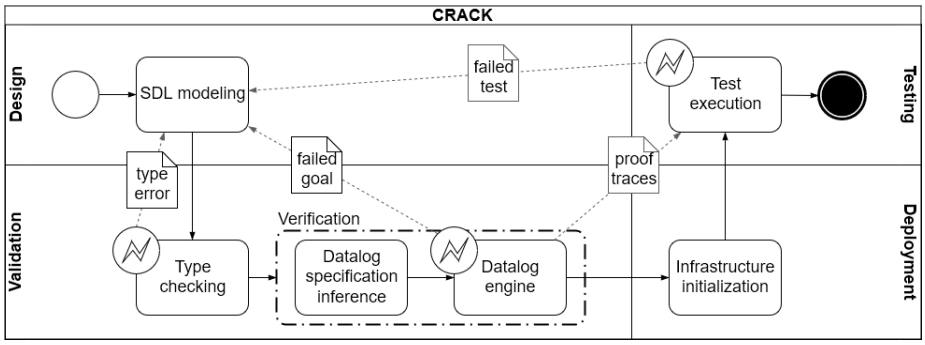


Figure 10: CRACK general workflow.

the process moves to the infrastructure initialization task. When the infrastructure is up and running, CRACK performs the test execution phase (described in Section 5.3). This task converts the proof traces into unit tests and executes them on the deployed infrastructure. If a test fails, a feedback is provided to the user in order to refine the scenario blueprint. Eventually, when all the tests are passed, the scenario is ready to be played.

5.2. Scenario Validation

SDL inherits the TOSCA type system (cf. Section 4.1). A well-typed blueprint enjoys some properties of interest, such as the coherence between nodes and relationships [18]. Although efficient, type checking cannot verify more complex properties. To overcome this limitation, we introduce a further verification phase that converts a well-typed SDL model into a Datalog specification (see Section 5.2.1). Objectives are then encoded as queries that must be satisfied by the specification.

A successful verification yields as a set of proof traces, one for each objective. Proof traces are later used as input to the deployment and testing phases (cf. Section 5.3). Interestingly, a verification failure is also useful: invalid queries can be productively used (e.g., see [19]) to identify bugs in the model that originated the specification.

5.2.1. Encoding

The encoding process generates a Datalog specification from a scenario blueprint. The Datalog terms refer to a set of predefined predicates. Some predicates are inspired by [20]. The most relevant ones are listed in Table 1.

Facts and clauses belong to three blocks, i.e., *constants*, *behaviors* and *goals*. Constants include Datalog terms that model the standard behavior common to any infrastructure. For instance, the clause

```
hasAccount(A,H,U) :- hasUser(U,H,P,R), knows(A,U), knows(A,P).
```

means that principal `A` owns account `U` on host `H` if user `U` exists on `H` and `A` knows both `U` and the associated password `P`.

Table 1: Predefined Datalog predicates (excerpt).

Predicate	Description
`knows(A,D)`	Principal `A` knows datum `D` (e.g., a password)
`hasAccount(A,H,U)`	Principal `A` has an account on host `H` as user `U`
`hasUser(U,H,P,R)`	Host `H` has user `U` with password `P` and role `R`
`listeningOn(H,Q,S)`	Host `H` has a software on port `S` with protocol `Q`
`isConnected(H,N)`	Host `H` is connected to network `N`
`hostACL(H,K,Q,S)`	Host `H` can access host `K` via protocol `Q` on port `S`

Behaviors blocks contains the terms introduced by the SDL elements through the behavior property (see Section 4). The behavior property consists of a mapping between identifiers and *term patterns*. A term pattern resembles a standard Datalog term, i.e., a fact or a clause, but its parameters can also be access patterns (see Section 4.4).

Example 5. Consider the SDL fragment of Example 3. We show the Datalog term generated by the EoP vulnerability. Let assume that `www_vuln1` has the following `behavior` property.

```
hasAccount(A, H, =>ToUser) :- hasUser(=>ToUser, H, P1, R1),
                              hasUser(=>FromUser, H, P2, R2),
                              hasAccount(A, H, =>FromUser).
```

The term above models the vulnerability prerequisites (clause premises) and effect. In practice, the vulnerability allows a principal `A` to obtain the control over a target high-privileged user `=>ToUser`[22] by leveraging a misconfiguration. In particular, the misconfiguration has to do with a low-privileged user `=>FromUser` that can impersonate `=>ToUser` when launching a certain command. For this to happen three conditions must be satisfied.

i) The high-privileged account `=>ToUser` exists on the target system.
ii) A low-privileged account `=>FromUser` also exists on the target system.
iii) Principal `A` has control over the low-privileged account.

Similarly, the behavior property of the relationship `GetsCredentials` is defined as follows.

```
knows(.src, .dst.node[password])
knows(.src, .dst.node[username])
```

Finally, also the nodes of type `User.Linux`, i.e., `alice` and `root`, define a term pattern in their behavior property.

```
hasUser(this[username],=>System,this[password],this[role])
```

According to the specification of Example 3, all the above term patterns contribute to the following Datalog specification.

[22]Recall that this is an abbreviation for `this=>ToUser[name]`.

```
/* Constants */
hasAccount(A,H,U) :- hasUser(U,H,P,R), knows(A,U), knows(A,P).
/* EoP vulnerability */
hasAccount(A, H, root) :- hasUser(root, H, P1, R1),
                          hasUser(alice, H, P2, R2),
                          hasAccount(A, H, alice).
/* GetsCredentials */
knows(eve, 9JmDGEr4).
knows(eve, alice).
/* root */
hasUser(root, www_system, , admin).
/* alice */
hasUser(alice, www_system, 9JmDGEr4, user).
```
□

Finally, the SDL goals result in queries to be evaluated against the Datalog model. Such queries denote that a certain configuration is reachable in the scenario.

Example 6. Consider the node `goal` of Example 3. Its behavior property contains the term `hasAccount(=>Principal, =>Privilege=>System, =>Privilege)?` which reduces to the query `hasAccount(eve, www_system, root)?`.

Notice that, in this particular example, the goal query contains no free variables. Thus, its evaluation results in a plain boolean value. □

5.2.2. Verification

The verification process boils down to running a Datalog engine against the goal queries. The verification fails when one or more queries cannot be satisfied. In such a case, the failure denotes that a principal cannot achieve one of its goals.

Example 7. Consider again the Datalog specification of Example 5 and the query of Example 6. The query is trivially satisfied by the specification. The generated proof trace is as follows.

```
1. hasUser(root, www_system, , admin).              /* Fact */
2. hasUser(alice, www_system, 9JmDGEr4, user).      /* Fact */
3. knows(eve, alice).                               /* Fact */
4. knows(eve, 9JmDGEr4).                            /* Fact */
5. hasAccount(eve, www_system, alice) :-
     hasUser(alice, www_system, 9JmDGEr4, user),    /* From 2 */
     knows(eve, alice),                             /* From 3 */
     knows(eve, 9JmDGEr4).                          /* From 4 */
6. hasAccount(eve,www_system,root) :-
     hasUser(root, www_system, , admin),            /* From 1 */
     hasUser(alice, www_system, 9JmDGEr4, user),    /* From 2 */
     hasAccount(eve, www_system, alice).            /* From 5 */
```

We discuss the steps of the proof trace in backward order. The last step (`6`) concludes the proof by inferring the goal query. The proof step consists of an application of the clause introduced by the EoP vulnerability (see Example 5). To apply the clause, three preconditions must be satisfied. Two of them amount to facts appearing in the specification (i.e., `1` and `2`), thus requiring no further proofs. Instead, the last one is

```
[...]
38:  interfaces:
39:    standard:
40:     configure:
41:      implementation: /scripts/eop-apt-config.sh
```

Figure 11: Extended YAML declaration of the `EoP` vulnerability.

inferred by applying the clause of the constants block (see Example 4). The inference step (`5`) is based on three premises, i.e., `2`, `3` and `4`. Since all of them are facts appearing in the specification, the proof is completed. □

5.3. Deployment and testing

Once validated, a blueprint can be instantiated and executed. In general, the deployment process is not guaranteed to preserve the model-validated properties. As a matter of fact, the abstract, high-level design purposely neglects some implementation aspects that may affect the scenario at runtime. For instance, a piece of software may behave differently from the expectations of the scenario designer.

To favor a prompt detection and debugging of the scenario, we leverage the verification proof traces to automatically generate and run tests (see Section 5.3.2). Each test aims at confirming that a model-validated property also holds in the deployed scenario.

5.3.1. Deployment

The deployment phase generates the directives for the IaaS provider. Many solutions exist for the interpretation/translation of the TOSCA specifications into the orchestration instructions of the major IaaS provider (e.g., see Cloudify [21], ARIA TOSCA [14], OpenTOSCA [22], Alien4Cloud [23] and Heat-Translator [24]). All of them only apply to the standard TOSCA. Clearly, to support our SDL we could customize one of them. However, this would make SDL incompatible with the other existing TOSCA implementations.

To avoid the customization and preserve the compatibility with the existing TOSCA-based technologies, we rely on the TOSCA interfaces (see Section **??**). In particular, for all the SDL node types we declare a standard interface. Recall that all of the SDL types have some relationship (either direct or indirect, i.e., through a relationships path) with one TOSCA Compute node (see Section 4.1). Such relationship is resolved at runtime to identify the platform where the interface task must be executed. Thus, all the tasks defined by our standard interfaces result in a configuration command to be executed on a certain TOSCA Compute node. To clarify we propose the following example.

Example 8. Consider again the EoP vulnerability (node `www_vuln1`) of Example 3. Figure 11 shows the continuation of the declaration of `linux.vulnerability.EoP` (given in Figure 9). Clearly, the vulnerability must be enabled by properly configuring `www`, i.e., the Compute node where the privilege escalation takes place. In this case, the node `www` is identified as the system hosting the `User` nodes (namely, `fromUser` and `toUser`) related to the vulnerability.

```
#!/bin/bash

# read username attributes
fromUser= #...
toUser= #...

echo "$fromUser ALL=($toUser) NOPASSWD: /usr/bin/apt-get" >> /etc/sudoers
```

Figure 12: Implementation of `eop-apt-config.sh`.

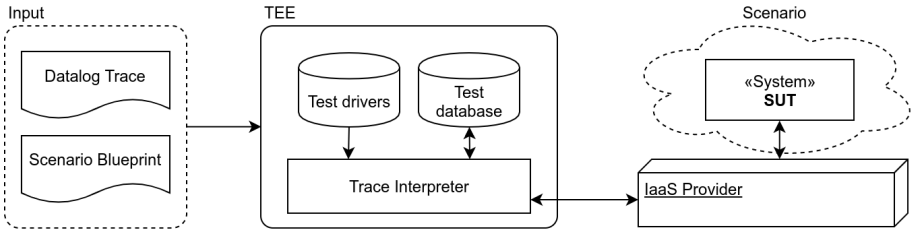


Figure 13: The Test Execution Engine of CRACK.

The implementation of the interface is given through a shell script (line 41). We report a core operations of the script in Figure 12. Briefly, the script enables a specific implementation of the EoP vulnerability for Debian-based OS. The vulnerability is activated through a misconfiguration of the `/etc/sudoers` file that allows underprivileged users to invoke the Debian package manager `apt-get` in passwordless mode.[23] In particular, the script retrieves the usernames of the two users. Usernames are obtained by reading the corresponding SDL attributes which contain actual, runtime values. Then, the vulnerability is enabled by appending the vulnerable configuration line to `/etc/sudoers`. □

5.3.2. Testing

All in all, the test execution process consists of translating a Datalog proof trace into an executable test and run it on the deployed scenario. The testing process is handled by a *Test Execution Engine* (TEE).

Figure 13 schematically depicts the TEE and its relationship with a running scenario. The test execution proceeds in this way. The next fact $F(\bar{v})$, namely the *Fact Under Test* (FUT), in the input Datalog trace is extracted and given to a *trace interpreter*. The interpreter retrieves a *test driver*, i.e., a script specifically designed to test facts referring to the predicate F, from an internal database. The test driver has a predefined interface and it may refer to values take from either the FUT, the scenario blueprint or runtime information generated by the test execution and stored in a *test database*. The structure of the test database is straightforward. In particular, it consists of a table for each Datalog predicate where each column corresponds to a parameter.

[23] `https://attack.mitre.org/techniques/T1169/`

```
ū := get_values_from_fut()
v̄ := get_values_from_blueprint()
w̄ := get_values_from_testdb()

sut := get_sut_from_blueprint()
test := get_runtime_from_blueprint(sut, fut)
insert_values_in_script(test, ū, v̄, w̄)
r := submit_script_to_iaas(sut, test)

if(is_not_successful(r)) then test_failed()
else insert_values_in_testdb(r) and test_passed()
```

Figure 14: Test driver pseudo-code.

Notice, however, that actual, runtime values may be considered when they simplify the testing operations. For instance, one might prefer to store the actual username of a user, rather then its SDL identifier as it appears in the Datalog trace (see Example 9).

The general structure of a test driver is given in Figure 14. Each driver starts by retrieving the necessary data from the FUT ($\bar{u}$), the blueprint ($\bar{v}$) and the test database ($\bar{w}$). Then it identifies the *system under testing* (SUT) by checking which SDL node declares the FUT in its behavior. Similarly, the test driver retrieves the corresponding $test$ script from the runtime property of the SUT. Before running the script, the actual test values must be inserted, i.e., passed as the input parameters of the script. Eventually, the script is submitted to the IaaS provider that executes it on the SUT and returns the output values. The output is a tuple on which a successful condition can be checked. If the check is not passed the test fails, otherwise the driver inserts the output values in the test database and terminates.

Example 9. Consider again the EoP vulnerability of our working example. Its *runtime* property contains the script of Figure 15.

The script starts by reading the inputs provided by the test driver (lines 6–9). For this purpose a utility function is used (`getParam`, line 3). There are four inputs passed by the driver. Two of them, i.e., `principal` and `host`, are taken from the FAU. Instead, `username` is taken from the corresponding attribute of the node `User`. The reason is that the script requires the actual username, rather than the `User` node identifier (which appears in the proof trace). The last parameter is the knowledge of the principal, i.e., the content of table `Knowledge` (for the current principal) in the test database. All in all, the script checks if the principal's knowledge contains a username that enables the exploitation, i.e., escalating the privileges. In details, the main body of the script amounts to a for loop iterating on each element `k` of `knowledge` (line 11). Then, through a regular expression matching (line 13), the script checks if the configuration file `/etc/sudoers` contains a line (i) starting with (username) `k`, (ii) containing the keyword `NOPASSWD`, i.e., indicating the passwordless command execution mode, and (iii) containing the exploitable command `apt` (see Example 8). If the

```
 1. #!/bin/bash
 2. # Test driver for hasAccount(principal, host, user)

 3. function getParam() {
 4.  # parses and reads the input tuples ū, v̄, w̄
 5. }

 6. principal=$(getParam $1)  # the principal
 7. host=$(getParam $2)       # the host
 8. username=$(getParam $3)   # user's username
 9. knowledge=$(getParam $4)  # principal's knowledge

10. r=""
11. for k in $knowledge
12. do
13.  if [ "$(grep -P "^$k.+NOPASSWD.+apt.+" /etc/sudoers)" ]; then
14.   r=$(su $k -c 'sudo /usr/bin/apt-get update \
15.    -o APT::Update::Pre-Invoke::="id"' | grep $username) && break
16. done

17. if [ ! "$r" ]; then
18.     username=""
19. fi

20. echo "($principal, $host, $username)"
21. # is_not_successful() iff $username == ""
```

Figure 15: Runtime script used to test the EoP vulnerability.

match occurs, the script executes the exploit (line 14) as the user `k` (`su $k -c`). The EoP exploit leverages a configuration option (`APT::Update::Pre-Invoke::`) for invoking arbitrary commands before updating the `apt` package index file. In particular, the script invokes the command `id` for printing the username of the current user. If the output matches `username`, i.e., the privileged user, the for loop breaks and `r` is assigned to the command output (lines 14–15). After the loop, `r` is left empty only if the script failed in running the exploit. Hence, the script overwrites `username` with en empty string (line 17-19). Eventually, the script returns the tuple `(principal, host, username)` (line 20). The driver checks whether the script failed by comparing `username` with the empty string. □

A test is successful when each step succeeds. In this case, we obtain an evidence that the proof trace has been preserved after the scenario deployment. Otherwise, we get a useful indication of what went wrong. In particular, a test failure amounts to the failure of a certain script. By reversing our mapping, we find which clause, appearing in the proof trace, is not satisfied by the deployed scenario. As a consequence, the blueprint can be inspected to understand and fix the error. When all the tests are successful the scenario is ready.

6. CRACK Demo

In this section, we provide a demonstration of CRACK applied to our working example. CRACK is available as a free open source software on GitHub[24]. The repository contains (i) the source code, (ii) the configurations and (iii) the library of SDL elements required for replicating the experiments described below. In particular, the SDL implementation consists of 56 node types, 20 capabilities and 11 relationships. All together, they amount to 3411 YAML lines of code (loc) for the type definitions, 1084 shell loc for the deployment interfaces and 500 shell loc for the runtime scripts.

CRACK is built on top of the Apache ARIA project (see Section 5.3.1). In particular, we rely on ARIA for supporting the design (see Section 4) and deployment (see Section 5.3.1) phases. Instead, the validation (see Section 5.2) and testing (see Section 5.3.2) phases are enabled through a plugin extension of ARIA. Moreover, we use pyDatalog[25] as the Datalog engine for the verification module.

Our testing environment runs on an Ubuntu Linux, version 16.04.5 LTS, installed on a two Intel Xeon Processors E5440 server with 32GB of RAM. Finally, we use DevStack[26] for installing OpenStack 3.16.0 (Rocky).

The outline of the demonstration follows. We start from the design of the first scenario of our case study in Section 6.1. Then, we validate (Section 6.2), deploy and test (Section 6.3) the scenario. In Section 6.4 we simulate the red team activity on the running scenario. Also, there we evaluate the effort for migrating between the first and the second scenario of the case study.

6.1. Design

For the first scenario, we carry out the design from scratch. That is, we define all the elements without assuming any prior scenario design. The design process starts from the theater elements, e.g., networks and compute nodes, and incrementally proceeds to the scenario aspects, e.g., vulnerabilities.

The entire scenario is encoded in 1018 lines of code over 10 YAML files. The files are structured as follows.

```
we.yaml           # top level of the working example
├── network.yaml  # network infrastructure specification
├── client.yaml   # client host specification
├── db.yaml       # db host specification
├── firewall.yaml # firewall host specification
├── ns.yaml       # ns host specification
├── provider.yaml # provider host specification
├── root-ns.yaml  # root-ns host specification
├── www.yaml      # www host specification
└── config.yaml   # input variables
```

Briefly, `we.yaml` is the entry point for the scenario specification. This file imports all the other YAML specifications and contains the declarations of the scenario

[24] https://github.com/enricorusso/CRACK

[25] https://sites.google.com/site/pydatalog/

[26] https://docs.openstack.org/devstack/latest/

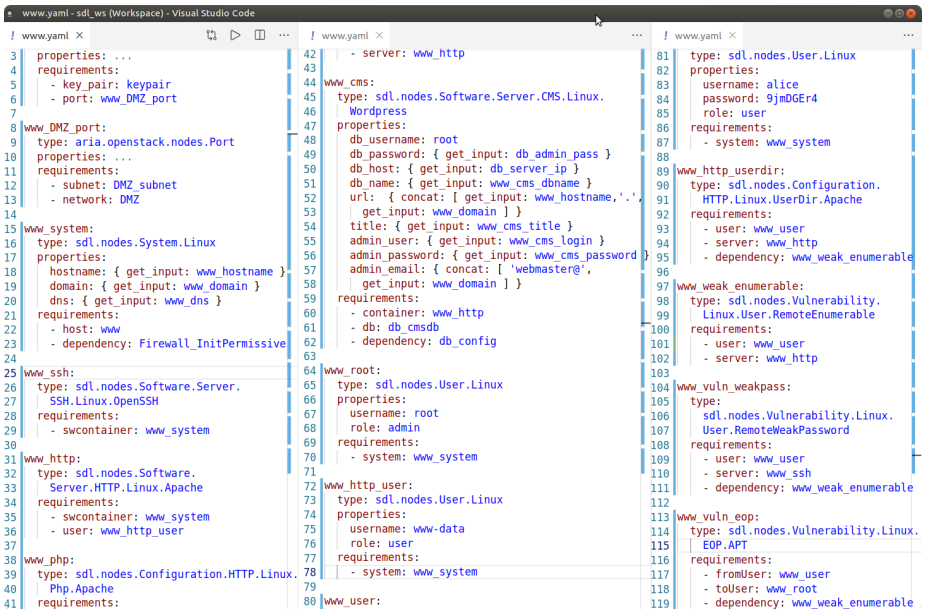


Figure 16: The content of `wwww.yaml`.

principals and goals. In particular, recall that in our scenarios we have only one goal for the `eve` (a.k.a. the red team), i.e., exfiltrating data from *db*. We use the label `DB_confidential` to denote the target data. As a consequence, the goal amounts to `knows('eve','DB_confidential')`. The network infrastructure is defined in `network.yaml`. Instead, each host is defined in a separate YAML file. A host file contains the definition of the corresponding Compute node and its configuration. Finally, we introduce a utility file `config.yaml`. Such a file contains values assigned to *common reconfigurable* properties, e.g., network addresses, domain names and usernames. These properties can be modified for quickly reshaping the scenario by acting on a single file.

Figure 16 shows the YAML specification of `www` for the first scenario of the working example. We start by defining the OpenStack Compute node running `www` and its connection to the DMZ subnetwork via `www_DMZ_port` (lines 1 and 8, respectively). Then, we connect `www_system` (line 15) to `www`. As stated in Section 4.2, `www_system` enables the relationships with the SDL nodes. For instance, the ssh server `www_ssh` (line 25), the Apache[27] HTTP server `www_http` (line 31), the Apache module for running PHP[28] pages `www_php` (line 38), and WordPress[29] `www_cms` (line 44) are software components running on `www`. Also, we define three users, i.e., `www_root` (line 64), the `www_http_user` (line 72) and `www_user` (line 80). Fi-

[27] `https://httpd.apache.org/`
[28] `https://www.php.net/`
[29] `https://wordpress.org/`

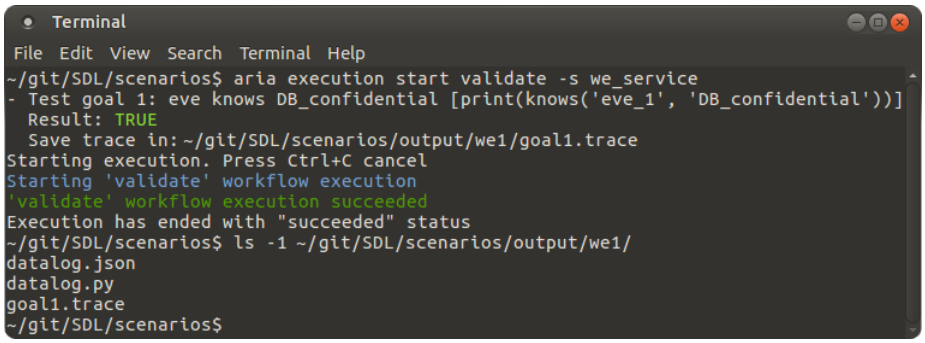


Figure 17: The execution of the *validate* workflow.

nally, we add one (mis)configuration and three vulnerabilities. Briefly, they implement they implement the three vulnerabilities introduced in Section 2.

1. The configuration `www_http_userdir` (line 89) exposes the users home directories and the vulnerability `www_weak_enumerable` (line 97) modifies the usernames to ensure that they are enumerable (in the sense explained in Section 2).
2. The node `www_vuln_weakpass` (line 104) configures the weak password for `www_user`.
3. The node `www_vuln_eop` (line 113) injects the EoP vulnerability (see Example 8).

At the end of the design step, we submit `we.yaml` to ARIA. This operation includes the type checking step (see Section 5.2). When the operation terminates, the scenario, called a *service* in the ARIA terminology, is saved as *we_service*. Although the scenario is technically deployable, we still have to validate it with CRACK.

6.2. Validation

CRACK provides a TOSCA workflow, called *validate*, that implements the validation procedure described in Section 5.2. We invoke the *validate* workflow through the ARIA workflow execution engine. Figure 17 shows the output for the first scenario of the working example. The generated Datalog specification in saved as `datalog.py` using the pyDatalog format.

Since the test goal is verified (`Result: TRUE`), the `goal1.trace` file containing the Datalog proof trace is generated. Also, the workflow creates `datalog.json`. Briefly, it contains a mapping between the Datalog terms and the SDL type declaring them. Such a mapping binds each FUT appearing in the proof trace with the corresponding SUT during the test execution process (see Section 5.3.2). The mapping is stored as a list of records of the form

```
Nat : { "node": SDL node identifier,
        "type": SDL node type,
        "key" : Behavior/runtime identifier }
```

```
+ isConnected(1, 'www', 'DMZ_subnet')
+ hasUser(2, 'www_root', 'www', 'None', 'admin')
+ listeningOn(3, 'www', 'tcp', '22')
hasAccount(4,A, 'www', 'www_user') <= knows(ID1,A, 'alice') &
 hasUser(ID2, 'www_user', 'www',P,R) &
 listeningOn(ID3, 'www', 'tcp', '22') &
 hostACL(ID4,K, 'www', 'tcp', '22') & hasAccount(ID5,A,K,V)
hasAccount(5,A, 'www', 'www_root') <=
 hasUser(ID1, 'www_root', 'www',P,R) &
 hasAccount(ID2,A, 'www', 'www_user')
knows(6,A, 'alice') <= listeningOn(ID1, 'www', 'tcp', '80') &
 hostACL(ID2,K, 'www', 'tcp', '80') & hasAccount(ID3,A,K,V)
+ hasUser(7, 'www_user', 'www', '9JmDGEr4', 'user')
+ listeningOn(8, 'www', 'tcp', '80')
knows(9,A, 'venerus') <= hasUser(ID1, 'www_http_user', 'www',P,R) &
 hasAccount(ID2,A, 'www', 'www_http_user')
knows(10,A, 'venerus') <= hasUser(ID1,U, 'www',P, 'admin') &
 hasAccount(ID2,A, 'www',U)
+ hasUser(11, 'www_http_user', 'www', 'None', 'user')
```

Figure 18: An excerpt of `datalog.py`.

```
[...]
New fact: hasUser(7,'www_user','www','9JmDGEr4','user')
/* proof of knows(6,'eve','alice') */
New fact: knows(6,'eve','alice')
/* and so on ... */
New fact: hasAccount(4,'eve','www','www_user')
[...]
New fact: hasAccount(5,'eve','www','www_root')
[...]
New fact: knows(10,'eve','venerus')
[...]
New fact: knows(21,'eve','DB_confidential')
```

Figure 19: An excerpt of `goal1.trace`.

where `Nat` is a unique natural number, `node` is the name of a node in the blueprint, `type` is the SDL type of the node and `key` denotes a valid entry in the behavior/runtime mapping of the node (see Section 4.3).

Figure 18 shows an excerpt of the `datalog.py` file. Briefly, it amounts to the Datalog facts and clauses generated for `www`. The syntax follows the pyDatalog format where facts are preceded by the '+' symbol and clauses use '<=' and '&' in place of ':-' and ',' (respectively, cf. Section 3.3). Furthermore, notice that each Datalog predicate has an extra argument, i.e., the first one, being a natural number. Such an argument is assigned to a unique constant in each fact and left term of the clauses. Instead, the argument appears as an unconstrained variable in the premises of each clause. All in all, this argument maps each step of a proof trace to a corresponding entry in the `datalog.json` file (through the values `Nat` discussed above).

Figure 19 contains a fragment of the proof trace generated by the *validate* work-

flow. For the sake of presentation, we omit the proof details and we only report the facts proved at each step by the pyDatalog deduction system. The proof succeeds by achieving the goal, i.e., `knows(21,'eve','DB_confidential')`, at the final step.

6.3. Deployment and testing

The deployment process results in the infrastructure depicted in Figure 20. The infrastructure contains the components discussed in Section 2. It is worth noticing that the actual topology also includes the networks that were originally only implicitly defined. For instance, `outside` that connects `firewall` to `provider`.

Another TOSCA workflow of CRACK, called *test*, executes the test for the proof trace of Figure 19. The result is partially reported in Figure 21. The structure of the test follows the facts of Figure 19, but it refers to actual runtime values. For instance, the username and password of `www_user` are dynamically configured by `www_weak_enumerable` and `www_weak_password` to `manager` and `qwerty` (respectively). Finally, notice that the test fails. When this happens, CRACK displays the execution log of the failed script for supporting the scenario debugging process.

We now interpret and fix the error that caused the failure of Figure 21. From the FUT identifier, i.e., 4, we find in `datalog.json` that the clause was declared by the node `www_weak_password`. The corresponding runtime consists of a script running a dictionary-based brute force over ssh by using Hydra[30]. From the log we discover that the error occurred because the password-based authentication is not enabled on ssh. The reason is that, by default, ssh is configured only to support key-based authentication. To solve this issue, one can add the property `PasswordAuthentication: "yes"` to the node `www_ssh` (see Figure 16). Once the property is added, the test concludes successfully.

6.4. Execution and evaluation

We now describe the scenario execution by simulating the attack of the red team. The steps of the attack are depicted in Figure 22. Initially, the red team scans the machine hosting the home page of ACME Corp (a). Running nmap shows that the server is open on ports 80 and 22. Then, they run the nmap script *http-userdir-enum*[31] and enumerate two users, i.e., *backup* and *manager* (b). The next step (c) is brute forcing the password of *manager* using *hydra* and the *rockyou*[32] wordlist (see Section 6.3). Once the red team has the password, they can log in `www` as *manager* (d) and execute the privilege escalation discussed in Example 8 (e). Finally, they leverage the root privileges to read the Wordpress configuration file `wp-config.php` and obtain the access credentials to the database (f) which accomplishes the data exfiltration (g).

Some statistics about the working example are reported in Table 2. In particular, for the two scenarios we provide (i) the size of the scenario blueprint (in terms of YAML

[30] `https://github.com/vanhauser-thc/thc-hydra`
[31] `https://nmap.org/nsedoc/scripts/http-userdir-enum.html`
[32] `https://github.com/danielmiessler/SecLists/tree/master/Passwords`

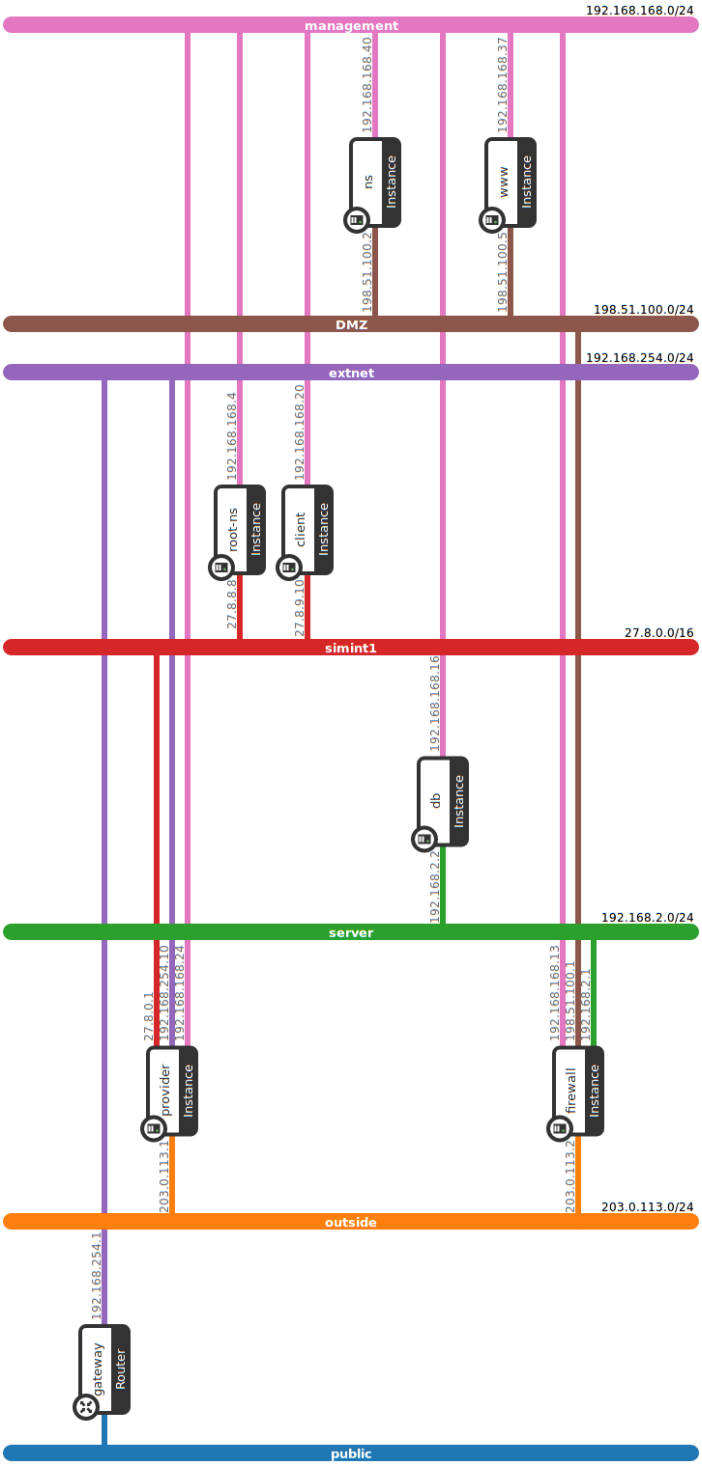


Figure 20: The network topology view of the working example theater in OpenStack.

```
[...]
Verify: hasUser(7,'www_user','www','9JmDGEr4','user')
Execution of test 'hasUser' on www
with parameters 'manager' 'www' 'qwerty' 'user'
Result: OK
[...]
Verify: knows(6,'eve','alice')
Execution of test 'knows' on www
with parameters 'eve' 'alice'
Result: OK
[...]
Verify: hasAccount(4,'eve','www','www_user')
Execution of test 'hasAccount' on www
with parameters 'eve' 'www' 'manager'
Result: FAILED
Log:
 Hydra v8.1 (c) 2014 by van Hauser/THC
 [...]
 [DATA] attacking service ssh on port 22
 [ERROR] target ssh://127.0.0.1/ does not support password authentication.
 [...]
```

Figure 21: An excerpt of `goal1-test.log` containing a failed test step.

	BLUEPRINT		SPECIFICATION		TEST EXECUTION	
#	size	types	size	time	FUTs	time
1	1018	31	142	9.51	108	157.88
2	991	27	138	8.42	92	101.49
	34	5	4	1.09	16	56.39

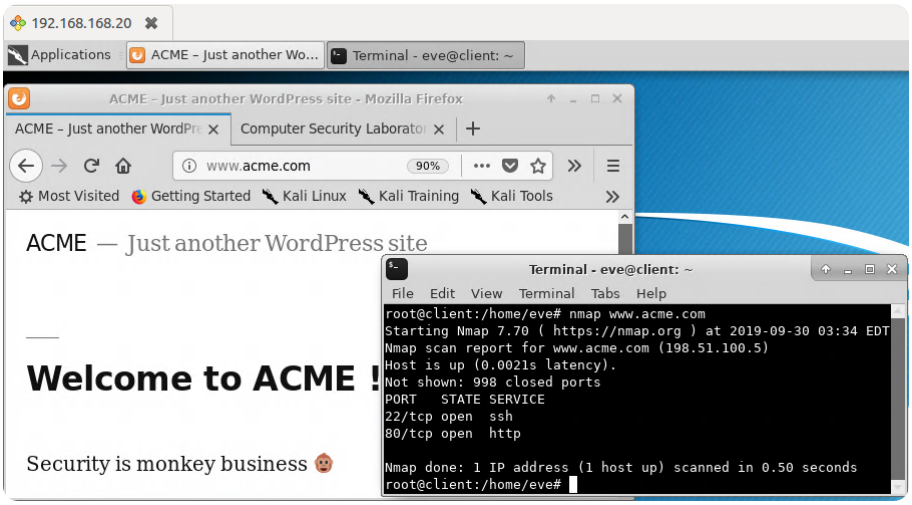
Table 2: Numbers of the two scenarios.

loc and number of node types), (ii) the size (Datalog loc) and and time[33] (seconds) for the Datalog specification verification, and (iii) the size (number of FUT) and time (seconds) for the test execution. Also, in the last line, we report the differences between the two scenarios. In particular, for the blueprint we report the number of loc and types that must be modified to obtain Scenario 2 starting from Scenario 1. It is worth noticing that this process only requires to add/remove 5 types (16.13%) which is achieved by modifying 34 loc, that is the 3.34% of the entire blueprint.

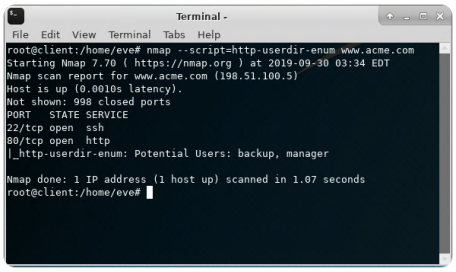
7. Related work

A preliminary version of our SDL appeared in [25]. The present work extends the previous proposal in a number of ways, being the development of CRACK the main one.

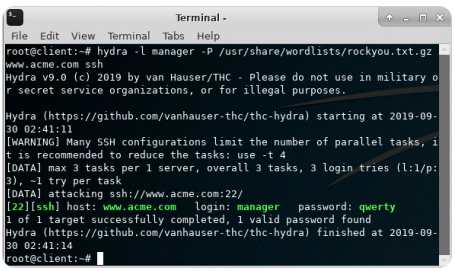
[33] Specification and test times include both the generation and the execution of the verification/test engine.



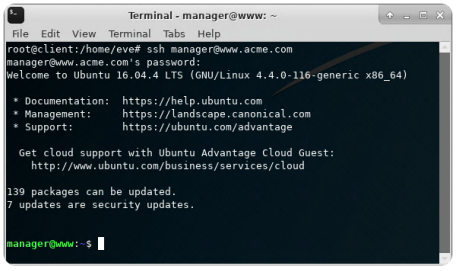
(a) Connecting to `client` (via remote desktop) and scanning `www`.



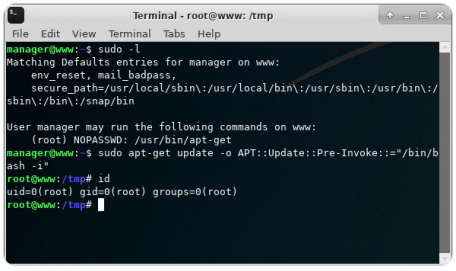
(b) Enumerating the users.



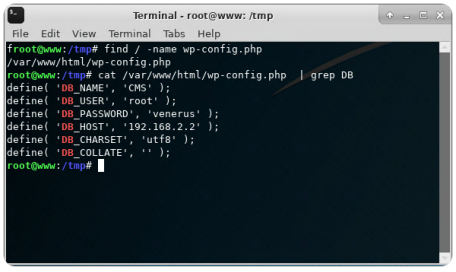
(c) Brute forcing the password.



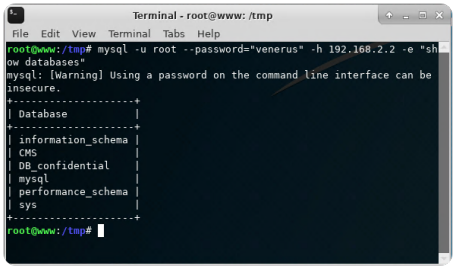
(d) Accessing `www` via ssh.



(e) Executing the privilege escalation.



(f) Reading the Wordpress configuration.



(g) Accessing the database content.

Figure 22: Scenario execution from the perspective of the red team.

The growing demand for cyber security professionals with hands-on skills is boosting the development of Cyber Ranges as well as training environments in general. In [26] Yamin et al. present a survey of Cyber Ranges and security testbeds. There, they also provide a taxonomy and an architectural model of a generic Cyber Range. CRACK (see Section 3.1) complies with their architectural requirements. Moreover, they mention a number of facilities, e.g., random traffic generators, that, although not yet implemented in CRACK, are compatible with our proposal.

In terms of exercises executed on a Cyber Range, Locked Shields [1] is possibly the most famous initiative. It is an annual event relying on a large, complex and heterogeneous scenario. The design phase of this exercise is based on a theater which is updated every year. Both design and testing require the effort of many experts for several months, while the execution phase only lasts for a few days. These are the conditions that inspired our work.

Similarly to CRACK, *ALPACA* [27] creates complex scenarios according to some user-specified constraints. In particular, it generates virtual machines containing sequences of predefined vulnerabilities which the trainee exploits to achieve a specific goal. Interestingly enough, the sequence generation is driven by a Prolog-based AI engine. Unlike CRACK, since [27] does not include a test phase, there is no guarantee that the deployed scenario preserves the verified properties.

Vykopal et al. [28] present their experience in organizing an exercise inspired by Locked Shields. Their main contributions are a lesson learned and a workflow for the organizers of the cyber exercises. Their exercises are specifically developed for the KYPO Cyber Range [7] that also provides a visual design interface. In this paper, we followed a similar line of reasoning and we proposed a methodology to automatize part of the workflow of [28]. Unlike [7], we use a standard IaC design language based on TOSCA. This allows for the integration with several cloud providers.

ADLES [5], *CyRIS* [6] and *SecGen* [29] are tools for the generation and management of cyber exercises. All of them introduce their own YAML-based language to describe the training environment and provide the corresponding module for deploying it. In particular, [6] also adds few advanced features such as attack emulation and traffic monitoring. Instead, [29] includes a catalog of vulnerabilities that are randomly injected in the generated scenario. Since none of them include a verification procedure, the generated exercises cannot be validated against general, fine-grained goals.

Both [30] and [31] propose a cyber scenario description language. The language of [30] provides a fast reconfiguration mechanism for shuffling a list of predefined scenarios, e.g., by changing the network address spaces. However, their language does not support the construction of new scenarios from scratch as we do with our SDL. Noticeably, [31] introduces several concepts that are also present in our SDL, e.g., software, artifacts, constraints, objectives, and actors. Nevertheless, in [31] these concepts are only informally stated, while our SDL provides a rigorous and operational definition.

8. Conclusion

In this paper we presented CRACK, an open source tool for modeling, validating and testing the scenarios for a Cyber Range. To this aim we proposed the *scenario*

definition language (SDL), an extension of TOSCA that introduces several, new features. For instance, we defined novel types for modeling, e.g., vulnerabilities and goals, as well as a Datalog semantics. The Datalog translation enables the validation of the scenario and generates proof traces that drive the automatic testing process.

There are several, future directions for this line of research. Among them, we will investigate the automatic generation of attack/defense strategies. These strategies can both improve the training process by mimicking the behavior of an attacker/defender in a simulation. Moreover, we plan to organize real training sessions to test the advantages of our approach in the organization of the cyber exercises.

References

[1] CCDCOE, Locked Shields, https://ccdcoe.org/exercises/locked-shields/, (Accessed on October 2019) (2019).

[2] T. E. Bell, Final Technical Report. Project Boeing SGS, Tech. rep., The Boeing Company, Seattle, WA (United States) (2014).
URL https://www.osti.gov/biblio/1177423

[3] Cyberbit, Cyber Range Training and Simulation, https://www.cyberbit.com/solutions/cyber-range/, (Accessed on October 2019) (2019).

[4] SANS, NetWars, https://www.sans.org/netwars/, (Accessed on October 2019) (2019).

[5] D. C. de Leon, C. E. Goes, M. A. Haney, A. W. Krings, ADLES: Specifying, deploying, and sharing hands-on cyber-exercises, Computers & Security 74 (2018) 12–40.

[6] R. Beuran, C. Pham, D. Tang, K. ichi Chinen, Y. Tan, Y. Shinoda, Cybersecurity Education and Training Support System: CyRIS, IEICE Transactions on Information and Systems E101.D (3) (2018) 740–749.

[7] J. Vykopal, R. Oslejsek, P. Celeda, M. Vizvary, D. Tovarnak, KYPO Cyber Range: Design and Use Cases, in: Proceedings of the 12th International Conference on Software Technologies - Volume 1: ICSOFT, SciTePress, Madrid, Spain, 2017, pp. 310–321.

[8] OASIS, OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC, https://www.oasis-open.org/committees/tosca/, (Accessed on October 2019) (2019).

[9] P. M. Mell, T. Grance, SP 800-145. The NIST Definition of Cloud Computing, Tech. rep., Gaithersburg, MD, United States (2011).
URL https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

[10] M. Artac, T. Borovsak, E. D. Nitto, M. Guerriero, D. A. Tamburri, Devops: Introducing infrastructure-as-code, 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C) (2017) 497–498.

[11] S. Ceri, G. Gottlob, L. Tanca, Syntax and Semantics of Datalog, Springer Berlin Heidelberg, Berlin, Heidelberg, 1990, pp. 77–93.

[12] E. Dantsin, T. Eiter, G. Gottlob, A. Voronkov, Complexity and Expressive Power of Logic Programming, ACM Computing Surveys 33 (3) (2001) 374–425.

[13] G. Gottlob, E. Grädel, H. Veith, Datalog LITE: A Deductive Query Language with Linear Time Model Checking, ACM Trans. Comput. Logic 3 (1) (2002) 42–79.

[14] ARIA TOSCA, Apache ARIA TOSCA orchestration engine, http://ariatosca.incubator.apache.org/, (Accessed on October 2019) (2019).

[15] M. Rutkowski, L. Boutier, C. Lauwers, TOSCA Simple Profile in YAML Version 1.2, Tech. rep., OASIS (January 2019).
URL https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/os/TOSCA-Simple-Profile-YAML-v1.2-os.pdf

[16] A. Brogi, J. Soldani, Reusing cloud-based services with TOSCA, in: GI-Jahrestagung, Citeseer, 2014, pp. 235–246.

[17] C. Bodei, P. Degano, R. Focardi, L. Galletta, M. Tempesta, Transcompiling firewalls, in: L. Bauer, R. Küsters (Eds.), Principles of Security and Trust, Springer International Publishing, 2018, pp. 303–324.

[18] A. Brogi, A. D. Tommaso, J. Soldani, Sommelier: A Tool for Validating TOSCA Application Topologies, in: Model-Driven Engineering and Software Development - 5th International Conference, MODELSWARD 2017, Porto, Portugal, February 19-21, 2017, Revised Selected Papers, 2017, pp. 1–22.

[19] S. Köhler, B. Ludäscher, Y. Smaragdakis, Declarative Datalog Debugging for Mere Mortals, in: P. Barceló, R. Pichler (Eds.), Datalog in Academia and Industry, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 111–122.

[20] X. Ou, S. Govindavajhala, A. W. Appel, MulVAL: A Logic-based Network Security Analyzer, in: Proceedings of the 14th Conference on USENIX Security Symposium, Vol. 14 of SSYM'05, USENIX Association, Berkeley, CA, USA, 2005, pp. 8–23.

[21] Cloudify, Cloudify — Cloud and NFV Orchestration Based on TOSCA, https://cloudify.co/, (Accessed on October 2019) (2019).

[22] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner, Opentosca — a runtime for tosca-based cloud applications, in: Proceedings of the 11th International Conference on Service-Oriented Computing - Volume 8274, ICSOC 2013, Springer-Verlag, Berlin, Heidelberg, 2013, pp. 692–695.

[23] Atos, Digital transformation with Atos alien4cloud and Cloudify, https://atos.net/wp-content/uploads/2017/07/Alien4Cloud-and-Cloudify-White-paper.pdf, (Accessed on October 2019) (2019).

[24] OpenStack, Heat Translator, https://wiki.openstack.org/wiki/Heat-Translator, (Accessed on October 2019) (2019).

[25] E. Russo, G. Costa, A. Armando, Scenario Design and Validation for Next Generation Cyber Ranges, in: 2018 IEEE 17th International Symposium on Network Computing and Applications (NCA), IEEE, 2018, pp. 1–4.

[26] M. M. Yamin, B. Katt, V. Gkioulos, Cyber ranges and security testbeds: Scenarios, functions, tools and architecture, Computers & Security 88.

[27] J. Eckroth, K. Chen, H. Gatewood, B. Belna, Alpaca: Building Dynamic Cyber Ranges with Procedurally-Generated Vulnerability Lattices, in: Proceedings of the 2019 ACM Southeast Conference, ACM, 2019, pp. 78–85.

[28] J. Vykopal, M. Vizvary, R. Oslejsek, P. Celeda, D. Tovarnak, Lessons learned from complex hands-on defence exercises in a cyber range, in: 2017 IEEE Frontiers in Education Conference (FIE), 2017, pp. 1–8.

[29] Z. C. Schreuders, T. Shaw, M. Shan-A-Khuda, G. Ravichandran, J. Keighley, M. Ordean, Security Scenario Generator (SecGen): A Framework for Generating Randomly Vulnerable Rich-scenario VMs for Learning Computer Security and Hosting CTF Events, in: 2017 USENIX Workshop on Advances in Security Education (ASE 17), USENIX Association, Vancouver, BC, 2017.

[30] R. S. Weiss, S. Boesen, J. F. Sullivan, M. E. Locasto, J. Mache, E. Nilsen, Teaching Cybersecurity Analysis Skills in the Cloud, in: Proceedings of the 46th ACM Technical Symposium on Computer Science Education, SIGCSE '15, ACM, New York, NY, USA, 2015, pp. 332–337.

[31] B. K. Fite, Simulating Cyber Operations: A Cyber Security Training Framework, The SANS Institute.
URL https://www.sans.org/reading-room/whitepapers/bestprac/paper/34510