Computers & Security 116 (2022) 102635

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

TC 11 Briefing Papers

Modeling and executing cyber security exercise scenarios in cyber ranges

Muhammad Mudassar Yamin*, Basel Katt

Department of Information Security and Communication Technology, Norwegian University of Science and Technology, Teknologivegen 22, Gjøvik 2815, Innlandet, Norway

ARTICLE INFO

Article history:
Received 15 April 2021
Revised 13 January 2022
Accepted 31 January 2022
Available online 9 February 2022

Keywords:
Cyber range
Security
Exercises
Scenario
Modeling

ABSTRACT

The skill shortage in global cybersecurity is a well-known problem; to overcome this issue, cyber ranges have been developed. These ranges provide a platform for conducting cybersecurity exercises; however, conducting such exercises is a complex process because they involve people with different skill sets for the scenario modeling, infrastructure preparation, dry run, execution, and evaluation. This process is very complex and inefficient in terms of time and resources. Moreover, the exercise infrastructure created in current cyber ranges does not reflect the dynamic environment of real-world systems and does not provide adaptability for changing requirements. To tackle these issues, we developed a system that can automate many tasks of the cybersecurity exercise life cycle. We used model-driven approaches to (1) model the roles of the different teams present in the cybersecurity exercises and (2) generate automation artifacts to execute their functions efficiently in an autonomous manner. By executing different team roles such as attackers and defenders, we can add friction in the environment, making it dynamic and realistic. We conducted case studies in the form of operational cybersecurity exercises involving national-level cybersecurity competitions and a university class setting in Norway to evaluate our developed system for its efficiency, adaptability, autonomy, and skill improvement of the exercise participants. In the right conditions, our proposed system could create a complex cybersecurity exercise infrastructure involving 400 nodes with customized vulnerabilities, emulated attackers, defenders, and traffic generators under 40 minutes. It provided a realistic environment for cybersecurity exercises and positively affected the exercise participants' skill sets.

© 2022 Elsevier Ltd. All rights reserved.

1. Introduction

Conducting operational cybersecurity exercises is a difficult and challenging task (Pham et al., 2016), and creating the environment for such exercises is error-prone and mostly manual (Beuran et al., 2018). Creating such cybersecurity exercise environments and executing exercise scenarios can be done using cyber ranges (Yamin et al., 2020). Exercise scenarios can help with conducting hands-on operational cybersecurity exercises, as well as discussion-based or table-top exercises for educational purposes. Although both types of cybersecurity exercises are very important, we identified that there were inefficiencies in operational-based exercises (Yamin and Katt, 2018b), hence hindering their capabilities to be widely used for cybersecurity education, as well as for other public or private institutions.

Multiple researchers have been trying to make the process of executing operational cybersecurity exercises more efficient and less manual labour intensive (Pham et al., 2016; Russo et al., 2020; Schreuders et al., 2017; Yamin and Katt, 2019). These researchers were successful in dealing with the inefficiencies in cybersecurity exercises. However, most of the proposed solutions are just limited to deploying the exercise infrastructure only. Moreover, because of the dynamic nature of evolving cybersecurity threats, there is a need to model scenarios so that they are adaptable enough to accommodate changes in scenario requirements before and after scenario deployment. Additionally, for a realistic cybersecurity exercise environment, there is a need to autonomously execute cybersecurity exercise operations (Jones et al., 2015; Yamin et al., 2020). These operations range from emulating virtual users and generating network traffic to executing offensive and defensive operations within the exercise environment. Traditionally, these tasks have been the responsibility of human teams; therefore, there is the need to increase the automation level to make the exercise life

* Corresponding author.
E-mail address: muhammad.m.yamin@ntnu.no (M.M. Yamin).

https://doi.org/10.1016/j.cose.2022.102635
0167-4048/© 2022 Elsevier Ltd. All rights reserved.

cycle more efficient. So in the current work, we investigate four research questions:

1. How can we model and execute realistic cybersecurity exercise scenarios more efficiently?
2. Is it possible to make cybersecurity exercise models adaptable to changing requirements?
3. What operations in cybersecurity exercises can be executed autonomously to reduce dependability on human teams?
4. How much do such exercise scenarios improve the skills of cybersecurity exercise participants?

In the present work, we developed a software-based solution that addresses these questions. We used our solution to model, verify, deploy, test, and execute cybersecurity exercise scenarios in a controlled and safe environment. We performed a case study with top cybersecurity talents in Norway to evaluate our solution's performance on a set of defined matrices. Our developed solution is now being actively used in research and educational activities at the *Norwegian University of Science and Technology* (NTNU), the details of which are presented in the current paper.

To demonstrate the capabilities of the proposed system, two case studies were conducted during *Norwegian Cyber Security Challenge*(NCSC) 2020. NCSC is a national competition in Norway in which individuals ranging from 16–25 years old participate. NCSC has multiple rounds, and our developed solution was used in its final rounds for two case studies. The first case study involves a penetration testing scenario in which a small organization network was orchestrated. The network had three subnets: public, demilitarized zone, and internal network. Red team members had direct access to the public network, but they had to exploit vulnerabilities in the public network to pivot into the demilitarized zone and internal network. The participants were asked to find vulnerabilities in the network and achieve a very specific objective: updating the content of a file in a specific machine present in the internal network.

The second case study was also conducted during the NCSC 2020 finals, in which an attack/defense scenario was created. The scenario topology comprised five identical isolated networks and the execution was divided into two parts. In the first part, teams were tasked with patching the vulnerabilities present in their corresponding networks in a particular amount of time. In the second part, the isolated networks were interconnected, and the teams were tasked with attacking each other's network.

The paper is structured as follows: In Section 2, we provide the necessary background and related work for this research. Continuing that, in Section 3, we present the methodology that we employed for this research. Following that, in Section 4, we present the requirement and our design for modeling and executing cybersecurity exercises. After that, in Section 6, we discuss the implementation details of our solution and its evaluation through a case study. Finally, in Section 7 and 8, we conclude the article with a discussion and conclusion respectively.

2. Background and related work

2.1. Background

2.1.1. Cyber ranges

The word "Cyber Range" was first used during the 1970s and 1980s to describe a class of mainframe super computers developed by Control Data Corporation (CDC) (Lord, 1985; Weeden and Cefola, 2010). These computers were used in mathematically intensive tasks and the modeling of complex natural phenomena. Moving forward, in 2004, the US Congress directed the Center for Technology and National Security Policy (CTNSP) to develop a program *"to find practical ways in which the defense IT community can gain a mutual understanding of defense needs and industry capabilities and identify opportunities to integrate IT innovations in the U.S. military strategy."* (Kramer et al., 2006). In the report findings, the US Northern Command suggested the creation of a "Cyber range" for homeland security and homeland defense. Cyber ranges provide an interactive representation of an organizational environment, including tools, application, network architecture, and people functions; they are used for cybersecurity training, testing, and educational scenarios in a controlled and safe environment for providing hands-on cybersecurity experiences (NIST, 2020).

We conducted a detailed study on unclassified cyber ranges (Yamin et al., 2020) in which we discussed the scenarios, functions, tools, and architecture of such platforms. We developed a taxonomy of cyber ranges and proposed a functional architecture for building future cyber ranges. In the proposed architecture, there are six modules for the cyber range, which are presented in Fig. 1. One of the modules that deal with the run-time environment has different functions such as running the exercise infrastructure on emulated, simulated, or hardware infrastructure, generating attacks, user behavior, and traffic to add necessary realism into the environment.

This run-time environment is the prime necessity for conducting operational cybersecurity exercises. Other modules of cyber ranges also depend on this run-time environment for the scenario management and the exercise monitoring. This run-time environment also facilitates the operations of training, testing, and educational modules. Because of their central role, a lot of research has been conducted on cyber ranges (Pham et al., 2016; Russo et al., 2020; Schreuders et al., 2017; Yamin and Katt, 2019), and creating such a dynamic environment is usually the responsibility of the different teams present in cybersecurity exercises, which we discuss in Section 2.1.2.

2.1.2. Cyber security exercises

There are multiple teams involved in the cybersecurity exercise life cycle that perform different roles, as follows:

1. White Team
   White team members are subject matter experts who define the cybersecurity exercises objective and plan the scenario. They can assist the other participating teams in understating the scenario and can provide hints.
2. Green Team
   Green team members are responsible for deploying the cybersecurity exercises infrastructure as per the specification of the white team members' developed scenario. They are also responsible for the monitoring and maintenance of exercise infrastructure during the cybersecurity exercises.
3. Red Team
   Red team members are the attackers in the cybersecurity exercises. They attack the cybersecurity infrastructure developed by the green team members for achieving the objectives defined by the white team members.
4. Blue Team
   Blue team members are the defenders in the cybersecurity exercises. They defend the cybersecurity infrastructure developed by the green team members for achieving the objectives defined by the white team members.

2.1.3. Cyber security exercises scenarios

There are different types of operational cybersecurity exercise scenarios. We categorized them based on the network topology that is employed to execute the scenarios.

1. Jeopardy Style CTF
   The jeopardy-style capture-the-flag (CTF) competition uses the simplest network topology in which individuals and teams of

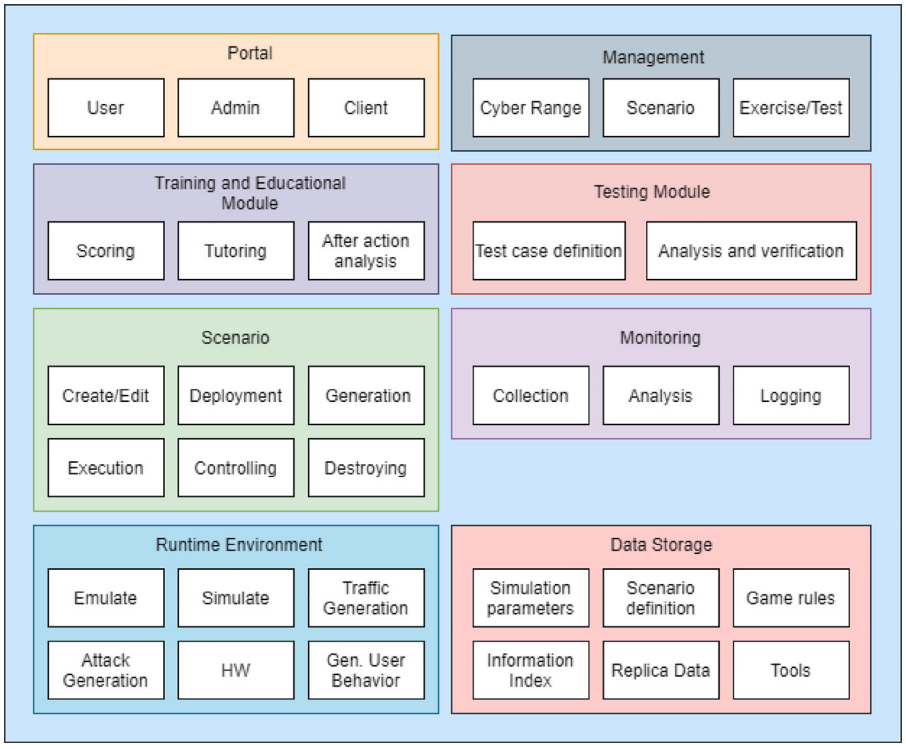


Fig. 1. Cyber range functional architecture Yamin et al. (2020).

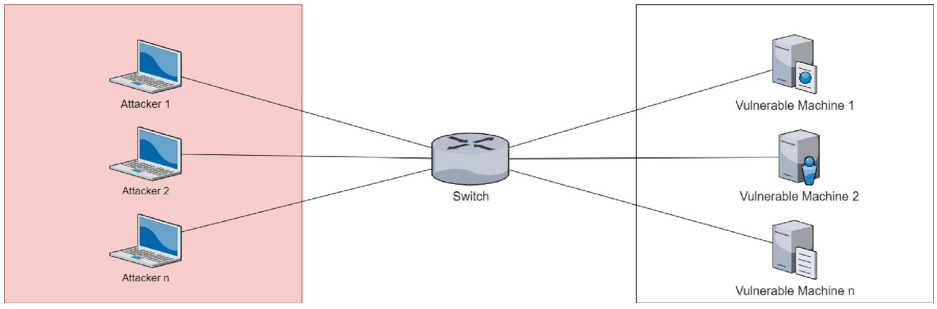


Fig. 2. Simple jeopardy-style CTF scenario network topology.

attackers have access to the machine over a network that has vulnerabilities. Attackers must exploit those vulnerabilities and retrieve a unique string, which is called the flag. The flag is then used for scoring purposes. The exercises are time-bound, so whoever scores the most will be declared the winner at the end. A simple representation of a jeopardy-style CTF competition network topology is presented in Fig. 2.

2. Attack/Defense
   Attack/defense cybersecurity exercises are team-based exercises in which each team has access to a network for which they are responsible for maintaining/defending the services of the different machines present in the network. The teams have the capability to launch attacks on the other team networks and disrupt the running of services. Flags and service availability statistics are mostly used for scoring purposes. A simple representation of an attack/defense cybersecurity exercise scenario network topology is presented in Fig. 3.
3. Red Team/Blue Team
   Red team/blue team exercises are objective-oriented, in which a team of attackers/red team is assigned an objective to penetrate into an organization and perform specific tasks such as data exfiltration and manipulation. The blue team performs actions related to incident response and forensics to figure out the red team's objectives. The exercise is conducted for skill improvement for the both red and blue teams and is mostly evaluated based on which team clearly achieved its objectives. A simple representation of the red team/blue team cybersecurity exercise scenario network topology is presented in Fig. 4.

2.1.4. Cyber security exercises life cycle

Developing, verifying, testing, and evaluating cybersecurity exercise scenarios is a challenge in and of itself. There is a whole life cycle involved in conducting cybersecurity exercises, Vykopal et al. (2017b) presented the lessons learned from an operation-based cybersecurity exercise in a cyber range. After analyzing multiple cybersecurity exercises, the researchers shared their *cybersecurity exercise life cycle*. The life cycle has five phases, as follows:

- Preparation
  In this phase, the exercises' objectives are defined, the scenario for the exercise is developed, and the necessary infrastructure for the scenarios is deployed. This phase takes from weeks to months in the cybersecurity exercise life cycle because it involves a lot of planning and development.

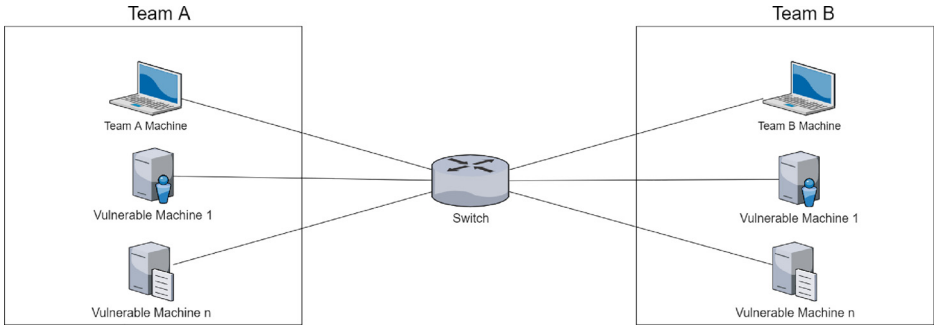


Fig. 3. Simple attack/defense scenario network topology.

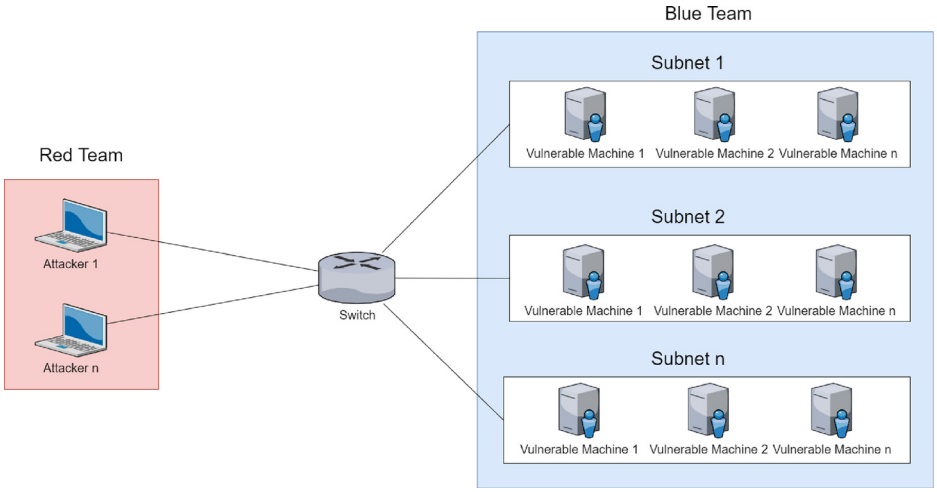


Fig. 4. Simple red/blue scenario network topology. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

- Dry run
  In this phase, the developed scenario and deployed infrastructure are tested by a team of experts. Changes are made to ensure that everything is working as planned. This phase also takes a few weeks because it involves debugging the exercises scenario and infrastructure for any error.
- Execution
  In this phase, the cybersecurity exercise is executed by different teams to try to achieve the objectives defined in the exercise scenario. This phase usually takes from a few days to few weeks, depending on the nature of the exercise.
- Evaluation
  In this phase, the different participating teams' performance in the cybersecurity exercise is evaluated based on the achieved objectives. This phase usually takes a few days to evaluate team performance.
- Repetition
  In this phase, the overall exercise is analyzed to identify any technical and nontechnical problems that need to be addressed before rerunning the exercise. This phase usually takes a few days to fix newly identified issues.

From our previous research, we have identified that the current way of conducting cybersecurity exercises is not efficient (Yamin and Katt, 2018b). Here, introducing automation at different phases of the cybersecurity exercise life cycle can greatly reduce the time required for such exercises (Yamin et al., 2018). Another problem with the current way of conducting cybersecurity exercises is their static nature because they do not offer active opposition to attackers and defenders (Jones et al., 2015) and are not adaptable to real-life environments. Hence, we propose a modern cybersecurity operation triad that can be applied to the cybersecurity exercise life cycle to make them more efficient and address these shortcomings.

2.2. Related work

A lot of research has been carried out in the development of security testbeds and cyber ranges; we will highlight some of the related works in the field. In 2000, the development of EMULAB/Netbed began (White et al., 2002), which is a combination of software and specialized hardware facilities. It has two parts: end nodes and core nodes. The end nodes host the experimental artifacts, while the core nodes are designed to be used as end nodes, simulated routers, traffic shaping nodes, and traffic generators. The core nodes allow the EMULAB/Netbed infrastructure to be reconfigured, making it useful for a variety of networking and cybersecurity experiments.

In 2004, the cyber-Defense Technology Experimental Research laboratory (DETER) (Mirkovic et al., 2010) started an over 300-node facility. DETER uses physical nodes and provides access to students over SSH DETER uses Emulab for the programming of routers present in the network to create new network topologies. At its inception, most of the work in DETER lab was done manually, with a later edition of the automation of traffic generation.

In 2008, the first phase of the Cyber Range and Training Environment (CRATE) (Almroth and Gustafsson, 2020) started. CRATE provides a hybrid cyber range environment; that is, it runs on both emulated virtual machines and dedicated hardware. CRATE supports the design, deployment, and execution of cybersecurity scenarios through a set of dedicated APIs that use JSON as the input.

CRATE is deployed over local servers, and access to users is provided through VPN. CRATE uses a high level of automation and divides the cyber range network into parts, first *event plane* and then *control plane*. In the event plane, internet traffic and attacks are emulated to add realism in the environment, while in the control plane, the cybersecurity exercise is conducted.

In 2012, researchers presented Telelab (Willems and Meinel, 2012), which was used to develop a single virtual machine-based lab environment for cybersecurity exercises. The interesting thing about this is that it uses an XML-based lab requirement specification to inject vulnerabilities in a virtual machine through a local agent running on the machine. Similar to Telelab, researchers created *SecGen* (Schreuders et al., 2017), which also uses XML configuration language for creating a virtual machine with vulnerabilities randomly selected from a catalog. They used Puppet and Vagrant for automating the process of vulnerability injection. Similar to Telelab and Secgen, in 2019, researchers developed *Alpaca* (Eckroth et al., 2019), which creates single virtual machines for cybersecurity exercises using Ansible. However, it has a prolog-based exercise planner, which is used to develop multi-step attack scenarios for complex training.

In 2016, researchers presented *Cyber Range Instantiation System for Facilitating Security Training* (CyRIS) (Pham et al., 2016). They automated the process of designing and deploying infrastructure for cybersecurity exercises using a YAML-based language. The researchers used Libvirt (lib, 2021) virtualization API for deploying exercises infrastructure on local servers. Continuing this, in 2017, they developed CyTrONE (Beuran et al., 2017), an integrated cybersecurity training framework. CyTrONE uses YAML-based language to do two things: first, create content for a learning management system (LMS), which is a mobile-based application and second is to design, deploy, and emulate attacks on a virtual environment. For the infrastructure orchestration, CyTrONE uses CyRIS.

In 2016, researchers presented (Yasuda et al., 2016) *a mimetic network environment construction system* Alfons. It was developed to mimic a realistic environment for malware execution and dynamic analysis in a local environment. It was written in Ruby using an XML-based environment composition file with SpringOS API calls to deploy the infrastructure on StarBed virtualized nodes. Alfons was not designed for conducting cybersecurity exercises, but it was developed to provide a platform to conduct forensic analyses for blue team members.

In 2017, researchers presented the KYPO Cyber Range (Vykopal et al., 2017a), which uses a JSON-based *scenario definition language*, which is used for designing and deploying the exercise infrastructure on an Openstack-based cloud environment. The scenario specification is translated into Ansible and Puppet scripts, which are then used for infrastructure orchestration. It supports the execution of attacks on infrastructures such as for phishing and DoS through SDL-defined templates. It is useful for conducting CTF-like competitions and has a physical facility for conducting cybersecurity exercises as well; however, it can also be accessed remotely.

In 2020, researchers presented the *Cyber Range Automated Construction Kit* (CRACK) (Russo et al., 2020), which supports the design, automated verification, deployment, and automated testing of complex cyber range scenarios. The CRACK framework is built on the extended version of a *scenario definition language*(SDL) (Russo et al., 2018), which is YAML based. SDL was developed to be compatible with open TOSCA (Ope, 2021) standards and is suitable for deploying infrastructure on the cloud in an agnostic manner. The researchers combined their SDL with Datalog logic programming for verification of the scenario properties, and they performed a case study for conducting a cybersecurity exercise on the infrastructure of a small organization. In the case study, three networks were behind firewall protection. Also, in 2020 researchers presented the AIT cyber range (Leitner et al., 2020), which uses Ansible and Terraform infrastructures for creating a cybersecurity exercise environment. The conducted exercises involved nearly 350 students.

We consider EMULAB, DETER, and CRATE as hardware-specific reliant testbeds that support cybersecurity exercises. We believe they can support large-scale cybersecurity exercises; however, this will require a lot of human effort for setting up the exercise environment. Telelab, Secgen, and Alpaca can be used for small-scale training exercises but are not suitable for large-scale exercise. The CYRIS, KYPO, CRACK, and AIT cyber ranges can support large-scale exercises using infrastructure as code and cloud technologies; however, they do not provide enough flexibility to change the exercise infrastructure after deployment. Moreover, the above solutions do not provide the necessary friction (Jones et al., 2015) for conducting realistic cybersecurity exercises.

In our solution, we utilized previous suitable techniques and added new elements to make the cybersecurity exercises more realistic. We utilized similar techniques from TELELAB to make virtual machines vulnerable in an agentless manner. We developed a JSON-based SDL similar to KYPO for infrastructure provisioning. We implemented a similar solution from CRACK for scenario infrastructure verification. We added similar capabilities in the CRATE event plane for emulating attacks, presenting user behavior, and generating network traffic. Finally, we introduced emulated defenders in the cyber range environment to make the exercise more realistic.

If we compare the proposed solution with other systems identified in the literature we can see that most systems are focusing on preparation of cybersecurity exercise environment. While *CRACK* is using formal verification for *Dry Run* purposes. *KYPO*, *CRACK* and our proposed system are using widely adapted cloud technologies that make them more computationally repeatable in terms of infrastructure deployment compare to other systems that are using very specific hardware and virtualization technologies. For dry run we used formal modeling and analysis for verification of different scenario properties before the scenario actual operational deployment. In terms of execution, the proposed system is introducing new capabilities like attacker and defender agents for conducting cybersecurity exercises in a more efficient manner. The process of evaluation can be automated in different ways, like flags style scoring. However, for complex exercises, detailed root cause analysis of system compromise is required which needs further investigation. A comparison between our proposed solution and other systems present in the literature based upon cybersecurity exercise life cycle is presented in Fig. 5.

3. Methodology

The overall research methodology that we used is **DSR** (*design science research*) (Vaishnavi and Kuechler, 2015). DSR focuses on the development and performance improvement of artifacts for increasing the functional performance of the artifact. These artifacts are usually algorithms and systems that involve human-computer interactions. DSR has three parts 1) knowledge flows, 2) process steps, and 3) output. Knowledge flows recursively integrate the information identified in the previous process steps into the next process steps. The process steps involve six processes: 1) awareness of problem, 2) suggestions, 3) development, 4) evaluation, and 5) conclusion. The execution of these processes results in the output in the form of 1) proposal, 2) tentative design, 3) artifacts, 4) performance measurement, and 5) results. We published our findings related to the proposal and tentative design in the following research articles (Yamin and Katt, 2018b; 2019; Yamin et al., 2020; 2018), in which we identified that the current way of executing cy-

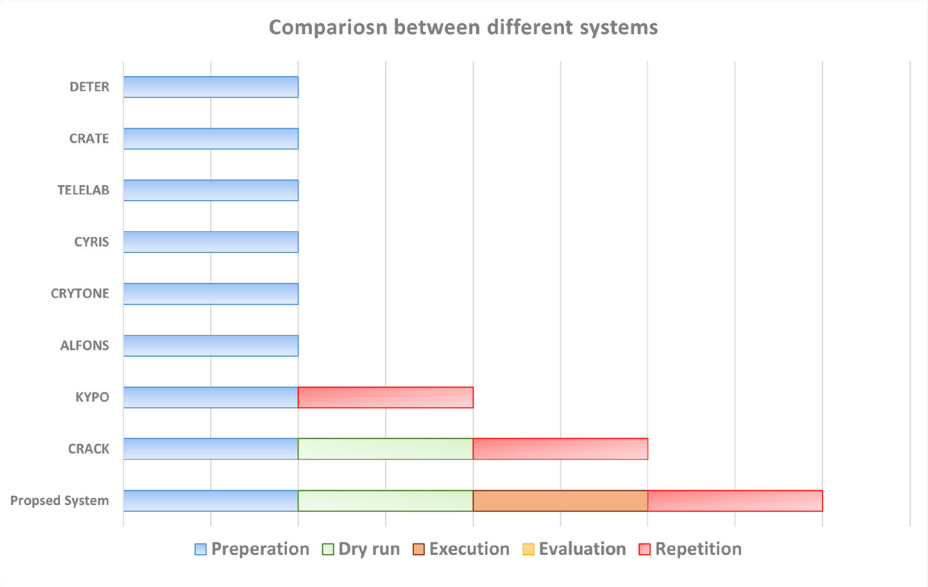


Fig. 5. Comparison between different cyber range systems with respect to cybersecurity exercise life cycle.

bersecurity exercises is not efficient and that automation can help to reduce this inefficiency.

For the artifact's development and to reduce these inefficiencies, we used **MDE** (*model-driven engineering*) (Schmidt, 2006). Model-driven approaches provide the ability to express complex domain-specific concepts in an abstract manner, which was very difficult in third-generation programming languages. This helps in increasing the productivity by a factor of 10 and improves consistency, traceability, and maintainability in the software development process (Chang et al., 2019). MDE combines two important technologies:

1. **DSML** (*Domain specific modeling languages*)
   In DSML, domain experts specify the domain knowledge in the form of a model. The model contains the concepts from the domain, their key semantics, relationship, and constraints associated with combining different concepts. DSML is used to specify the specific problem related to a specific domain in a DSML's instance, which is the abstract and human-readable representation of the problem.
2. Transformation engines and generators
   Transformation engines and generators take the DSML's instance and transform them into concrete software artifacts in an automated manner. This automated process of software artifact generation involves multiple steps, which include, but are not limited to, the following:
   - Text-to-model transformation
     In this step, the human-readable DSML instance is transformed into a computer-readable model. The DSML instance is usually the source code of a program that is transformed into a computer model using different parsing rules.
   - Model validation
     In this step, the model is verified using a set of defined semantic rules to ensure it is correct, which is done by construction implementation of the model.
   - Model to model, or model to code, transformation
     In this step, the model is transformed into another model, or a concrete code, that represents the problem specified in the DSML's instance.

The proposed system has an abstract scenario modeler that can generate two artifacts: one is a concrete DSL *domain specific language* instance that can be deployed to execute the cybersecurity exercise scenario, and the other is as a formal model of the scenario in Datalog, which can be used to analyze the different properties of the scenario before deployment. A model-to-model translation methodology was used in scenario modeler, which gave us the ability to logically verify some of the scenario properties. We developed a JSON-based **DSL** and used *text-to-model transformation* for orchestrating the exercise infrastructure and executing different cybersecurity operations. We used logic programming (Lloyd, 2012) to formally verify and analyze and verify different properties of the scenario for *model validation* before actual deployment. We used *model-to-model transformation* to combine the DSL model with a formal model, hence providing *meta-model conformance*. For the practical implementation of the artifact, we used different programming languages and operating system automation techniques, ranging from Python, Bash, Power shell, HEAT templates, and many more. The programming languages and techniques were selected with no particular preferences and were employed as the functionality's need arose. The artifact was developed in a very modular way. Each module can work independently from each other, providing us with a lot of flexibility in executing different cybersecurity operations.

For performance measurement of the developed artifact, we employed **applied experimentation** in operational cybersecurity exercises (Edgar and Manz, 2017). In *applied experimentation*, the performance of the developed artifact is measured against a set of predefined test cases and benchmarks. These were used to evaluate the overall performance of the artifact. Because the developed artifact involved system performance for deploying the exercise infrastructure and skill improvement of exercise participants, we used a mixed methods approach to gather quantitative and qualitative research data. We conducted a case study in which a cybersecurity exercise scenario was deployed and executed. We measured different quantitative matrices such as the scenario deployment efficiency, its usability in cybersecurity exercises, its flexibility to accommodate new changes, its adaptability to be used in contexts other than the defined context, and its scalability. We conducted pre and post-exercise surveys to measure the qualitative matrices,

for example, realism and skill improvement from the cybersecurity exercise scenario execution.

4. Design

4.1. Requirements for modern cyber security operations

With the rapid advancement of technologies such as *IaC* infrastructure as code (Wha, 2020) and *SOAR* security orchestration, automation, and response (Wha, 2020), it has become increasingly apparent that cyber operations are evolving. This advancement was made to execute such operations in an efficient, adaptable, and autonomous manner. This enabled us to reduce the cybersecurity exercise life cycle's inefficiencies and execute them in an efficient, adaptable, and autonomous manner.

4.1.1. Efficient

As stated earlier, preparing the environment for cybersecurity exercises takes anywhere from weeks to months. Efficiency in this context implies that the process of creating the exercise scenario and infrastructure should not take more than a couple of hours to a few days and should be applicable and accurate, as per the specified requirements. Efficiency means reducing the time required for creating the scenario, which does not affect the scenario's applicability and accuracy. ***Applicability*** measures whether the deployed scenario is employable for conducting practical cybersecurity exercises. In comparison, ***accuracy*** measures whether the deployed scenario fulfills the specified requirements in the scenario model. Multiple factors can affect the timeline, which may include the size and complexity of the exercise, but these factors need to be addressed in a systemic manner to remove inefficiency.

4.1.2. Adaptable

Here, adaptability implies that the deployed cybersecurity exercise infrastructure is flexible and scalable enough to adapt to new changes per the chaining scenario requirements. ***Flexibility*** measures the deployed scenario's capability to accept changes after deployment; in comparison, ***Scalability*** measures whether the deployed scenario is expandable enough to accommodate additional teams in the scenario. The adaptability will enable the cybersecurity exercises scenario developers to adapt the scenario for participants with different skill levels, creating a balanced environment for different participants.

4.1.3. Autonomous

In this context, autonomous implies that most of the cybersecurity operations are executed with minimum or no human interference. If we consider the example of autonomous cars, we can identify six levels of autonomy: *0: no automation, 1: driver assistance, 2: partial automation, 3: conditional automation, 4: high automation*, and *5: full automation* (Taxonomy, 2020). Cyber operations such as an attack and defense scenario can be autonomous in a cybersecurity exercise environment. Human intervention is still possible in monitoring the situation; however, this intervention must be conditional, which would be in contrast to the second level, where human monitoring is required.

4.2. Integrating modern operations with cyber security exercise life cycle

After an in-depth analysis of the five phases of the cybersecurity exercise life cycle, we divided these phases in to eight independent modular activities, and those eight modular process have 11 technical functions. The updated cybersecurity exercise life cycle is presented in Fig. 6, and details of the new activities and functions of the cybersecurity exercise life cycle are given below.

4.2.1. Preparation

Scenario Modeling In this phase of the scenario, a logical network topology with vulnerabilities was modeled, attacker and defender capabilities to exploit or defend those vulnerabilities were defined, and probable attack and defense strategies were analyzed and logically verified. We developed a DSL to specify different cybersecurity operational requirements during the exercise. A DSL provides a layer of abstraction to solve domain-specific problems without dealing with the necessary overhead of general purpose programming language. We simplified the cybersecurity operation in an exercise into five general operations: *infrastructure orchestrator, vulnerability injector, attacker agent, defender agent*, and *traffic generator*. These operations have their specific properties in a cybersecurity exercise scenario; to simplify things, the properties of our scenario modeling language are presented in BNF (Backus-Naur form). BNF is a notation technique for context-free grammar and is also used to describe the syntax of languages used in computing, such as computer programming languages.

1. **Scenario Language Design**
   The scenario modeling was performed through our developed scenario language, which was verified logically by Datalog. Our scenario language has multiple parts whose requirements are presented in the coming sections. Before defining the actual scenario modeling language, we first define some of the basic variables that were used in the language:
   These basic variables are used to define the characters, strings, integers, IP addresses, ranges, and CIDR used in rest of the language.
   - **Infrastructure Orchestrator**
     The infrastructure orchestrator has two parts *subnet* and *machine*. Subnet is used to represent the network with which machines are connected. It requires three things to be specified in the language:
     (a) CIDR (CIDR value like *10.10.0.10/24*)
     (b) Name (String value that indicates the name of subnet, like *Public*)
     (c) A network interface (String value of *Network ID* from Openstack)
     The second part *machine* is a host; hosts are virtual machines that are connected to the specified *subnet* and in which vulnerabilities are injected. It requires r things to be specified in the language:
     (a) Name (String value that indicates the name of a machine like *Machine1*)
     (b) Operating system (String value *operating system id* that is already uploaded over Openstack)
     (c) Key (String value *SSH key* that can be used for maintenance and monitoring)
     (d) Depends (String value name of *subnet* with whine *Machine* is connected)
     The BNF representation for specifying the requirements of infrastructure orchestration is as follows:
   - **Vulnerability Injector**
     When the *machines* are deployed, then the vulnerability injection can be done. The process *vulnerability injection* requires the following properties:
     (a) MachineIP (IP address value of a deployed machine like *10.10.1.10*)
     (b) MachineUserID (String value that indicates admin user account on a deployed machine like *root*)
     (c) MachineUserPassword (String value that indicates the admin user password of the deployed machine like *toor*)
     (d) OS (String value that indicates the name of a machine like *Machine1*)

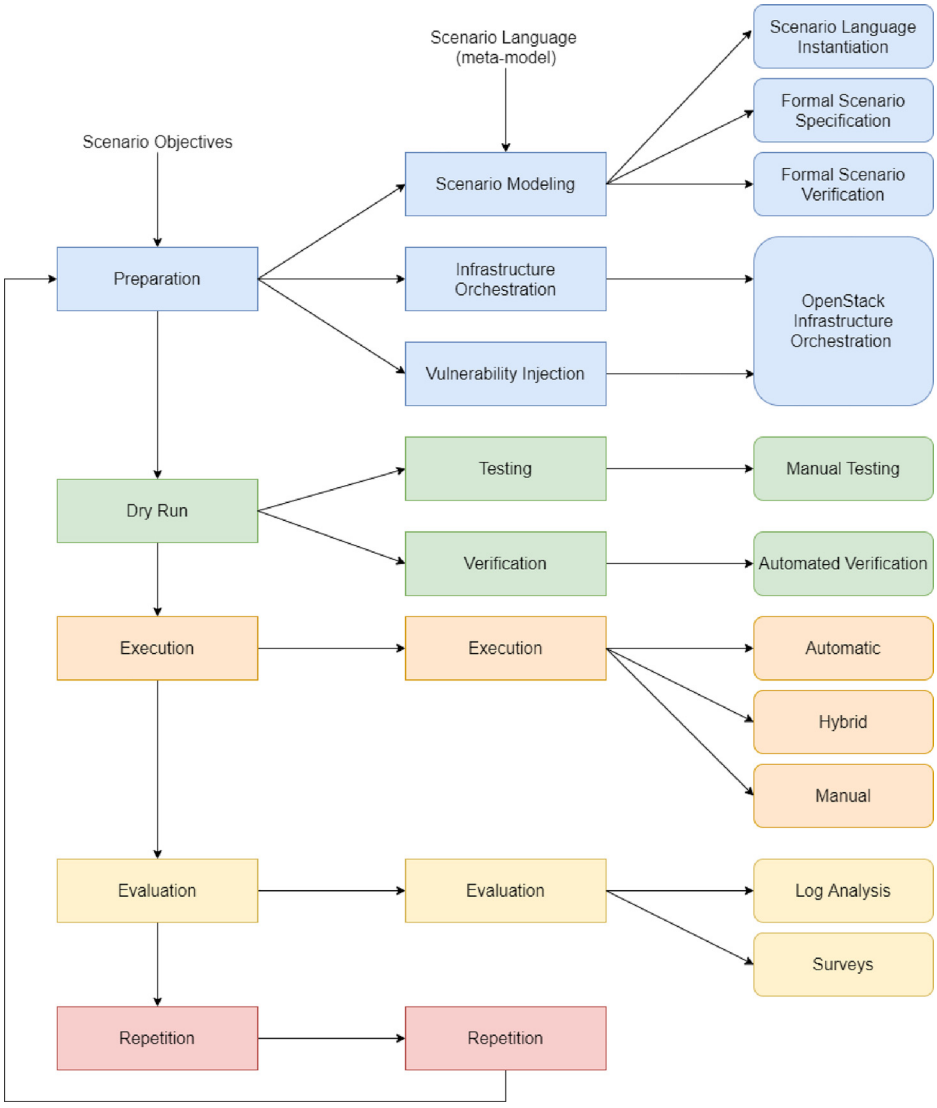


Fig. 6. Modern cybersecurity operations integrated with cybersecurity exercise life cycle.

(e) Vulnerability (String value that indicates the type of vulnerability that needs to be injected in the machine like *WeakPassword*)
(f) Parameter (String value that indicates the vulnerability-specific parameter like "passwd|apollo")

The BNF representation for specifying the requirements of the vulnerability injection is as follows:

- Attacker Agent After the vulnerabilities are injected, they can be verified by an *attack agent*. It can also be used to emulate attacker behavior during a cybersecurity exercise. The attacker agent requires six properties to be specified before its execution.
(a) ToolName (String value that indicates the tool name to be used by the agent like *nmap*)
(b) AgentIP (IP address value of a *Kali Linux*-based machine present in the network topology like *10.10.1.5*)
(c) AgentUserID (String value that indicates admin user account of the *Kali Linux* machine like *root*)
(d) AgentUserPassword (String value that indicates admin user password of the *Kali Linux* machine like *toor*)
(e) Argument ((String value that indicates attacker agent action-specific parameter like *-sS -sV*)
(f) Target (IP address value of a deployed machine like *10.10.1.10*)

The BNF representation for specifying the requirements of attacker agent behavior is as follows:

- **Defender Agent**
To add friction and realism in cybersecurity exercises, a host-based defender agent is developed that can be injected into the deployed machines. It has the following five properties.
(a) MachineIP (IP address value of a deployed machine like *10.10.1.10*)
(b) MachineUserID (String value that indicates the admin user account on deployed machine like *root*)
(c) MachineUserPassword (String value that indicates the admin user password of the deployed machine like *toor*)
(d) OS (String operating system name of the deployed machine like *Windows*)
(e) Parameter (String value that indicates the action-specific parameters of the defender agent in a CSV file like *netstat -ano| taskkill /F /PID ??*)

The BNF representation for specifying the requirements of defender agent behavior is as follows:

- Traffic Generator

To make the scenario execution dynamic with realistic traffic, two modules have been developed. First, there is TcpRelay, which can replay traffic from prerecorded network traffic using PCAP files. Second, there is VncBot, which can emulate user behavior from prerecorded VDO files. These modules have similar six requirements as those of the attack agent:

(a) ToolName (String value that indicates the tool name to be used by the agent like *VncBot*)
(b) AgentIP (IP address value of a *Kali Linux*-based machine present in network topology like *10.10.1.5*)
(c) AgentUserID (String value that indicates the admin user account of the *Kali Linux* machine like *root*)
(d) AgentUserPassword (String value that indicates the admin user password of the *Kali Linux* machine like *toor*)
(e) Argument ((String value that indicates an attacker agent action-specific parameter like *Test.vdo|toor* where *toor* is the password of the VNC-enabled machine deployed in the exercise infrastructure)
(f) Target (IP address value of a deployed machine like *10.10.1.10*)

The BNF representation for specifying the requirements of the traffic generator is as follows:

2. **Formal Scenario Specification**

Multiple models have been proposed for modeling the attackers' and defenders' behavior during a cyber engagement. These models focus on the chain of events that lead up to compromising the computer systems. Lockheed Martin Hutchins et al. (2011) put forward the *cyber kill chain* methodology to protect computer network damage espionage. The *cyber kill chain* consists of the following steps and stages:

(a) Reconnaissance: Looking out for intrusions, via email, conferences, and so forth.
(b) Weaponization: Malicious intent realized through PDF and word files for intrusion purposes.
(c) Delivery: Sending the intrusion payload 2004–2010 according to Lockheed Martin mostly sent through email attachments, USB, and websites.
(d) Exploitation: Intrusion is in and focuses on its target
(e) Installation: Providing a means of access to the compromised system to the adversary
(f) Command and control: After getting access, obtaining all the controls of the compromised, intruded system, and controlling it, done manually, not automatically via an internet controller server.
(g) Actions on objectives: To get access to the information and resources for which the last six phases took place.

They (Hutchins et al., 2011) also proposed the defender's course of action against the attacker in *cyber kill chain*, including what type of visibility the defender has and what tools and techniques a defender can use to stop the attacker. The course of action matrix is presented in Fig. 7.

Other models like *MITRE* (MIT, 2020) and the *unified kill chain* (Pau, 2020) provide more technical details of the attacker's and defender's steps, but at this stage, we chose the *cyber kill chain* for two main reasons: First, it is very well established and well-known modeling technique, and second, it offers a layer of simplicity and abstraction compared with other models, which focus more on core technical steps.

(a) **Scenario Formalization Background**

We used Datalog (Dat, 2020) for formal modeling of the scenario and to verify the different scenario properties. Datalog is a programming language based on a declarative logic (Lloyd, 2012). It is employed by researchers for large-scale software analyses (Naik, 2020), automatic evaluations of cybersecurity matrices (Zaber and Nair, 2020), and the verification of cybersecurity exercise scenarios (Russo et al., 2020), making it suitable as a formal model for cybersecurity exercise scenarios. It consists of two parts: facts and clauses. A fact conforms to the parts of the elements of the predicated phenomenon. A clause refers to information deriving from other subsets of information. Clauses rely on terms, which can contain variables; however, facts cannot. It adjudicates whether the specific term is adherent to the specified facts and clauses. If it happens to be so, the specific query is validated via a query engine, providing the prerequisite facts and clauses.

When running a Datalog operation, the specified conditions include a combination of two facts along with a singular clause. We assign a condition that if the query is valid, a specific response is to be expected at the end. The conclusion of the said experiment is that the specific response is received and that the query is satisfied. By utilizing the clauses via their variables, the engine can pinpoint and find the result. For a concrete example (Ceri et al., 1989), consider the facts *"John is the father of Harry"* and *"Harry is the father of Larry"*. A clause will allow us to deduce facts from other facts. In this example, consider we want to know *"Who is the grandfather of Larry?"*. We can use three variables *X,Y* and *Z* and make a deductive clause: If *X* is the parent of *Y* and *Y* is the father of *Z*, then *X* will be the grandfather of *Z*. To represent facts and clauses, Datalog uses *horn clauses* in a general shape:

$$L_0 : -L_1 ..., Ln$$

Each instance of L represents a *literal* in the form of a *predicate* symbol that contains one or multiple *terms*. A *term* can have a constant or variable value. A Datalog clause has two parts: the left hand side part is called the head, while the right hand side part is called the body. The body of the clause can be empty, which makes the clause a fact. A body that contains at least literal represents the rules in the clause. Lets us represent the above mentioned facts that *"John is the father of Harry"* and *"Harry is the father of Larry"* as follows:

$$Father(John, Harry)$$

$$Father(Harry, Larry)$$

The clause if *X* is the father of *Y* and *Y* is the father of *Z*, then *X* will be the grandfather of *Z* can be represented as follows:

$$GrandFather(Z, X) : -Father(Y, X), Father(Z, Y)$$

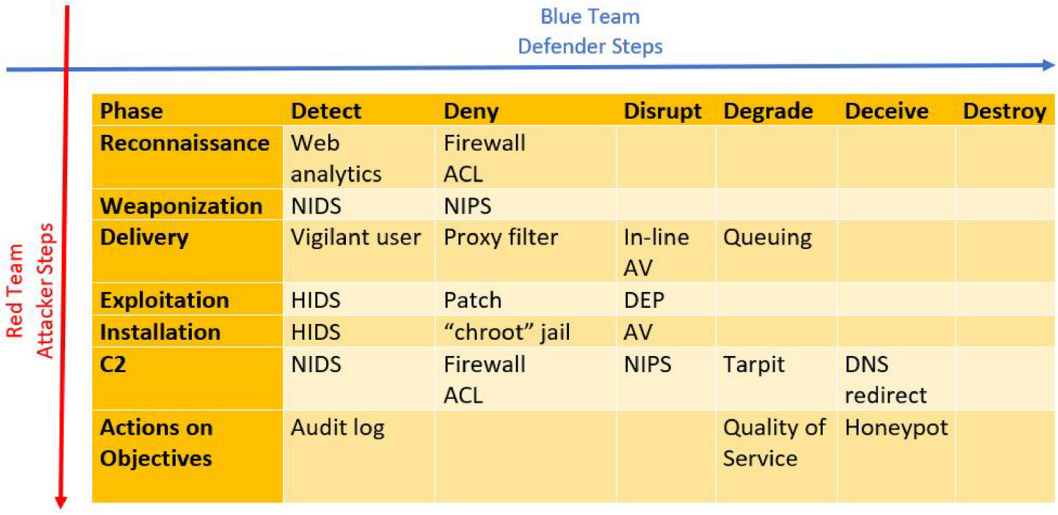
(b) **Scenario Formalization**

We have defined four basic predicates for our scenario modeling, which are *1) link, 2) vulnerable, 3) capability,* and *4) killchain*. The facts for the scenario model are presented as follows:

The *Link* predicate is logically represented as *Link(H,N)*, and it has the two variables of host *H* and network *N*. *H* is a string value that indicates the machine name (virtual machine name), while *N* is a string value that indicates the name of the network with which it is connected. For a concrete example, say a *Host* name *'Machine1'* connected with network name *'Public'* can be represented as follows:

$$Link('Machine1', 'Public')$$

The *Vulnerable* predicate is logically represented as *vulnerable(H,V)*, it has two variables host *H*, which is the specified machine name and *V*, a string value that indicates the presence of a particular vulnerability in *H*. A concrete example



Phase	Detect	Deny	Disrupt	Degrade	Deceive	Destroy
Reconnaissance	Web analytics	Firewall ACL				
Weaponization	NIDS	NIPS				
Delivery	Vigilant user	Proxy filter	In-line AV	Queuing		
Exploitation	HIDS	Patch	DEP			
Installation	HIDS	"chroot" jail	AV			
C2	NIDS	Firewall ACL	NIPS	Tarpit	DNS redirect	
Actions on Objectives	Audit log			Quality of Service	Honeypot	

Fig. 7. Cyber kill chain course of action matrix for attacker and defender Hutchins et al. (2011).

could be that *'Machine1'* is vulnerable to a *'SSHBruteforce'* attack and can be represented as follows:

$$Vulnerable('Machine1', 'SSHBruteForce')$$

The *capability* predicate is logically represented as *capability('V','A','DE')*, and it has three variables *V*, which is the vulnerability present in *H*, *A*, which is a *Bool* value that indicates whether a particular vulnerability *V* is exploitable by the attacker and *DE* that indicates whether a particular vulnerability *V* is defendable by the defender. A concrete example of a *'SSHBruteforce'* vulnerability that can be exploited by an attacker but cannot be defended by the defender is represented as follows:

$$Capability('SSHBruteForce', 'YES', 'NO')$$

The *KillChain* predicate is logically represented as *KillChain(H,R,W,D,E,C,O)*. It has seven variables host *H* and raw *cyber kill chain* process of reconnaissance *R*, weaponization *W*, delivery *D*, exploitation *E*, command and control *C*, and actions and objectives *O*. The *cyber kill chain* process variables have *Bool* values that were assigned based on *V* present in *H*. A concrete example for a *host 'Machine1'* that is completely exploitable as per the *cyber kill chain* can be represented as follows:

$$KillChain\big('Machine1', 'YES', 'YES', 'YES', 'YES', 'YES', 'YES', 'YES'\big)$$

A detailed scenario model with all its facts and clauses for the scenario presented in Fig. 10 is given in Appendix B. We used the clauses for logical verification of the scenario, which are presented in next section.

3. **Formal Scenario Verification**

We developed a tool for scenario modeling and verification. The tool integrates the concepts of our scenario language with Datalog modeling and provided us with the capability to model and verify cybersecurity exercise scenarios at the same time. In the scenario modeler, a user can specify the following:

(a) The networking topology that is required for the scenario
(b) The type of machines that are present in the network
(c) The type of vulnerabilities present in the machine
(d) The capabilities of the attacker and defender who can exploit or defend those vulnerabilities

After the specification is given to the modeler, the modeler can generate the instance required for the orchestration of cybersecurity exercise operations and a formal model in Datalog. Different type sof logical analyses can be performed before the actual deployment; some examples of security properties and questions that can be verified include the following:

(a) Which machines are reachable from a specific point in the network?
(b) Which machines are vulnerable to attack and can be reached by an attacker?
(c) Which machines are vulnerable to an attack but can be defended by the defender to limit attacker ingress into the network?

This is achieved by defining clauses that contain specific rules related to the scenario. First, we need to logically inter *link* different *hosts*. This can be done by creating a rule for a direct bidirectional link connection between the hosts using variables *X* and *Y*, as follows:

$$CanReach(X, Y) \leq Link(Y, X)$$

Second, similar to the grandfather and grandchildren case, to identify which *hosts* are indirectly connected in the network, we can create a new *CanReach* with variable *Z*. This can be used to find a direct link between *X* and *Y*, as well as an indirect link between *X* and *Z* through *Y*:

$$CanReach(X, Y) \leq Link(X, Y)$$
$$CanReach(X, Y) \leq Link(X, Z)$$

To check which machines are connected to a machine, for example, *Machine1*, with a specific vulnerability, for example, *BufferOveflow*, we can verify this by the following clause:

$$CanReach('Mahine1', Y) \,\&\, Vulnerable(Y, 'BufferOverFlow')$$

To check which *hosts* are connected to a vulnerable *host*, for example, Machine1, which is not defendable by a defender, we can verify it with the following clause:

$$Capability(V, 'YES', 'NO') \,\&\, CanReach('Mahine1', Y)$$
$$\&\, Vulnerable(Y, V)$$

To integrate *cyber kill chain* concepts into the model, we can specify the impact of a *vulnerability* injected in the *host* regarding whether it allows the attacker to perform steps like *reconnaissance*, *exploitation*, and so forth. This impact is a bool value that suggests the *cyber kill chain* stage the attacker can theoretically reach. We can create the following clause:

$$Capability(V, 'YES', 'NO') \,\&\, CanReach('Mahine1', Y)$$

$$\&Vulnerable(Y,V)\,\&\,KillChain(Y,'YES','YES','YES','YES','YES','YES','YES')$$

The verification of the scenario can be done on run time before the actual deployment of the exercise infrastructure. The verification process used the facts and clauses generated in Datalog syntax through our developed tool. With the help of mathematical logical operation, specially transitive relation, the Datalog engine can return properties verification results based upon given quires. The queries are similar to SQL quires and can use logical operators like AND or OR for verification of different properties. If the property is verified then the Datalog engine will return the output containing the elements that were verified by the query. If the output is empty then it is considered that the property is not verified. A complete scenario model with its properties verification steps are presented in Appendix B.

Infrastructure Orchestration In this phase, the modeled scenario that has been formally verified is transformed into an emulated network topology. This transformation is achieved by utilizing infrastructure as code technologies in which the template for the infrastructure orchestration is generated; this is deployed over a cloud instance. Multiple cloud providers like *Microsoft, Google, and Amazon* provide infrastructure orchestration technologies, but they are a closed source and paid solution. We opted for an open source solution called *Openstack*, which provides a functionality similar to *Microsoft, Google, and Amazon*; however, it also provides the capability to set up local cloud infrastructures without relying on third-party infrastructure. Openstack provides infrastructure orchestration to *HEAT* templates. HEAT templates provide an interface to specify the requirement for the network topology and the type of system present in the network. *Vulnerability Injection* Vulnerability injection in this type of network topology is a difficult process (Russo et al., 2018). Researchers have used different infrastructure configuration technologies like Ansible and Puppet for vulnerability injection (Leitner et al., 2020). However, we opted for a fundamentally different technique for vulnerability injection and developed our own custom vulnerability injector. Vulnerabilities are injected per the scenario model requirement using different operating system automation techniques. These OS automation techniques basically open an SSH connection in the machines deployed in the emulated network and manipulate software, services, and configuration using *Bash*, *Powershell*, and *Python* scripts. This enabled us to modify the scenario after infrastructure deployment and inject new vulnerabilities to make the scenario more flexible and balanced, if required.

4.2.2. Dry run

In the dry run phase, different scenario properties are checked to identify whether the deployed scenario fulfills the specified requirements in the scenario model. We divided this phase into two parts. *Manual Testing* Manual testing is performed as a quality assurance process for verifying different scenario requirements. In this phase, the deployed infrastructure is manually checked for any abnormalities. This is done by manually executing a dry run, which involves checking the network topology and exploiting the injected vulnerabilities. *Automated Verification* A manual dry run of the exercises usually takes a lot of time as well. To address this issue, we used the attacker agent to verify different scenario properties automatically. The attacker agent is a Kali Linux-based host machine present in the deployed network infrastructure. It receives the instruction of what actions to take from the scenario language. When a vulnerability is modeled to be injected into a host, a model for the attacker agent action is also generated to verify the vulnerability properties. This automatic verification includes different network link connections and vulnerability presence, along with exploitability.

4.2.3. Execution

Preparation and the resulting dry run take the most time in the cybersecurity exercise life cycle. When these parts are completed, the exercise can be executed; however, finding the right people for the exercise was a challenge because if you want to conduct a blue team exercise, you need a red team or vice versa. To address this issue, we added automation in the execution part as well, so our proposed exercises could be executed in the below ways. *Automatic* In automatic execution, attacker and defender actions can be specified for testing different cybersecurity scenarios in an automated manner. We used agent-based techniques for this purpose, in which we can inject attacker and defender agents within the exercise infrastructure. This agent follows the requirements specified in the DSL to execute attacker and defender actions. *Hybrid* In hybrid execution, an automated agent can emulate an attacker or defender against a human team in a cybersecurity exercise. In a hybrid execution, an adversary team's requirement is removed, making the cybersecurity exercise life cycle less reliant on human input and reducing the inefficiencies related to finding human teams. *Manual* In manual execution, a normal cybersecurity exercise is conducted in which all participants are human. Depending on the training requirements, the proposed system supports the manual execution of cybersecurity exercises. In manual execution, both red and blue teams consist of human participants who perform a cyber-attack and defense within the exercise infrastructure.

4.2.4. Evaluation

Most operational cybersecurity exercises are evaluated using flags. Flags are a textual string that the participants must capture from a system to receive a score. A different variation of this method is called dynamic scoring. Time is taken to capture the flag, and the number of times the flags are captured are also taken into account for awarding higher and lower scores. We could easily integrate such a scoring mechanism into our system. However, we opted for more systemic evaluation methods, which are given below. *Log Analysis* We have collected the command line history of exercise participants, which can be used to analyze the participants' capability and what type of skills they have. We plan to train an AI model for this purpose and use it to classify exercise participants' skill sets based on the *cyber kill chain*. Because we are still collecting data from the exercises and working on this part, this is not included in this paper. *Surveys* We used pre and post-exercise surveys to identify any skill improvements of the exercise participants and get qualitative data about the exercises. Researchers have previously used these methods (Moore et al., 2017) for such purposes.

4.2.5. Repetition

In this phase, the feedback from the surveys is analyzed, and problems are identified in the scenarios, including whose solutions were incorporated in the next iterations of the exercise.

4.3. Full system workflow

We have presented the whole system workflow in Fig. 8 which we discussed in Section 4.2. In the workflow, different parts of the cybersecurity exercise life cycle are presented in different colors. The system uses our scenario language to model the scenario in a coherent, logical model that is platform independent. The logical model is used to verify different scenario properties, and if they fulfill the scenario requirement, then the platform-independent model is transformed into platform-dependent artifacts. These artifacts create the network topology, inject vulnerabilities, generate traffic, and emulate various exercise teams. After this, the scenario is manually tested and automatically verified from the attacker

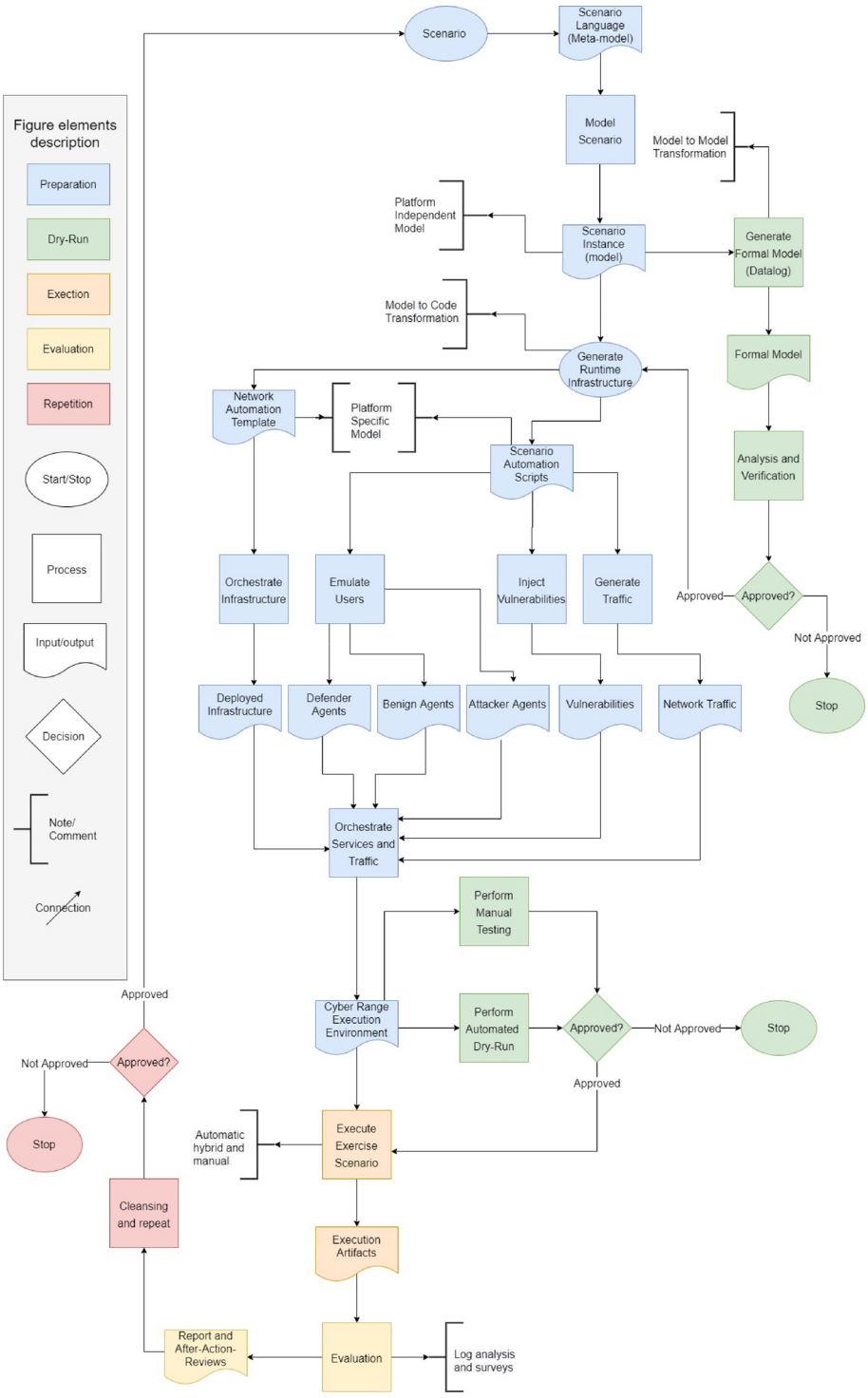


Fig. 8. Cyber security exercise operation workflow.

agent, and if it satisfies the scenario specification, then the exercise is executed. After exercise execution, relevant data are collected for exercise experience improvement in the next exercises.

The concept defined in the our scenario language instance is presented in concrete syntax for the orchestrator to understand and generate the necessary artifacts. The concrete syntax is JSON *Java script object notation* representations, which is specifically chosen because of its excellent capability to represent different models into objects with minimum or no changes in the code data structure. The details of the concrete syntax with respect to the specific concept is presented below:

Our scenario language input is transformed into emulated artifacts, and these artifacts can be used together or independently in five different processes for the execution of the cybersecurity exercise life cycle. The orchestrator is developed using the.Net framework, in which C# played a major role; secondary components of the orchestrator were developed using python scripts, whose calls were controlled by the C# base program. This modular design approach allowed us to include many existing system automa-

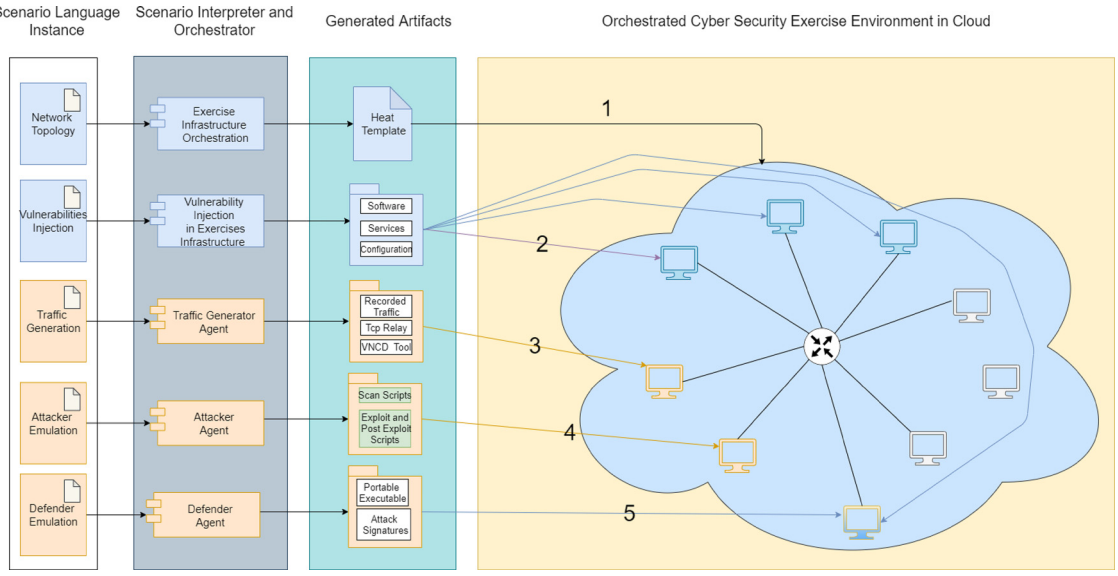


Fig. 9. Cyber security exercise operation orchestrator.

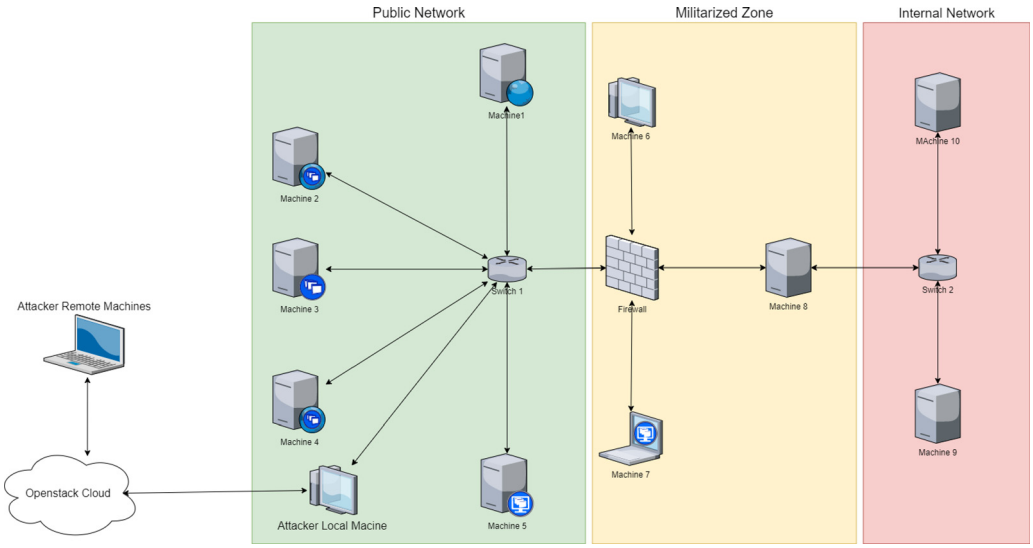


Fig. 10. Case study scenario.

tion techniques inside a single platform that provided much flexibility from scenario generation to scenario execution in a semiautonomous manner. The logical operation of the scenario orchestrator is presented in Fig. 9. The details of each component are given below.

Although some technical functions in the cybersecurity exercise life cycle are still manual, such as manual testing and survey, they are included to increase the exercise quality from a human perspective.

5. System implementation

The system is developed in a highly modular way, and different modules are used to automate different phases of the cybersecurity exercise lifecycle. Each module can run independently, or all the modules can run as a whole through a common API that uses our developed DSL for the exercise environment orchestration. The system is implemented in a web application that provides the interface to the developed API. It can take input from the developed DSL for orchestrating different cybersecurity operations roles. The website can be deployed in a local environment or on the cloud. The developed solutions require Openstack base cloud for performing infrastructure provisioning. While Vulnerability injection, attacker and defender emulation, and traffic generation can be done on a system supporting standard SSH protocol.

A variety of programming languages were used for the development of the application. The front end of the application was developed in Asp.net. Similarly, the API was developed in C#; however, the API runs multiple Python, Bash, and HEAT scripts in the back end. Package managers were used to installing vulnerabilities in Linux-based systems, while a silent install technique was used to install vulnerable software on Windows-based machines. In the case of the configuration and services, SSH-based automation techniques were used to make the system vulnerable. The source code of the developed orchestrator application can be found here NCR (2021). The repository contains pre-requisite information and detailed installation instruction for its easy deployment.

The system has three interfaces that are used to provide input and platform access. The first interface is for the administrator; the administrator can design and deploy customized cyber ranges hav-

ing access to detailed network configuration and vulnerability injection modules. The second interface is for a teacher or a coach who need to deploy a scenario in a very short amount of time, so the teacher or a coach can specify the type of vulnerabilities and the number of machines that are needed to be deployed for the scenario and the things deployed automatically. The final interface is for students who need to access the platform for practicing cybersecurity skills. They can access the platform and have access to multiple predefined scenarios that they can deploy by themselves and practice in the environment.

If we want to start the exercise from scratch, we can use all the modules that are presented in Fig. 9. We can perform (1) scenario modeling and orchestration for deploying exercise infrastructure, (2) vulnerabilities' injection and verification, (3) traffic generation, and (4) attacker and (5) defender agents for executing the exercise scenarios. If the infrastructure is already present, the implemented system can be used to inject vulnerabilities based upon the given requirement. Additionally, if the infrastructure is vulnerable, the developed solution can generate an attack model to verify those vulnerabilities. This modular system gives us the flexibility to adapt to various cybersecurity scenarios for dynamic cybersecurity exercises. The system module's usage for modeling an exercise scenario is presented in the form of a case study. It highlights the different modules developed during the research and their usage for conducting this research work.

6. Case studies

The competition organizers gave the scenario requirements, in which they requested two scenarios: a penetration testing scenario and an attack and defenses scenario. For the penetration testing scenario, the competition organizers gave the following scenario requirements:

1. The scenario should represent the IT infrastructure of a small- or medium-sized organization.
2. The scenario should have vulnerabilities that are exploitable in a particular amount of time
3. The scenario should be suitable for participants with various skill set levels.

We created a sample scenario description, which is provided in Appendix A and a rough network topology presented in Fig. 10 to translate the high-level requirements in to low-level technical artifacts based on our experience for conducting such exercises (Yamin and Katt, 2019; Yamin et al., 2018). We presented these ideas to the competition organizers and after their feedback and approval, we used it for the penetration testing scenario. We will used this network topology, as presented in Fig. 10, to showcase that a scenario can be modeled and orchestrated on a virtualized environment in the cloud. The scenario was logically verified and tested by an attacker agent, and the scenario also incorporated a defender agent in one of the scenario machines, which was included to add the friction and make the scenario more realistic. This case study was used to evaluate the *efficiency* and *autonomy* offered by our proposed solution.

In the second case study, the competition organizers provided the machines required for an attack/defense scenario with the diagram of the network topology. The requirement from the organizers was the following:

1. Deploy the machines in identical isolated networks for the first phase of the attack/defense scenario.
2. Update the network topology in the second phase of the attack/defense scenario so that the deployed networks can be interconnected.

```
<digit>          ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter>         ::= a | b | c | ... | y | z
<integer>        ::= <digit> { <digit> }
<string>         ::= " <char> { <char> } "
<char>           ::= <letter>  |  <digit>
<IP-Address>     ::= <Class-Address>"."<VLAN-Location>"."
                     <Device-Code>"."<Node>
<Class-Address>  ::= 001 to 255
<VLAN-Location>  ::= 001 to 255
<Device-Code>    ::= 001 to 255
<Node>           ::= 001 to 255
<CIDR>           ::= <Class-Address>"."<VLAN-Location>"."
                     <Device-Code>"."<Node>"/"<Range>
<Range>          ::= 0 to 24
```

Listing 1. Defining the basic variables.

```
<Subnet>       ::= <SubnetName> <CIDR> <NetworkID>
<SubnetName>   ::= <string>
<CIDR>         ::= <CIDR>
<NetworkID>    ::= <string>

<Machine>      ::= <MachineName> <OS> <Key> <Depends>
<MachineName>  ::= <SubnetName>
<OS>           ::= <string>
<Key>          ::= <string>
<Depends>      ::= <string>
```

Listing 2. Defining infrastructure.

```
<Vuln>                  ::= <MachineIP> <MachineUserID>
                            <MachineUserPassword> <OS>
                            <Vulnerability> <Parameter>
<MachineIP>             ::= <IP-Address>
<MachineUserID>         ::= <string>
<MachineUserPassword>   ::= <string>
<OS>                    ::= <string>
<Vulnerability>         ::= <string>
<Parameter>             ::= <string>
```

Listing 3. Defining vulnerabilities.

Because this scenario involved integrating external machines and updating network topology at run time, we used the *adaptability* to evaluate our proposed solution.

6.1. Preparation

6.1.1. Formal scenario modeling and analysis

We used our developed scenario modeler and verifier tool for the preparation of the scenario presented in Fig. 10. The scenario description is presented in Appendix A. The formal model generated by our tool is presented in Appendix B. The formal model of the scenario allowed us to verify different scenario properties. A Datalog analysis engine execution allowed us to logically verify different scenario properties, such as which machines present in the network were directly or indirectly accessible to the attackers. This allowed us to identify different edge cases like attacker accessibility without machine exploitation so that we could update the scenario before emulated network deployment. In Appendix B of the formal scenario model, we present the process of verifying a condition. An example of Datalog[1] query execution for identification of an attacker accessibility to different subnets is presented in Listing 7:

Similarly, in another example presented in Appendix B of the formal scenario model, we used the Datalog analysis engine exe-

[1] Definition in pydatalog ref:https://sites.google.com/site/pydatalog/home.

```
<Agent>                 ::= <ToolName> <AgentIP>
                            <AgentUserID>
                            <AgentUserPassword>
                            <Target> <Argument>
<ToolName>              ::= <string>
<AgentIP>               ::= <IP-Address>
<AgentUserID>           ::= <string>
<AgentUserPassword>     ::= <string>
<Argument>              ::= <string>
<Target>                ::= <IP-Address>
```

Listing 4. Defining attacker agent behavior .

```
    <Agent>::= <AgentIP> <AgentUserID>
    <AgentUserPassword> <OS>
    <Parameter>
    <AgentIP>               ::= <IP-Address>
    <AgentUserID>           ::= <string>
    <AgentUserPassword>     ::= <string>
    <OS>                    ::= <string>
    <Parameter>             ::= <string>
```

Listing 5. Defining defender agent behavior.

```
<Agent>                 ::= <AgentIP> <AgentUserID>
                            <AgentUserPassword> <Argument>
                            <Target>
<AgentIP>               ::= <IP-Address>
<AgentUserID>           ::= <string>
<AgentUserPassword>     ::= <string>
<Argument>              ::= <string>
<Target>                ::= <IP-Address>
```

Listing 6. Defining traffic generator behavior.

```
Query
--------------------
CanReach(Attacker1,Y)

Output
--------------------
***Public***
Machine1
Machine2
Machine3
Machine4
Machine5
***Demilitarized Zone***
Machine6
Machine7
Machine8
***Internal***
Machine9
Machine10
```

Listing 7. Analysis of an attacker that can reach other machines through direct and indirect links in different subnets like public, demilitarized zone, and internal.

cution for identifying which hosts were vulnerable to specific vulnerabilities and were reachable by attacker machines. This allowed us to determine the probable attacks' paths based on attacker capabilities and remove any edge cases where an attacker could not exploit the machines present in the logical representation of the exercise environment. A sample execution of the machines that were exploitable by a particular attacker capability is presented in Listing 8:

When different scenario properties are verified then the infrastructure is orchestrated.

6.1.2. Infrastructure orchestrator

The infrastructure orchestrator takes the JSON input from our scenario language and transforms it into HEAT templates. The

```
Query
--------------------
CanReach(Attacker1,Y) &Vulnerable(Y,BufferOverFlow)

Output
--------------------
Machine4
Machine7
```

Listing 8. Analysis of a host *Machine1* that can reach machines vulnerable to *BufferOverflow* vulnerability.

```
[
  {
    "Subnet 1": {
      "Name": "Public",
      "CIDR": "10.10.0.0/24",
      "NetworkID": "e18b412c-75c0-44a3-a326-708659d04152"
    },
    "Machine 0": {
      "Name": "Linux1",
      "OS": "721b1bc5-430e-44b9-89e3-45c92f3617fb",
      "key": "test",
      "Depends": "Public"
    }
  }
]
```

Listing 9. Concrete syntax for infrastructure generation.

HEAT templates were used to deploy the infrastructure using Openstack orchestration API. It should be noted that the required operating system images for the exercise are needed to be uploaded on Openstack before running the HEAT template. The orchestrator is currently only generating the exercise infrastructure in Openstack, but it is possible to transform our scenario language instance to other cloud orchestration technologies.

An example of the infrastructure is presented in Listing 9:

6.1.3. Vulnerability injector

The vulnerability injector can inject three types of vulnerabilities into the deployed infrastructure: software, services, and configuration. For software, the vulnerability injector reads the executable of the vulnerable program from the local drive and uses SSH to move it to a specified remote machine and then install it using different OS automation techniques. For services, the vulnerability injector reads a docker container file containing vulnerable services and moves it through SSH to the specified remote machine and deploys it automatically using different OS automation techniques. For configuration, a set of predefined bad configurations are integrated on the orchestrator, which can be specified in our scenario language and employed on remote machines using SSH. The configuration ranges from setting a user with a weak password to open directory shares for exploiting different vulnerabilities.

An example of our developed scenario language instance for vulnerability injection is presented in Listing 10:

In "Vuln 1," the vulnerability is "VulnerableProgram," and the parameter is "BufferOverflow.exe." This code will take the buffer overflow vulnerable program of SDL orchestrator machine and deploy it over a remote machine with the specified IP address using SSH and OS automation techniques.

In "Vuln 2," the vulnerability is "WeakPassword," and the parameter is "root2,toor." This code will create a new user account "root2" with "toor" as a password on a remote machine using an SSH connection and OS automation techniques.

In "Vuln 3," the vulnerability is "DockerInject," and the parameter is ""docker run -d -p 80:80 -p 3306:3306 -e MYSQL_Pass=mypass vulnerables/."" It will take "web-dvwa.tar" from the orchestrator machine and automatically deploy the docker on the remote machine using a SSH connection and OS automation techniques.

Table 1
Vulnerability type, property and parameter mapping.

Vulnerability Type	Vulnerability Property	Parameter Property	Description
Software	VulnerableProgram	IntgratedHomePro.exe	Name of executable
Software	VulnerableProgram	Icecast.exe	Name of executable
Software	VulnerableProgram	WingFTP.exe	Name of executable
Service	DockerInject	docker run -d -p 80:80 -p 3306:3306 -e MYSQL_Pass=\"mypass\" vulnerables/	Service command to be run or deploy and its parameters
Service	EnableTelnet	Install-WindowsFeature -name Telnet-Client	Service command to be run or deploy and its parameters
Service	EnableRDP	Invoke–Command –Computername “server1”, “Server2” –ScriptBlock {Set–ItemProperty –Path “HKLM:\System \CurrentControlSet \Control \Terminal Server” –Name “fDenyTSConnections” –Value 1}	Service command to be run or deploy and its parameters
Configuration	WeakPassword	root,toor	Usename and password
Configuration	DisableFirewall	NetSh Advfirewall set allprofiles state off	Disabling security service
Configuration	EnableLocalShare	net share Docs=E:\Documents /grant:everyone,FULL	Changing local drive access settings

```
[
  {
   "Vuln 1": {
      "MachineIP": "192.168.81.128",
      "MachineUserID": "root",
      "MachineUserPassword": "toor",
      "OS": "Windows",
      "Vulnerability": "VulnerableProgram",
      "Parameter": "BufferOverflow.exe"
    },
    "Vuln 2": {
      "MachineIP": "192.168.81.130",
      "MachineUserID": "root",
      "MachineUserPassword": "toor",
      "OS": "Linux",
      "Vulnerability": "WeakPassword",
      "Parameter": "root2,toor"
    },
    "Vuln 3": {
      "MachineIP": "192.168.81.128",
      "MachineUserID": "root",
      "MachineUserPassword": "toor",
      "OS": "web-dvwa.tar",
      "Vulnerability": "DockerInject",
      "Parameter": "docker run -d -p 80:80 -p 3306:3306 -e
      ↪  MYSQL_Pass=\"mypass\" vulnerables/"
    }
  }
]
```

Listing 10. Concrete syntax for vulnerability injection.

The orchestrator contains a mapping list between potential values of the *vulnerability* property and the vulnerability type such that the orchestrator understands the type of the vulnerability that needs to be injected by reading that property value. Consequently, the orchestrator expects a specific information in the parameter property. Table 1 shows a sample keys of these parameters and the vulnerability types they represent, including *software, services* and *configuration* vulnerability types. The complete list of the orchetrator include over 800 vulnerabilities related to both Windows and Linux environments. This list is constantly expanding with new vulnerabilities. For example, the *vulnerability* property `VulnerableProgram` is mapped to the *software vulnerability* type, and indicates a program that contains a software problem (c.f. Table 1). In this case, the *parameter* property indicates the name of the vulnerable executable.

If we want to allow multiple vulnerabilities of the same type, then we add two vulnerabilities with the same *vulnerability* property and different *parameter* properties. For example, if we want to allow two “bufferoverflow” software vulnerabilities in same machine, (1) we define two vulnerabilities, whose *vulnerability* property is `VulnerableProgram`, and (2) each one refer to different buffer overflow executable to be deployed, e.g., `bufferoverflow1.exe` and `bufferoverflow2.exe`. The rest of the values like *MachineIP*, *MachineUserID*., will remain same in case both vulnerabilities are being deployed on the same machine. It should be noted that this simple classification of vulnerabilities can be mapped to other types of classifications, like CWE and CVE, however this is out of scope of this work.

6.1.4. Attacker agent

The attacker agent is a Kali Linux machine deployed within the exercise infrastructure. The Kali Linux machine is controlled through SSH by the orchestrator and performs the steps specified in our scenario language. These steps include performing network scanning, launching actual exploits, and post-exploitation. Different automation techniques have been used to achieve this process. One such technique for automating Metasploit is presented in Fig. 11. Here, we used Metasploit resource scripts to specify and pre and post-attacker steps during exploitation. In Fig. 11 vulnserver.rb is the Metasploit exploit, which is then automated in step 1 and transformed into vul.rc, which is the Metasploit resource script. The resource script is sufficient for launching the attack, but we integrated post-exploitation steps in it to mimic a real attacker. In Step 2, gather.rc is integrated, which emulates post-exploitation steps such as capturing network information, as indicated in Fig. 11. It should be worth mentioning that there can be multiple attacker agents in the scenario, with each one performing different types of attacks independent of each other.

An example of our scenario language for attacker behavior emulation in the scenario is presented in Listing 11:

6.1.5. Defender agent

The defender agent is a portable executable that can be injected into the machine that is present in the scenario environment. A configuration file is also injected with the agent, and the CSV file contains a list of actions that an attacker can perform and a list of reactions against those actions that the defender agent can take. For a Windows-based environment, the defender agent analyzes Windows security event logs to identify the attacker actions specified in the configuration file and execute the appropriate reaction for stopping the attacker. The defender agent’s internal working is based on our exploit chain detection algorithm (Yamin et al., 2019). Here, we looked for a particular Windows security event ID 4688 and analyzed the command line argument for the identification of malicious behavior, as we presented in (Yamin and Katt, 2018a). It should be worth mentioning that there can be multiple defender agents running with different configurations within the scenario, reacting to different types of attacks independent of each other.

An example of our scenario language for defender behavior emulation in the scenario is presented in Listing 12:

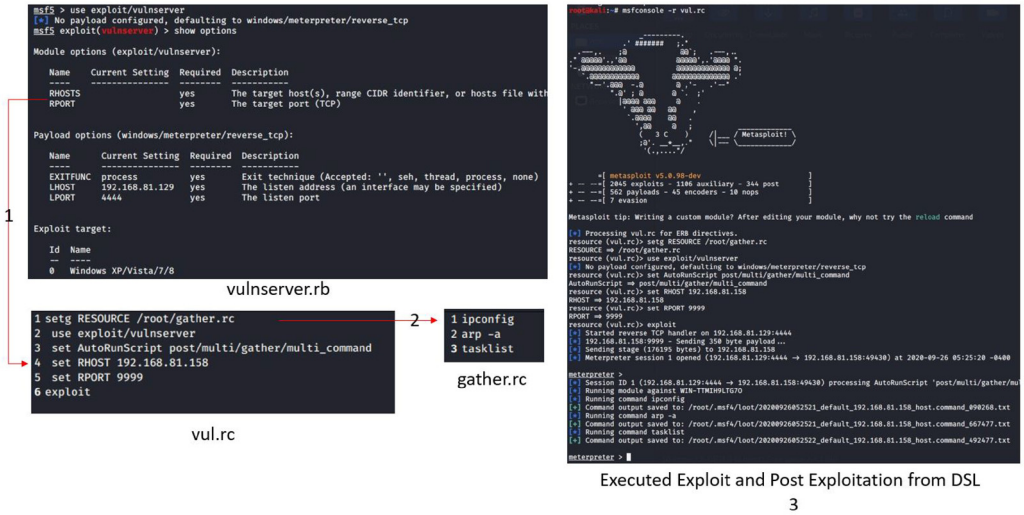


Fig. 11. Automated exploitation and post exploitation using metasploit.

```
[
  {
    "ActiveScan": {
      "AgentIP": "192.168.81.128",
      "AgentUserID": "root",
      "AgentUserPassword": "toor",
      "Argument": "SV",
      "Target": "192.168.81.130"
    },
    "BruteForce": {
      "AgentIP": "192.168.81.128",
      "AgentUserID": "root",
      "AgentUserPassword": "toor",
      "Argument": "SSH",
      "Target": "192.168.81.130"
    },
    "MetaSploit": {
      "AgentIP": "192.168.81.128",
      "AgentUserID": "root",
      "AgentUserPassword": "toor",
      "Argument": "Vulnserver",
      "Target": "192.168.81.130"
    }
  }
]
```

Listing 11. Concrete syntax for attacker behavior emulation.

```
[
  {
   "Defender 1": {
      "MachineIP": "192.168.81.132",
      "MachineUserID": "root",
      "MachineUserPassword": "toor",
      "OS": "Windows",
      "Parameter": "Actions1.csv"
    },
    "Defender 2": {
      "MachineIP": "192.168.81.134",
      "MachineUserID": "root",
      "MachineUserPassword": "toor",
      "OS": "Windows",
      "Parameter": "Actions2.csv"
    }
  }
]
```

Listing 12. Concrete syntax for defender behavior emulation.

6.1.6. Traffic generator

The traffic generator is a Kali Linux machine deployed within the exercise infrastructure. The Kali Linux machine is controlled through SSH by the orchestrator and performs the steps specified in our scenario language. These steps included replaying already captured network traffic and exercising specific email generation

```
[
  {
    "TcpRelay": {
      "AgentIP": "192.168.81.128",
      "AgentUserID": "root",
      "AgentUserPassword": "toor",
      "Argument": "Traffic.pcap",
      "Target": "Null"
    },
    "VncBot": {
      "AgentIP": "192.168.81.130",
      "AgentUserID": "root",
      "AgentUserPassword": "toor",
      "Argument": "Userbehavior.vdo|toor",
      "Target": "192.168.81.133",
    }
  }
]
```

Listing 13. Concrete syntax for traffic generation.

to emulate benign user behavior. For emulating benign users, Vncdotool (vnc, 2020) is used, enabling us to mimic user behavior in the GUI over VNC-enabled remote machines using a prerecorded VNC session. This added an extra layer of realism within the cybersecurity exercise environment. It should be noted that there can be multiple traffic generators in the scenario, generating different types of traffic independent from each other.

An example of our scenario language for the traffic generation and user behavior emulation in presented in Listing 13.

6.2. Dry run

The attacker agent developed during the research has two uses. Besides its role during the execution of the exercise, it can also perform a dry run on the developed infrastructure to test and verify that everything is working as expected. Some of the attacker agent's functionality tested during the dry run is presented in the form of the logs at Appendix C. Because our attacker agent is a Kali Linux machine, it can perform a network scan both actively and passively to check and verify that the machines in the scenario are up and running. Listing 14 shows the log of a passive scan that identifies the running machine in the scenario. Listing 15 shows the log generated by launching a brute-force attack on one of the deployed machines in the scenario. Similarly, Listing 16 shows the log of an automated exploit execution using Metasploit.

Additionally, the attacker agent can play the role of traffic generator using different Kali Linux functionalities. An example of an agent generating and replaying random network traffic is pre-

sented in Listing 13. As stated earlier, for emulating user behavior, VNCDOTOOL was used. A sample script for emulating a user to an open notepad and writing a few lines is presented in Listing 18. Moreover, a sample of the successful and unsuccessful email generation from our scenario language is presented in Listing 19. These functionalities can be used in different ways, depending on the scenario requirements.

6.3. Execution and evaluation

The platform was used for multiple qualifying rounds at NCSC 2020. The challenge had multiple rounds of qualifications, and in the first round, which was online, 150 people participated from all around Norway. Twenty-five individuals were invited for the second and third rounds, which were deployed by the proposed system. In the second round, 17 individuals participated in a penetration testing scenario. In the third and final round, the participants took part in an attack and defense scenario. The platform was evaluated for its efficiency in creating a realistic cybersecurity exercise infrastructure. Its capability to execute operations that are traditionally performed by human participants, along with its capability to adapt changes based on changing requirements. Moreover, it was also evaluated for improving the skill set of the cybersecurity exercise participants.

6.3.1. Platform deployment evaluation

The platform was evaluated by deploying the scenario presented in Fig. 10. The scenario had nine exploitable machines, one machine with no known vulnerability, and one machine running the defender agent. The machine running the defender agent had a similar vulnerability present in one of the exploitable machines. The machines were divided into three subnets: public, demilitarized zone, and internal network. Each team had five Kali machines present in the public network, which they could access over the internet using SSH. The scenario was deployed within five minutes using the orchestrator, but this does not accurately reflect the complexities involved in deploying such a scenario. For deploying the scenario, we needed to (1) collect the required operating system, vulnerable programs, services, and configuration details, which took a few hours to days, depending on the scenario requirement; (2) upload the required operating system on the cloud in the form of RAW images, which took a few minutes to hours depending on the internet speed; (3) specify the scenario according to the scenario language, which took a few minutes to hours depending on scenario complexity; (4) deploy the scenario on the cloud using the proposed system, which also took a few minutes to hours depending on scenario complexity; and (5) verify scenario properties using the emulated dry run, which took a few minutes to hours depending on scenario complexity. When all the perquisites for deploying and testing the scenario were fulfilled, our scenario language was able to provide the functionality and was very efficient in deploying the scenario. We then evaluated the proposed solution based on a set of five qualitative matrices: efficiency, usability, completeness, flexibility, scalability, and adaptability.

6.3.2. Efficiency

As stated earlier, multiple factors are involved in the efficiency of deploying cybersecurity scenarios. Consider the case of cooking a dish as an example of those factors. First, you must gather all the ingredients and then follow a recipe to make a dish. Here, we can measure the efficiency with respect to time in three ways: (1) cooking time, (2) the time required to gather the ingredients, and (3) the time required to grow the ingredients. The standard way of measuring how fast a dish is made is based on the cooking time, so we are ignoring the time required to gather the necessary components and artifacts for deploying the scenario and are only measuring the time our scenario language took to deploy the scenario, which was approximately five minutes. This figure can vary greatly because we deployed the scenario on a highly customized cloud infrastructure (Ope, 2021) that is specifically optimized for such infrastructure orchestration. Technically, the cloud infrastructure comprised 608 CPUs, 5.5 TB RAM for general purpose and 84 CPUs, 1792 GB RAM, 2 Tesla v100, 2 Tesla a100 for GPU-accelerated workloads with 133 TiB of total SSID storage. We suspect that the deployment efficiency result will be different in different cloud infrastructures, but we will investigate this in future research. *Applicability* Applicability measures whether the deployed scenario is employable for conducting operational cybersecurity exercises without manual tuning of the infrastructure. We tested the developed orchestrator in two different cybersecurity exercises that involved the top cybersecurity talent present in Norway, finding that the deployed scenario's performance was up to par with similar systems (Leitner et al., 2020; Russo et al., 2020; Vykopal et al., 2017a). The deployed infrastructure faced some failures during the exercises like some services failing to respond after continuing attacks; however, these issues were mitigated on the spot to ensure smooth running of the exercises.

Accuracy Accuracy measures whether the deployed scenario fulfilled the requirements specified in our scenario language accurately, such as the network topology and vulnerabilities present in the machines. Our developed scenario language fulfilled the technical requirement for deploying the infrastructure, injecting vulnerabilities, and executing a dry run as specified; the experimentally validated details of which are presented in Section 6.2 and Section 6.3.5.

6.3.3. Adaptability

Adaptability refers to the capability of developed solutions to accept and accommodate changes from different sources and implement the exercise scenario. We used another qualifying round for NCSC to check this capability. This qualifying round was an attack/defense scenario in which the vulnerable machine was developed by one of our colleagues using another source code vulnerability injector solution. Our colleague shared the four vulnerable machines with us, and using our scenario language, we deployed the scenario in an attack/defense network topology. Five teams participated in the attack/defense scenario, and they were assigned five identical isolated networks with four vulnerable machines each. The attack/defense scenario had two parts: in the first part, the teams had to patch the vulnerabilities in their assigned network. In the second part, the teams' networks were interconnected and were tasked to exploit the vulnerabilities in the other teams' machines. This changing network topology, while accommodating unknown machines, was used to verify the adaptability of our developed system. The deployed attack/defense scenario on Openstack is presented in Fig. 12.

Flexibility Flexibility measures the capability of the deployed scenario to accept changes after deployment. We developed the scenario orchestrator in a modular way, in which the infrastructure was independent of the vulnerability injection steps. This enabled us to inject new vulnerabilities after the scenario had been deployed. We consider this a highly useful feature because it enabled us to use the same network topology with different types of vulnerabilities for individuals with different skill sets. Moreover we could change the network topology, add additional machines with new vulnerabilities, and launch new attacks to make the scenario more dynamic based on the given requirements. *Scalability* Scalability measures whether the deployed scenario is expandable and can accommodate additional teams in the scenario. There were a total of 15 machines that were allocated to one team, but there were a total of five teams involved in NCSC, so the scenario was

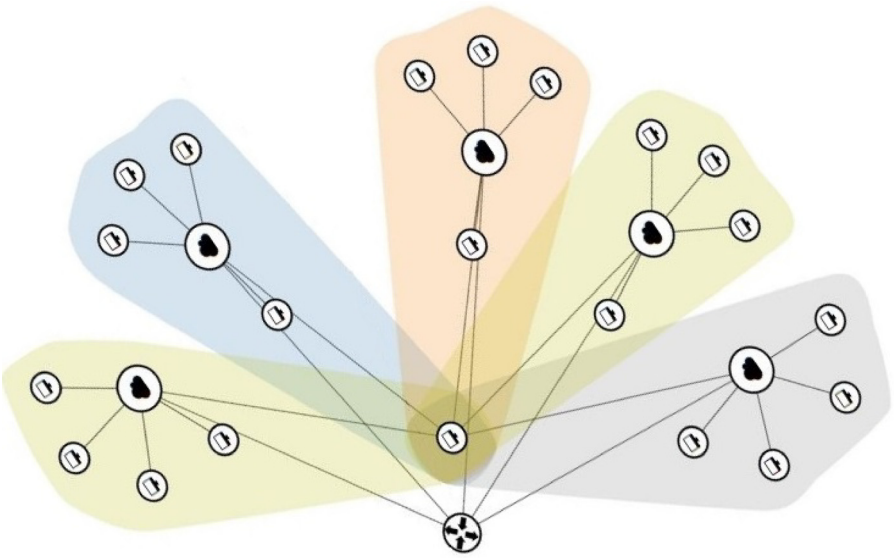


Fig. 12. Deployed Attack/defense scenario.

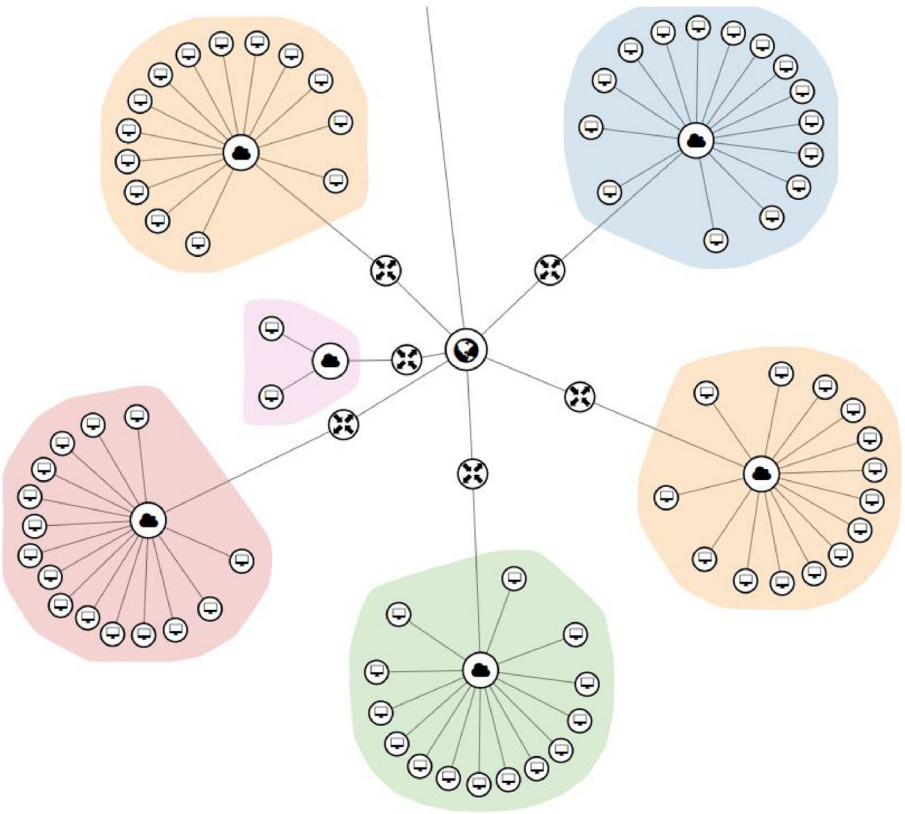


Fig. 13. Deployed penetration testing scenario.

replicated within five minutes to instantiate 75 virtual machines, which enabled us to assign a completely segregated network to each team. The fully deployed five networks comprising 75 machines that implemented the case study scenario are presented in Fig. 10 and can be seen in Fig. 13:

Because we conducted this research to perform cybersecurity exercises that are scalable for a large number of participants, we used the developed tool to create and deploy the exercise infrastructure for one of the cybersecurity courses taught at NTNU. The course had 84 students, and the infrastructure was required for a four-week-long red and blue team exercise. The infrastructure included nearly 400 machines with 19 teams and one research network and was deployed in approximately 37 minutes. The deployment time was noted form the Openstack *SystemUsageData* which includes the timestamps for when the *Stack CREATE started* and when the *Stack CREATE is completed* Sys (2021). It should be noted that each network was deployed as a separate HEAT template, and we added a one-minute break after each network deployment to manage Openstack API calls. The deployed network topology is presented in Fig. 14.

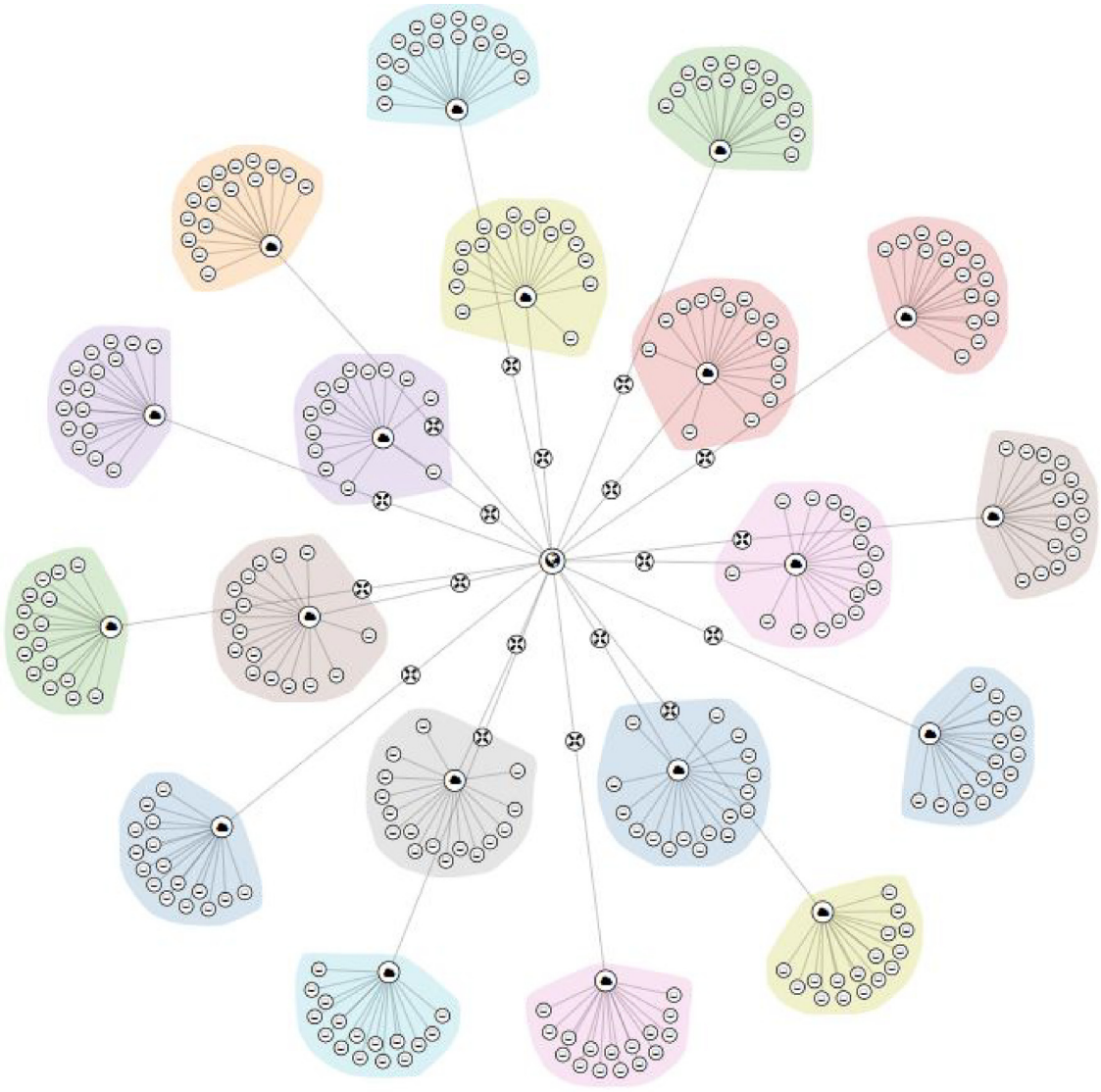


Fig. 14. Deployed red and blue team cybersecurity exercise scenario. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

6.3.4. Autonomy

Autonomy refers to the capability of a developed solution to perform tasks that have been traditionally performed by a human. We used the developed solution to perform an emulated dry run automatically on the scenario infrastructure, as discussed in Section 6.2. This provided us with the capability to replace humans in the dry run phase of cybersecurity exercises. Second, two machines in the scenario had the same vulnerability, but one machine was running the proposed defender agent. Five teams played the scenario, and two teams were able to exploit the vulnerability and get the flag on the machine that was not running the defender agent. The teams did not have any knowledge about a defender agent running on the system and were incentivized to exploit the machine to get additional points, as stated in Appendix A. With this experiment, we concluded that the defender agent was working as expected and could be a useful addition to cybersecurity exercises to add friction for increased realism.

6.3.5. Preliminary skill improvement evaluation

The skill improvement was measured in the final round of NCSC, in which 17 individuals, who qualified from the nationwide CTF of Norway, participated. The participants aged between 16–25, more than half of the participants were high school and university students, while others were employed in different public and private sectors. They participated in the form of randomly assigned groups in the CTF consisting of 3–4 individuals each. We employed a multitude of quantitative and qualitative methods for identifying skill improvements in exercise participants that were highlighted in Maennel (2020). First, we conducted pre and post-exercise surveys to identify any skill improvement in the individuals. This methodology was employed in a case study where individuals measured specific skill set levels over the passage of time (Moore et al., 2017). In summary a total of 7 technical skills were measured that were required to solve the challenges present in the exercise. The pre and post self-classification of skill level were presented in Fig. 15 and details of the results are provided in Section 6.3.5.1 and 6.3.5.2.

Secondly, we used CTF instrumentation suggested in Chothia and Novakovic (2015); however, each group was assigned their own separate exercise infrastructure and unique flags were injected manually to avoid *Flag Plagiarism*. Additionally, we asked the exercise participants to provide a *Penetration Test Report* to evaluate their performance in a qualitative manner. The

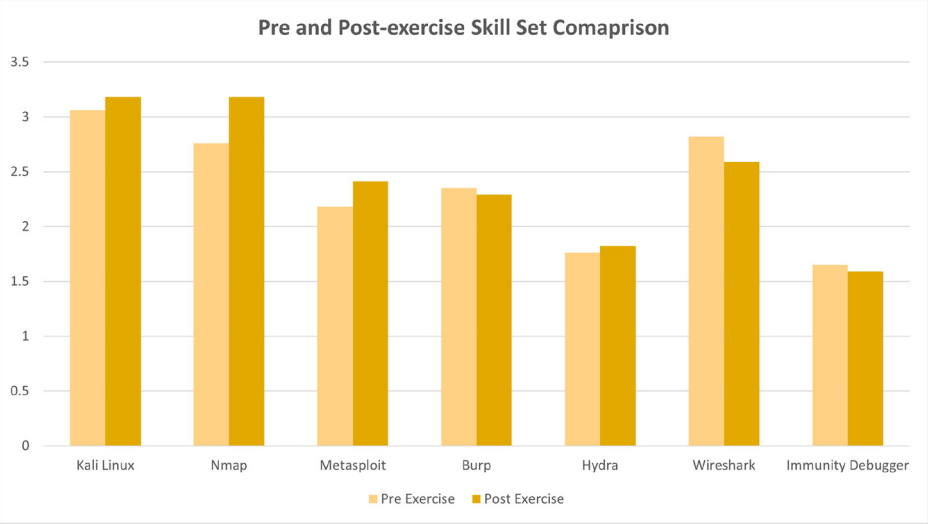


Fig. 15. Self-classification of skill level Pre and Post Exercise.

You classify yourself as a low/medium/high skilled individual in cyber security?

	Response Total	Response Percent
low	5	29%
medium	6	35%
high	6	35%
Total Respondents	17	100%

Fig. 16. Self-classification of skill level.

How many CTF you previously played

	Response Total	Response Percent
1-5	2	12%
5-10	8	47%
>10	7	41%
Total Respondents	17	100%

Fig. 17. Experience in CTF competitions.

survey questions are presented in Appendix D, and the results of the survey and CTF instrumentation are given below:

Pre-exercise Survey For the first question, we asked the participants about their self-classification of skill sets. The majority of the participants had medium and high cybersecurity skills, 35% each, while 29% reported that they had low cybersecurity skills. Details of the participants' answers about their self-classification of skill level are presented in Fig. 16:

The second question we asked was about the participants' experiences in CTF competitions. The question was intended to get information that could be used to measure the CTF quality in post-exercises surveys. Most participants (47%) replied that they had participated between in five and 10 CTFs before, 41% participated in more than 10 CTFs, while only 12% participated between one and five CTFs. Experience in CTF competitions is presented in Fig. 17:

The third question we asked was rating their skill-specific cybersecurity tools from 1 to 5, where 1 is the lowest and 5 is the highest value. These tools included Kali Linux, Nmap, Meatsploit, Burp, hydra, Wireshark, and Immunity Debugger. Most of the participants were familiar with Kali Linux, with a mean score of 3.06 out of 5, while most were not familiar with Immunity Debugger, with a mean score of 1.65 out of 5. Details of their pre–exercise skill set response related to specific tools are presented in Fig. 18.

Post-Exercise Survey After the pentest scenario, we conducted a post-exercise survey to measure the skill improvement of the scenario participants. The first question we asked was related to skill improvement with respect to specific tools. The participants reported that they saw skill improvements in Kali Linux, Nmap, Metasploit, Burp, and Hydra but did not see any improvements in their Wireshark and Immunity Debugger skills. Details of their post–skill set responses related to specific tools are presented in Fig. 19.

The second question we asked was related to their overall skill improvement, in which 24% reported that they had a definite skill improvement, while 41% reported that they saw some skill improvement after playing the scenario. Here, 35% reported that they did not have any skill improvement, which was expected because 35% represented the six seniors who were already highly skilled. Details of their post–exercise overall skill improvement are presented in Fig. 20.

Rate your skills in different cyber security tools from 1 to 5, where 1 is the lowest value and 5 is the highest value:

	1	2	3	4	5	Response Total	Response Average
Kali linux	5,88% (1)	23,53% (4)	29,41% (5)	41,18% (7)	0% (0)	17	3,06
Nmap	23,53% (4)	17,65% (3)	23,53% (4)	29,41% (5)	5,88% (1)	17	2,76
Metasploit	41,18% (7)	23,53% (4)	11,76% (2)	23,53% (4)	0% (0)	17	2,18
Burp	41,18% (7)	11,76% (2)	23,53% (4)	17,65% (3)	5,88% (1)	17	2,35
hydra	58,82% (10)	17,65% (3)	17,65% (3)	0% (0)	5,88% (1)	17	1,76
Wireshark	11,76% (2)	41,18% (7)	17,65% (3)	11,76% (2)	17,65% (3)	17	2,82
Immunity Debugger	76,47% (13)	5,88% (1)	5,88% (1)	0% (0)	11,76% (2)	17	1,65
					Total Respondents		17

Fig. 18. Pre-exercise skill set specific to tools.

Rate your skills in different cyber security tools from 1 to 5, where 1 is the lowest value and 5 is the highest value:

	1	2	3	4	5	Response Total	Response Average
Kali linux	0% (0)	23,53% (4)	41,18% (7)	29,41% (5)	5,88% (1)	17	3,18
Nmap	0% (0)	35,29% (6)	29,41% (5)	17,65% (3)	17,65% (3)	17	3,18
Metasploit	23,53% (4)	41,18% (7)	17,65% (3)	5,88% (1)	11,76% (2)	17	2,41
Burp	47,06% (8)	17,65% (3)	0% (0)	29,41% (5)	5,88% (1)	17	2,29
hydra	52,94% (9)	29,41% (5)	5,88% (1)	5,88% (1)	5,88% (1)	17	1,82
Wireshark	29,41% (5)	23,53% (4)	23,53% (4)	5,88% (1)	17,65% (3)	17	2,59
Immunity Debugger	76,47% (13)	5,88% (1)	5,88% (1)	5,88% (1)	5,88% (1)	17	1,59
					Total Respondents		17

Fig. 19. Post-exercise skill set.

Did you havve any skill improvement after playing the CTF?

	Response Total	Response Percent
Yes	4	24%
No	6	35%
Somewhat	7	41%
Total Respondents	17	100%

Fig. 20. Post-exercise skill improvement.

How realistic was the CTF compare to other CTF you played before? Give it rating from 1 to 5, where 1 indicates the lowest value and 5 indicates the highest value.

	1	2	3	4	5	Response Total	Response Average
Realistic level:	11,76% (2)	41,18% (7)	41,18% (7)	0% (0)	5,88% (1)	17	2,47
					Total Respondents		17

Fig. 21. Scenario realism.

The third question was related to the realistic level of the scenario; here, we asked the participants to rate the scenario's realism from 1 to 5. The scenario received a total rating of 2.47 out of 5. We consider this sufficient because most of the scenario was generated by our scenario language. Details of scenario's realism rating are presented in Fig. 21:

The last question we asked was related to scenario difficulty level, in which 59% of the participants considered the scenario to be difficult, 24% considered the difficulty level as medium, while 18% considered the difficulty level as easy. Details of the scenario difficulty level are presented in Fig. 22:

CTF Instrumentation We used CTFd CTF (2021) for scoring and instrumentation purposes. CTFd is widely used for scoring purposes for such exercises and is very user friendly and easy to deploy. The CTFd score board indicated how each group is performing in solving different challenges with respect to time as indicated in Fig. 23. This helped us to compare the skill set required to solve a challenge with the skill set stated by the participants in pre and post-exercises surveys.

For instance, if we analyze the task complemented by the one of the leading group we can see that it completed a task related to debugging of an application and brute force attack on another application as indicated in Fig. 24 where *NewBie* is for brute-force and *Debugger1* is for debugging. Other groups were struggling with those tasks, which was reflected in post exercise survey where participant self assessed their skills lower then compare to pre-exercise survey. Similarly majority of exercise participants were able to complete tasks that involved *Metasploit*, *Nmap* and *Kali Linux* which was positively reflected in post exercise survey. This helped us to measure the actual skill set compare to perceived skill set of the exercise participants. Finally a thematic analysis on the submitted penetration test reports were performed. Individual feedback for each participating group was given for focusing on particular skill set in future.

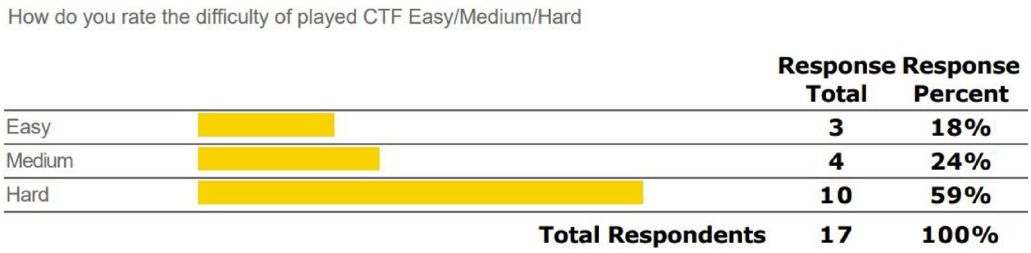


Fig. 22. Scenario difficulty level.

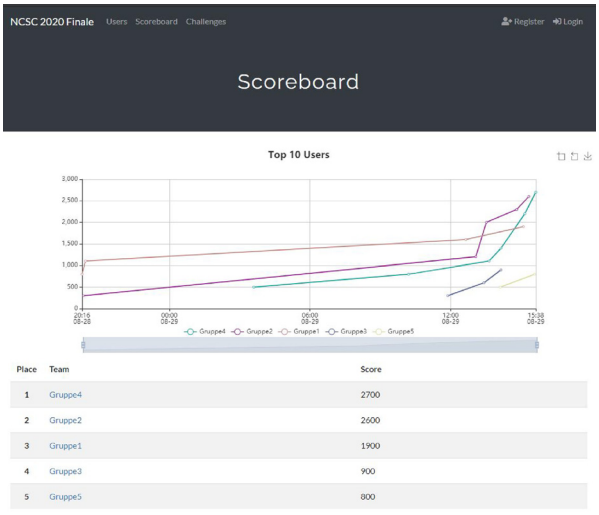


Fig. 23. Final Score Board of CTF.

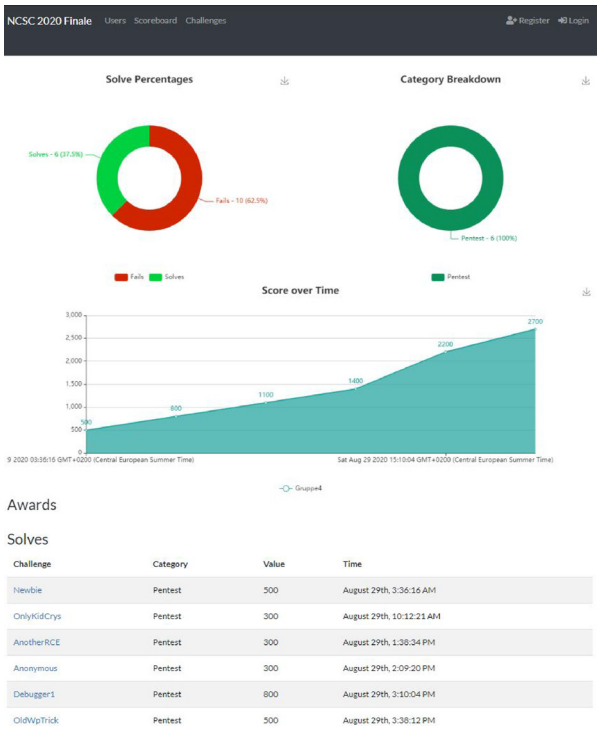


Fig. 24. Challenge Completion Analysis.

7. Discussion

In the course of this research, we investigated inefficiencies in the cybersecurity exercise lifecycle. Conducting cybersecurity exercises is a complex and challenging task involving a whole cycle which includes phases of *Preparation*, *Dry run*, *Execution*, *Evaluation* and *Repetition*. Each phase brings a new set of challenges and inefficiencies for cybersecurity exercise organizers. The challenges range from preparing operational exercise infrastructure to finding skilled individuals to perform dry run on it. Moreover, identifying skilled individuals to play the role of Red or Blue team members is also challenging during the exercise execution phase which adds additional inefficiencies. We proposed a system to address those inefficiencies and make the execution more efficient. The results and limitations of the prospered research work are presented below:

7.1. Results

We investigated 4 *Research Questions* in this work, *Question 1* was to identify systems and methods with which we can model and execute realistic cybersecurity exercise scenarios more efficiently. We proposed our developed orchestrator for addressing this question, we used model-driven engineering to develop a DSL that can use to specify cybersecurity exercise models involving realistic adversaries and traffic generation. Using our orchestrator, the DSL can computationally execute various functions of different teams present in the cybersecurity exercise environment to conduct exercises with minimal human effort, which makes the exercise execution efficient.

In *Research Question 2* we investigated the following query: *Is it possible to make cybersecurity exercise models adaptable to changing requirements?*. We addressed this question by conducting three cyber security exercises involving around 100 people. The exercises were conducted over the period of 6 months with changing requirements. The first exercise *Penetration testing* exercise was conducted to test the developed platform. The second *Attack and Defense* exercise was conducted to check the adaptability and flexibility of the developed platform, in which the developed platform incorporated machines from other sources. The third exercise which was a large *Red VS Blue* team exercise was conducted to test and validate the scale-ability and performance of developed autonomous agents.

In *Research Question 3* we investigated *What operations in cybersecurity exercises can be executed autonomously to reduce dependability on human teams?*. Our work indicated that in terms of automation, White, Green, and partially Red teams that are involved in the *Preparation* and *Dry run* phases are automated completely. For exercise execution, forensic traces for a “Red vs Blue” team exercise can be created, which removes the need for a human Red team in such scenarios. Similarly, autonomous defense agents can be injected into the machines present in the network to prevent the human attacker to achieve their objectives in a penetration testing scenario. A separate case study highlighting the internal working of attack and defense agents will be published soon. We are currently working on an AI model to analyze cybersecurity

skills based on the participants' bash history. We are in the advanced stages of its development but lack sufficient data for proper training of the AI model. To address this, we are conducting cybersecurity exercises, collecting relevant data from them, and planning to publish a separate study.

In *Research Question 4* we investigated *How much do such exercise scenarios improve the skills of cybersecurity exercise participants?*. Our work indicated that the developed cybersecurity exercise execution platform has a positive impact on exercise participants. The participants reported skill improvement on 5 out 7 tested technical skills which was co-related with the type of challenges completed in the exercises. The results are positive, yet not definitive, and further studies are required for better assessment of the skill improvement.

7.2. Limitations

Although the proposed system addresses the problems in conducting cybersecurity exercises efficiently and realistically, nevertheless it has few limitations. One of the first limitations of the proposed system is in vulnerability injection. The proposed system uses its own classification of vulnerabilities based upon operational requirements of the software, services, and configuration, which isn't commonly used in standards like CVE and CWE. We will address this in our future work by integrating other classification standards into the vulnerability classification of the vulnerability injection function. Secondly, for the evaluation, only 17 people participated and provided their feedback on the system for qualitative evaluation. However, we would like to emphasize that the 17 survey participants were selected from the pool of 150 people who competed in a nationwide CTF in Norway. The 17 participants were actively participating in the study and the exercises. On the other hand, one of ENISA report from 2021 Tow (2022) indicated that while the number of participants in cyber security competitions can be high, the active participants are a small portion of the total number (ca. 11%). We are targeting very specific skill sets and incorporating a large group is very challenging, especially during the pandemic. Similarly, from the same report, it was also indicated that the average training group for different European countries is around 20 people. Based upon this information we believe that the data presented in the qualitative evaluation surveys might not be conclusive, but it has meaningful insights. We would further like to highlight similar studies Abbott et al. (2015); Moore et al. (2017); Peker et al. (2016); Yamin et al. (2021) that involve 26, 30, 28, and 25 people, respectively.

8. Conclusion and future work

In the current work, we presented a cybersecurity exercise modeling and execution tool for cyber ranges. At the core of the tool, we have proposed a scenario language that unifies concepts of different operations present in cybersecurity exercises. This enabled us to computationally orchestrate their functionality. We used our tool to conduct multiple cybersecurity exercises in an efficient, adaptable, and autonomous manner, which resulted in decreasing the inefficiencies in the cybersecurity exercise life cycle. This will enable the utilization of operational cybersecurity exercises on a wide scale for education and training purposes, helping overcome the ever-growing cybersecurity skill shortage.

This research work resulted in the development of a DSL that can be used to model cybersecurity exercises scenarios. The modeled scenarios can be executed in an efficient and realistic manner using the orchestrator developed during this research. The proposed models are adaptable based upon changing cybersecurity exercise scenario requirements. This is achieved through the automation of various roles in cybersecurity exercises. The white team role is automated to specify the scenario requirements and automatic deployment of exercise infrastructure. The red team role is automated to perform dry runs and act as an automated adversary during exercise execution. While blue team role was automated to provide active defense during cybersecurity exercise execution. We evaluated the developed system by conducting multiple cybersecurity exercises and measured exercises participant skill improvement.

In the future, we are planning to expand the current version of our scenario language and add a new concept called "module." The "module" concept will specify a template of an organizational infrastructure, for example, an *bank* or an *internet service provider* ISP. When the "module" specification is given to our scenario language, the infrastructure will be provisioned for automatic deployment. One key feature that we are planning to integrate into such infrastructure provisioning is multi-cloud orchestration because we are going to emulate the whole infrastructure of an organization, which will be quite resource-demanding for single cloud deployment. This will also help us realistically model different segregated organizations on the internet for cybersecurity scenarios. Finally, we are planning to use TLA+ to model the attacker and defender steps during a cybersecurity exercise. This will enable us to perform a more formal analysis on different attack and defense strategies. We will conduct more in-depth case studies for autonomous cybersecurity exercise execution and then analyze such technology's effectiveness in training human attackers and defenders.

Declaration of Competing Interest

The authors declare no conflict of interest in publishing the article "Modeling and Executing Cyber Security Exercise Scenarios in Cyber Ranges".

Appendix A. Scenario Description

NCSC 2020 Penetration Test Scenario

A1. Good corp LLC

Good Corp LLC is a SME (Small and Medium Enterprise) working in the Management Consultancy area. It provides assistance to various government and non-government entities in making decisions that are economically feasible, environmentally sustainable and socially acceptable. Good Corp LLC is currently conducting a study that is focusing on economic benefits of increased Baltic sea area by forecasting success of recent DS (Delete Sweden) movement.

While the DS movement got allot of support from Nordic regions and is economically feasible and environmentally sustainable. However, it is not socially acceptable by some people, so they launched an anti-DS movement Ref: https://www.change.org/p/sweden-anti-delete-sweden.

But the anti-DS movement didn't get any significant public support, so a group of 5 social activist hackers decided to take things in their own hands. They planned to launch a cyber-attack on Good Corp LLC and tamper with the data on its servers in order to change its report that it is going to present to the government.

A2. Good corp LLC employees

Good Corp LLC is an equal opportunity organization and provide people from diverse backgrounds and environments to grow and achieve success. New employees come and go. Some learn new things while other earn. Due to current COVID situation Good Corp LLC is struggling to provide orientation sessions to new employees

which includes overview of company values and basic cybersecurity awareness. Mike is a young and passionate researcher who recently joined Good Corp LLC but missed the orientation sessions. Good Corp LLC is facing a lot of cybersecurity challenges as it is involved in very high-profile research, so it invited a cybersecurity firm to pentest their employees and infrastructure. The firm identified a lot of security issues in Good Corp LLC infrastructure. Some of them were related to very famous ransomware vulnerabilities while others were related system configuration and bad password policies. One shocking finding of the pentest was a data breach that was identified but thankfully it didn't contain some important information. Just some old file and custom software programs that Good Corp LLC uses in its day to day operations. The good thing that was identified in the pertest was that the CEO Jason was found to be very well secured while one of the mangers has no zero day vulnerabilities. The CEO decided to continue the operations of the company while patches were implemented and issues with data breach were resolved. Link to data breach:

A3. Rules

1. You are tasked to discover, exploit, analyze and report vulnerabilities in Good Corp LLC infrastructure.
2. Every system has a file flag.txt. Locate it and post the content of it in the scoreboard.
3. Take a screenshot of the flag content with IP address of exploited machine and use it as evidence in the report.
4. Bonus point for putting a file in CEO computer with content "DS is not recommended"
5. Not allowed to use any sort of DoS Attacks
6. Not allowed to share any information about the scenarios with anyone other than your teammates

Appendix B. Formal Scenario Model and Verification for the Use Case

Lets us consider this scenario in which 10 machines are connected to 3 sub-nets. In the scenario two subnets are directly connected *PUblic* and *MilitrizedZone* while a machine present in the network have dual network interface which provide access to subnet *Internal*. We can define the scenario model the following way.

1. **Defining Links**

Link Facts

$\%Specifying\ facts\ which\ Hosts\ are\ connected\ to\ which\ Network$
$Link('Machine1', 'Public')$
$Link('Machine2', 'Public')$
$Link('Machine3', 'Public')$
$Link('Machine4', 'Public')$
$Link('Machine5', 'Public')$
$Link('Public', 'DemilitarizedZone')$
$\%Specifying\ two\ Networks\ are\ directly\ connected$
$Link('Machine6', 'DemilitarizedZone')$
$Link('Machine7', 'DemilitarizedZone')$
$Link('Machine8', 'DemilitarizedZone')$
$Link('Machine8', 'Internal')$
$\%Specifying\ a\ Host\ is\ connected\ with\ multiple\ network\ interface$
$Link('Machine9', 'Internal')$
$Link('Machine10', 'Internal')$

To check which machines are connected to which network and can reach which machines we can define a new *Term CanReach* with variable *X*, *Y* and *Z*.

Link Clauses

$\%A\ clause\ which\ creates\ a\ bidirectional\ link\ between\ two\ Hosts\ X\ and\ Y$
$Link(X, Y) \leq Link(Y, X)$
$\%A\ clause\ to\ check\ direct\ link\ between\ two\ Hosts\ X\ and\ Y$
$CanReach(X, Y) \leq Link(X, Y)$
$\%A\ clause\ to\ check\ link\ between\ two\ Hosts\ X\ and\ Z\ via\ Host\ Y$
$CanReach(X, Y) \leq Link(X, Z)$

2. **Defining Vulnerabilities**

Vulnerability Facts

$\%Specifying\ facts\ which\ Hosts\ are\ vulnerable\ to\ which\ vulnerability$
$Vulnerable('Machine1', 'SSHBruteForce')$
$Vulnerable('Machine2', 'EasyFTPExploit't)$
$Vulnerable('Machine3', 'MS17-010')$
$Vulnerable('Machine4', 'BufferOverFlow')$
$\%A\ Host\ can\ be\ vulnerable\ to\ multiple\ vulnerabilities$
$Vulnerable('Machine4', 'MS17-010')$
$Vulnerable('Machine5', 'XXE')$
$Vulnerable('Machine6', 'AppacheExploit')$
$Vulnerable('Machine7', 'MS14-068')$
$Vulnerable('Machine7', 'BufferOverFlow')$
$Vulnerable('Machine8', 'RDPBrutforce')$
$\%A\ Host\ can\ have\ no\ known\ vulnerability\ as\ well$
$Vulnerable('Machine9', 'NoVulnerability')$
$Vulnerable('Machine10', 'EasyFTPExploit')$

To check which machine is connected to machine with a specific vulnerability e.g. *BufferOveflow* we can verify it by the following clause:

Vulnerability Clause

$CanReach('Attacker1', Y)\ \&\ Vulnerable(Y, 'BufferOverFlow')$

3. **Defining Capabilities**

Capabilities Facts

$\%Specifying\ which\ vulnerabilities\ are\ exploitable\ and\ which\ are\ defendable$
$Capability('SSHBruteForce', 'YES', 'NO')$
$Capability('WebExploit', 'YES', 'NO')$
$Capability('MS17-010', 'YES', 'YES')$
$Capability('BufferOverFlow', 'YES', 'NO')$
$Capability('MS14-068', 'YES', 'YES')$
$Capability('XXE', 'YES', 'NO')$
$Capability('AppacheExploit', 'YES', 'NO')$
$Capability('RDPBrutforce', 'YES', 'NO')$
$Capability('EasyFTPExploit', 'YES', 'YES')$
$Capability('NoVulnerability', 'NO', 'YES')$

To check which machine is connected to vulnerable machines which are not defendable by defender we can create the following clause:

Capabilities Clause

$Capability(V, 'YES', 'NO')\ \&\ CanReach('Attacker1', Y)\ \&\ Vulnerable(Y, V)$

4. Defining KillChain

KillChain Facts

%Specifying facts that which host is exploitable to different stages of CKC.
$KillChain('Machine1', 'YES', 'YES', 'YES', 'YES', 'YES', 'YES', 'YES')$
$KillChain('Machine2', 'YES', 'YES', 'YES', 'YES', 'YES', 'YES', 'YES')$
$KillChain('Machine3', 'YES', 'YES', 'YES', 'YES', 'YES', 'YES', 'YES')$
$KillChain('Machine4', 'YES', 'YES', 'YES', 'YES', 'YES', 'YES', 'YES')$
$KillChain('Machine5', 'YES', 'YES', 'YES', 'YES', 'YES', 'YES', 'YES')$
$KillChain('Machine6', 'YES', 'YES', 'YES', 'YES', 'YES', 'YES', 'YES')$
$KillChain('Machine7', 'YES', 'YES', 'YES', 'YES', 'YES', 'YES', 'YES')$
$KillChain('Machine8', 'YES', 'YES', 'YES', 'YES', 'YES', 'YES', 'YES')$
$KillChain('Machine9', 'YES', 'YES', 'YES', 'YES', 'YES', 'NO', 'NO')$
$KillChain('Machine10', 'YES', 'NO', 'NO', 'NO', 'NO', 'NO', 'NO')$

To integrate Cyber Kill Chain concepts to the model we can create the following clause:

KillChainClause

%A clause to check which specific Host is connected to Hosts
%in a network that are vulnerable and are not defendable
%and are exploititable to different stages of Cyber Kill Chain .
$Capability(V, 'YES', 'NO') \& CanReach('Mahine1', Y) \& Vulnerable(Y, V) \&$
$KillChain(Y, 'YES', 'YES', 'YES', 'YES', 'YES', 'YES', 'YES')$

Appendix C. Dry Run Logs

```
Date and Time:12:20 PM
 17 Captured ARP Req/Rep packets, from 4 hosts.   Total size: 1020
 ------------------------------------------------------------------
   IP            At MAC Address     Count     Len  MAC Vendor
 ------------------------------------------------------------------
 192.168.81.2    00:50:56:fc:37:72      3     180  VMware, Inc.
 192.168.81.130 00:50:56:29:2a:2e      1      60  VMware, Inc.
 192.168.81.1    00:50:56:c0:00:08     12     720  VMware, Inc.
 192.168.81.254 00:50:56:e0:48:87      1      60  VMware, Inc.
```

Listing 14. Sample log for passively checking network connection.

```
Date and Time:12:26 PM
Hydra v8.6 (c) 2017 by van Hauser/THC - Please do not use in
↪  military or secret service organizations, or for illegal
↪  purposes.

Hydra (http://www.thc.org/thc-hydra) starting at 2019-02-28
↪  15:25:16
[DATA] max 64 tasks per 1 server, overall 64 tasks, 5084 login
↪  tries (l:1/p:5084), ~80 tries per task
[DATA] attacking ssh://192.168.243.130:22/
[STATUS] 1022.00 tries/min, 1022 tries in 00:01h, 4191 to do in
↪  00:05h, 64 active
[22][ssh] host: 192.168.243.130   login: root   password: jason1
1 of 1 target successfully completed, 1 valid password found
[WARNING] Writing restore file because 62 final worker threads did
↪  not complete until end.
Hydra (http://www.thc.org/thc-hydra) finished at 2019-02-28
↪  15:26:33
```

Listing 15. Sample log generated by launching a brute-force attack during dry run.

```
      .:okOOOkdc'           'cdkOOOko:.
    .xOOOOOOOOOOOc       cOOOOOOOOOOOx.
   :OOOOOOOOOOOOOOk,   ,kOOOOOOOOOOOOOO:
 'OOOOOOOOOkkkkOOOOO: :OOOOOOOOOOOOOOOO'
 oOOOOOOOO.MMMM.oOOOOoOOOOl.MMMM,OOOOOOOOo
 dOOOOOOOO.MMMMMM.cOOOOOc.MMMMMM,OOOOOOOOx
 lOOOOOOOO.MMMMMMMMM;d;MMMMMMMMM,OOOOOOOOl
 .OOOOOOOO.MMM.;MMMMMMMMMM;MMMM,OOOOOOOO.
  cOOOOOOO.MMM.OOc.MMMMM'oOO.MMM,OOOOOOOc
   oOOOOOO.MMM.OOOO.MMM:OOOO.MMM,OOOOOOo
    lOOOOO.MMM.OOOO.MMM:OOOO.MMM,OOOOOl
     ;OOOO'MMM.OOOO.MMM:OOOO.MMM;OOOO;
      .dOOo'WM.OOOOocccxOOOO.MX'xOOd.
        ,kOl'M.OOOOOOOOOOOOO.M'dOk,
          :kk;.OOOOOOOOOOOO.;Ok:
            ;kOOOOOOOOOOOOOOk:
              ,xOOOOOOOOOOx,
                .lOOOOOOOl.
                   ,dOd,
                     .

       =[ metasploit v5.0.98-dev                          ]
+ -- --=[ 2045 exploits - 1106 auxiliary - 344 post       ]
+ -- --=[ 562 payloads - 45 encoders - 10 nops            ]
+ -- --=[ 7 evasion                                       ]

Metasploit tip: View advanced module options with advanced

[*] Processing vul.rc for ERB directives.
resource (vul.rc)> setg RESOURCE /root/gather.rc
RESOURCE => /root/gather.rc
resource (vul.rc)> use exploit/vulnserver
[*] No payload configured, defaulting to
↪  windows/meterpreter/reverse_tcp
resource (vul.rc)> set AutoRunScript
↪  post/multi/gather/multi_command
AutoRunScript => post/multi/gather/multi_command
resource (vul.rc)> set RHOST 192.168.81.158
RHOST => 192.168.81.158
resource (vul.rc)> set RPORT 9999
RPORT => 9999
resource (vul.rc)> exploit
[*] Started reverse TCP handler on 192.168.81.129:4444
[*] 192.168.81.158:9999 - Sending 350 byte payload...
[*] Sending stage (176195 bytes) to 192.168.81.158
[*] Meterpreter session 1 opened (192.168.81.129:4444 ->
↪  192.168.81.158:49172) at 2020-10-09 04:05:26 -0400

meterpreter >
[*] Session ID 1 (192.168.81.129:4444 -> 192.168.81.158:49172)
↪  processing AutoRunScript 'post/multi/gather/multi_command'
[*] Running module against WIN-TTMIH9LTG7O
[*] Running command ipconfig
[+] Command output saved to: /root/.msf4/loot/
↪  20201009040527_default_192.168.81.158_host.command_553149.txt
[*] Running command arp -a
[+] Command output saved to: /root/.msf4/loot/
↪  20201009040527_default_192.168.81.158_host.command_236368.txt
[*] Running command tasklist
[+] Command output saved to: /root/.msf4/loot/
↪  20201009040527_default_192.168.81.158_host.command_168109.txt
```

Listing 16. Sample log generated from launching a Metasploit exploit from our scenario language.

```
Date and Time:5:10 AM
Actual: 136045 packets (13973936 bytes) sent in 3.39 seconds
Rated: 4114044.4 Bps, 32.91 Mbps, 40052.79 pps
Flows: 0 flows, 0.00 fps, 136045 flow packets, 0 non-flow
Statistics for network device: eth0
        Successful packets:        136045
        Failed packets:            0
        Truncated packets:         0
        Retried packets (ENOBUFS): 0
        Retried packets (EAGAIN):  0
```

Listing 17. Sample log for network traffic generation.

```
key tab
pause 1
key enter
pause 1
type notepad
pause 1
key enter
pause 1
type "Testing Automation"
pause 1
key enter
type "which is very basic proof of concpet"
pause 1
key enter
type "showing a user behavior on notepad"
pause 1
key enter
type "in a very limited manner"
pause 1
key enter
key alt
key enter
```

Listing 18. Sample VDO configuration for emulating a user behavior.

```
Date and Time:8:22 AM
Jan 22 11:20:39 kali sendemail[3707]: Email was sent successfully!


Date and Time:2:42 AM
Jan 23 05:40:29 kali sendemail[1722]: ERROR => Received:
↪          554 5.7.1 [2] Message rejected under suspicion of SPAM;
↪  https://ya.cc/1IrBc 1579776029-NTJI8wwzTr-eSWu5qsR
```

Listing 19. Sample log for successful and unsuccessful phishing email generation.

Appendix D. Survey Questions

D1. Pre-Exercise

1. You classify yourself as a Senior or Junior?*
   - Senior
   - Junior
2. You classify yourself as a low/medium/high skilled individual in cybersecurity?*
   - Low
   - Medium
   - High
3. How many CTF you previously played*
   - 1 to 5
   - 5 to 10
   - More then 10
4. Rate your skills in different cybersecurity tools from 1 to 5, where 1 is the lowest value and 5 is the highest value:*
   - Kali linux
   - Nmap
   - Metasploit
   - Burp
   - hydra
   - Wireshark
   - Immunity Debugger

D2. Post-Exercise

1. Rate your skills in different cybersecurity tools from 1 to 5, where 1 is the lowest value and 5 is the highest value:*
   - Kali linux
   - Nmap
   - Metasploit
   - Burp
   - hydra
   - Wireshark
   - Immunity Debugger
2. How do you rate the difficulty of played CTF Easy/Medium/Hard*
   - Low
   - Medium
   - High
3. How realistic was the CTF compare to other CTF you played before? Give it rating from 1 to 5, where 1 indicates the lowest value and 5 indicates the highest value.
4. Did you havve any skill improvement after playing the CTF?*
   - Yes
   - No
   - Somewhat

CRediT authorship contribution statement

Muhammad Mudassar Yamin: Conceptualization, Resources, Investigation, Writing – original draft. **Basel Katt:** Supervision, Writing – review & editing.

References

Abbott, R.G., McClain, J.T., Anderson, B.R., Nauer, K.S., Silva, A.R., Forsythe, J.C., 2015. Automated Performance Assessment in Cyber Training Exercises. Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

Almroth, J., Gustafsson, T., 2020. Crate exercise control–a cyber defense exercise management and support tool. In: 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). IEEE, pp. 37–45.

Beuran, R., Pham, C., Tang, D., Chinen, K.-i., Tan, Y., Shinoda, Y., 2017. Cytrone: an integrated cybersecurity training framework. SCITEPRESS–Science and Technology Publications 157–166.

Beuran, R., Tang, D., Pham, C., Chinen, K.-i., Tan, Y., Shinoda, Y., 2018. Integrated framework for hands-on cybersecurity training: cytrone. Computers & Security 78, 43–59.

Ceri, S., Gottlob, G., Tanca, L., et al., 1989. What you always wanted to know about datalog(and never dared to ask). IEEE Trans Knowl Data Eng 1 (1), 146–166.

Chang, W., Zhao, S., Wei, R., Wellings, A., Burns, A., 2019. From java to real-time java: a model-driven methodology with automated toolchain. In: Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, pp. 123–134.

Chothia, T., Novakovic, C., 2015. An offline capture the flag-style virtual machine and an assessment of its value for cybersecurity education. 2015 {USENIX} Summit on Gaming, Games, and Gamification in Security Education (3GSE 15).

Ctfd : The easiest capture the flag platform. 2021. (Accessed on 10/13/2021) https://ctfd.io/.

Datalog: Deductive database programming. 2020. (Accessed on 09/30/2020) https://docs.racket-lang.org/datalog/index.html.

Eckroth, J., Chen, K., Gatewood, H., Belna, B., 2019. Alpaca: Building dynamic cyber ranges with procedurally-generated vulnerability lattices. In: Proceedings of the 2019 ACM Southeast Conference, pp. 78–85.

Edgar, T.W., Manz, D.O., 2017. Research methods for cyber security. Syngress.

Hutchins, E.M., Cloppert, M.J., Amin, R.M., 2011. Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains. Leading Issues in Information Warfare & Security Research 1 (1), 80.

Jones, R.M., O'Grady, R., Nicholson, D., Hoffman, R., Bunch, L., Bradshaw, J., Bolton, A., 2015. Modeling and integrating cognitive agents within the emerging cyber domain. In: Proceedings of the Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC), Vol. 20. Citeseer.

Kramer, F., Starr, S., Wentz, L., 2006. Actions to enhance the use of commercial information technology (it) in department of defense (dod) systems. In: Fifth International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS'05). IEEE, pp. 8–pp.

Leitner, M., Frank, M., Hotwagner, W., Langner, G., Maurhart, O., Pahi, T., Reuter, L., Skopik, F., Smith, P., Warum, M., 2020. Ait cyber range: Flexible cyber security environment for exercises, training and research. In: Proceedings of the European Interdisciplinary Cybersecurity Conference, pp. 1–6.

Lloyd, J.W., 2012. Foundations of logic programming. Springer Science & Business Media.

Libvirt: The virtualization api. 2021. (Accessed on 02/25/2021)https://libvirt.org/.

Lord, D., 1985. Worldwide networking for academics. Data Processing 27 (5), 27–31.

Maennel, K., 2020. Learning analytics perspective: Evidencing learning from digital datasets in cybersecurity exercises. In: 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). IEEE, pp. 27–36.

Mirkovic, J., Benzel, T.V., Faber, T., Braden, R., Wroclawski, J.T., Schwab, S., 2010. The deter project: Advancing the science of cyber security experimentation and test. In: 2010 IEEE International Conference on Technologies for Homeland Security (HST). IEEE, pp. 1–7.

Moore, E., Fulton, S., Likarish, D., 2017. Evaluating a multi agency cyber security training program using pre-post event assessment and longitudinal analysis. In: IFIP World Conference on Information Security Education. Springer, pp. 147–156.
Mitre att&ck®. 2020. (Accessed on 09/30/2020) https://attack.mitre.org/.
Ncr orchestrator application. 2021. (Accessed on 10/15/2021) https://tinyurl.com/4y57t3vb.
Naik, M., 2020. Petablox: Large-Scale Software Analysis and Analytics Using Datalog. Technical Report. GEORGIA TECH RESEARCH INST ATLANTA ATLANTA United States.
NIST, 2020. Cyber ranges. (Accessed on 11/09/2020) https://www.nist.gov/system/files/documents/2018/02/13/cyber_ranges.pdf.
Openstack at ntnu - skyhigh - ntnu wiki. 2021. (Accessed on 03/24/2021) https://www.ntnu.no/wiki/display/skyhigh/Openstack+at+NTNU.
Opentosca. 2021. (Accessed on 02/25/2021) https://www.opentosca.org/.
Paul-pols—the-unified-kill-chain.pdf. 2020. (Accessed on 09/30/2020) https://www.csacademy.nl/images/scripties/2018/Paul-Pols---The-Unified-Kill-Chain.pdf.
Peker, Y.K., Ray, L., Da Silva, S., Gibson, N., Lamberson, C., 2016. Raising cybersecurity awareness among college students. Journal of The Colloquium for Information Systems Security Education, Vol. 4. 17–17
Pham, C., Tang, D., Chinen, K.-i., Beuran, R., 2016. Cyris: A cyber range instantiation system for facilitating security training. In: Proceedings of the Seventh Symposium on Information and Communication Technology, pp. 251–258.
Russo, E., Costa, G., Armando, A., 2018. Scenario design and validation for next generation cyber ranges. In: 2018 IEEE 17th International Symposium on Network Computing and Applications (NCA). IEEE, pp. 1–4.
Russo, E., Costa, G., Armando, A., 2020. Building next generation cyber ranges with crack. Computers & Security 101837.
Schmidt, D.C., 2006. Model-driven engineering. Computer-IEEE Computer Society-39 (2), 25.
Schreuders, Z.C., Shaw, T., Shan-A-Khuda, M., Ravichandran, G., Keighley, J., Ordean, M., 2017. Security scenario generator (SecGen): A framework for generating randomly vulnerable rich-scenario vms for learning computer security and hosting {CTF} events. 2017 {USENIX} Workshop on Advances in Security Education ({ASE} 17).
Systemusagedata - openstack. 2021. (Accessed on 10/12/2021) https://tinyurl.com/a8y3bed.
Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles - sae international. 2020. (Accessed on 09/25/2020) https://www.sae.org/standards/content/j3016_201806/.
Towards a common ecsc roadmap – enisa. 2022. https://www.enisa.europa.eu/publications/towards-a-common-ecsc-roadmap. (Accessed on 01/11/2022).
Vaishnavi, V.K., Kuechler, W., 2015. Design science research methods and patterns: innovating information and communication technology. Crc Press.
Vykopal, J., Ošlejšek, R., Čeleda, P., Vizvary, M., Tovarňák, D., 2017. Kypo cyber range: design and use cases. SciTePress.
Vykopal, J., Vizváry, M., Oslejsek, R., Celeda, P., Tovarnak, D., 2017. Lessons learned from complex hands-on defence exercises in a cyber range. In: 2017 IEEE Frontiers in Education Conference (FIE). IEEE, pp. 1–8.
vncdotool. pypi. 2020. (Accessed on 10/08/2020) https://pypi.org/project/vncdotool/.
What is infrastructure as code (iac)? | ibm. 2020. (Accessed on 09/25/2020) https://www.ibm.com/cloud/learn/infrastructure-as-code.
What is soar? security definition | fireeye. 2020. (Accessed on 09/25/2020) https://www.fireeye.com/products/helix/what-is-soar.html.
Weeden, B.C., Cefola, P.J., 2010. Computer systems and algorithms for space situational awareness: history and future development. Advances in the Astronautical Sciences 138 (25), 2010.
White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A., 2002. An integrated experimental environment for distributed systems and networks. ACM SIGOPS Operating Systems Review 36 (SI), 255–270.
Willems, C., Meinel, C., 2012. Online assessment for hands-on cyber security training in a virtual lab. In: Proceedings of the 2012 IEEE Global Engineering Education Conference (EDUCON). IEEE, pp. 1–10.
Yamin, M.M., Katt, B., 2018. Detecting malicious windows commands using natural language processing techniques. In: International Conference on Security for Information Technology and Communications. Springer, pp. 157–169.
Yamin, M.M., Katt, B., 2018. Inefficiencies in cyber-security exercises life-cycle: A position paper. CEUR workshop proceedings.
Yamin, M.M., Katt, B., 2019. Modeling attack and defense scenarios for cyber security exercises. In: 5th interdisciPlinary cyber research conference 2019, p. 7.
Yamin, M.M., Katt, B., Gkioulos, V., 2019. Detecting windows based exploit chains by means of event correlation and process monitoring. In: Future of Information and Communication Conference. Springer, pp. 1079–1094.
Yamin, M.M., Katt, B., Gkioulos, V., 2020. Cyber ranges and security testbeds: scenarios, functions, tools and architecture. Computers & Security 88, 101636.
Yamin, M.M., Katt, B., Nowostawski, M., 2021. Serious games as a tool to model attack and defense scenarios for cyber-security exercises. Computers & Security 110, 102450.
Yamin, M.M., Katt, B., Torseth, E., Gkioulos, V., Kowalski, S.J., 2018. Make it and break it: An IoT smart home testbed case study. In: Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control, pp. 1–6.
Yasuda, S., Miura, R., Ohta, S., Takano, Y., Miyachi, T., 2016. Alfons: A mimetic network environment construction system. In: International Conference on Testbeds and Research Infrastructures. Springer, pp. 59–69.
Zaber, M., Nair, S., 2020. A framework for automated evaluation of security metrics. In: Proceedings of the 15th International Conference on Availability, Reliability and Security, pp. 1–11.

Muhammad Mudassar Yamin is currently doing his Ph.D. at the Department of Information and Communication Technology at the Norwegian University of Science and Technology. He is the member of the system security research group and the focus of his research is system security, penetration testing, security assessment, intrusion detection. Before joining NTNU, Mudassar was an Information Security consultant and served multiple government and private clients. He holds multiple cyber security certifications like OSCE, OSCP, LPT-MASTER, CEH, CHFI, CPTE, CISSO, CBP.

Basel Katt is currently working as an Associate Professor at the Department of Information and Communication Technology at the Norwegian University of Science and Technology. He is the technical project leader of Norwegian cyber range. Focus of his research areas are: • Software security and security testing • Software vulnerability analysis • Model driven software development and model driven security • Access control, usage control and privacy protection • Security monitoring, policies, languages, models and enforcement