Computers & Security 122 (2022) 102892

Contents lists available at ScienceDirect

Computers & Security

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

Use of cyber attack and defense agents in cyber ranges: A case study

Muhammad Mudassar Yamin*, Basel Katt

Norwegian University of Science and Technology, Gjøvik 2815, Norway

ARTICLE INFO

Article history:
Received 15 September 2021
Revised 8 July 2022
Accepted 6 August 2022
Available online 27 August 2022

Keywords:
Cyber attack agent
Cyber defense agent
Cyber range
Security exercise

ABSTRACT

With the ever-changing cybersecurity landscape, the need for a continuous training for new cybersecurity skill sets is a requirement. Such continuous training programs can be delivered on platforms like cyber ranges. Cyber ranges support training by providing a simulated or emulated representation of a computer network infrastructure, besides additional training and testing services. Cyber attack and defense skills can be gained by attacking and defending a simulated or an emulated infrastructure. However, to provide a realistic training in such infrastructures, there is a need for necessary friction in the environment. Human teams, playing both attackers' and defenders' roles, provide this friction. Involving human teams in large-scale cybersecurity exercises is relatively inefficient and not feasible for standardizing training because different teams apply different tactics. Currently, the proposed solutions for cyber range training platforms focus on automating the deployment of the cybersecurity exercise infrastructure but not on the execution part. This leaves a room for improving exercise execution by adding realism and efficiency. This research presents an agent-based system that emulates cyber attack and defense actions during cybersecurity exercise execution; this helps provide realistic and efficient cybersecurity training. To specify agents' behavior and decision making, a new formal model, called the execution plan (EP), was developed and utilized in this work.

1. Introduction

Conducting operational cybersecurity exercises is a difficult and inefficient task (Yamin and Katt, 2018b). We have found that automating the different roles involved in cybersecurity exercises can reduce these inefficiencies (Yamin et al., 2018). These roles primarily involve a human team required to set up the exercise technical network infrastructure. Additionally, there is a team that attacks the deployed infrastructure as an attacker and a team that defends it as a defender (Yamin et al., 2020). There can be multiple ways with which a cybersecurity exercise can be executed that may or may not involve both attackers and defenders at the same time. However, in a realistic environment, to train attackers, the systems being attacked are expected to be defended by somebody. Because of shortage in the cybersecurity skills, it is very difficult to find people with the relevant skill set (Yamin and Katt, 2019a) to conduct continuous cybersecurity exercises. Moreover, different people have different tactics and techniques in cybersecurity operations, making a standardized assessment of cybersecurity exercises difficult (Herold et al., 2017). Therefore, there is a need for automating attack and defense roles in cybersecurity exercises. Despite its importance, there is a lack of research dealing with realism and efficiency in cybersecurity exercise execution in cyber ranges. Most of the related work deals with automating the creation and deployment of the exercise infrastructure. This leaves a room for researchers to improve the realism and efficiency of cybersecurity exercise execution. We tackle this issue by proposing an agent-based system, which models the attacker and defender roles and automate their execution as required. In particular, we developed a new modeling technique: execution plan (EP). EP is a multi-level model for specifying behavior and decision-making processes for attacking and defending agents. We argue that our solution (1) adds the necessary friction in the cybersecurity exercise environment to make it realistic and (2) reduce the human input of attackers and defenders to make exercise execution more efficient. Therefore, in the current research, we aim to answer the following research question (RQ):

***RQ** : How can cybersecurity attack and defense scenario models be executed autonomously in a cybersecurity exercise to make cybersecurity exercise execution realistic and efficient?*

We present our experience in developing and using cyber-attack and defense agents during cybersecurity exercises against human teams. The current paper focuses on the conceptual design, agent decision modeling, practical implementation and user experience with cyber-attack and defense agents. The system is evaluated using a case study against defined benchmarks. The case

* Corresponding author.
E-mail addresses: muhammad.m.yamin@ntnu.no (M.M. Yamin), basel.katt@ntnu.no (B. Katt).

https://doi.org/10.1016/j.cose.2022.102892
0167-4048/

study involved an operational cybersecurity exercise in which the attack and defense agents were deployed along with human participants. The attack agents were used to create forensic traces for blue team members, which were verified in their forensic reports. At the same time, the defense agents were used to add friction, or realism, in the exercise environment and were evaluated based on the compromised status of the systems on which they were deployed. The paper is structured as follows: First, we share the research background and related work. After that, we state our research methodology. We then present the conceptual design and technical implementation of the attack and defense agents. Finally, we present the experimental results and conclude the article.

2. Research background and related work

Cybersecurity skills are essential for both against individual inexperienced hackers and against coordinated teams of hackers that might or might not have governmental support. The conventional methods of teaching cybersecurity skills include lectures, assignments, seminars, and hands-on labs. Hands-on methods include competitions, challenges, and exercises such as the following:

1. Capture the flag (CTF), which focus on attacking, defending, or attack and defense at the same time.
2. Cyber defense exercises (CDX), which focus primarily on defending.

These competitions, challenges, and exercises are conducted in isolated and safe environments, which are called *cyber ranges*. Cyber ranges can host single standalone challenges for CTF competitions or represent a complex computer network infrastructure of one organization or a set of organizations within a CDX exercise (Yamin et al., 2020).

An important element missing from current virtual environments is an element of active opposition. A training simulation environment has static defenses (access control, firewalls, fixed set of intrusion methods, etc.) but lacks active opposition (e.g., monitor logs, block connections, exploit switching, or gather information). This results in cyber operators behaving as though opponents do not have a tangible existence and higher-level goals. Second, it ignores an opportunity to tailor the student's learning experience through adjustable adversary behavior (Jones et al., 2015).

Cyberwarfare is an imminent threat to military and civilian systems; it could damage the economy and the whole national security. Cyber aggressors are guided by cognitive behavior (script-kiddies, ideological activists, investigators, financial criminals, intelligence agents, or cyber warfighters). Building an effective training system for cyberwarfare currently faces many barriers. Current training environments are unable to capture the purpose, creativity, and adaptability of cyber warfighters, and cyber warfighters need to be effectively trained against a cunning and adaptive opponents.

We conducted a detailed study on cyber ranges (Yamin et al., 2020) and identified that after 2014, different operations in cybersecurity exercises have become automated.These operations focused on setting up a cyber range environment and automating the roles of attackers and defenders.

2.1. Related work

Multiple researchers have worked on the development of cyber attack and defense agents. Here, we discuss some of the work relevant to our research. For emulating attacker steps, a lot of research work has been carried out, resulting in open source, free, and commercially available solutions. Some of them are the following:

Splunk attack range[1] is a limited cyber range deployment tool in which a small infrastructure can be deployed on cloud and local machines. The infrastructure is monitored by various Splunk attack monitoring and detection engines. Different attacks of the infrastructures are simulated using ART *(Atomic Red Team)*[2]. ART follows the MITRE attack chain model[3] and can simulate an attack on Windows, Linux, and Mac OS systems. It uses YAML-based inputs to execute atomic tests for adversary actions on the target systems.

APTSimulator [4] is an open source advanced persistence threat simulation tool. The tool uses batch scripts with various hacking utilities to create system compromise traces like command and control, defense evasion, lateral movement, and so on. It is used for endpoint detection agent assessment, testing security monitoring and detection capabilities, and preparing the digital forensic class environment. It roughly follows the MITRE attack chain model and is also limited to emulating attacks on Windows-based host machines.

Metta [5] is an open source information security preparedness tool. The tool uses Virtualbox with different development tools and frameworks, like Redis/Celery, python, and vagrant, to simulate adversarial actions. Input is given to the tool in the form of a YAML file, which is parsed to execute the attacker's actions on the host- and network-based systems. Metta follows the MITRE attack chain model and is limited to emulating attacks on Windows-based systems.

In the case of cybersecurity exercises, there is a need for the repeatable (Holm and Sommestad, 2016) execution of attacker steps for standardized training. Moreover, there are legal and ethical concerns in developing autonomous cyber-attack agents (Yamin and Basel, 2018), so for an attack agent, the attack execution steps need to be planned while keeping a human in the loop before executing them in a cybersecurity exercise environment.

For the defense agent, most related work has focused on network- and host-based detection systems, while some have looked into introducing active attack-repellent features in defense agents. Randolph el al. (Jones et al., 2015) conducted research about modeling and integrating cognitive agents in the cyber domain. The purpose was to develop agents that can produce the necessary friction during cybersecurity exercises to create the needed realism. They developed a novel modeling approach for cyber offense and defense; they used the proposed models to create software adapters that translate from task-level actions to network-level events to support agent-network interoperability for cybersecurity operations. They presented a high-level defender goal hierarchy in which the defender has to (1) establish a baseline, (2) detect an ongoing attack, (3) stop an ongoing attack, and (4) prevent future attacks.

Kott et al. (2018b) put forward the idea of the development of a *"Hello world"* program for the intelligent autonomous defense agent. The researchers stated that an autonomous agent should have the following six capabilities to be considered intelligent:

1. The agent should be strictly associated with its environment and should be useless outside its designed environment.
2. The agent should interact with its environment using appropriate sensors.

[1] Splunk attack range ref: https://github.com/splunk/attack_range (Accessed on 01/27/2021

[2] Atomic red team ref: https://github.com/redcanaryco/atomic-red-team (Accessed on 01/25/2021)

[3] MITRE attack chain model ref: https://attack.mitre.org/ (Accessed on 01/25/2021)

[4] APTSimulator ref: https://github.com/NextronSystems/APTSimulator/tree/master/test-sets (Accessed on 01/27/2021)

[5] Metta ref: https://github.com/uber-common/mett (Accessed on 01/27/2021)

3. The agent should act to achieve its stated goals.
4. The agent's activities should be sustained overtime to be autonomous.
5. The agent should have an internal world model that can express its states with performance measures.
6. The agent should learn new knowledge and modify its model and goals over time based on the new acquired knowledge.

Theron et al. (2018) proposed the reference architecture for autonomous intelligent cyber defense agents (AICA). In their proposed architecture, the researchers stated that AICA has five main high-level functions:

1. Sensing world state information
2. Planning and action selection
3. Collaboration and negotiation
4. Action execution
5. Learning and knowledge improvement

The researchers also suggested a functional architecture for AICA and stated that according to their assessment, the development of such agents is not beyond the current technical capabilities and can be developed within ten years.

Although other researchers are also investigating autonomous intelligent cyber defense agents, the work is still in the design stage (Kott et al., 2018b; Theron et al., 2020). One implementation of such an agent was proposed by Jones et al. (2015), in which the researchers suggested the idea of adding friction in cybersecurity exercises. Their approach utilized cognitive modeling of human cybersecurity experts to model the behavior of the agent based on human expertise. We consider such an approach not suitable for agent development because human experts have different subjective experiences, which can result in unintentional bias in their behavior, as observed in data-driven AI algorithms (Yamin et al., 2021). Kott et al. (2018b) and Theron et al. (2018) provided the baseline requirements and the functional needs of a cyber defense agent, which we considered suitable to fulfill our requirements. For the attack agent, we find the ART, APTSimulator, and META approaches suitable for the usage in cybersecurity to conduct cybersecurity exercises. We integrated multiple ideas and approaches suggested by various researchers for the development of our attack and defense agents. Our attack agent follows a systemic step-by-step execution of attacks similar to ART, APTSimulator, and META, but it is integrated with a cybersecurity exercise orchestrator, making it suitable for computationally repeatable cybersecurity exercises and experiments. Our defense agent is also integrated with our cybersecurity exercise orchestrator and provides example implementation of the design presented by Kott et al. (2018b).

3. Methodology

We employed numerous research methodologies during this research activity. We used the **DSR** (*design science research methodology*) (Hevner and Chatterjee, 2010) for the overall development of the necessary artifacts for this research. DSR is a well-known research methodology that has five phases *1) awareness of the problem, 2) suggestion, 3) development, 4) evaluation,* and *5) conclusion*. These phases are iterative in nature, and the results of these phases are used to improve the overall design to produce a research artifact that addresses the research problem (Kuechler and Vaishnavi, 2008).

In the awareness phase, the research problem is studied and identified. This was conducted in previous studies, e.g., in Yamin and Katt (2018b); Yamin et al. (2020; 2018). In the suggestion phase, the solution's designs are proposed to address the research problem. This was conducted previously in Yamin and Katt (2019b); Yamin et al. (2020). We are currently developing and evaluating the artifacts, and the present paper is presenting the results of this phase. For the development of such an artifact, model-driven engineering methodology is widely employed (Braghin et al., 2019). In such an approach, a complex problem is abstractly defined in the form of a model, and automation techniques are used to generate low-level artifacts from the abstract model. In our previous work, we developed a **DSL** (*domain specific language*) to specify the cybersecurity exercise scenario environment (Yamin and Katt, 2022). The scenario environment contains the exercise infrastructure and agents running within the environment. In this work, we used the DSL developed for automating the creation and deployment of the exercise environment. Additionally, we augmented the DSL with a new modeling technique based on attack/defense trees that we call execution plans (EPs). EPs enable a designer to model the behavior and decision making of attack and defense agents. Finally, we use **applied experimentation** in operational cybersecurity exercises against defined benchmarks to gather the analytical data for evaluating the performance of developed cybersecurity defense agents (Edgar and Manz, 2017).

4. Conceptual design

This research work is a part of a larger initiative in which the whole process of the cybersecurity exercise life cycle is automated. To achieve this, a DSL is developed to transform abstract concepts related to the cybersecurity exercise life cycle (Yamin and Katt, 2022) into concrete artifacts. These abstract concepts include defining the network topology, the vulnerabilities in the nodes connected to the network, the benign network traffic, and the attacker and defender behavior. In this work, our scope is limited to attacker and defender behavior, so we only focus on the concepts involved in attack and defense agent development. Fig. 1 illustrates the DSL meta-model for defining attacker and defender properties used to generate agent artifacts. Later in Section 5, a concrete syntax will be presented, which will give an example of an instantiation of this DSL meta-model.

The attack agent comprises of a total six concepts. The first one is the *attack-action-id*, which is an identifier of a potential attack action. In case the attack action causes a particular tool to be executed, the tool name can be used as an identifier. The second, third, and fourth concepts are *Agent IP*, *Agent User ID*, and *Agent User Password*. These concepts provide the information about the agent from which the attack action is going to be performed. The fifth concept is *Argument*, which represents one of the properties that are needed for the attacker's action to be executed. This concept includes the tool to execute the attacker's action as well as the tool's specific arguments. The sixth and final concept for the attack agent is *Target*, which is the IP address of a machine on which the attack agent performs its actions. Similarly, the defense agent has five properties. The first three are *Agent IP*, *Agent User ID*, and *Agent User Password*, which provide the information necessary to install the defense agent on a system. The fourth property is the *Operating System*, which provides the information about the operating system that the defense agent is working on. The final property is the *Parameter*, which provides the defender with a specific knowledge base. The parameter contains a list of pairs, each of which consists of an attacker action and a defender reaction to it.

The DSL uses an orchestrator (Yamin and Katt, 2022) that implements the abstract concepts defined in it to concretely create operational artifacts for establishing the necessary exercise infrastructures, generating network traffic, emulating benign users, launching cyber attacks, and defending against these attacks. The orchestrator has a master control unit connected to the attack agents and used to control them in a semi-autonomous manner. On the other hand, the orchestrator configures the defense agents

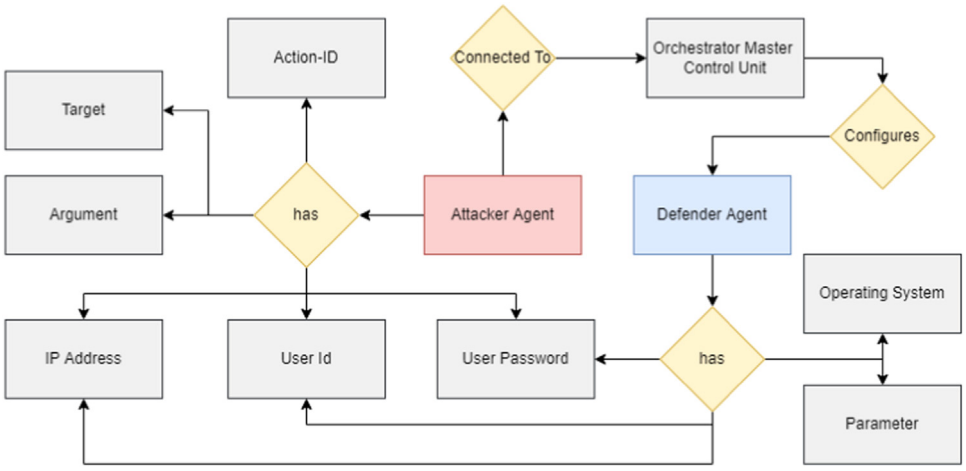

Fig. 1. A meta-model of the attack and defense agents.

before deployment so that they can work in an autonomous mode. The operational artifacts contain (1) network topology templates for cloud deployment, (2) specification of the vulnerabilities in the form of software, services and configurations, and (3) the specification of the benign, attack, and defense agents' behaviors present in the deployed network. The specification is given to the orchestrator in JSON, and it starts generating the necessary artifacts in five steps. First, the exercises network infrastructure is generated through a HEAT template for infrastructure deployment in Opestack-based cloud. In the second step, software, services, and configurations are invoked in the deployed infrastructure using a custom SSH-based installation and configuration module to make the infrastructure vulnerable. In the third step, a part of the deployed infrastructure is used to generate benign traffic using various automated tools like TCP relay and VNCD tool. In the fourth step, an attack agent is used to test and verify the vulnerabilities present in the exercise infrastructure. In the fifth and final step, a defense agent is deployed in part of the infrastructure to add the necessary friction in the cybersecurity exercise. The DSL implementation related to exercise network infrastructure generation and generating benign user behavior and traffic is part of another research work. In the current work, we focus on the DSL instance of attackers and defenders.

There can be multiple ways attack and defense agents can be designed and deployed. This depends on the eventual goal of the agent, that is, what is expected from the agent. We can model the goals of the attack and defense agents based on the responsibilities of the red and blue teams. Lockheed Martin's *cyber kill chain* (CKC) course of action matrix (Hutchins et al., 2011) provides a simplified way to model the attack and defense phases. For the attacker, there are seven phases *reconnaissance, weaponization, delivery, exploitation, installation, command and control, and actions on objective*. These attack phases utilize a set of tools and techniques to achieve the attacker's eventual objectives and goals, which could be the disruption of services or extraction of data. On the defense side, there are six phases to stop the attacker: *detect, deny, disrupt, degrade, deceive, and destroy*. The defender uses different network/host intrusion detection and prevention systems, firewalls, antivirus software, and honeypots to achieve the objective of stopping the attacker.

Let us analyze the course of action matrix for attackers and defenders. Here, the attacker's reconnaissance and weaponization goals can be detected by external information sources like web analytics and NIDS (network intrusion detection systems). In contrast, exploitation and installation can be detected by HIDS (host intrusion detection system). Although detecting an attack is desirable at an early stage, a host-based system can be better suited to respond directly to the attack; it can detect a security event and automatically respond to it by making operational changes such as applying local firewall rules and installing patches through its knowledge base without relying on an external input to deny the attacker actions. The knowledge base can contain information about the expected attacker's actions and the appropriate defender response. This knowledge can be useful for known attack tactics and techniques; however, it needs to be updated for new attack detection and responses, which require some intelligence. This intelligent behavior can be learned by analyzing the attack vectors and implementing security actions against them, manually first and automatically later. The attack vectors can be learned by constantly monitoring the system state and detecting changes. When a change is detected, the events that lead to that change can be fetched for identifying the malicious actions. A set of predefined reactions can be specified for implementation against a particular set of actions to deny the attacker from using them for further exploitation.

All components and parts of a cybersecurity exercise environment are considered as a system, and each system is running software and services with system-specific configurations. The system for the attack agent is a Kali Linux machine controlled by another system running our developed orchestrator software and using SSH as a service for communication with the Kali machine. The orchestrator can control multiple Kali machines to lunch multiple attacks at the same time. Similarly, for the defender part, the orchestrator can inject a defense agent with its knowledge base as a software that can independently run on the injected system to protect its software, services, and configurations. Additionally, there are traffic generators that are present in the cybersecurity exercise environment, which are basically attack agents performing benign activities such as replying PCAP files and automating GUI user behavior using VNCDtool. The agents and their interactions are presented in Fig. 2, which is mapped with the third, fourth, and fifth steps of the orchestrator (Yamin and Katt, 2022).

The developed agents operate in a cloud-based cybersecurity exercise environment. The environment has attack and vulnerable machines on which the attack and defense agents are operating. The vulnerable machines have vulnerabilities related to software, services, and configurations that an attack agent can remotely sense. The behavior of the agents in the environment is governed by the world state. The *world state* represents the software's, services', and configurations' specific information provided to the agents. New information about the world state is gathered from the environment in which the agents are operating by using their sensing capability. The sensing capability indicates which

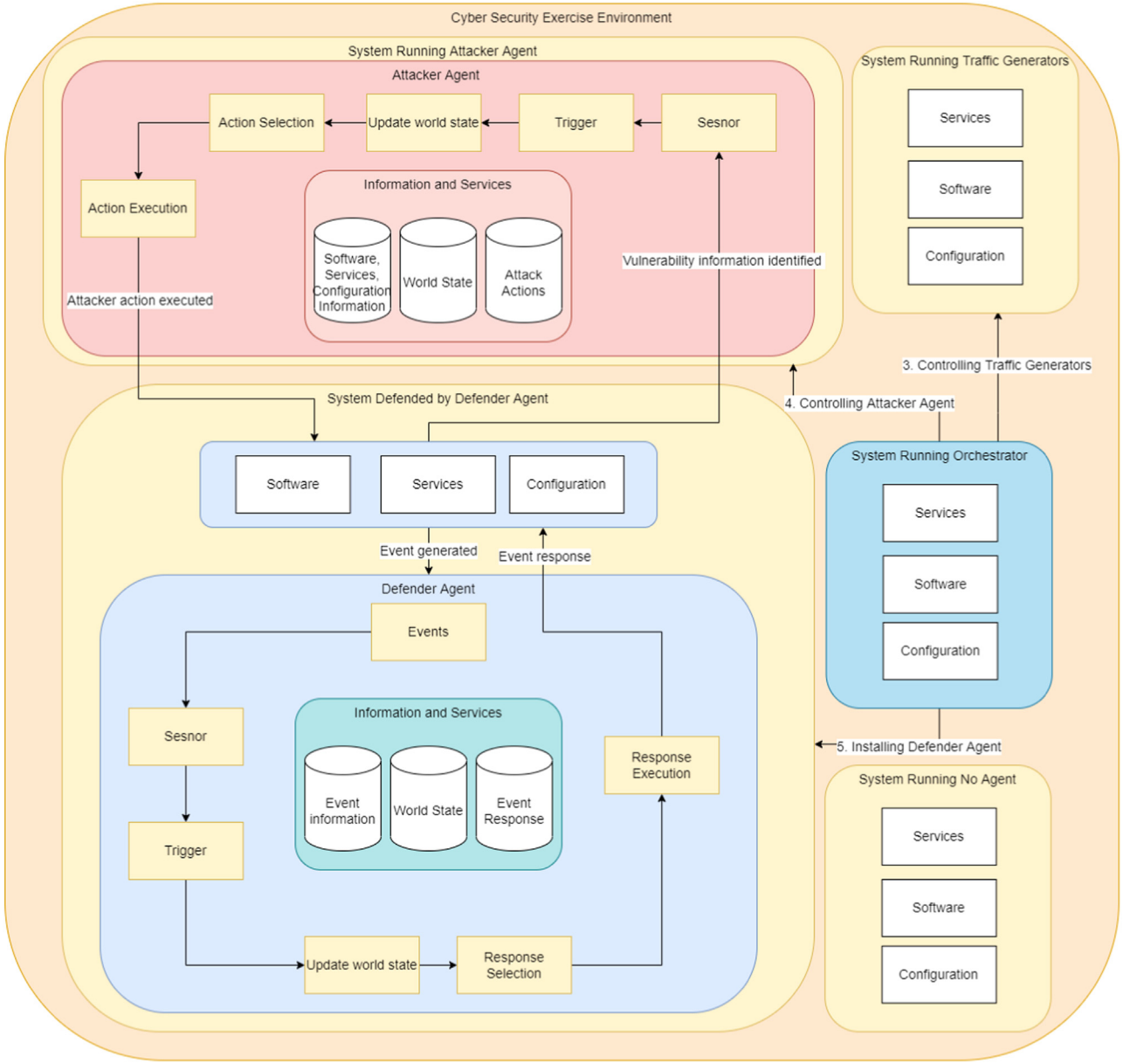

Fig. 2. Attack and defense agent environment and interactions.

type of systems the agent is interacting with and triggers an action when it finds some specific information about the world state (Kott et al., 2018a; 2018b).

For the attack agent, active and passive reconnaissance tools like arp-scan and Nmap are used to gather information about the services running in the network. This information is used to create the world model for the attack agent, and changes in these services will update the world state for the attacker. When the attack agent senses a vulnerable service, it triggers a change in its world state, upon which an attack action is selected and executed on the vulnerable machine. Similarly for the defense agent, Windows security logs provides an active sensing capability to create a model of the world state, which includes the type of software and services running on the system and any changes in their configuration. When an attacker interacts with the system defended by a defense agent, it creates logs that are then used to update the world state and trigger a reaction from the defense agent. The decision-making process of the proposed agents are discussed in Section 4.1.

When attacker's actions are executed, events are created in the vulnerable machine environment. The defense agent has a list of malicious events, the current world state, and responses to those events. The defense agent can sense the known events generated by the attacker actions, which can trigger a change in its world state. The defense agent then selects an appropriate response and executes it to counter the attack agent's actions. The interactions between the attack and defense agents in the cybersecurity exercise environment is presented in Fig. 2, and details of the attack and defense agent workflows are presented in Fig. 4 and Fig. 5 respectively.

In terms of deploying the agents in the cybersecurity exercise environment, there are certain considerations. Some researchers have implemented script execution techniques on machines [6] to generate attack logs; however, we deployed the attack agents on

[6] Realistic non player characters for training cyber teams ref: https://insights.sei.cmu.edu/blog/generating-realistic-non-player-characters-for-training-cyberteams/ (Accessed on 04/19/2021)

a remote machine to emulate realistic adversary behavior. On the other hand, the defense agents were deployed locally on the machines because a central command and control unit could have been compromised to disable the defenders' functionality.

4.1. Agent decision modeling

On an abstract level, our agents have five properties— *Knowledge*, *Goals*, *Conditions*, *Actions*, and *Commands*, where the knowledge is provided through the DSL. The *Knowledge* of an agent contains information about the world state, like the software, services, and configurations running in the environment. An agent's *Condition* is used to perform condition-specific actions on the software, services, and configurations. These conditions are triggered based on events that change the world state. An *Action* is executed using a set of *Commands*. A set of successful action executions result in the achievement of a *Goal*, which is modeled based on the *CKC* course of action matrix. For the attacker, these goals are *Reconnaissance, Delivery, Exploitation, Installation, Command and Control* and *Actions on Objective*. Similarly, for the defender, these goals are *Detect* and *Deny*.

The DSL instance is translated into EPs (Execution Plans) for the achievement of specific *Goals*. We adapted the concepts from the attack and defense trees (Kordy et al., 2014), as well as the hierarchy of action plans (Kotenko, 2005), to develop the EP models. The EP models consist of three levels of decisions *high, medium,* and *low*.

4.1.1. EP model

EPs are tree-structured models that represent the agent's decision-making process. An EP describes the goals, conditions, and commands of an agent, as well as showing the path that needs to be taken to reach the final conditions and fulfill a goal. These conditions result in one of the following EP outputs: plan fulfilled, plan not fulfilled, or plan maybe impractical.

The root of an EP tree is the goal of an agent, and an end-leaf of an EP tree represents the commands that will fulfill an agent's goal. An EP consists of three decision levels: Level 1, Level 2, and Level 3. Each level is represented as a sub-tree of the EP tree. The root of Level 1 of an EP tree is the root of the EP tree. The leaves of one sub-tree are the roots of the next level sub-tree. A parent node is connected with its children nodes using two possible operators, *AND* and *OR*, which are represented by $\wedge$ and $\vee$, respectively. The semantics of the nodes and operators in an EP tree depend on the level where the node exists.

Level 1 The Level 1 sub-tree of an EP tree is the first high level sub-tree of the an EP tree. The root node of a Level 1 sub-tree of an EP tree represents the main goal of the EP tree, and the leaves represent a set of sub-goals. The operator $\wedge$ is used if all the sub-goals needs to be fulfilled for the parent goal to be achieved. On the other hand, $\vee$ can be used if the fulfillment of one sub-goal will result in the fulfillment of the parent goal.

Level 2 The Level 2 sub-tree of an EP tree is the second medium-level sub-tree of an EP tree. The root of a Level 2 sub-tree is a leaf in the Level 1 sub-tree or the root of the EP tree where the Level 1 sub-tree consists of one node. Level 2 represents a sequence of conditions that need to be checked to decide which actions should be executed. The nodes of a Level 2 sub-tree are conditions with two possible outputs *Yes/No*, meaning that only $\vee$ operators are allowed in a Level 2 sub-tree. Each parent node is connected to, at most, two children nodes, which represents the next conditions to be checked. A special type of condition is called "Not Fulfilled." It is a final condition with no children, and it is denoted by the symbol **|**; reaching this condition means that the plan is not or cannot be fulfilled. A leaf in a Level 2 sub-tree can be either an NF "Not Fulfilled" condition or an action.

Level 3 The Level 3 sub-tree of an EP tree is the third low-level sub-tree of the EP tree. Level 3 roots are actions represented in the leaf nodes of the Level 2 sub-tree, and the nodes represent concrete commands. Both $\wedge$ and $\vee$ are allowed in a Level 3 sub-tree. $\wedge$ means all the children's commands need to be executed to achieve the action, while $\vee$ means that the execution of any of the commands can achieve the action.

Output Plan output "Fulfilled" is reached when all the commands in the EP tree leaves are executed successfully and the goal is achieved. Plan output "Impractical" is reached when the result of the execution of one command leaf is not successful. Plan output "Not Fulfilled" is reached when the agent cannot reach an action leaf because of knowledge or resource limitations. The EP plan in this case will stop at the Level 2 sub-tree.

4.1.2. EP formal model

We use Datalog[7] for formal modeling of the agents decisions and to verify the different decision properties like: *is the goal fulfilled or not?*. 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 a 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 a 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 a literal represents the rules in the clause. We can represent the above mentioned facts that *"John is the father of Harry"* and *"Harry is the*

[7] Datalog ref: https://docs.racket-lang.org/datalog/index.html (Accessed on 09/30/2020)

father of Larry" as follows:

$Father(Harry, John)$

$Father(Larry, Harry)$

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:

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

For our agents we define 4 basic predicates for decision modeling, which are *1) Goal, 2) Condition, 3) Action* and *4) Fulfilled*. The facts for the decision model with their definitions are as follows:

Definition 1. A *Goals* predicate is logically presented as *Goals (Goal, SubGoal)*, and it has two variables *Goal* and *SubGoal*. The *Goal* is a string value that indicates an attacker's goal or a defender's goal, like *'Exploit System'* for attackers and *'Prevent Attacks'* for defenders. The *SubGoal* is also a string values which contains sub goals for achieving the *Goal* like *'Reconnaissance'* and *'Exploitation'* for attack and *'Detect'* and *'Deny'* for defense. In the following are concrete examples of the predicate *Goals* for attackers and defenders:

For attackers:

$Goals(\text{'}ExploitSystem\text{'}, \text{'}Reconnaissance\text{'})$

$Goals(\text{'}ExploitSystem\text{'}, \text{'}Exploitation\text{'})$

For defenders:

$Goals(\text{'}PreventAttacks\text{'}, \text{'}Detect\text{'})$

$Goals(\text{'}PreventAttacks\text{'}, \text{'}Deny\text{'})$

Definition 2. A *Condition* predicate is logically presented as *Condition (Goal, Parameter, Command)* and it has three variables *Goal*, *Parameter* and *Command*. *Goal* is a string value which is the leaf of Level 1 sub-tree consisting of goals and sub-goals like *'Reconnaissance'*. A *Parameter* is a string value that is condition specific to achieve the goal like *'Active'* for performing active reconnaissance. As *Command* contains parameters for performing a low-level actions like service and version scan using Nmap *'-sS -sV'*. A concrete example of a *Condition* predicate is presented as follows:

$Condition(\text{'}Reconnaissance\text{'}, \text{'}Actvie\text{'}, \text{'} - sS - sV\text{'})$

Similarly a defense *Condition* can be represented as follows:

$Condition(\text{'}Deny\text{'}, \text{'}shell.exe\text{'}, \text{'}taskkill/IM''shell.exe''/F\text{'})$

Definition 3. An *Action* predicate is logically presented as *Action (ActionName, ActionTarget)*, and it has two variables *ActionName* and *ActionTarget*. *ActionName* is a string value that contains a low-level goal action execution like *Nmap* for attacker and *Deny* for defender. *ActionTarget* is a string value that contains the information of a machine address on which the action is to be executed for the attacker and the pattern on which the action is triggered by the defender. A concrete example of *Action* predicate is presented as follows:

For attackers:

$Action(\text{'}Nmap\text{'}, \text{'}172.168.2, 2\text{'})$

For defenders:

$Action(\text{'}Deny\text{'}, \text{'}shell.exe\text{'})$

Definition 4. A *Fullfilled* predicate is logically presented as *Fullfilled (Goal, ActionName)* and it has two variables *Goal* and *ActionName*. A *Goal* is a string value that indicates high-level goal like *'Reconnaissance'* and *ActionNamel* is a string value that indicates a concrete tool or action to achieve the high-level goal, like *'Nmap'*. A concrete examples of *Fullfilled* predicates is presented as follows:

For attackers:

$FullFilled(\text{'}Reconnaissance\text{'}, \text{'}Nmap\text{'})$

For defenders:

$FullFilled(\text{'}Deny\text{'}, \text{'}shell.exe\text{'})$

4.1.3. EP model verification

The decision model presented in Section 4.1 help us to verify various agent properties before their execution, like:

- How a high-level goal can be translated into to low-level actions
- Can an agent fulfill a given goal?
- What information is missing to achieve a given goal?

To verify the decision model we define a new predicate, called *CheckGoals*, which takes two variables a *Goal* and a *SubGoal*, and is logically presented as *CheckGoals (Goal,SubGoal)*. A logical relationship is defined between the *Goal* and *SubGoal* so it can be established whether the *Goal* is a leaf for *Level 2* conditions or the *SubGoal*.

CheckGoals (Goal, SubGoal) $\leq$ *CheckGoals (SubGoal, Goal)*

FullFilled predicate is used to link the *Goal* and *SubGoals* which is presented as fallows:

FullFilled (Goal, SubGoal) $\leq$ *CheckGoals (Goal, SubGoal)*

Furthermore, it is defined whether a *SubGoal* needs to be completed in order to achieve the *Goal*. A relationship is established between *Goal*, *SubGoal* and *Action* using *transitive* property which is presented as fallows:

FullFilled (Goal, SubGoal) $\leq$ *CheckGoals (Goal, Action) & FullFilled (SubGoal, Action) & (Goal! = SubGoal)*

To verify the attack decision for achieving a high-level goal using low-level actions based upon certain conditions, the following clause can be defined:

FullFilled ('Reconnaissance ', SubGoal) & Condition (Goal, 'Active', Commands) & *Action (SubGoal, ActionTarget)*

The clause will return the high-level goal, and the low-level specific actions to be executed based upon the specific command. Similarly, for the defense decision, the following clause can be defined:

FullFilled ('Deny', SubGoal) & Condition ('Deny', SubGoal, Commands)

The clause will return (1) the low-level pattern through which the action is triggered and (2) the low-level command to deny that action.

4.1.4. EP model representation

A schematic representation of an attack and a defense agents' EP models with all three levels are presented in Figure 3. A high-level *Goal* is given to an agent that has the aim of performing system exploitation. The sub-goals are *Reconnaissance* and *Exploitation*. The agent will check the information in its knowledge base to make medium-level decisions, for example, whether the information about the target is provided or not. If the target information is provided, it will check whether it is accessible or not; if it is accessible, then the agent will check if some specific arguments are present to perform a specific kind of reconnaissance actions, like *nbtscan*, which is a low-level decision. Otherwise, it will use a default reconnaissance technique like *nmap* or *netcat*. Similarly, if the target information is not provided, the agent will check whether there is a network interface and if on that

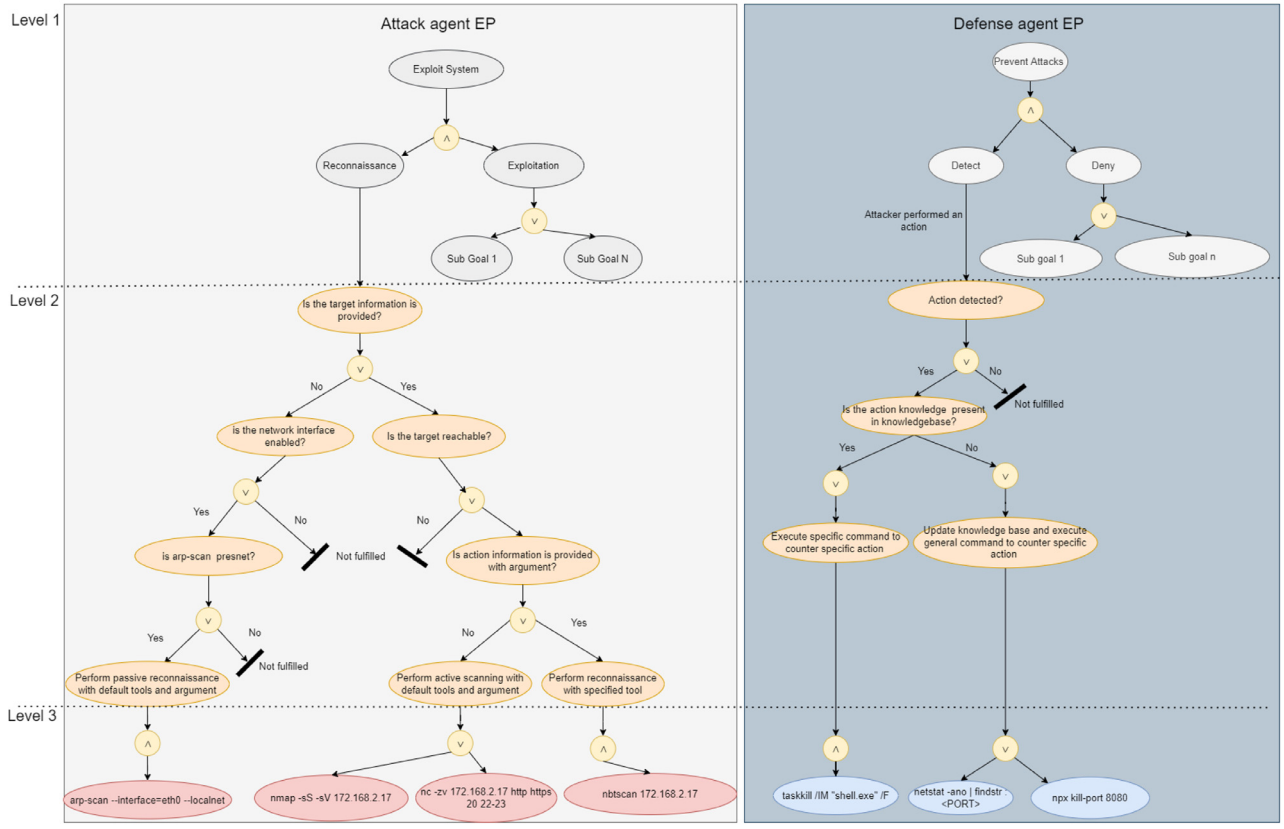

Fig. 3. Attack/Defense agent EP.

network interface it can perform *arp-scan*, which is a medium-level decision. If the condition returns true, then the agent can perform an arp-scan using default commands, which is a low-level decision.

For defenders, a high-level *Goal* is given to an agent to prevent attackers from performing any action. The sub-goals are to *Detect* and *Deny* attacker actions. Whenever an attacker performs an action, it creates an event. If the event is detected, then it is checked in the knowledge base of the defender; this is considered a medium-level decision. If the knowledge base contains information about the reaction to this specific action, then it will execute a specific command to the counter the attacker's action; this is considered a low-level decision, for example, killing a specific malicious process using *taskkill*. In another case, if there is no specific command to react to the attacker's action, then the agent will execute a general defense command to counter the action, such as closing the ports using *netstat* or *npx-kill-port*. If the action of the attacker is not detected, then the agent will fail to defend the system.

4.1.5. Attack agent

Conceptually, the attack agent's goal is to perform the steps involved in exploiting vulnerable systems during a cybersecurity exercise. The steps involve performing scanning, identifying vulnerable services, and launching an attack on them. Multiple adversary emulation tools already exist in academia and industry. In this work, various techniques ranging from logical programming (Yuen, 2015) to AI (Stoecklin, 2018) can be utilized for achieving this goal.

A model-driven approach for executing the attacks during a cybersecurity exercise can provide repeatable and standardized training. The model needs to follow a penetration testing execution standard [8] to leave realistic attack and exploitation traces for the defender or blue team members to identify. This can be achieved by specifying the attacker's actions in a scenario DSL, hence enabling the precise and systematic execution of attack steps and helping in the evaluation by the defenders in incident response and forensic analysis.

We combined complex attacker operations into six components of the DSL. An example of an instance of the DSL concrete syntax is presented in Listing 1. These components specify the attack techniques that are going to be used by the attacker and on which target it needs to be performed. The DSL instance components are used to provide the necessary information to the EP model to specify the behavior of the attacker based on the *cyber kill chain*. Once the model has been created, it is verified by executing it in different operational cybersecurity exercises with the same network topology to check whether its execution is repeatable for standardized training.

The attack agent DSL model has the following properties:

1. It can perform actions from the following list: *Reconnaissance, Delivery, Exploitation, Installation, Command and Control,* and *Actions on Objectives*.
2. It has six attributes: agent action name, agent to use, agent credential user ID and password, action-specific parameters and the target on which action is to be performed.
3. It runs in a separate attack machine in the exercise network environment, where it has network-level access to vulnerable machines present in the exercise network.

[8] Penetration testing execution standards ref: https://tinyurl.com/6cgn3cu (Accessed on 01/20/2021)

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

Listing 1. Concrete syntax for attacker DSL instance.

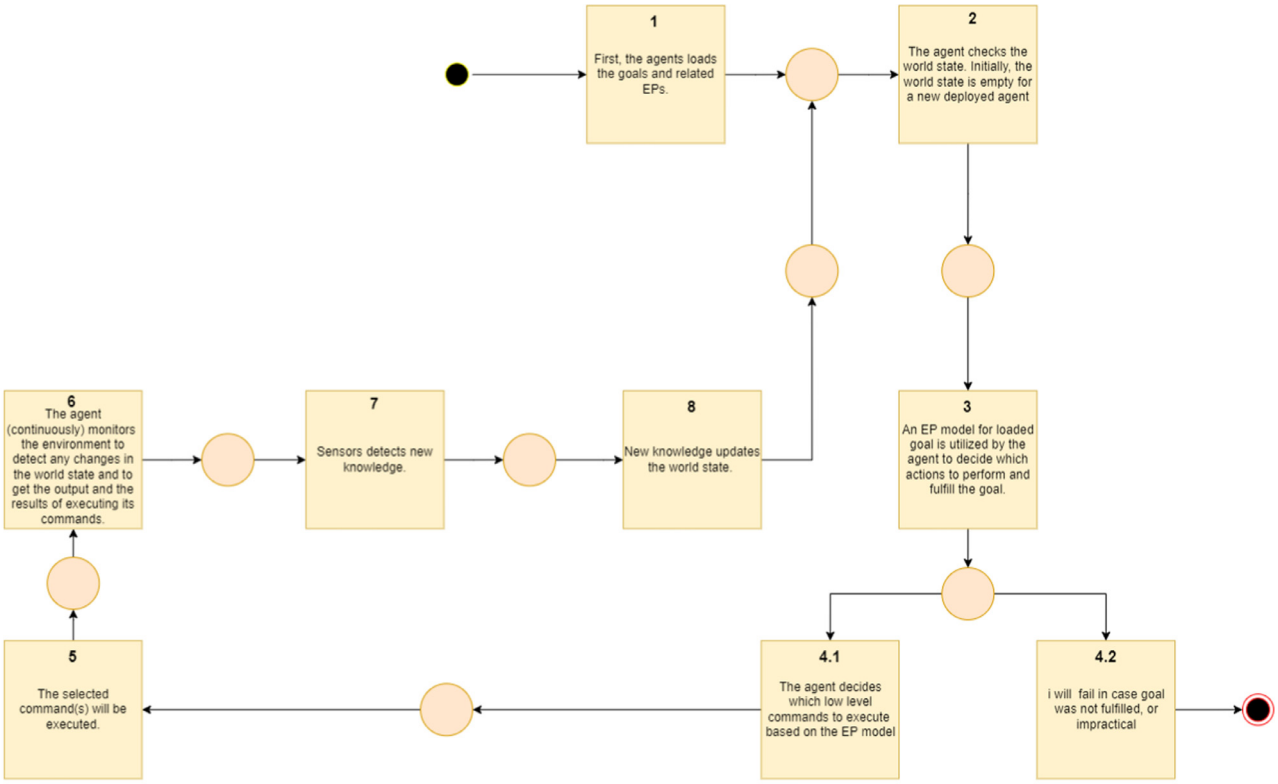

Fig. 4. Attack agent's workflow.

4. It interacts with exercise networks using a specific actions, which can collect information about the software, services, and configurations present in exercise machines and then preform other actions to exploit those machines. Deciding on which actions and commands to execute is specified in the EP model.

Utilizing the above-mentioned properties, an attack agent launches attacks in a semi-autonomous manner, as defined in the EP models. These EPs model the execution of attack phases presented in *CKC* by utilizing its attack agent properties. The attack agent's overall workflow is represented in Fig. 4.

An agent performs the following: (1) First, it loads the goals and related EPs. (2) it checks the world state. Initially, the world state is empty for a newly deployed agent. (3) An EP model for the loaded goal is utilized by the agent to decide which actions to perform and fulfill the goal. (4) The agent decides which low-level commands to execute based on the EP model (4.1), and it will fail in case the goal was not fulfilled or was impractical (4.2). (5) The selected command(s) will be executed. (6) The agent (continuously) monitors the environment to detect any changes in the world state and to get the output and results of executing its commands. (7) The sensors detect new knowledge. (8) New knowledge updates the world state. Finally, (2) the agent checks the new world state.

4.1.6. Defense agent

The defense agent's primary goal is to defend its system against external and internal attackers. This primary goal has additional sub-goals in which the defense agent has to *Detect, Deny, Disrupt, Degrade, Deceive, and Destroy* an attacker. The focus of this work is on *detect* and *deny*. These sub-goals are achieved by the usage of different tools and techniques at different stages of an attack. These tools and techniques include, but are not limited to, network and host intrusion detection and prevention systems, web analytics, security configuration, and system user training.

Our DSL instance is used to specify the defense agent's properties. Based on these properties, the EPs are executed by our orchestrator. The orchestrator inserts the agent in the machine present in the exercise network with the knowledge base of the events generated by the attacker's actions. Conceptually, the defense agent has the following properties:

1. It can detect and deny the actions performed by the attacker.
2. It has a knowledge base that contains information about the attacker's actions and their countermeasures.
3. It runs on the exercise machines being attacked.
4. It interacts with the events generated by the attacks and executes specified countermeasures on the machine it is running.

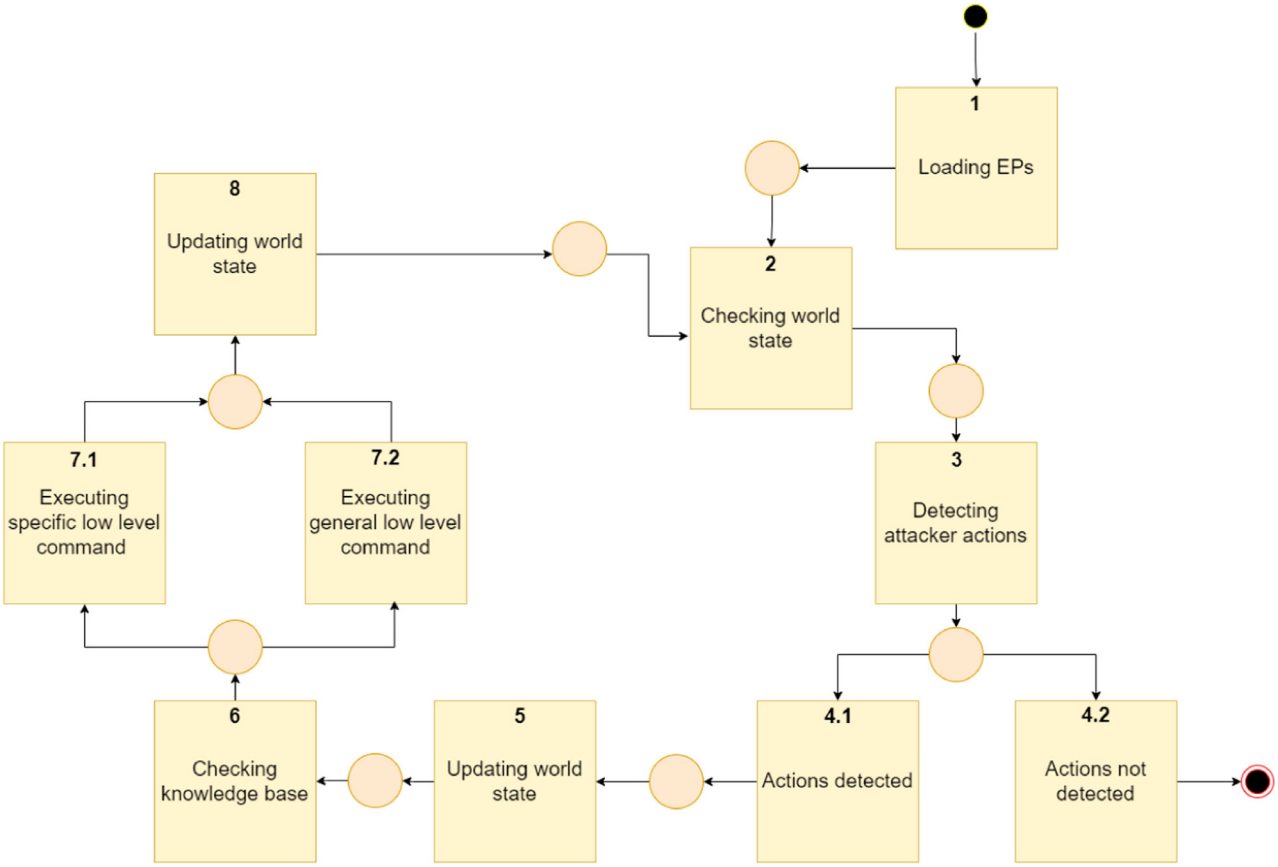

Fig. 5. Defense agent's workflow.

Utilizing the above-mentioned properties, the defense agent can perform defense measures against launched attacks in a semi-autonomous manner as defined in the EP models. These EPs model the execution of the defense phases presented in *CKC* by utilizing the defense agent properties. One key difference between the attack and defense agents is that the defense agent is not controlled by a Master and is independent in its execution. The Master only configures the knowledge base of the defense agent one time and uploads it on the machine that needs to be defended. The defense agent's overall workflow, uploaded to a machine, is represented in Fig. 5.

According to this workflow, the agent does the following: (1) First, the agent loads the goals and related EPs.(2) The agent checks the world state based on the EP model of detecting the attacker. Initially, the world state is empty for a newly deployed agent. (3) The agent will use its sensors to detect an event generated by attacker actions based on its knowledge base. (4) In the detection phase, if the attacker's action event is detected, then the agent will check its knowledge base to counter the attacker's action (4.1). If the attacker's actions were not detected by the agent, then it will fail to deny the attacker (4.2). (5) The agent will update its world state and act upon a new EP model to deny the attacker. (6) To deny the attacker, the agent will check its knowledge base. (7) Then, the agent will execute a low-level command that changes the world state. If the agent's knowledge base has information about the specific action, then it will execute a specific command (7.1). If the agent does not have specific countering information, then it will execute a general command (7.2). (8) New knowledge is updated in the world state of the agent. Finally, (2) the agent checks the new world state.

5. Technical implementation

5.1. Attack agent

An attack agent is a Kali Linux automation utility that can automate most of the Kali Linux environment tools. These tools can perform red team operations such as reconnaissance, weaponization, delivery exploitation, installation, command and control, and actions on objectives. In a cybersecurity exercise environment, one or more Kali Linux machines can be deployed to perform the attacker's actions. The orchestrator has a remote Master control unit that controls these machines using a dedicated SSH connection. The Kali Linux agents and the Master control unit work in a *Botnet Command and Control* (Feily et al., 2009; Zeidanloo and Manaf, 2009) manner. The attacker's actions are modeled in the DSL instance, which the Master control unit interprets and forwards to the Kali Linux machines as *EP*.

The Master control unit contains the *EP* of various attack stages, such as Nmap scripts for scanning and Metasploit scripts for exploitation. The resource script[9] of Metasploit is used to perform post-exploitation on the exploited machines. An extensive logging mechanism is integrated into the Master control unit, which can collect logs from the Kali machines to confirm whether the launched attack steps were successful or not. A schematic diagram presenting the main components existing in an attack agent is shown in Fig. 6. Each component represents a category of tools that can be used by the attack agent. For example, these compo-

[9] Resource scripts ref: https://docs.rapid7.com/metasploit/resource-scripts/ (Accessed on 01/19/2021

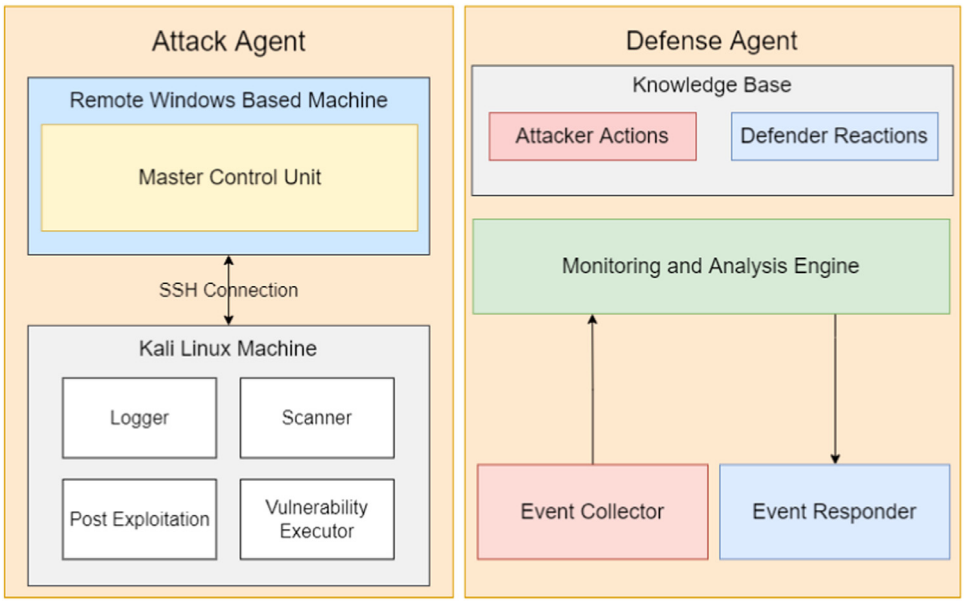

Fig. 6. Attack and defense agent technical implementation.

nents include scanners to collect information about the exercise environment and vulnerability executors to perform a particular attack or exploit. The results of the attacks are logged to update the world state and to inform the Master control unit about successful attacks.

5.1.1. Scanner

The scanner can work both passively and actively. In a passive mode, the scanner uses ARP resonance techniques for the identification of targets using Netdiscover[10]. When the targets are identified, it can switch into an active scanning mode and use Nmap for the identification of vulnerable services running on the system. The information of vulnerable services and their exploitation methods are provided in the DSL parameters.

5.1.2. Vulnerability executor

The vulnerability executor can take the information from the scanner to launch an attack based on predefined conditions, or it can follow the concrete action steps provided in the DSL instance and the EP. DSL instance contains the general static information for the attack agent, and the EP contains the execution plan to fulfill the attacker's goals. The conditions include finding a specific service or application signature and launching a relevant and well-known attack or exploit. On the other hand, the concrete action steps from EP provide a repeatable execution of vulnerability exploits. The DSL constructs include (1) the tool or the action name needed to be executed, (2) the agent's IP and credentials from which the action is to be executed, and (3) the specific vulnerability parameters and the target address on which the attack is to be executed.

5.1.3. Post exploitation

When a vulnerability is exploited, the post-exploitation module performs different tasks like credentials and memory dumps, backdoor injection, pivoting and lateral movement, and so forth. The post-exploitation steps are predefined; however, because the cyber kill chain does not incorporate post-exploitation steps, concepts from MITRE Attack[11] are utilized. For a Windows-based environment, most of the post-exploitations are performed through predefined Mimikatz commands with standard Meterpreter modules[12] and Powershell scripts[13]. For a Linux-based environment, a set of bash scripts[14] are used in an automated manner for post-exploitation.

5.1.4. Logger

The logger logs all the different agents' activities with respect to time, the commands used, and their results in textual format. The logs are used to verify different attack agent success parameters in scanning, exploiting, and post-exploitation of the vulnerabilities in the cybersecurity exercise environment.

5.2. Defense agent

The defense agent is a portable executable that can be deployed in a Windows-based machine. The defense agent's *EP* is generated based on the DSL instance, that is, on which system it is needed to be deployed on and what kind of actions it needs to take, as presented in the Listing 2.

The defense agent has multiple components, such as knowledge base, monitoring, analysis engine, event collector, and event responder. The knowledge base of the defender can be configured to adjust the agent behavior based on the scenario requirements, the details of which are given below and presented in Fig. 6. The defense agent is deployed in a *Windows*-based environment. It uses a custom monitor and analysis engine that collects security events from Windows event logs and that act as sensors to collect information about the environment. When an event is detected, whose information is present in the defense agent knowledge base, a trigger will change its world state. This results in the selection and the execution of different responses against the attacks using the information present in the knowledge base. The execution of this step is mapped to different *CKC* phases and is conceptually represented in Fig. 2.

[10] Netdiscover ref: https://manpages.debian.org/unstable/netdiscover/netdiscover.8.en.html (Accessed on 01/23/2021)

[11] MITRE Attack ref: https://attack.mitre.org/ (Accessed on 01/25/2021)

[12] Mimikatz ref: https://www.offensive-security.com/metasploit-unleashed/mimikatz/ (Accessed on 01/25/2021)

[13] PowerSploit ref: https://github.com/PowerShellMafia/PowerSploit (Accessed on 01/25/2021

[14] Linux post exploitation ref: https://github.com/mubix/post-exploitation/wiki/Linux-Post-Exploitation-Command-List (Accessed on 01/25/2021)

```
#Commands used for detecting file manipulation
type <filename.txt>
more <filename.txt>
#Commands used for detecting user manipulation
whoami /all
net user
net user <user> <pass> /add
net localgroup administrators <user> /add
#Commands used for detecting host information gathering
```

Listing 2. Concrete syntax for defender behavior emulation.

```
w^h^oami
w^h^o"ami
w^h^o"am""""""""""i
```

Listing 3. Process command line information.

```
[{"ActiveScan": {
   "AgentIP": "10.10.4.96",
   "AgentUserID": "root",
   "AgentUserPassword": "toor",
   "Argument": "SV",
   "Target": "10.10.1.1/24"},
  "MetaSploit": {
   "AgentIP": "10.10.4.96",
   "AgentUserID": "root",
   "AgentUserPassword": "toor",
   "Argument": "FTPexploit",
   "Target": "10.10.1.4"},
  "MetaSploit": {
   "AgentIP": "10.10.4.96",
   "AgentUserID": "root",
   "AgentUserPassword": "toor",
   "Argument": "FTPexploit",
   "Target": "10.10.1.5"},
  "MetaSploit": {
   "AgentIP": "10.10.4.96",
   "AgentUserID": "root",
   "AgentUserPassword": "toor",
   "Argument": "Vulnserver",
   "Target": "10.10.1.6"}}]
```

Listing 4. Defender reaction to the detected event.

5.2.1. Knowledge base

The knowledge base for the defense agent is a simple CSV file that contains the list of attacker actions and defender reactions. This approach of segregating the knowledge of the defense agent from the program provides the flexibility to use different levels of knowledge against the different skill set levels of attackers. For example, in a cybersecurity exercise for novice and expert hackers, the knowledge base can be adjusted to create a balanced environment (Mirkovic et al., 2015). We analyze some examples of attacker's actions and defender's reactions below.

5.2.2. Attacker actions

We can consider the example of *Pass the hash attack* on a remote Windows-based system. The attack will generate a 4688 Windows security event log that contains the following command line information:

This event and command line information can be inserted into the knowledge base to give the capability to the defender to detect such an attack signature. If the attacker is skilled enough, then the attacker can use various payload obfuscation techniques to bypass the defender's detection.

5.2.3. Defender reaction

The defender's reaction can be variable based on the scenario requirements, the knowledge for preventing the above attack can be added in the CSV file.

The defense agent can disable the service being exploited to prevent the known attack; however, it will not be able to prevent attacks that it has no information about. To address this, the defense agent has a monitoring and an analysis engine to detect and respond to new attack patterns. Most of the attacks result in events that have similar patterns as the defined attack action; for exam-

```
#Defining necessary term for the model
pyDatalog.create_terms('Goal','SubGoal','Condition','Action',
'CheckGoals','FullFilled','ActionName','ActionTarget','Commands')
#Defining root goal with sub goals of attack agent with an AND relation
+Goal('Exploit System','Reconnaissance')
+Goal('Exploit System','Exploitation')
#Defining sub goal of attack agent with an OR relation
+Goal('Exploitation','Service' or 'Configuration')
#Defining attack actions
+Action('Default', '172.168.2.1')
+Action('ping', '172.168.2.1')
+Action('Nmap', '172.168.2.1')
#Defining fulfillment requirements for attack agent
+FullFilled('Reconnaissance','ping')
+FullFilled('Reconnaissance','Nmap')
+FullFilled('Reconnaissance','WirShark')
+FullFilled('Reconnaissance','NetCat')
+FullFilled('Reconnaissance', 'Default')
#Defining conditions for attack agent
##Checking action target information is provided
FullFilled('Reconnaissance', SubGoal) & Action(SubGoal,ActionTarget) or
↪  print ('NotFullFilled')
##Condition to check is target accessible or not
+Condition('Reconnaissance', 'TargetAccess', '-i 4' )
##Validating target access
Condition(Goal, 'TargetAccess', Commands) & FullFilled('Reconnaissance',
↪  SubGoal) & Action(SubGoal,ActionTarget) or print ('NotFullFilled')
##Condition to check is network interface enabled
+Condition('Reconnaissance', 'NetworkInterface', 'netstat -i' )
##Validating network interface is enabled
Condition(Goal, 'NetworkInterface', Commands) &
↪  FullFilled('Reconnaissance', SubGoal) & Action(SubGoal,ActionTarget)
↪  or print ('NotFullFilled')
##Condition to check is arp-scan is present
+Condition('Reconnaissance', 'arp-scan-check', 'man arp-scan' )
##Validating network arp-scan is present
Condition(Goal, 'arp-scan-check', Commands) & FullFilled('Reconnaissance',
↪  SubGoal) & Action(SubGoal,ActionTarget) or print ('NotFullFilled')
##Checking action name information is provided
FullFilled('Reconnaissance', SubGoal) & Action(SubGoal,ActionTarget) or
↪  Action('Default',ActionTarget)
##Defining the attacker action to execute
+Condition('Reconnaissance', 'Active', '-sS -sV' )
+Condition('Reconnaissance', 'Passive', 'arp-scan -interface=eth0
↪  -localnet' )
+Condition('Reconnaissance', 'Default', 'nc -zv ' )
```

Listing 5. Sample commands used for detecting file manipulation, user manipulation, and system information gathering.

ple, opening a port from different exploits will have similar signature. The defense agent can utilize such information to prevent the attack.

5.2.4. Monitoring and analysis engine

The monitoring and analysis engine provides the defense agent the capability to acquire and apply new knowledge. We can consider a case where the attacker was able to bypass the detection mechanism of the defender; then, the attacker will try to achieve its goals and objectives. For instance, an attacker can try to tamper with the content of the file, which was specified in the knowledge base to be monitored. This action will also generate an event log that can be traced back to the original process using our exploit chain detection algorithm (Yamin et al., 2019). Using this information, the knowledge base of the defense agent can be updated automatically to kill the exploited process or close the vulnerable port using standard system commands. Some of the command set that is used for file manipulation, user manipulation, and system information gathering is presented in Listing 5.

5.2.5. Event collector

The event collector collects *Windows* security event logs. The event logs contain a lot of information that can be used in event correlation and for a process analysis. The event collectors parse the event logs, remove irrelevant information, and forward them to the monitoring and analysis engine for further processing. The

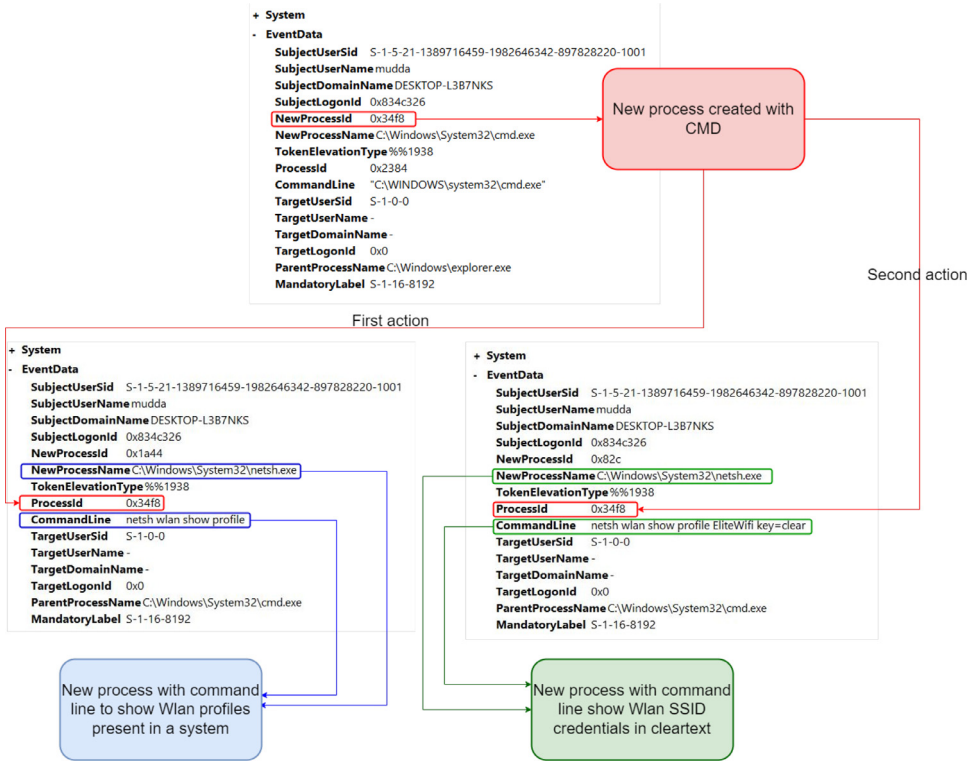

Fig. 7. Sample event collector.

event collector can be configured to collect the security logs from the active directory to perform network-centric cyber defense. However, currently, it is only working on Windows-based host systems.

Figure 7 represents sample attacker actions for the event collector in the defense agent in which an attacker is trying to retrieve clear text WLAN credentials. One attacker action is to open a CMD shell on a compromised system; then, the attacker can fetch the WLAN profiles present on the system through a CMD command. The WLAN profiles contain information about the WIFI networks with which the system is or was connected. The attacker then fetches the clear text credentials of a WIFI network SSID through another command. The attack scenario involves the total execution of three commands. In Fig. 7, it can be seen that the process ID for initiating the CMD console is 0x34f8, which then executed two child processes with the process IDs 0x1a44 and 0x82c.

5.2.6. Event responder

The event responder is running on the host system with system-level privileges. It merely takes input from the monitoring and analysis engine and performs relevant operating system security tasks. These tasks involve executing *Windows command line* and *Powershell* for managing and implementing security setting changes on the system.

All these processes in Fig. 7 create events in the Windows security logs and can be monitored by the monitoring and analysis engine. If the command entered in the CMD console is being monitored and detected, then the parent process ID 0x34f8 will be used to kill the process. If the action of the attacker is not detected and a secure resource is retrieved, here being the credentials of the WLAN SSID, then the process is traced back, and the command parameters used for leading to the information retrieval are saved in the knowledge base to prevent future exploitation.

An attacker can bypass the defender command and event monitoring capability using different command-line obfuscation techniques[15]. The techniques use special characters and encoding schemes to evade any pattern matching algorithms; an example of such a technique is presented in Listing 6. In this example, the command *whoami* is executed in a CMD shell of a Windows 10 machine using various obfuscation techniques. Although machine learningbased techniques are developed to detect such obfuscated commands (Hendler et al., 2018; Yamin and Katt, 2018a), we did not integrate such a solution in the defense agent yet.

6. Experimentation

6.1. Experimental setup of the cyber range

The experimental environment setup was created using our cybersecurity exercises scenario modeling language (Yamin and Katt, 2019b). The experimental setup was used to conduct three cybersecurity exercises, in which one was against an attack agent and two were against defense agents. A total of 101 people, ranging in age from 20 to 25 and from Norway, participated in the exercises; quantitative methods were used to evaluate the agents' performance.

Experiment 1: The attack agents were used in a digital forensic and incident response cybersecurity exercise at the Norwegian University of Science and Technology [16], in which 84 people participated in 17 groups on a multi-subnet exercise network environment of 408 machines. Each group was provided with individual networks compromised by human attackers and attack agents. Each network contained 11 Windows- and Linux-based machines, and 2 out of 17 networks were compromised by the attack agent.

[15] DOS command obfuscation ref: https://i.blackhat.com/briefings/asia/2018/asia-18-bohannon-invoke_dosfuscation_techniques_for_fin_style_dos_level_cmd_obfuscation-wp.pdf (Accessed on 01/28/2021)

[16] NTNU course ref: https://www.ntnu.edu/studies/courses/IMT3004 (Accessed on 01/28/2021)

```
#Defining goals and sub goals of defense agent
+Goal('Prevent Attacks','Detect')
+Goal('Prevent Attacks','Deny')
#Defining defense actions
+Action('Deny', 'shell.exe')
+Action('Deny', 'port 8080')
#Defining fulfillment requirements for defense agent
+FullFilled('Detect','shell.exe')
+FullFilled('Detect','rootkit.exe')
+FullFilled('Detect','chroot.exe')
+FullFilled('Detect','port 8080')
#Check the attack action is detected or not
FullFilled('Detect', SubGoal) & Condition('Deny', SubGoal, Commands) or
↪  print ('NotFullFilled')
#Defining conditions for defense agent to prevent attack action
+Condition('Deny', 'shell.exe', 'taskkill /IM "shell.exe" /F' )
+Condition('Deny', 'Default', 'npx kill-port 8080' )
```

Listing 6. Sample techniques for defense agent detection bypass.

The participants did not have any knowledge about the attacker and how they exploited the machine.

Experiment 2: The defense agents were used in two cybersecurity exercises, which were conducted at the Norwegian Cyber Range[17]. The first exercise was a 48-hour long qualification round for the Norwegian national team for the European Cyber Security Challenge[18],[19] in which 17 people participated in 5 groups on a multi-subnet exercise network environment of 75 machines. The second exercise was a 2-week-long exercise conducted during the Ethical Hacking course taught at the Norwegian University of Science and Technology, in which 84 people participated in 17 groups on a multi-subnet exercise network environment of 408 machines. Both exercises were focused on a penetration testing scenario of a small organization.

The organization infrastructure has a multi-subnet network containing 11 Windows and Linux-based machines. The participants had access to the public network through 5 dedicated Kali machines. For the attack agents, two Kali machines were used for launching attacks and creating forensic traces. In the organization's infrastructure, 2 out of those 11 machines had the same vulnerabilities, but a single machine was running the proposed defense agent. Each group was assigned a segregated replica of the organization network and tasked with doing a pen-test. The scenario's ultimate goal was to tamper with the content of a file in the scenario machine running the defense agent, and the participants were incentivized with extra points to achieve the goal. However, they did not know the presence of the defense agents. A schematic diagram representing the experimental infrastructure is presented in the Fig. 8.

6.2. Test cases

6.2.1. Attack agent

We defined four test cases to evaluate the attack agent performance. These test cases were selected based on the information we gathered from cybersecurity exercises (Yamin et al., 2020). These include scanning the network, performing exploitation, post-exploitation, and launching attacks that were not successful. Details of the test cases are as follows:

1. Perform network scan on all the machines present in the exercise network.
2. Exploit $\boldsymbol{n}$ of the machines present in the exercise network.
3. Perform post-exploitation on $\boldsymbol{k}$ of the machines present in the exercise network.
4. Launch unsuccessful attacks on $\boldsymbol{m}$ of the machine present in the exercise network.

In Listing 7, a snippet of the test cases execution is provided. The agent's goal according to *EP* was to exploit the system. The agent with the IP address of 10.10.4.96 first performed a full network scan using Nmap on subnet 10.10.1.1/24 to emulate a realistic adversary. Then, the second action was to launch a successful FTP exploit on 10.10.1.4; the FTP exploit was designed as Metasploit resource script, so it performed the post-exploitation steps automatically. After that, the agent launched an unsuccessful attack on 10.10.1.5 using the same exploit and a successful attack on 10.10.1.6 using a different exploit.

6.2.2. Defense agent

In case of the defense agent, we used three cases to evaluate their performance indicators:

1. Number of machines exploited not running the defense agent.
2. Number of machines exploited running the defense agent.
3. Files that are tampered with and that were monitored by the defense agent.

We used the knowledge base similar to Listing 4 and 5 in the case study for test case execution. The defender's goal according to *EP* was to detect and deny the attacker using its knowledge base. In the knowledge base, different attacker actions such as information gathering and user and file manipulation were presented, and the defender's actions against those activities were given.

6.3. Evaluation

We employed both quantitative and qualitative evaluation metrics to evaluate the agents. The quantitative metrics include (1) the formal analysis of the agents properties, and (2) the efficiency evaluation of the developed artifacts in which the performance of humans was compared with the proposed agent in similar task with respect to time and resources; this is discussed in

[17] Norwegian Cyber Range ref: https://www.ntnu.no/ncr (Accessed on 01/28/2021)
[18] Norwegian Cyber Security Challenge ref: https://www.ntnu.no/ncsc (Accessed on 01/28/2021)
[19] European Cyber Security Challenge ref: https://europeancybersecuritychallenge.eu/ (Accessed on 01/28/2021)

```
#Establishing links between goals, sub goals
CheckGoals(Goal,SubGoal) <= CheckGoals(SubGoal,Goal)
FullFilled(Goal,SubGoal) <= CheckGoals(Goal,SubGoal)
#Establishing links between sub goals and conditions
CheckGoals(SubGoal,Condition) <= CheckGoals(SubGoal,Condition)
FullFilled(SubGoal,Condition) <= CheckGoals(SubGoal,Condition)
#Establishing links between conditions and action
CheckGoals(Condition, Action) <= CheckGoals(Action,Condition)
FullFilled(Condition, Action) <= CheckGoals(Action,Condition)
#Check a goal can be full filled to perform specific action
FullFilled(Goal,SubGoal) <= CheckGoals(Goal,Action) &
↪  FullFilled(SubGoal,Action) & (Goal != SubGoal)
#Sample analysis condition for attack agent EP decision
Condition(Goal, 'Actvie', Commands) & FullFilled('Reconnaissance',
↪  SubGoal) & Action(SubGoal,ActionTarget)
#Sample analysis condition for defense agent EP decision
FullFilled('Detect', SubGoal) & Condition('Deny', SubGoal, Commands)
```

Listing 7. Concrete syntax for attacker behavior emulation.

Sections 6.3.2 and 6.3.3. The qualitative metrics used to measure the realism through a survey conducted on the participants who took part in the experimental scenario. The questions we asked are given in Appendix B, and its evaluation details are presented in Appendix C.

6.3.1. Agent decision modeling and verification

The EP model presented in Section 4.1 supports analyzing different test cases before their actual execution by the agents. This analysis helps us to verify different model properties like

- How high-level goals can be translated into to low-level actions
- Can the agent fulfill a given goal?
- What information is missing to achieve a goal?

This enables us to fine-tune agent decisions based upon model analysis for their precise execution. Listings 8 and 9 presents the implementation and logical verification conditions of the EP model for attack and defense decisions in PyDatalog[20]. While Listing 10 presents a sample analysis of the presented models. PyDatalog is a python-based implementation of Datalog and is used due to its interoperability with the rest of the technology stack used in this research for artifact development.

While translating high-level goals to low-level actions and tasks is a complex and challenging task, our model can perform it based upon the given conditions. Furthermore, the agents' different decisions were verified, highlighting the decisions that can result in goals not fulfilled or impractical. This allowed us to plan agent decisions based upon the scenario requirements. Like in some scenarios, the agents need to make wrong decisions against human adversaries to maintain realism. Similarly, in some scenarios, the agents were required to execute the actions as fast as possible, like performing dry runs on exercise infrastructure, so their decision model can be tuned to avoid unfulfilled and impractical decisions to save time.

6.3.2. Attack agent results

The Task performed by humans: Human teams of attackers were given the task to perform penetration testing on the segregated exercise networks presented in Fig. 8. They had to discover, exploit, analyze, and report the identified vulnerabilities in a penetration testing report. The penetration testing report was used for their evaluation and comparison with the attack agent's performance. ***The Task performed by agents:*** The attack agent was tasked with performing penetration testing on the similar segregated exercise network presented in Fig. 8. First, the attack agent performed a full network scan of the network to emulate an adversary's scanning. Then, the attack agent created forensic evidence by launching attacks and performing post-exploitation. Out of the 11 machines, the attack agent was tasked to compromise 4 machines, perform post-exploitation, launch failed attacks on 3 machines, and launch no attack on 4 machines. The attack agent was programmed to emulate a human adversary, so it created successful and unsuccessful attack traces for forensic investigators. ***Comparison of human and attack agent performance:*** The humans and attack agents were given the same task, but the humans participated in teams of five, and most teams compromised a minimum of four machines, while one team compromised eight machines in the exercise network. The human teams took around 50 hours to complete the assigned task, while the attack agent were able to emulate the human performance in approximately 10 minutes. ***Verification of performance:*** Human and attack agent performance were measured in the same way. The human participants in the digital forensic and incident response exercise were tasked with performing the forensic analysis of the compromised network by the human teams and the attack agents, and to present their findings in a digital forensic and incident response report. The report was used to assess the performance of the attack agent in the cybersecurity exercise. The summary of the findings are presented in Table 1.

Table 1
Results of the cybersecurity exercise against the attack agent .

Exercise 1			
Group task	Compromised machines identified	Post-exploitation identified	Attack attempts identified
Forensic analysis of machine compromised by humans	3	3	3
Forensic analysis of machine compromised by attack agent	4	4	3

[20] Definition in pydatalog ref: https://sites.google.com/site/pydatalog/home (Accessed on 09/03/2021)

```
#Defining necessary term for the model
pyDatalog.create_terms('Goal','SubGoal','Condition','Action',
'CheckGoals','FullFilled','ActionName','ActionTarget','Commands')
#Defining root goal with sub goals of attack agent with an AND relation
+Goal('Exploit System','Reconnaissance')
+Goal('Exploit System','Exploitation')
#Defining sub goal of attack agent with an OR relation
+Goal('Exploitation','Service' or 'Configuration')
#Defining attack actions
+Action('Default', '172.168.2.1')
+Action('ping', '172.168.2.1')
+Action('Nmap', '172.168.2.1')
#Defining fulfillment requirements for attack agent
+FullFilled('Reconnaissance','ping')
+FullFilled('Reconnaissance','Nmap')
+FullFilled('Reconnaissance','WirShark')
+FullFilled('Reconnaissance','NetCat')
+FullFilled('Reconnaissance', 'Default')
#Defining conditions for attack agent
##Checking action target information is provided
FullFilled('Reconnaissance', SubGoal) & Action(SubGoal,ActionTarget) or
↪  print ('NotFullFilled')
##Condition to check is target accessible or not
+Condition('Reconnaissance', 'TargetAccess', '-i 4' )
##Validating target access
Condition(Goal, 'TargetAccess', Commands) & FullFilled('Reconnaissance',
↪  SubGoal) & Action(SubGoal,ActionTarget) or print ('NotFullFilled')
##Condition to check is network interface enabled
+Condition('Reconnaissance', 'NetworkInterface', 'netstat -i' )
##Validating network interface is enabled
Condition(Goal, 'NetworkInterface', Commands) &
↪  FullFilled('Reconnaissance', SubGoal) & Action(SubGoal,ActionTarget)
↪  or print ('NotFullFilled')
##Condition to check is arp-scan is present
+Condition('Reconnaissance', 'arp-scan-check', 'man arp-scan' )
##Validating network arp-scan is present
Condition(Goal, 'arp-scan-check', Commands) & FullFilled('Reconnaissance',
↪  SubGoal) & Action(SubGoal,ActionTarget) or print ('NotFullFilled')
##Checking action name information is provided
FullFilled('Reconnaissance', SubGoal) & Action(SubGoal,ActionTarget) or
↪  Action('Default',ActionTarget)
##Defining the attacker action to execute
+Condition('Reconnaissance', 'Active', '-sS -sV' )
+Condition('Reconnaissance', 'Passive', 'arp-scan -interface=eth0
↪  -localnet' )
+Condition('Reconnaissance', 'Default', 'nc -zv ' )
```

Listing 8. EP Decision Model implementation for attack.

Summary of the results: The human participants detected most of the successful attacks, post-exploitation, and unsuccessful attack attempts. The attack traces were identical to the attack traces generated by an human attacker, and the participants were not able to identify that they were generated by an attack agent. This indicates that the attack agent was providing the necessary realism and removing the need for a red team member for launching attacks, thus increasing the efficiency by automating different CKC phases like *Reconnaissance*, *Delivery*, *Exploitation*, *Installation*, *Command and Control*, and *Actions on Objectives*. It was identified that the proposed agent was working as expected and suitable in a cybersecurity exercise for creating digital forensic traces. We didn't include the factor of time in this work, which can be used by an experienced forensics analysts for distinction between humans and attack agents generated forensic traces. However, over the passage of time we are incorporating the concepts of technical injects that are executed according to a scenario timeline to make the generation of forensic traces as realistic as possible.

6.3.3. Defense agent results

Tasks performed by humans: Teams of humans were tasked to compromise a vulnerable machine that was or was not running the defense agent. In the scenario, the human teams were incentivized with additional points to exploit a particular machine known as the *CEO Machine*, which had the same vulnerabilities present in another machine, i.e., *Machine9*, but was running the

```
#Defining goals and sub goals of defense agent
+Goal('Prevent Attacks','Detect')
+Goal('Prevent Attacks','Deny')
#Defining defense actions
+Action('Deny', 'shell.exe')
+Action('Deny', 'port 8080')
#Defining fulfillment requirements for defense agent
+FullFilled('Detect','shell.exe')
+FullFilled('Detect','rootkit.exe')
+FullFilled('Detect','chroot.exe')
+FullFilled('Detect','port 8080')
#Check the attack action is detected or not
FullFilled('Detect', SubGoal) & Condition('Deny', SubGoal, Commands) or
↪  print ('NotFullFilled')
#Defining conditions for defense agent to prevent attack action
+Condition('Deny', 'shell.exe', 'taskkill /IM "shell.exe" /F' )
+Condition('Deny', 'Default', 'npx kill-port 8080' )
```

Listing 9. EP Decision Model implementation for defense.

```
#Establishing links between goals, sub goals
CheckGoals(Goal,SubGoal) <= CheckGoals(SubGoal,Goal)
FullFilled(Goal,SubGoal) <= CheckGoals(Goal,SubGoal)
#Establishing links between sub goals and conditions
CheckGoals(SubGoal,Condition) <= CheckGoals(SubGoal,Condition)
FullFilled(SubGoal,Condition) <= CheckGoals(SubGoal,Condition)
#Establishing links between conditions and action
CheckGoals(Condition, Action) <= CheckGoals(Action,Condition)
FullFilled(Condition, Action) <= CheckGoals(Action,Condition)
#Check a goal can be full filled to perform specific action
FullFilled(Goal,SubGoal) <= CheckGoals(Goal,Action) &
↪  FullFilled(SubGoal,Action) & (Goal != SubGoal)
#Sample analysis condition for attack agent EP decision
Condition(Goal, 'Actvie', Commands) & FullFilled('Reconnaissance',
↪  SubGoal) & Action(SubGoal,ActionTarget)
#Sample analysis condition for defense agent EP decision
FullFilled('Detect', SubGoal) & Condition('Deny', SubGoal, Commands)
```

Listing 10. EP Decision Model analysis.

defense agent. ***Tasks performed by agents:*** The defense agent was tasked to block or prevent the attacks launched by the human attackers. The defense agent had a knowledge base to prevent particular attacks from specific vulnerabilities also present in *Machine9*. Additionally, the defense agent was also monitoring a local file to prevent access to the attacker in case the attacker exploited a vulnerability not present in the defense agent's knowledge base. ***Comparison of human and attack agent performance:*** In the exercise environment, there were no machines actively defended by human adversaries, so comparing the performance regarding efficiency of the human and defense agents is a bit difficult. However, the machines that were defended by the proposed agents were difficult to exploit compared with the undefended machines. According to IBM, the average detection time for a data breach is 206 days [21]; we are not arguing that our proposed solution will drastically improve this, but we are arguing that having an active agent running in a system will restrict the attacker's actions and improve system security. ***Verification of performance:*** In the second exercise, three out of five groups compromised the vulnerable machine not running the defense agent. In comparison, one group compromised the machine running the defense agent using a vulnerability that was not in the defense agent's knowledge base but could not tamper with the file because of the defense agent's actions. Similarly, in the third exercise, 8 out of 17 groups compromised the vulnerable machine, and 2 groups were able to compromise the machine running the defense agent using a vulnerability not present in the defense agent's knowledge base but were not able to tamper with the content of the file. The results of the experiments are presented in Table 2.

Table 2
Results of the cybersecurity exercise against the defense agent.

Exercise 2			
Number of Groups	Groups Exploited Vulnerable Machine	Groups Exploited Vulnerable Machine Running Defense Agent	Groups Tampered with the File
5	3	1	0
Exercise 3			
17	8	2	0

[21] IBM data breach report ref: https://www.ibm.com/security/data-breach (Accessed on 06/16/2021)

Summary of the results: The results indicate that the defense agent created the necessary friction and added realism by preventing attacks present in its knowledge base during an operational cybersecurity exercise. The defense agent removed the need for an active blue team member during the exercise, thus increasing the efficiency by automating certain tasks in CKC like *Detect* and *Deny* for the defender. The inclusion of the defense agent made the exercise environment more dynamic and challenging because some groups were not able to compromise the machines running the defense agents. We want to highlight that a smart attacker who can exploit a vulnerable machine remotely in some rare cases can analyze the functionality of the defense agent and use it for local exploits like elevating privileges; however, as it is an exercise and training environment, so such risks are acceptable.

7. Discussion and conclusion

In this work, we investigated the attack and defense agents' usage in cyber ranges for improving the realism and efficiency of cybersecurity exercise execution. We identified that such agents could provide the necessary level of friction during an exercise. For example, in a red team exercise, a defense agent will try to prevent attackers from achieving their objectives. On the other hand, in a blue team exercise, an attack agent will conduct various attacks and create forensics traces, such that the need for a human red team is reduced. This makes cybersecurity exercise execution more realistic and efficient.

We proposed EP models for specifying the agents' decision making. An EP model contains three levels of decisions: *high, medium,* and *low*. These decisions were translated using a DSL instance into goals, actions, and commands. We presented the workflows of the attack and defense agents to showcase how they made their decisions during the execution of a cybersecurity exercise. We employed the proposed agent-based system in cybersecurity exercises and presented their performance results in the form of a case study.

For the attack agent, we consider its performance satisfactory when applied in a semi-autonomous manner. The attack agents create realistic forensic traces during a cybersecurity exercise, which is verified by the human participants. On the other hand, our developed cyber defense agents currently have the six capabilities highlighted by Kott (Kott et al., 2018b). However, we do not consider these agents suitable for deployment in the actual production environment for active cyber defense because they were tested in a controlled environment of cyber ranges. Yet, they can be considered a first step to achieve autonomous cyber defense. One of the limitation of the proposed defense agents is in the way they respond to attacks. If an agent detects a new attack on a specific port, it will update the knowledge base with some information related to the attack's signature. Then, it will close the port or service for the next attack with a similar signature without analyzing the impact of its action, which is not suitable for real-world systems. We will address this issue in our future research, and we are planning to develop state machines that can help the defense agent taking optimal actions to deal with attackers.

Declaration of Competing Interest

None.

CRediT authorship contribution statement

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

Appendix A. Experimental Setup

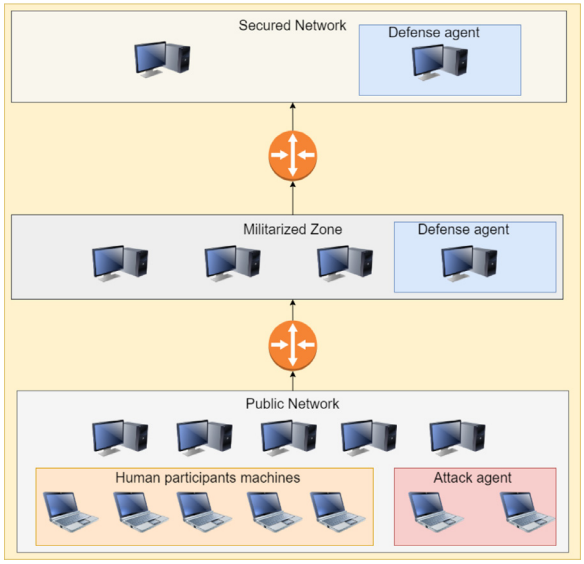

Fig. 8. Experimental setup of the cyber range.

Appendix B. Survey Question

1. Did you find the scenario realistic?
2. Did you find the scenario difficulty hard, medium, or easy?
3. How many machines did you exploited?
4. Did you find similarities between Machine9 and CEO machines?
5. Did you identify any of your attacks get blocked?
6. If yes, did you exploited both or only one and why?
7. What can be improved in the scenario?

Appendix C. Qualitative feedback from the exercise participants

The qualitative feedback consists of survey responses from the 3 teams, which represent 15 students. The survey was conducted in a semi-informal way. The participants were asked to answer a list of pre-defined open-ended questions using digital communication apps, hence following COVID-19 restrictions. Fallow-up questions were asked if the answers needed further explanation. The survey was conducted in a relaxed and friendly environment, and the participants were given sufficient time to reflect on their experience and properly answer the question to avoid any biases. No personally indefinable information was collected during the survey to avoid any GDPR *(General Data Protection Regulation)*-related issues. During the survey, when we asked the participants if they found the scenario realistic, one survey participant stated the following:

Scenarios were pretty realistic for the hacking phase

This sentiment was shared by the majority of the survey participants. The exercise was conducted in two phases *Ethical Hacking* and *Incident response & Forensics*. The participants were very impressed by the complexity and dynamism of the scenario in the *Ethical Hacking* phase; this can be attributed to the presence of a dynamic defense agent in the exercise environment. However, for the *Incident response & Forensics*, they were not that impressed because the environment was static. The participants expected continuing attacks during this phase to make it more dynamic. The feedback of the participants was noted for implementing active attack execution in the *Incident response & Forensics* phase in the future. When we asked about the difficulty of the scenario, one participant stated the following:

I think it is good that the scenario is large and consist of both easy machines and more difficult ones. This allows weaker students to be able to get points and provides a challenge for stronger students with much experience. In my opinion, the project is good from a grading perspective

This sentiment was also shared by most of the survey respondents. The participants indicated that they found the machines to exploit having a variety of difficulty levels *easy, medium,* and *hard*, which allowed the participants with ranging skill sets to practice their skills. This indicates that the presence of the developed agents in the scenario provided balance in the lab, which made some machines difficult to exploit because of them having similar vulnerabilities to other easily exploitable machines. When we asked about the number of machines exploited by their teams, two teams stated that they exploited four machines, while one team exploited nine machine. Continuing from this, we asked a specific question about the machine that was not running the agent *Machine9* and the machine that was running the defense agent *CEO machine*. Two teams were not able to exploit both machines, while one team was able to exploit it *Machine9*. When we asked why they were not able to exploit, it they responded with the following:

No exactly each planned attack went through except for one where we were trying to do an smb exploitation but we couldn't figure out and came to the conclusion that it was rabbit hole and moved on

The rest of the questions were asked to improve the quality of the exercise scenario and are not relevant to this study. From the qualitative feedback, it can be concluded that the exercise scenario that incorporated our agents are quite realistic and offer the opportunity to exercise participants to practice their skills against realistic computational adversaries.

References

Braghin, C., Cimato, S., Damiani, E., Frati, F., Mauri, L., Riccobene, E., 2019. A model driven approach for cyber security scenarios deployment. In: Computer Security. Springer, pp. 107–122.

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

Edgar, T.W., Manz, D.O., 2017. Research methods for cyber security. Syngress, pp. 271–297.

Feily, M., Shahrestani, A., Ramadass, S., 2009. A survey of botnet and botnet detection. In: 2009 Third International Conference on Emerging Security Information, Systems and Technologies. IEEE, pp. 268–273.

Hendler, D., Kels, S., Rubin, A., 2018. Detecting malicious powershell commands using deep neural networks. In: Proceedings of the 2018 on Asia Conference on Computer and Communications Security, pp. 187–197.

Herold, N., Wachs, M., Dorfhuber, M., Rudolf, C., Liebald, S., Carle, G., 2017. Achieving reproducible network environments with insalata. In: IFIP International Conference on Autonomous Infrastructure, Management and Security. Springer, Cham, pp. 30–44.

Hevner, A., Chatterjee, S., 2010. Design science research in information systems. In: Design Research in Information Systems. Springer, pp. 9–22.

Holm, H., Sommestad, T., 2016. Sved: Scanning, vulnerabilities, exploits and detection. In: MILCOM 2016-2016 IEEE Military Communications Conference. IEEE, pp. 976–981.

Hutchins, E.M., Cloppert, M.J., Amin, R.M., et al., 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), 20. Pennsylvania State University.

Kordy, B., Mauw, S., Radomirović, S., Schweitzer, P., 2014. Attack–defense trees. Journal of Logic and Computation 24 (1), 55–87.

Kotenko, I., 2005. Agent-based modeling and simulation of cyber-warfare between malefactors and security agents in internet. 19th European Simulation Multiconference æSimulation in wider Europe.

Kott, A., Théron, P., Dražar, M., Dushku, E., LeBlanc, B., Losiewicz, P., Guarino, A., Mancini, L., Panico, A., Pihelgas, M., et al., 2018. Autonomous intelligent cyber-defense agent (AICA) reference architecture. release 2.0. arXiv preprint arXiv: 1803.10664.

Kott, A., Thomas, R., Dražar, M., Kont, M., Poylisher, A., Blakely, B., Theron, P., Evans, N., Leslie, N., Singh, R., et al., 2018. Toward intelligent autonomous agents for cyber defense: report of the 2017 workshop by the north atlantic treaty organization research group IST-152-RTG. arXiv preprint arXiv: 1804.07646.

Kuechler, B., Vaishnavi, V., 2008. On theory development in design science research: anatomy of a research project. European Journal of Information Systems 17 (5), 489–504.

Lloyd, J.W., 2012. Foundations of logic programming. Springer Science & Business Media, pp. 1–31.

Mirkovic, J., Tabor, A., Woo, S., Pusey, P., 2015. Engaging novices in cybersecurity competitions: A vision and lessons learned at {ACM} tapia 2015. 2015 {USENIX} Summit on Gaming, Games, and Gamification in Security Education (3GSE 15).

Naik, M., 2020. Petablox: Large-Scale Software Analysis and Analytics Using Datalog. Technical Report. Georgia Tech Research Institute Atlanta United States. Report Number: AD1098764

Russo, E., Costa, G., Armando, A., 2020. Building next generation cyber ranges with crack. Computers & Security 95, 101837.

Stoecklin, M.P., 2018. Deeplocker: how ai can power a stealthy new breed of malware. Security Intelligence, August 8.

Theron, P., Kott, A., Dražar, M., Rzadca, K., LeBlanc, B., Pihelgas, M., Mancini, L., De Gaspari, F., 2020. Reference architecture of an autonomous agent for cyber defense of complex military systems. In: Adaptive Autonomous Secure Cyber Systems. Springer, pp. 1–21.

Theron, P., Kott, A., Dražar, M., Rzadca, K., LeBlanc, B., Pihelgas, M., Mancini, L., Panico, A., 2018. Towards an active, autonomous and intelligent cyber defense of military systems: The NATO AICA reference architecture. In: 2018 International Conference on Military Communications and Information Systems (ICMCIS). IEEE, pp. 1–9.

Yamin, M.M., Basel, K., 2018. Ethical problems and legal issues in development and usage autonomous adversaries in cyber domain. In: Proceedings of the EXplainable AI in Law Workshop (XAILA 2018). CEUR Workshop Proceedings, pp. 33–41.

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. In: AAAI Fall Symposium: ALEC, pp. 41–43.

Yamin, M.M., Katt, B., 2019. Cyber security skill set analysis for common curricula development. In: Proceedings of the 14th International Conference on Availability, Reliability and Security, pp. 1–8.

Yamin, M.M., Katt, B., 2019. Modeling attack and defense scenarios for cyber security exercises. In: 5th InterdisciPlinary Cyber Research Conference, p. 7.

Yamin, M.M., Katt, B., 2022. Modeling and executing cyber security exercise scenarios in cyber ranges. Computers & Security 116, 102635.

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., 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.

Yamin, M.M., Ullah, M., Ullah, H., Katt, B., 2021. Weaponized ai for cyber attacks. Journal of Information Security and Applications 57, 102722.

Yuen, J., 2015. Automated cyber red teaming. Technical Report. Defence Science And Technology Organisation Edinburgh (Australia).

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.

Zeidanloo, H.R., Manaf, A.A., 2009. Botnet command and control mechanisms. In: 2009 Second International Conference on Computer and Electrical Engineering, Vol. 1. IEEE, pp. 564–568.

Muhammad Mudassar Yamin is currently doing his Post-doc at the Department of Information and Communication Technology at the Norwegian University of Science and Technology. He is a member of the system security research group, and the focus of his research is system security, penetration testing, security assessment, and 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, and 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, Model driven security Access control, Usage control and privacy protection. He holds multiple cyber security certifications like CISSP, CSSLP.