SoftwareX 19 (2022) 101120

Contents lists available at ScienceDirect

SoftwareX

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

Original software publication

irs-partition: An Intrusion Response System utilizing Deep Q-Networks and system partitions

Valeria Cardellini [a], Emiliano Casalicchio [b], Stefano Iannucci [c], Matteo Lucantonio [b], Sudip Mittal [d], Damodar Panigrahi [d,*], Andrea Silvi [a]

[a] *University of Rome Tor Vergata, Italy*
[b] *Sapienza University of Rome, Italy*
[c] *Roma Tre University, Italy*
[d] *Mississippi State University, United States of America*

ARTICLE INFO

Article history:
Received 16 February 2022
Received in revised form 25 April 2022
Accepted 30 May 2022

Keywords:
Intrusion Response System
Self-protection
Self-adaptation

ABSTRACT

Intrusion Response is a relatively new field of research. Recent approaches for the creation of Intrusion Response Systems (IRSs) use Reinforcement Learning (RL) as a primary technique for the optimal or near-optimal selection of the proper countermeasure to take in order to stop or mitigate an ongoing attack. However, most of them do not consider the fact that systems can change over time or, in other words, that systems exhibit non-stationary behaviors. Furthermore, stateful approaches, such as those based on RL, suffer from the curse of dimensionality, due to the state space growing exponentially with the size of the protected system. In this paper, we introduce and develop an IRS software prototype, named *irs-partition*. It leverages the partitioning of the protected system and Deep Q-Networks to address the curse of dimensionality by supporting a multi-agent formulation. Furthermore, it exploits transfer learning to follow the evolution of non-stationary systems.

© 2022 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

Code metadata

Current code version	V1(tag:irs-partition-v2)
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-22-00049
Code Ocean compute capsule	N/A
Legal code license	Apache License 2.0
Code versioning system used	git
Software code languages, tools, and services used	Java, Python, Shell scripts
Compilation requirements, operating environments & dependencies	openjdk 11.0.13, maven
If available link to developer documentation/manual	https://github.com/dpanigra/irs-partition
Support email for questions	dp1657@msstate.edu

1. Motivation and significance

Intrusion Detection Systems (IDSs) are widely used to detect threats to computer systems. However, they are just one of the two parts of an automatic self-protecting system, as shown in Fig. 1. Indeed, while IDSs are fundamental to identify ongoing threats, they generally offer trivial response capabilities, usually based on a static mapping between the attack that has been identified and a response (e.g., Snort [1]). Unfortunately, such an approach exhibits evident limitations mainly related to scalability [2] and lack of generalizability [3]. For this reason, in the last decade, research on Intrusion Response Systems (IRSs) started to gain traction. The purpose of an IRS is to automatically identify the proper response to an ongoing attack, usually by exploiting additional knowledge of the attacker behavior and of the protected system.

We investigated existing IRS methodologies (e.g., [1,4–14]), and the most recent comprehensive survey on this topic, published in 2017 by Nespoli et al. [15]. We found that, with the exception of [3], upon which this work is based, all of them assume that the behavior of the protected system does not change

[*] Corresponding author.
E-mail address: dp1657@msstate.edu (Damodar Panigrahi).

https://doi.org/10.1016/j.softx.2022.101120
2352-7110/© 2022 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

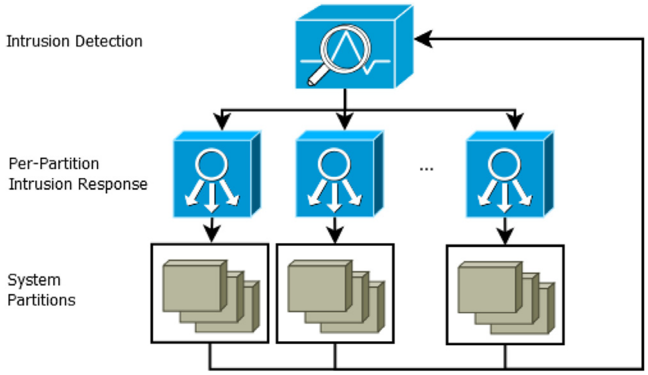


Fig. 1. Role of Intrusion Detection and Intrusion Response in self-protecting systems.

over time or, in other words, that the protected system is *stationary*. Indeed, most IRSs (e.g., [8,12–14]) use either a *rule-based static configuration* or a combination of static attacker and system models (e.g., [16,17]) to formulate a set of responses for the entire system. However, modern systems exhibit a non-stationary behavior, and therefore need the ability to automatically adapt to changes while dynamically predicting a near optimal response to an intrusion.

Moreover, to the best of our knowledge, none of the existing works are based on an openly accessible software prototype, therefore limiting the reproducibility of the experimental results.

For this reason, in this work, we describe as our *main contribution* an open-source licensed software prototype that implements an IRS, named *irs-partition*, which builds upon the methodology introduced in [3]. It uses Deep Q-Networks [18] (DQN), Reinforcement Learning (RL) [19], and transfer learning [20] to cope with the non-stationary behavior of computer systems. To address the curse of dimensionality, its formulation supports the partitioning of the system model, therefore enabling the usage of different local modeling techniques and solvers, e.g., approaches based on Markov Decision Processes, such as, DQN and Dynamic Programming [19], or other types of optimization, such as, Mathematical Programming. To the best of our knowledge, our IRS software implementation is the first to be released with an Apache 2.0 license.

The high-level architecture of the proposed prototype, and how it fits in the intrusion detection (ID) and IR chain, is depicted in Fig. 1. In particular, the defended system is divided into independent subsystems (partitions), and an IR agent is responsible to control each of them. Furthermore, the IR agents receive the attack details from the IDS, which in turn is in charge of collecting and analyzing the data using sensors deployed into the system partitions. The prototype focuses on the IR and assumes an already existing IDS component. The response is generated upon reception of an alert from the IDS using exclusively a model of the system, i.e., without using an attack model. This is a common trend in recent works on IR (e.g., [3,21]) and it allows the IRS to handle zero-day attacks, while providing a less targeted response if compared to an IRS based on the attacker model, when the attack is known.

The rest of the paper is organized as follows: we describe the system model and the design of its software implementation in Section 2. Then, we showcase the functionalities of the developed software with a case study based on the open-source Online Boutique application [22] in Section 3. Finally, we discuss the impact of the software followed by conclusions and future works in Sections 4 and 5, respectively.

Table 1
Main notation used in this paper.

Symbol	Meaning
i	A *component type*
p_i	A *partition* corresponding to the ith component type
i_j	The jth *component* of the ith component type
p_{i_j}	The jth *component* of the ith type of the ith partition
S	The computer system model
V	The set of state variables of system S
v_i	The set of state variables of component type i
$v_{i_{j_T}}$	The state of the jth component of type i at time T
p_{i_T}	The ith partition state at time T
S_T	The state of system S at time T
Σ	The state space
A	The set of actions available to system S
A_i	The set of valid actions for the ith component type
a_i	A valid action ($a_i \in A_i$) for the ith component type
$E(a_i)$	The execution time for action a_i
$C(a_i)$	The cost for taking action a_i
$R(\cdot)$	The reward function
τ	The termination function
τ_i	The termination function for partition i

2. System model and IRS design

We developed and published under the Apache 2.0 license an IRS prototype, named *irs-partition*. Even though the software is flexible enough to support different optimization techniques for different system partitions, at the current stage of development we introduced the support for a single solver, based on DQN. The latter uses a *training environment* to train agents that are defined on a per-partition basis. Each agent works toward the overall system goal of keeping the system *secure* by predicting the near-optimal action for its *partition* using a customizable DQN.

Software dependencies of the application include *Eclipse Deeplearning4J* (DL4J) [23], and *Reinforcement Learning for Java* (RL4J) [24]. Both are Java implementations of deep neural network algorithms and of the RL framework.

2.1. System model

In this section we introduce the system model and its notation. The latter is summarized in Table 1.

A system contains components of different types. Each *component type* can be defined at a different granularity level, as deemed necessary. Examples of component types are hardware devices, virtual appliances, software modules, web servers, application servers, database servers, network switches, load balancers, and container images. We define a *component* as an instance of component type. Furthermore, we define the concept of *partition* as the set of all the components of a given type i, i.e., $p_i = \cup_{j=1}^{m} i_j$, where i_j represents component j of type i, and m is the total number of components of type i. The system S is the set of all the *partitions*, that is, $S = \{p_1, p_2, \ldots, p_n\}$, where n is the total number of partitions. In addition, given any two partitions $p_a, p_b \in S$, they do not share any component, that is, $\forall a.\forall b.a \neq b \rightarrow p_a \cap p_b = \emptyset$. In other words, partitions are disjoint. This restriction, which has been introduced to simplify the development of the prototype, has important implications: on one hand, it eases the design, development and run-time administration of the proposed prototype. On the other hand, it could not fully capture the dynamics of a complex system, if components belonging to different partitions have some interaction. As a consequence, given the current formulation, the near-optimality of the response is guaranteed only if components

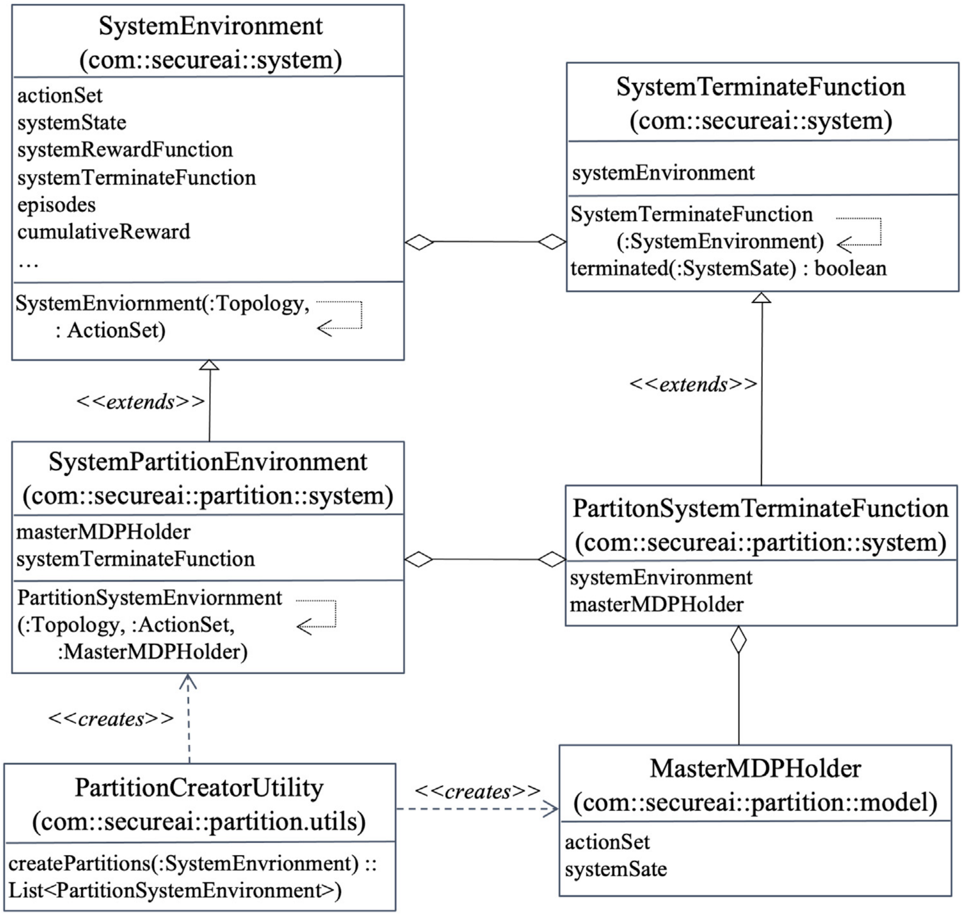


Fig. 2. Class diagram of the main classes of *irs-partition* software.

belonging to different partitions do not have any interaction. This limitation will be addressed in a future release of the software prototype.

2.2. System state

We define a set of boolean *state variables* $V = \cup_{i=1}^{n} v_i$, where $v_i = \{v_1, v_2, \ldots, v_q\}$, and where each variable $v \in v_i$ defines a specific characteristic of component type i and q is the total number of variables used to model the state for such component type. For example, following the case study scenario we will describe in Section 3, the variable *corrupted*$_i$ is applied to all the components of type i, and its instances represent whether or not each component of type i has been compromised. The set of the variable values of all the components of a given partition i at a given discrete time T represents the partition state, that is, $p_{i_T} = \cup_{j=1}^{m} v_{i_{j_T}}$. Similarly, the system state is represented by the set of the states of its component partitions, that is, $S_T = \cup_{i=1}^{n} p_{i_T}$. Finally, $S_T \in \Sigma$, where Σ represents the state space.

2.3. System actions

We define a set of *actions* which, when executed on a given component i_j, change the state of its corresponding partition p_i, and hence the system state. Each component type i of the system has its set of valid actions, i.e., $A_i = \{a_1, a_2, \ldots, a_r\}$, where r is the total number of actions executable on component type i. Furthermore, by design, we have that $\forall j.A_i = A_{i_j}$. Hence, the set of actions available to the entire system is the union of all of the actions defined for each component type, i.e., $A = \cup_{i=1}^{n} A_i$. Furthermore, each action is associated with a pre-condition and a post-condition. The former, $Pre(S_T, a_{i_j})$, where $a_{i_j} \in A_{i_j}$, determines if action a_{i_j} can be executed on component j of partition i when the system is in state S_T. The latter modifies the partition state, taking it from p_{i_T} to $p_{i_{T+1}}$, and thus from S_T to S_{T+1}.

2.4. Reward and termination functions

For each action $a_i \in A_i$, we define its *execution time*, $E(a_i)$, and *cost*, $C(a_i)$, as two criteria of a *reward function*. The latter returns the immediate reward obtained by a reinforcement learning agent upon its execution, and it is defined as:

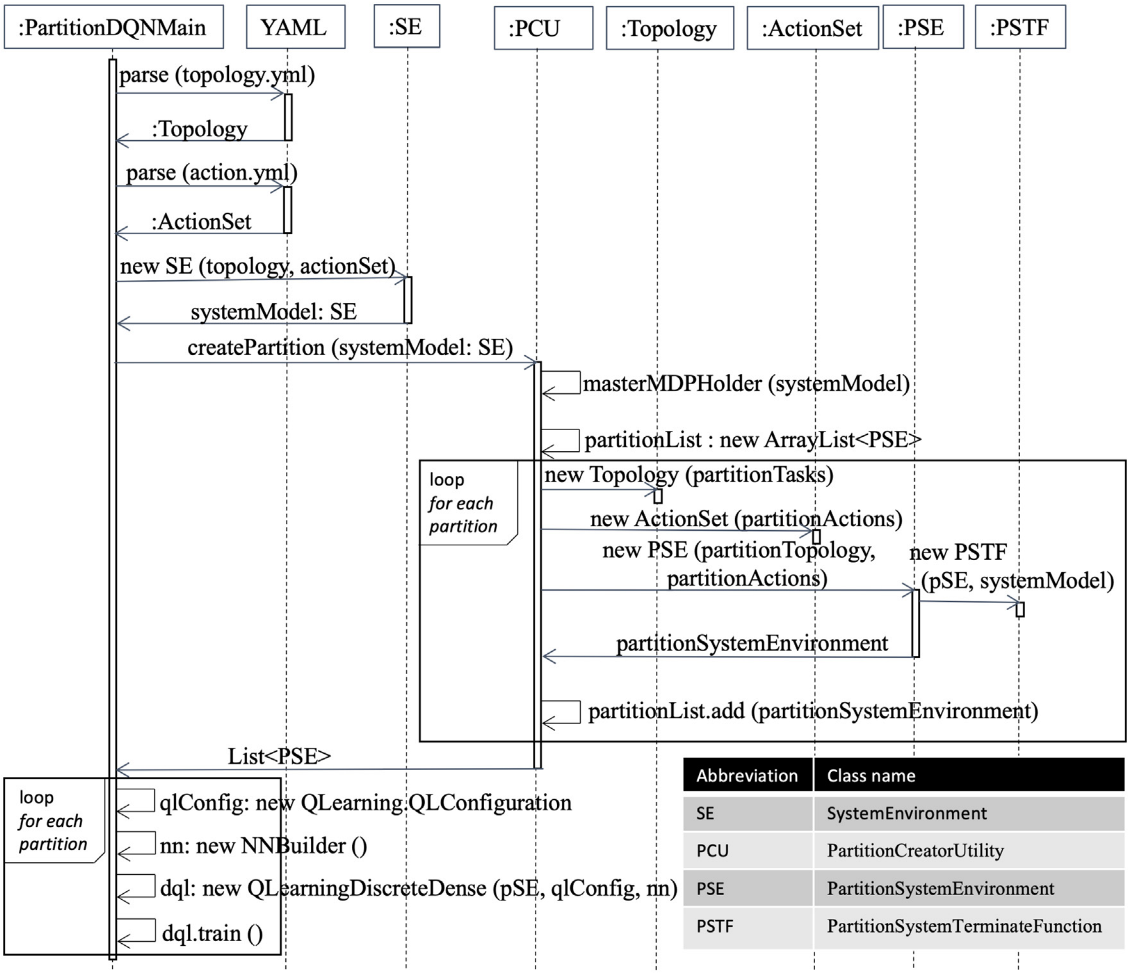
$$R(p_{i_T}, a_i, p_{i_{T+1}}) = \begin{cases} -2, & \text{if } p_{i_T} = p_{i_{T+1}} \\ -w_E \frac{E(a_i)}{E_{max}} - w_C \frac{C(a_i)}{C_{max}}, & \text{otherwise.} \end{cases} \quad (1)$$

where E_{max} and C_{max} are respectively the maximum execution time and the maximum cost; $w_E, w_C \in [0, 1]$ are the corresponding optimization weights. $R(p_{i_T}, a_i, p_{i_{T+1}})$ returns a high penalty score of -2 if an action, a_i, cannot be run because the preconditions are not met. This specific formulation is a technical requirement of the DQN solver implementation of the DL4J library.

Finally, the *termination function* is used to identify the set of states in which the system is considered *secure*. We define a per-partition termination function as $\tau_i : p_{i_T} \rightarrow \{true, false\}$, and a system-level termination function as $\tau = \bigwedge_{i=1}^{n} \tau_i(p_{i_T})$.

2.5. Software design

We implement the system model S, system state variables V, actions A, partition state p_{i_T}, reward function R, termination


:PartitionDQNMain
YAML
:SE
:PCU
:Topology
:ActionSet
:PSE
:PSTF
parse (topology.yml)
:Topology
parse (action.yml)
:ActionSet
new SE (topology, actionSet)
systemModel: SE
createPartition (systemModel: SE)
masterMDPHolder (systemModel)
partitionList : new ArrayList<PSE>
loop for each partition
new Topology (partitionTasks)
new ActionSet (partitionActions)
new PSE (partitionTopology, partitionActions)
new PSTF (pSE, systemModel)
partitionSystemEnvironment
partitionList.add (partitionSystemEnvironment)
List<PSE>
loop for each partition
qlConfig: new QLearning.QLConfiguration
nn: new NNBuilder ()
dql: new QLearningDiscreteDense (pSE, qlConfig, nn)
dql.train ()

Abbreviation	Class name
SE	SystemEnvironment
PCU	PartitionCreatorUtility
PSE	PartitionSystemEnvironment
PSTF	PartitionSystemTerminateFunction

Fig. 3. Sequence diagram to create deep neural nets.

function τ, and partition termination function τ_i respectively, in the SystemEnvironment (SE), SystemState, SystemAction, SystemPartitionEnvironment (PSE), SystemRewardFunction, SystemTerminateFunction, and PartitionSystemTerminateFunction (PSTF) classes. We decompose the system model S into multiple *partitions*, where each partition stores only its own state variables and actions in SystemPartitionEnvironment, which is a subclass of SystemEnvironment. All the partitions are then stored in the List<PartitionSystemEnvironment> list. We use PartitionCreatorUtility (PCU) to decompose the SystemEnvironment into multiple PartitionSystemEnvironment based on component type i, as shown in Fig. 2, which represents the class diagram of the main classes of the software. The references to the full system state variables V, and action set A are stored in MasterMDPHolder, which is a *singleton object* that acts as a central store and provides the state of the system at a discrete time T, S_T, and the set of actions, A, to objects of classes SystemPartitionEnvrionment and PartitionSystemTerminateFunction.

The execution of our software starts with the main function of PartitionDQNMain, where we create the system model (S) in SystemEnvironment from the .yml configuration files, store the system state (S_T) in MasterMDPHolder, decompose S into partitions, store each partition in SystemPartitionEnvironment, and create one DNN for each partition as shown in the sequence diagram of Fig. 3.

We train one agent on each partition p_i. Each agent is responsible for providing the local near-optimal next action, according to the current partition state. Given the formulation of the system model as a set of disjoint partitions, the set of predicted optimal local actions leads to a global optimum. We use DQN with Monte Carlo simulation to train the agents. We utilize QLearningDiscreteDense [24] for DQN with configurable parameters. The simulation begins with an initial system state configured in SystemState by the system administrator. Then, based on the initial state, a set of actions, ActionSet, (at most one for each partition) is executed on the environment, represented by PartitionSystemEnvironment, which returns a set of rewards (from SystemRewardFunction) and the next system state. Such actions are chosen by the agent by either exploiting the acquired knowledge, and therefore trying to maximize the expected discounted reward, or by exploring actions whose outcome, in terms of reward and transition, is still unknown. The latter case occurs with a probability $\epsilon = 0.01$ during the first epoch, and the parameter is gradually reduced to 0 after 1500 epochs. We store the state S_T, the action a_{T+1}, and the reward $R(S_T, a, S_{T+1})$ in the memory called *experience*. We configured the maximum size of *experience* to 5000 in a parameter expRepMaxSize. Finally, the epoch continues until it either terminates when the environment reaches a *secure state* (as determined by the partition termination function, PartitionSystemTerminateFunction) or when it reaches its maximum length (as configured in maxStep.) After storing a batch (configured as 128 in batchSize parameter) of experiences, we train multiple DNNs, one (implemented in NNBuilder with parameters layers, hiddenSize, and learningRate) for each partition, p_i, with episodes drawn from the

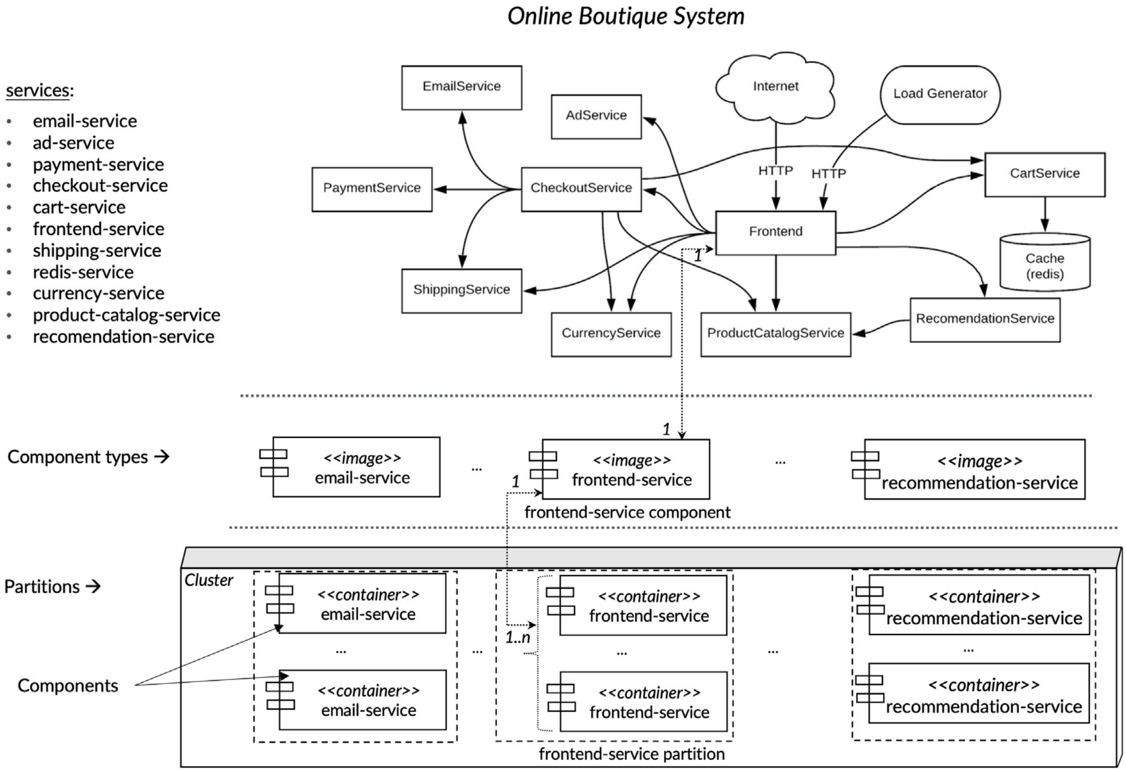


Fig. 4. Architecture of the OB System.

memory using the *experience replay* technique. We run many batches of episodes to retrain the DNNs to increase accuracy in the prediction of the action.

3. Case study: Online boutique

A proper validation and comparison of different IRS techniques is usually undermined by the lack of a standardized cyber-range [25]. For this reason, and in order to improve the reproducibility of our scenario and results, we illustrate the functionalities of our IRS software using a use-case scenario based on the open-source Online Boutique (OB) 2.0 system [22]. OB is a web application used by Google to showcase cloud-enabling technologies like Kubernetes/GKE, Istio, Stackdriver, gRPC, and OpenCensus [26]. It is a cloud-native application based on the microservice architectural style and is composed of 11 services, written in different languages that communicate over gRPC, plus a workload generator. It implements an online shop where users can browse items, add them to the cart, and purchase them. Fig. 4 shows the OB system architecture, along with a representation of a possible partitioning scheme, according to the definition of partition introduced in Section 2.1. There are 11 partitions, one for each service. For the sake of simplicity and without loss of generality, we report experimental results showing the time needed to converge to a near-optimal solution for a scenario in which a sub-system with 2 partitions is considered. We used a machine of type c220g2 from CloudLab [27] to run our experiments. We used the following JVM parameters: `-Xms102400 m -Xmx102400 m -XX:MaxMetaspaceSize=40960m`. For space reasons, we do not report experimental results on the non-stationary aspects. However, the interested reader can find a detailed analysis in [3].

We now describe the system model of the case study and analyze the experiments.

3.1. Case study system model

The system administrator describes the system model containing the partition information in the `topology-containers.yml` configuration file.

```
frontend-service:
  replication: 1
  state:
    - start
    - active
    - restarted
    - corrupted
    - shellCorrupted
...
```

Listing 1: Configuration snippet from *topology-containers.yml*

Listing 1 shows an example configuration of the *frontend-service* partition, where the number of components in the partition is represented by the parameter `replication`, and its state variables are listed in the `state` section. This specific configuration instance shows that the component type has the following 5 state variables: `start, active, restarted, corrupted, shellCorrupted`.

For space reasons, we only list the configuration of one component type. However, we list in Table 2 all the state variables (and their corresponding meanings) that we used to model the OB system.

```
start:
  execution-time: 300
  execution-cost: 100
  pre-condition: state[active] == false
  post-condition: state[active] = rand(1)
  components:
    - frontend-service
    - cart-service
    - redis-service
...
```

Listing 2: Configuration snippet from *action-set-containers.yml*

Table 2
OB System State variables list.

State variable	Meaning
`start`	If *true*, the container has started
`active`	If *true*, the container is running
`corrupted`	If *true*, the container is under attacker control
`restarted`	If *true*, the container has been restarted after the agent requested to do so
`shellCorrupted`	If *true*, the attacker has overwritten the shell /bin/sh in the container
`cartCorrupted`	If *true*, the content of Redis data store has been altered by the attacker
`confVuln`	If *true*, the current configuration of Redis data store is vulnerable to potential attacks and is subject to loss of confidentiality
`intVuln`	If *true*, the current configuration of Redis data store is vulnerable to potential attacks and is subject to loss of integrity
`passwordRequired`	If *true*, it mandates a password before accepting a command on Redis data store
`dangerousCmdEnabled`	If *true*, dangerous commands, such as *flushall*, that can potentially compromise the Redis data store, are enabled.
`accessRestricted`	If *true*, it only permits access from permitted sources, such as *cart-service*, to the Redis data store.

Table 3
Actions list.

Action Name	Description	Pre-Condition	Post-Condition	$E(a_i)$	$C(a_i)$
$start_i$	Start a stopped microservice	$\neg active_i$	$P = 1 \rightarrow active_i = true$	300	100
$restart_i$	Restart a malfunctioning service	$active_i \wedge corrupted_i \wedge \neg restarted_i$	$P = 0.75 \rightarrow corrupted_i = false; P = 1 \rightarrow restarted_i = true$	500	300
$heal_i$	Restore a malfunctioning service from a container image	$active_i \wedge corrupted_i \vee shellCorrupted_i$	$P = 1 \rightarrow corrupted_i = false; P = 1 \rightarrow shellCorrupted_i = false$	1000	500
$healRedisSecure_i$	Restore a malfunctioning Redis server from a container image	$active_i \wedge cartCorrupted_i \wedge \neg intVuln_i$	$P = 1 \rightarrow cartCorrupted_i = false$	1000	500
$healRedisInsecure_i$	Restore a malfunctioning Redis server from a container image	$active_i \wedge cartCorrupted_i \wedge intVuln_i$	$P = 1 \rightarrow cartCorrupted_i = true$	1000	500
$enablePassword_i$	Configure the Redis server to request a password before a user can issue commands	$active_i \wedge \neg passwordRequired_i \wedge confVuln_i \vee intVuln_i$	$P = 1 \rightarrow passwordRequired_i = trueP = 1 \rightarrow confVuln_i = false; P = 1 \rightarrow intVuln_i = false$	1000	1000
$disableDangerousCmd_i$	Configure the Redis server to disable dangerous commands	$active_i \wedge dangerousCmdEnabled_i \wedge intVuln_i$	$P = 1 \rightarrow dangerousCmdEnabled_i = false; P = 0.85 \rightarrow intVuln_i = true$	50	500
$restrictAccess_i$	Configure firewall rules to permit access from authorized services	$active_i \wedge \neg accessRestricted_i \wedge confVuln_i \vee intVuln_i$	$P = 1 \rightarrow accessRestricted_i = true; P = 0.7 \rightarrow confVuln_i = true; P = 0.7 \rightarrow intVuln_i = true$	50	300

The administrator also defines a set of actions and provides the following parameters for each action: the reward parameters (execution time and cost), the pre-condition and the post-condition in the `action-set-containers.yml` configuration file. Listing 2 shows the configuration of the action `start`, consisting of: its reward parameters (`execution-time` and `execution-cost`); the component types (`frontend-service` and `redis-service`) whose components can choose `start` as one of the action under the `components` section; the pre- and post-conditions under their respective sections. Table 3 defines all the actions along with their pre-condition, post-conditions, execution time and cost, that we modeled for the protection of the OB system.

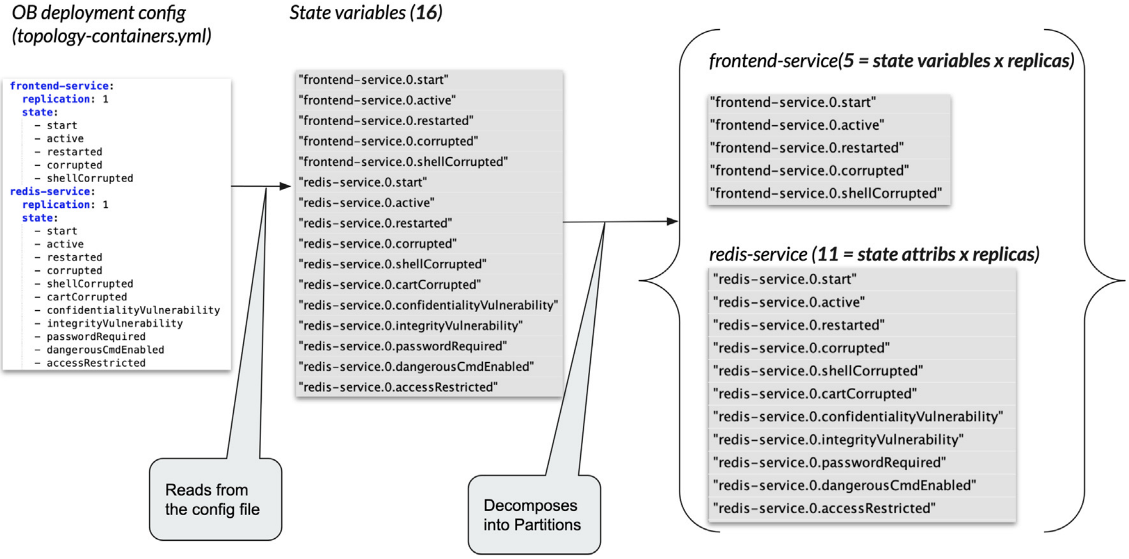
We use a total of *16 state variables* and decompose the system state as shown in Fig. 5. Furthermore, we implement `PartitionSystemTerminateFunction.terminate()` as the conjunction of the subset of the state variables reported in Table 4. In addition, the input to each DQN is the set of the state variable values of the its corresponding partition, and the output is one action from the set of valid actions.

Table 4
Termination condition.

State Variable Condition
`active` = *true* `corrupted` = *false* `cartCorrupted` = *false* `confVuln` = *false* `intVuln` = *false* `shellCorrupted` = *false*

3.2. Case study experiments

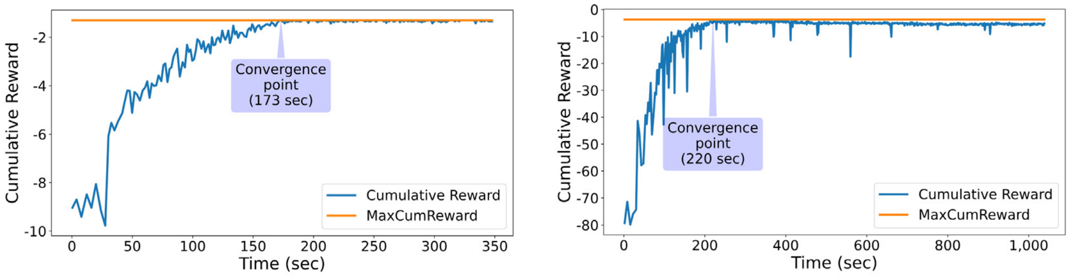
We initialize the system state to simulate an exploit based on the common vulnerability CVE-2019-5736 [28], based on the lack of authentication of Redis server. We measure the effectiveness of the proposed IRS prototype in terms of cumulative reward and convergence time, as typical in IRSs based on Reinforcement Learning (e.g., [3,21]). We carried out experiments to gather the



```
frontend-service:
  replication: 1
  state:
    - start
    - active
    - restarted
    - corrupted
    - shellCorrupted
redis-service:
  replication: 1
  state:
    - start
    - active
    - restarted
    - corrupted
    - shellCorrupted
    - cartCorrupted
    - confidentialityVulnerability
    - integrityVulnerability
    - passwordRequired
    - dangerousCmdEnabled
    - accessRestricted
```

OB deployment config
(topology-containers.yml)
State variables (16)
"frontend-service.0.start"
"frontend-service.0.active"
"frontend-service.0.restarted"
"frontend-service.0.corrupted"
"frontend-service.0.shellCorrupted"
"redis-service.0.start"
"redis-service.0.active"
"redis-service.0.restarted"
"redis-service.0.corrupted"
"redis-service.0.shellCorrupted"
"redis-service.0.cartCorrupted"
"redis-service.0.confidentialityVulnerability"
"redis-service.0.integrityVulnerability"
"redis-service.0.passwordRequired"
"redis-service.0.dangerousCmdEnabled"
"redis-service.0.accessRestricted"
frontend-service(5 = state variables x replicas)
"frontend-service.0.start"
"frontend-service.0.active"
"frontend-service.0.restarted"
"frontend-service.0.corrupted"
"frontend-service.0.shellCorrupted"
redis-service (11 = state attribs x replicas)
"redis-service.0.start"
"redis-service.0.active"
"redis-service.0.restarted"
"redis-service.0.corrupted"
"redis-service.0.shellCorrupted"
"redis-service.0.cartCorrupted"
"redis-service.0.confidentialityVulnerability"
"redis-service.0.integrityVulnerability"
"redis-service.0.passwordRequired"
"redis-service.0.dangerousCmdEnabled"
"redis-service.0.accessRestricted"
Reads from the config file
Decomposes into Partitions

Fig. 5. Relationship between the OB System state and Partition state variables.



(a) *frontend-service* partition

(b) *entire system*

Fig. 6. DQN training time vs cumulative rewards.

cumulative rewards in training the DQNs for both, the entire system and the *front-end partition* only. As depicted in Fig. 6, the training time to converge to a near-optimal cumulative reward of the *front-end partition*, 173 sec, is smaller than the convergence time for the case in which the entire system is considered, 220 sec. We calculated the optimal cumulative reward using our implementation of the Value Iteration algorithm [19] (classes VIMain and PartitionVIMain). Fig. 6(a) and 6(b) respectively show the cumulative reward obtained according to the time spent on training for both, the single *front-end partition* and the *system*. We do not provide a detailed analysis of the time overhead introduced by the IRS, because it is negligible with respect to the execution time of the response actions. Indeed, once the model has been trained, the IRS overhead consists in a single forward pass on the neural network, which can be accomplished in the order of milliseconds, while the execution time of the response actions is in the order of seconds or minutes.

4. Impact

The *irs-partition* system described in this paper further advances the state of the art in IRS software. We take a significant step forward in creating self-protecting systems that support non-stationary behavior, allow complex system partitioning, and near-optimal mitigation of local threats using multiple model types, including DQNs with customizable hyper-parameters. Our IRS software implementation with these capabilities is also the first to be released with an Apache 2.0 license.

Our software uses a training environment with a simulated system to train the IRS agents. Thus, it makes it possible to pre-train agents in a training environment and deploy them in a live environment. We train each agent with a dedicated deep neural network, where each network can be customized to a different architecture with its own set of hyperparameters. In addition, each agent could configure different types of modeling approaches, including DQNs, which we have used in our prototype.

5. Conclusions

Cyber threats are still evolving, and the security industry needs systems that can both, detect and respond, automatically. This need requires further investigation into automatic self-protecting systems, which can help secure real-world systems exhibiting non-stationary behavior. In this paper, we introduced a software tool to train multiple agents in a training environment using customizable deep neural networks to build an IRS, named *irs-partition*. We focused on leveraging multiple deep neural networks that predict a set of optimal actions. Moreover, the pre-trained agents immediately enhance system security using the transfer learning technique from their experience gained in a simulated system. In the future, we plan to monitor the impact and quality of the predictions, and to provide a mechanism to self-tune the deep neural networks.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

All the experiments have been conducted on the NSF-sponsored CloudLab platform [27]. E. Casalicchio is funded by the project Smart Defense, Italy (000090_19RS).

References

[1] What is snort?. 2021, https://www.snort.org/faq/what-is-snort (Accessed: 12 May 2021).

[2] Iannucci S, Abdelwahed S. A probabilistic approach to autonomic security management. In: Proc. of 2016 IEEE Int'l Conf. on Autonomic Computing. ICAC '16, 2016, p. 157–66.

[3] Iannucci S, Cardellini V, Barba OD, Banicescu I. A hybrid model-free approach for the near-optimal intrusion response control of non-stationary systems. Future Gener Comput Syst 2020;109:111–24.

[4] Guo Y, Zhang H, Li Z, Li F, Fang L, Yin L, et al. Decision-making for intrusion response: Which, where, in what order, and how long? In: Proc. of 2020 IEEE Int'l Conf. on Communications. ICC '20, 2020, p. 1–6.

[5] Hughes K, McLaughlin K, Sezer S. Dynamic countermeasure knowledge for intrusion response systems. In: Proc. of 31st Irish Signals and Systems Conf.. ISSC '20, IEEE; 2020, p. 1–6.

[6] Li X, Zhou C, Tian Y-C, Qin Y. A dynamic decision-making approach for intrusion response in industrial control systems. IEEE Trans Ind Inf 2018;15(5):2544–54.

[7] Iafarov R, Gad R, Kappes M. Improving attack mitigation with a cost-sensitive and adaptive intrusion response system. In: Proc. of 14th Int'l Conf. on Networks. ICN '15, 2015, p. 135–9.

[8] Foo B, Wu Y-S, Mao Y-C, Bagchi S, Spafford E. ADEPTS: Adaptive intrusion response using attack graphs in an e-commerce environment. In: Proc. of 2005 Int'l Conf. on Dependable Systems and Networks. DSN '05, IEEE; 2005, p. 508–17.

[9] Douligeris C, Mitrokotsa A. DDoS attacks and defense mechanisms: Classification and state-of-the-art. Comput Netw 2004;44(5):643–66.

[10] Koutepas G, Stamatelopoulos F, Maglaris B. Distributed management architecture for cooperative detection and reaction to DDoS attacks. J Netw Syst Manage 2004;12(1):73–94.

[11] Ryutov T, Neuman C, Dongho K, Li Z. Integrated access control and intrusion detection for web servers. IEEE Trans Parallel Distrib Syst 2003;14(9):841–50.

[12] Armstrong D, Carter S, Frazier G, Frazier T. Autonomic defense: Thwarting automated attacks via real-time feedback control. Complexity 2003;9(2):41–8.

[13] Armstrong D, Frazier G, Carter S, Frazier T. A controller-based autonomic defense system. In: Proc. of DARPA Information Survivability Conf. and Exposition, Vol. 2. IEEE; 2003, p. 21–3.

[14] Kreidl OP, Frazier TM. Feedback control applied to survivability: A host-based autonomic defense system. IEEE Trans Reliab 2004;53(1): 148–66.

[15] Nespoli P, Papamartzivanos D, Mármol FG, Kambourakis G. Optimal countermeasures selection against cyber attacks: A comprehensive survey on reaction frameworks. IEEE Commun Surv Tutor 2017;20(2):1361–96.

[16] Tanachaiwiwat S, Hwang K, Chen Y. Adaptive intrusion response to minimize risk over multiple network attacks. ACM Trans Inf Syst Secur 2002;19(1–30):95–6.

[17] Toth T, Kruegel C. Evaluating the impact of automated intrusion response mechanisms. In: Proc. of 18th Ann. Computer Security Applications Conf.. IEEE; 2002, p. 301–10.

[18] Mnih V, Kavukcuoglu K, Silver D, Graves A, Antonoglou I, Wierstra D, et al. Playing Atari with deep reinforcement learning. 2013, arXiv preprint arXiv:1312.5602.

[19] Sutton R, Barto A. Reinforcement learning: an introduction. 2nd ed.. Cambridge, MA, USA: MIT Press; 2018.

[20] Olivas ES, Guerrero JDM, Martinez-Sober M, Magdalena Benedito JR, Serrano Lopez AJ. Handbook of research on machine learning applications and trends: Algorithms, methods, and techniques. IGI Global; 2009.

[21] Hughes K, McLaughlin K, Sezer S. A model-free approach to intrusion response systems. J Inf Secur Appl 2022;66:103150.

[22] Online boutique. GitHub; 2021, GitHub Repository https://github.com/GoogleCloudPlatform/microservices-demo.

[23] DeepLearning4j. GitHub; 2020, GitHub Repository https://github.com/deeplearning4j/deeplearning4j.

[24] DeepLearning4j. RL4J: REinforcement learning for Java. GitHub; 2020, GitHub Repository https://github.com/deeplearning4j/rl4j.

[25] Montemaggio A, Iannucci S, Bhowmik T, Hamilton J. Designing a methodological framework for the empirical evaluation of self-protecting systems. In: Proc. of 2020 IEEE Int'l Conf. on Autonomic Computing and Self-Organizing Systems Companion. ACSOS-C '20, IEEE; 2020, p. 218–23.

[26] Bernstein D. Containers and cloud: From LXC to Docker to Kubernetes. IEEE Cloud Comput 2014;1(3):81–4.

[27] Duplyakin D, Ricci R, Maricq A, Wong G, Duerig J, Eide E, et al. The design and operation of CloudLab. In: Proc. of USENIX Ann. Tech. Conf.. ATC '19, 2019, p. 1–14.

[28] CVE-2019-5736 Detail, https://nvd.nist.gov/vuln/detail/CVE-2019-5736.