electronics

Article

An Adaptive Dynamic Defense Strategy for Microservices Based on Deep Reinforcement Learning

Yuanbo Li [1,2], Yuanmou Li [1], Guoqiang Wang [1,3,*] and Hongchao Hu [2]

[1] College of Computer, Luoyang Institute of Science and Technology, Luoyang 471023, China; 200900501775@lit.edu.cn (Y.L.); 200900501781@lit.edu.cn (Y.L.)
[2] Institute of Information Technology, Information Engineering University, Zhengzhou 450002, China; hhc@ndsc.com.cn
[3] Henan Key Laboratory of Green Building Materials Manufacturing and Intelligent Equipment, Luoyang 471023, China
* Correspondence: wgq@lit.edu.cn

Abstract

Aiming at the problem that it is difficult to balance security defense and quality of service in a dynamic cloud-native environment, an adaptive dynamic defense strategy (AD2S) for microservices based on deep reinforcement learning is proposed. First, a microservice attack graph model is constructed to extract security threats from multiple dimensions. Combined with queuing theory, the relationships among security performance, quality of service, cleaning cycle, and replica quantity are established to quantitatively model the effectiveness of defense. Subsequently, an adaptive defense framework is designed, which includes state monitoring, policy deployment, and optimization algorithms based on deep reinforcement learning, providing a rapid update solution for the optimal system configuration of microservices under dynamic traffic requests. The experimental results show that under dynamic traffic requests, compared with the existing DSEOM and OADSF strategies, AD2S improves the defense effectiveness by 34.38% and 10.29%, respectively, while ensuring the quality of service, significantly enhancing the system's security adaptive ability.

Keywords: cloud-native; microservices; deep reinforcement learning; dynamic defense; quality of service

Academic Editor: Myung-Sup Kim

Received: 15 September 2025
Revised: 15 October 2025
Accepted: 16 October 2025
Published: 19 October 2025

Citation: Li, Y.; Li, Y.; Wang, G.; Hu, H. An Adaptive Dynamic Defense Strategy for Microservices Based on Deep Reinforcement Learning. *Electronics* **2025**, *14*, 4096. https://doi.org/10.3390/electronics14204096

1. Introduction

Microservice architecture and container technology have transformed the deployment and operation modes of cloud applications [1–3], but they have also introduced new security threats. Microservices have intricate mutual invocation relationships. Microservices can be invoked at the application level through interactive interfaces and can also carry out lateral movement attacks in container environments, which makes the attack surface of microservices difficult to control [4,5]. Virtualization technology allows multiple microservices to share operating system resources at the container level; once an attacker exploits a container vulnerability to escape, it becomes easy to launch lateral attacks on other containerized microservices on the same host. Furthermore, with the changes in microservice request traffic, operational status, service migration, etc., the attack surface of containerized microservices is in a dynamic state, which poses higher requirements for the system's security defense strategy [6–8]. Consequently, in the face of microservice scenarios with dynamic traffic requests, traditional security protection strategies such as distributed authorization and intrusion detection cannot be dynamically adjusted along

with the changes in containerized microservices and are difficult to effectively adapt to new scenarios [9,10]. Even when communication is encrypted or devices appear physically isolated, adversaries can still infer fine-grained user activities through side-channel attacks—such as app fingerprinting under open-world settings or exploiting RF energy harvesting signals—revealing sensitive operations tied to specific UI components with high accuracy [11,12]. These emerging threats demonstrate that attackers can bypass conventional encryption and isolation mechanisms, thereby exacerbating the challenge of maintaining both strong security and service quality in dynamic systems.

Moving Target Defense (MTD) [13–15] effectively increases the difficulty for attackers to penetrate the system and enhances the defense effect of the system by adding dynamic characteristics to the system and constantly changing the attack surface of the system, but it also brings new problems. MTD can effectively enhance the security protection capability of the system [16–18]. However, due to the introduction of technologies such as dynamic cleaning and service migration, the response time of services is increased, resulting in a decline in Quality of Service (QOS), which seriously affects the user experience. Therefore, how to optimize the dynamic and redundant endogenous security capabilities provided by MTD and further improve the defense efficiency has become the focus of academic attention.

To solve the above problems, Gao et al. [5] incorporated MTD dynamics at the microservice programming language and container image levels. By continuously changing the microservice attack surface, they effectively mitigated attack threats caused by the same vulnerability. Bardas et al. [9] proposed a cloud computing platform that can capture the service dependencies in the cloud environment and find the best service instance replacement strategy to maximize the difficulty of attacks. Connell et al. [19] constructed the security evaluation framework of MTD and quantitatively analyzed the security performance and resource availability of the system after introducing the MTD strategy. However, the above MTD strategies mainly focus on the optimal security defense configuration in the static environment and fail to fully consider the dynamic characteristics of the cloud environment. In order to meet the demand of defense efficiency optimization in a dynamic environment, Jin et al. [20] built an attack graph model for container cloud, identified key microservice nodes by analyzing the mediation centrality of the graph and transformed the MTD security defense configuration problem into an optimization problem to realize the dynamic protection of key microservice nodes. Li et al. [21] used the Deep Reinforcement Learning (DRL) method to construct OADSF framework to realize dynamic protection of microservice nodes, which significantly improved system security. Although the existing research has made progress in security enhancement, it is difficult to balance the QOS under dynamic traffic requests. Specifically, the security defense configuration in microservices will lead to dynamic changes in the QOS. In order to maintain the QOS, the system configuration of microservices is dynamically optimized. This process causes frequent changes in the status of microservices, resulting in mismatches between security defense strategies and the actual system state, and ultimately compromising the real-time effectiveness of security defenses.

Although existing studies have attempted to enhance the security of microservices through dynamic cleaning mechanisms, dynamic defense mechanisms lack coordination with resource allocation strategies. Therefore, the elasticity of cloud-native applications is influenced by the synergy of resource allocation and dynamic defense mechanisms. To address this issue, this paper proposes AD2S, which changes the attack surface of the microservice by dynamically altering the cleaning cycle, cutting off the attack process, and enhancing the security of the microservice. However, the dynamic cleaning cycle will affect the number of available microservice replicas, and thereby influence the QOS.

By dynamically changing the number of microservice replicas, the response time can be optimized in real time while taking into account the QOS. Finally, the AD2S strategy with both security and QOS under dynamic traffic requests was implemented by using the DRL algorithm.

The main research work and contributions of this paper are as follows:

(1) By analyzing the complex attack process of microservices, the microservice attack graph model is established to extract specific risks and threats in the cloud. Using the attack graph, a security quantification model is constructed to establish the relationship between the dynamic cleaning cycle and security performance. Based on queuing theory, the QOS model is constructed, and the relationship between QOS, the cleaning cycle, and the number of redundant replicas is established. Finally, considering both security performance and QOS, the defense effectiveness of the system is defined.

(2) In order to realize the adaptive optimization of microservice defense strategy, this paper designs the AD2S framework, which includes the container cloud cluster, the status monitoring module and the security policy deployment module. Through collaborative work, the optimal cleaning cycle and the number of replicas of the microservices at the current time are updated in real time, which effectively improves the real-time effectiveness of the security defense strategy in this scenario.

(3) The simulation experiment results show that the DRL algorithm in the AD2S framework has good convergence and scalability. Compared with existing security defense strategies, AD2S provides an effective update solution for microservice security defense strategies under dynamic traffic requests, improving defense effectiveness by 34.38% and 10.29%, respectively, which verifies the effectiveness of the proposed strategy.

2. Problem Analysis

2.1. Threat Model

This paper adopts the Cyber Kill Chain (CKC) model [22] to describe the attacker's attack process against microservices. In this model, a complete attack process includes multiple steps such as preliminary reconnaissance, vulnerability exploitation, permission acquisition, backdoor installation, and influence expansion. Based on the above analysis, in a cloud-native environment, attackers launch attacks from the outside.

We make the following explicit assumptions regarding the attacker's capabilities and limitations to model a realistic yet challenging threat scenario:

(1) Attackers possess knowledge of publicly known vulnerabilities (e.g., CVEs) in microservices and containerized components, and can leverage tools to craft and deliver exploit payloads. However, they do not have prior insider knowledge of internal system configurations, authentication credentials, or encrypted communication keys.

(2) Attackers can compromise a microservice only if it exposes a known vulnerability and is reachable from the external network. Once a microservice is hijacked, the attacker may attempt lateral movement to adjacent microservices—but only if there exists a legitimate calling relationship that allows network reachability. This reflects the principle of least privilege in service mesh architectures.

(3) Lateral movement is constrained by the service dependency graph; attackers cannot jump arbitrarily between unrelated services. Instead, they are limited to traversing paths defined by actual invocation relationships, and typically prefer attack paths with lower cumulative cost (e.g., shortest weight path), where edge weights reflect exploit difficulty, detection risk, or access control strength.

Based on this model, at the application layer, the target may be a vulnerable microservice within a call chain [23,24], while at the container layer, the target could be the underlying container runtime. After reconnaissance and exploitation, attackers hijack an

initial foothold [25–27] and perform lateral penetration to escalate privileges and move closer to the ultimate objective—typically the microservice that interfaces with critical data storage, such as MongoDB [28,29].

In cloud-native environments, unauthorized access to target microservices in the cloud poses significant security threats to cloud assets [30–32]. Figure 1 illustrates the threat model. MongoDB stores sensitive business data and is a common final target. Attackers may reach it through multi-step attack paths, for example, Nginx, Payments, CustomerInfo, Personal Lending, MongoDB. Each hop represents a potential vulnerability exploitation enabled by calling dependencies. The edge weight computation in Section 3.2 is designed to increase the cost of such high-risk paths, thereby deterring attackers from following the shortest or most vulnerable routes.

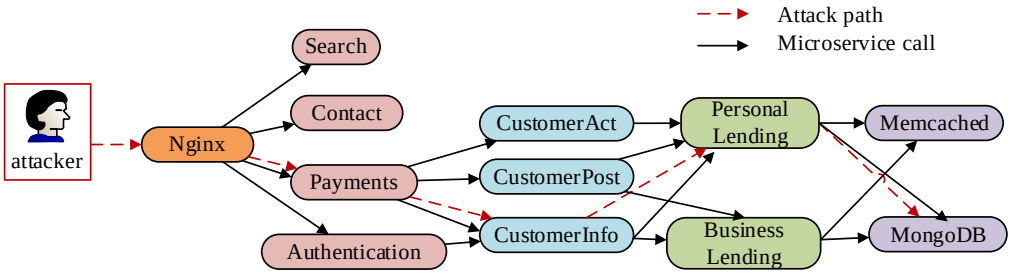

Figure 1. Microservice Attack Threat Model in Banking Systems.

2.2. Key Challenges

Based on the above analysis, in dynamic traffic requests scenarios, the dynamic defense strategies of microservices face the following challenges:

(1) As the scale of microservices expands, the complexity of solving security defense configurations for microservices increases. In scenarios with dynamic traffic requests, it is necessary to balance security performance and QOS to improve system defense effectiveness. A critical issue to be addressed is how to adjust the cleaning cycle to ensure security performance while dynamically adjusting the number of microservice replicas according to different traffic requests to meet user response time constraints.

(2) In cloud-native environments, the dynamic nature of microservices imposes higher requirements on the real-time update of system security defense strategies. As access request traffic changes, the status of microservices changes accordingly, leading to dynamic changes in the system's microservice attack surface and a decline in defense effectiveness. Therefore, another key challenge is how to quickly adjust the dynamic cleaning cycle and replica quantity of microservices to ensure that the system's defense effectiveness can be dynamically updated in real time in response to changes in microservice status.

In response to these key challenges, during an attacker's attack process, endogenous security defense capabilities such as dynamic and redundancy are invoked to achieve adaptive optimization of security resources. This paper adopts a dynamic cleaning strategy for all microservices to block the attack chain and ensure the security performance of the system. The dynamic cleaning strategy can be interpreted as: stateless microservices undergo dynamic cleaning during operation to change the attack surface, prevent the residence of attack payloads injected by attackers, and cut off attackers' further penetration along the microservice call chain. During the implementation process, the container orchestration platform, based on the dynamic cleaning cycle of microservices, creates new replicas of microservice one by one based on heterogeneous read-only container images, and simultaneously takes the previous replicas offline one by one, thus achieving the purpose of cleaning microservice replicas. V_i creates n_i replicas every T_i time and delete the old replicas after the new ones are successfully created. At the same time, the deployment

strategy of redundant replicas is adopted to ensure the QOS under high traffic requests. Based on the MTD strategies, in dynamic traffic requests scenarios, dynamic adaptive updates of microservice security defense strategies are realized, which can effectively block attackers' intrusion processes and enhance the real-time defense effectiveness of microservice systems.

3. Problem Modeling

Based on the analysis of the above problems and the existing security threats, this paper constructs a microservice attack graph model (MAGM), a security quantification model, and a QOS model. Finally, the problem is formally defined, describing the optimization of security defense strategies from multiple dimensions, including system security performance and QOS. The common symbols used in this paper are shown in Table 1.

Table 1. Meanings of Common Symbols.

Symbol	Meaning
$MAGM$	Microservices attack the graph model
V	Set of microservices
E	Set of edges representing dependency relationships
N_t	Set of microservice replica quantities
T_t	Set of microservice cleaning cycles
λ_t	Set of microservice request arrival rates
$e_{i,j}$	Dependency relationship between microservice V_i and V_j
$DF(e_{i,j})$	Difficulty of exploiting vulnerabilities in microservices V_j
$p(t)$	Attack success rate of the microservice
$D(e_{i,j})$	Weight between microservice V_i and V_j
φ_t	Security performance of the microservice system
c_i	The number of available replicas of microservice V_i
p_k^i	Probability that k replicas are in the cleaning state
p_0^i	Probability that no replicas are in the cleaning state
μ	Service rate of each microservice replica
ψ_t	User Quality of Service
DE	Defense effectiveness

3.1. Microservice Attack Graph Model

$MAGM$ is constructed based on the dependency relationships between microservices. At t time, $MAGM = (V, E, T_t, N_t, \lambda_t, \varphi_t, \psi_t)$ characterizes the edge relationships, dynamic cleaning cycles, redundant replicas, user request traffic, security performance, and QOS of microservices, with detailed descriptions as follows:

$V = \{V_1, V_2, \ldots, V_n\}$ represents n microservices with dependency relationships, where V_i denotes the *i-th* microservice node. These n microservice nodes form MAGM through the edge set of dependency relationships.

$E = \{e_{i,j} | V_i, V_j \in V, e_{i,j} \in \{0,1\}\}$ represents edges with dependency relationships in the graph. If $e_{i,j} = 0$, it indicates that microservice V_i is not a predecessor of V_j, and there is no invocation. If $e_{i,j} = 1$, it indicates that microservice V_i is a predecessor of V_j, and there exists invocation. Meanwhile, attackers can exploit vulnerabilities in V_j to deliver attack payloads. If the attack is successful, they can move laterally, and the weight of the edge represents the success rate of defense.

$T_t = \{T_1, T_2, \ldots, T_n\}$ denotes the dynamic cleaning cycle of each microservice, which is also the time interval at which each microservice can be called normally. That is, V_i undergo dynamic cleaning every T_i time, changing the IP address or configuration file of the microservice, or creating new replicas to replace the old ones.

$N_t = \{n_1, n_2, \ldots, n_n\}$ represents the number of dynamically configured microservice replicas based on QOS requirements. Replicas of the same microservice are generated from heterogeneous images with the same function and can perform identical tasks.

$\lambda_t = \{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ indicates the dynamic request arrival rate of each microservice.

φ_t represents the security of the microservice system, which is used to evaluate the defense effect of multi-dimensional dynamic defense strategies. ψ_t represents the response delay of the microservice system, which is used to assess the QOS of the microservice system.

3.2. Security Quantification Model

Definition 1. *Given microservices $V_i, V_j \in V, i \neq j$, edge $e_{i,j} = (V_i, V_j)$ indicates that V_i is a predecessor of V_j, and there are invocation relationships between them. If an attacker has gained control of V_i, they can continue to launch an attack on V_j using the edge relationship between V_i and V_j. The weight $D(e_{i,j})$ of edge $e_{i,j}$ is defined to represent the difficulty of attacking V_j.*

To quantitatively describe the difficulty of successfully attacking V_j, the vulnerability scoring index of microservice V_j is used for evaluation in the Common Vulnerability Scoring System (CVSS). The Exploitability Metrics (EM) and the temporary metric w were used to evaluate the difficulty of attacking microservice V_j. Among them, EM is composed of AV, AC, PR and UI, which is used to quantitatively describe the possibility of a single vulnerability being exploited, and can be specifically expressed as Equation (1).

$$EM = (8.22 \times AV \times AC \times PR \times UI)^{-1} \tag{1}$$

However, each microservice node or its hosting container environment may contain multiple exploitable vulnerabilities, and attackers can use any of these vulnerabilities to launch detection and attacks. Therefore, temporal metrics w are adopted to quantify the probability and possibility of selecting each vulnerability. By calculating the weighted average of all actually observable vulnerabilities in the microservice node, with the weight being the probability that each vulnerability might be selected, the exploitation difficulty of a microservice node is quantitatively modeled in Equation (2).

$$DF(e_{i,j}) = \frac{\sum_{vul \in VUL} (w \times EM)}{\sum_{vul \in VUL} w} \tag{2}$$

where VUL represents the set of all vulnerabilities in the attack surface represented by this microservice node. As shown in Figure 1, it is an instance of a microservice attack threat. The attacker can launch an attack from the external interface Nginx, carry out the attack along the red line, and eventually successfully penetrate the database node. In this paper, CVSS is used to obtain information on exploitable vulnerabilities in microservices. Equations (1) and (2) are utilized to calculate $DF(e_{i,j})$. The attack chain theory is used to analyze the attacker's state transition. Assuming that the system does not adopt the dynamic cleaning strategy, the attacker can always find the vulnerability in a certain period and successfully attack. Due to the dynamic cleaning strategy, when the microservice is successfully attacked and needs to move laterally along the microservice invocation and communication relationship, the injected attack payload is cleaned by the dynamic cleaning strategy, which blocks the attack chain and prevents further attack diffusion. The attack success probability under different cleaning cycles [33] is formally defined in Equation (3) using an exponential decay function that captures the effect of dynamic cleaning.

$$p(t) = 1 - \frac{1 - e^{(t - T_s)}}{1 - e^{-T_s}} \tag{3}$$

In this model, attackers initially need to obtain vulnerability information of the target microservice, and the probability of successful attacks is relatively low. As the amount of vulnerability information obtained increases, attackers grasp the key vulnerabilities of the target microservice, and the probability of successful attacks rises rapidly. When the attack succeeds at T_s time, T_s is the maximum time for the attack to succeed and also a reflection of the difficulty of exploiting the vulnerability. $T_s = f\left(DF(e_{i,j})\right)$ represents the maximum time required for an attacker to successfully invade this microservice node in a static environment. Function $f(.)$ represents the mapping relationship between the difficulty $DF(e_{i,j})$ of an attacker attacking microservice V_j and the maximum time for a successful attack. Assuming the microservice node with the greatest attack difficulty is MongoDB. According to reference [33,34], the maximum cleaning cycle of MongoDB is mapped to T_s. Based on the relative difficulty and attacker's capability, the parameters in $T_s = f\left(DF(e_{i,j})\right)$ quantify the strength of attacks that the service is designed to withstand. It can be calculated that the maximum cleaning cycle of other microservice nodes is $\left(\frac{DF(e_{i,m})}{DF(e_{i,j})}\right) * T_s$. At t time, when the system sets different cleaning cycles, the defense weights of the edges are different. The defense weights of each edge in the microservice attack graph can be calculated by Equation (4).

$$D(e_{i,j}) = 1 - p(T_i) \tag{4}$$

In this paper, the call sequence of multiple microservices is referred to as the microservice call chain, denoted as $P = \{V_{i_1}, V_{i_2}, \ldots, V_{i_r}\}$, where $d \in [1, r-1]$, $e_{i_d, i_{d+1}} = 1$. $V_{i_d} \in V$ represents a microservice node, and d represents the call depth of microservice V_i in microservice call chain P. For attackers, on the one hand, the more microservice nodes are captured, the greater the attack benefits; on the other hand, the last node of the microservice call chain is usually the microservice storing data [35], and the closer to the end of the microservice call chain, the weaker the security performance of the system. Therefore, the security performance of the system is shown in Equation (5).

$$\varphi_P = \sum_{V_i \in P} \left(1 - e^{-\xi d}\right) D(e_{i,j}), P \subseteq MAGM \tag{5}$$

Equation (5) indicates that as the depth of microservice invocation d increases, the attacker gradually approaches the end node of the microservice call chain, the attacker's attack gains gradually increase, and the security performance of the system becomes weaker. Among them, $\xi (0 < \xi < 1)$ is the adjustment factor for the growth rate of attack benefits. The smaller ξ is, the faster it increases with the increase in microservice call depth d.

For the microservice attack graph, it is assumed that each microservice has its own cleaning cycle. Once the attack graph is determined, attackers will choose the attack path with the minimum cost to target the system, which is measured by the shortest path in the graph. Since the exact attack path cannot be determined, it is assumed that attackers act rationally and select the path with the minimum cost to launch attacks on the target microservice in the graph. Therefore, the path with the minimum defense cost is used to characterize the system's security performance, as shown in Equation (6).

$$\varphi_t = \min_{P \subseteq MAGM} \{\varphi_p\} \tag{6}$$

The security performance of the system is defined in two complementary ways: in Equation (5), it is expressed as a function of microservice call chain depth, reflecting increasing attack gain and decreasing system resilience as the attacker approaches critical nodes. In Equation (6), it is defined as the minimum defense cost across all possible attack paths in the graph, the shortest path in terms of cumulative edge weights.

As shown in Figure 2, it is an example of an attack graph, where each microservice contains some vulnerability information that can be exploited, which is obtained through CVSS. The difficulty of attacking the microservice nodes is calculated using Equations (1) and (2). Assuming that the dynamic cleaning strategy is not adopted, attackers can launch probes against microservices. According to Equations (3) and (4), at T_s time, the attack success probability is 1, and the defense weight of edges is 0. When the dynamic cleaning cycle is T_i, assuming that the attack success probability and the defense weight of the edge are shown in Figure 2, the attacker is rational and will move laterally along the edge with the smallest defense weight. By comparing the red and blue attack paths, the shortest paths in the figure are A, B1, C3, and D, and the defense weights of the corresponding edges are 0.5, 0.3, 0.3, and 0.3, respectively. Since the target node is usually a microservice that stores data, the closer to microservice D, the greater the attacker's gain. By adjusting the attack gain according to Equation (5), the updated defense weights can be calculated, which are 0.28, 0.24, 0.27, and 0.29, respectively. According to Equation (6), the sum of the paths with the smallest defense weights represents the security performance of the system in Figure 2. Since the dynamic cleaning cycle needs to be dynamically adjusted according to security and QOS constraints, the defense weights of the edges change dynamically, and the optimal cleaning cycle needs to be set.

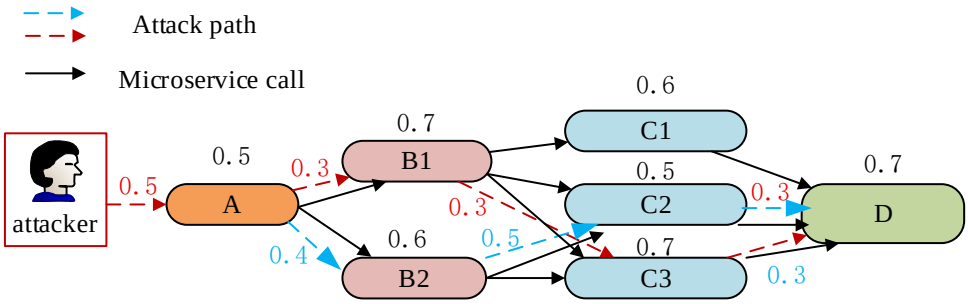

Figure 2. Example of an Attack Graph.

3.3. QOS Model

To enhance the security performance of microservice systems, it is necessary to reduce the dynamic cleaning cycle of microservices. However, an inappropriate cleaning cycle can affect system performance and further impact QOS. Therefore, this paper needs to ensure QOS while considering security performance. Generally, the response time of microservices is a key indicator of QOS, and queuing theory is used to model the response time of microservices.

First, assuming a microservice in the system has multiple replicas, if no cleaning cycle is set, the response time of these multiple microservice replicas can be modeled as an M/M/N queuing system. However, after introducing a dynamic cleaning cycle, microservice replicas in the system may be in either an available state or a cleaning state. In this case, it is necessary to recalculate the number of available microservice replicas in the system.

Suppose the dynamic cleaning cycle of microservice V_i is T_i, and T_i is the available time of the microservice. Microservices independently perform cleaning operations at the rate of $1/T_i$. Suppose c_i is the average number of replicas of available microservice V_i, and k is the number of replicas that are being re-cleaned. Therefore, the total quantity is $n_i = c_i + k$. X is the aggregation rate at which microservice V_i in the system completes its dynamic cleaning, and S is the average time required for microservice V_i to complete its dynamic cleaning. According to the little theorem, which describes the relationship among the available microservices, available time and arrival rate of the system, we can obtain

$k = X \times S$, $c_i = X \times T_i$. The dynamic cleaning process may involve the time for changing a new IP address, modifying the configuration file, and regenerating a new microservice image. Usually, S is a fixed value, and the dynamic cleaning cycle is dynamically set according to the security performance requirements of its microservice system.

A Continuous-Time Markov Chain (CTMC) shown in Figure 3 is used for calculation X. State k represents the number of replicas of microservices V_i that are being dynamically cleaned, and $n_i - k$ represents the number of replicas of microservices V_i that are in an available state. Assuming that the arrival rate and departure rate are stationary, they, respectively depend on the cleaning cycle T_i and the fixed value of S. The probability distribution of the number of microservice replicas in the cleaning state is obtained using the general survival equation, as shown in Equations (7) and (8).

$$p_k^i = p_0^i \cdot \prod_{r=0}^{k-1} \frac{(n_i - r)}{(r+1) \cdot T_i \cdot S} = p_0^i \cdot \left(\frac{S}{T_i}\right)^k \cdot \begin{pmatrix} n_i \\ k \end{pmatrix} \quad k = 1, 2, \cdots, n_i \tag{7}$$

$$p_0^i = \left[1 + \sum_{k=1}^{n_i} \left(\frac{S}{T_i}\right)^k \cdot \begin{pmatrix} n_i \\ k \end{pmatrix}\right]^{-1} \tag{8}$$

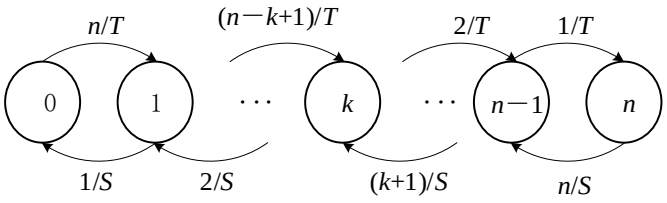

Figure 3. CTMC State Transition Process.

From this, the aggregate rate of dynamic cleaning X can be calculated, as shown in Equation (9).

$$X = \sum_{k=1}^{n_i} \left(\frac{k}{S}\right) \cdot p_k^i = \frac{1}{S} \sum_{k=1}^{n_i} k \cdot p_k^i \tag{9}$$

where the number of available microservice replicas in the system can be calculated using Equation (10).

$$c_i = X \cdot T_i = \frac{T_i}{S} \sum_{k=1}^{n_i} k \cdot p_k^i \tag{10}$$

The response time of microservice V_i with a single replica arrival rate of μ_i is calculated using the MMN queuing theory, and the calculation is carried out by Equation (11).

$$RT_i = \frac{1}{\lambda_i} \left(\frac{\lambda_i}{\mu_i} + \delta_i \frac{(c_i \delta_i)^{c_i}}{c_i!} \frac{p_0^i}{(1 - \delta_i)^2} \right), V_i \in V \tag{11}$$

where $\delta_i = \frac{\lambda_i}{c_i \mu_i}$, and p_0^i represents the steady-state probability that there is no request arrival in the request queue of the microservice V_i. Suppose the request arrival rate of the system root node is λ. Microservice V_i receives all traffic requests from its predecessor nodes and distributes the traffic requests to the subsequent nodes with equal probability. Therefore, based on the request arrival rate of the root node, the request arrival rate of the entire system's microservice nodes can be calculated. The microservice call chain with the

largest response delay in the system characterizes the system delay of the microservice. It can be expressed by Equation (12).

$$\psi_t = \max_{P \subseteq MAGM} \left\{ \sum_{V_i \in P} RT_i \right\} \tag{12}$$

3.4. Defense Effectiveness Definition

Based on the MAGM and CKC models, the attacker's attack process can be described as: the initial stage, lateral movement, service hijacking, and other stages. In the initial stage, the difficulty of the attack is determined by Equations (1) and (2), the probability of lateral movement is decided by Equation (3), and the selection of the attack path is based on the defense weight given by Formula (4) to complete the service hijacking. To block the attack process, it can be concluded from Equation (3) that the smaller the dynamic cleaning cycle, the better. However, the dynamic cleaning cycle will directly affect the number of available replicas. The smaller the dynamic cleaning cycle, the fewer the number of available replicas, which further affects the QOS. To enhance the QOS, it is necessary to increase the number of available replicas, which will further increase resource consumption and cause difficulties in the configuration of system security defense. In the face of dynamic traffic requests, it is necessary to balance the two aspects of factors, ensuring the QOS while enhancing the security performance of the system as much as possible. To comprehensively consider these factors, the defense effectiveness of the system is defined as shown in Equation (13).

$$\begin{aligned} &\max DE = \omega_1 \cdot \vartheta_1(\varphi_t) + \omega_2 \cdot \vartheta_2\left(\tfrac{1}{\psi_t}\right) \\ &\text{s.t.} \psi_t \leq \psi_{SLO}, T_i \leq T_s \end{aligned} \tag{13}$$

where ψ_{SLO} represents the response time constraint of the microservice, T_s represents the maximum cleaning cycle of the microservice, ω_1 and ω_2 are the weight coefficients of the indicators, and ϑ_1 and ϑ_2 are the normalization functions of the indicators.

4. AD2S Strategy Framework

This section first provides a detailed description of the overall framework of AD2S. This framework acquires the operational status of microservices in the cloud-native platform through the state monitoring module, then solves the security defense configuration at the current moment through the DQN algorithm, and dynamically deploys the obtained decision vectors to the microservices through the security policy module.

4.1. The Overall Framework of AD2S

In this paper, an AD2S framework for microservices based on DRL is proposed, which includes a state monitoring module, a computing node, an initial state vector, a DQN algorithm, and a decision vector. The framework is shown in Figure 4.

The container cloud cluster is mainly composed of the modified Kubernetes extracting real-time orchestration details, and is also responsible for real-time monitoring of the microservice cleaning cycle $T_t = \{T_1, T_2, \ldots, T_n\}$, the number of replicas $N_t = \{n_1, n_2, \ldots, n_n\}$, and the request arrival rate $\lambda_t = \{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ in the cloud. The three together form the initial state input vector of the AD2S framework. The DQN algorithm obtains the optimal decision vector at the current moment through multiple training processes. According to the optimization objective of Equation (13), it solves the optimal security deployment strategy of the microservice under the current state conditions in real time and sends the optimal security configuration decision vector to the security policy deployment module.

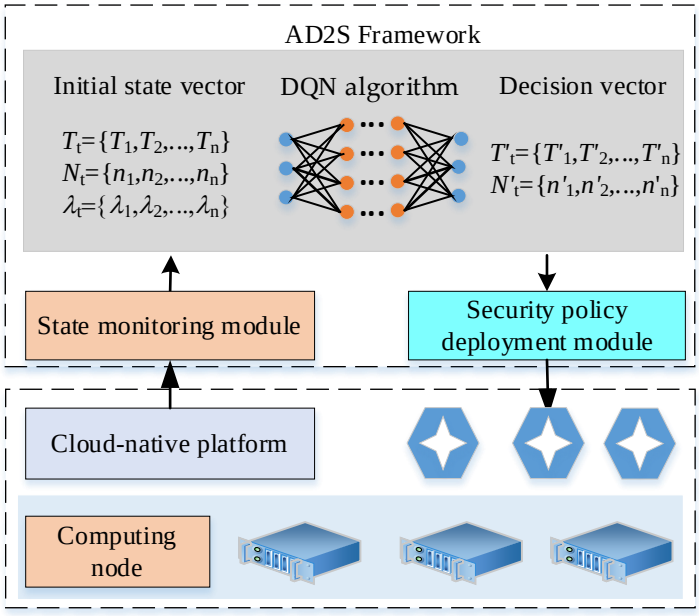

Figure 4. The Overall Framework of AD2S.

When the status monitoring module detects changes in the status of a microservice, it includes the request arrival rate, the number of replicas, node failures, etc. The status monitoring module will resend the latest status to the DQN algorithm module, obtain the latest security decision vector, guide the dynamic scaling unit in the security policy deployment module to manage the number of microservice replicas, and guide the microservice dynamic cleaning unit to complete the cleaning operation of replicas. When performing the cleaning operation, in order to minimize the adverse impact on the cleaning of microservice replicas, while providing the decision vector, determine whether to update the dynamic cleaning cycle. Specifically, if the cleaning cycle at the current moment is greater than that at the present moment, it indicates that the security performance of the system is better at this time. Microservices that have not completed cleaning at the current moment choose not to update the cleaning cycle. If the cleaning cycle at the current moment is less than that at the present moment, it indicates that the security performance of the system at this time is poor. Microservices that have not completed cleaning at the current moment should immediately update their cleaning cycles. The core of AD2S lies the fact in that the decision module based on the DQN algorithm can input the initial state vector obtained by the state monitoring module through the neural network, quickly obtain the security configuration vector of the microservice under the current user request, and guide the security configuration strategy module to optimize the deployment of microservices in the cloud. Section 4.2 will provide a detailed introduction to the optimal security defense configuration algorithm based on DQN.

4.2. Adaptive Dynamic Defense Strategy Optimization Algorithm Based on DQN

This section solves the problem based on the DQN network and proposes an adaptive dynamic strategy optimization algorithm. DQN estimates the action value in the state through the Q-network, avoiding the problem of Q-table explosion caused by traditional Q-learning algorithms in the high-dimensional state-action space. PPO and A3C perform well in continuous action Spaces. The action selection in this paper is a high-dimensional action space problem that is discrete and highly combinable. DQN can effectively alleviate the problems of Q value overestimation and dimension explosion through Action Factorization and the Double DQN, Dueling Network structure. Therefore, in this paper, the DQN algorithm is utilized to solve optimization problems.

In the sample exploration phase of the training process, the DQN selects the action to be executed based on the "ε-greedy strategy" according to the Q-value estimate output by the $Q(S_t, a; \theta)$ network, that is, the action with the largest Q-value estimate in the current state is selected with probability $1 - \varepsilon$, and the action is randomly selected with probability ε. As the training proceeds, ε starts from $\varepsilon_{\max}$ and decreases linearly to $\varepsilon_{\min}$ in M steps, coordinating the relationship between the "exploration" ability of DQN to unknown actions and the "utilization" of the action value, making the untried actions more widely explored in the early stage of training and the training experience more fully utilized in the late stage of training. In addition, to improve sample utilization, DQN uses an experience replay pool $\langle S_t, a_t, r_t, S_{t+1} \rangle$ to store samples. When the number of samples reaches the batch processing requirement, DQN starts to train the Q-network, so that it can estimate the state action value estimation more accurately. At t time, the loss function can be expressed by Equation (14).

$$Loss(\theta) = \mathrm{E}\left[\frac{1}{2}(y_t - Q(S_t, a; \theta))^2\right] \tag{14}$$

Here, let $Q(S_t, a; \theta)$ be the output values for state S_t and action a when the neural network parameter is θ. y_t is the learning objective of the evaluation network, which consists of the current moment's reward r_t and the estimated value of the target network for the next moment $Q(s_{t+1}, a; \theta^-)$. Specifically, it can be expressed by Equation (15).

$$y_t = r_t + \gamma Q\left(S_{t+1}, \max_a Q(S_{t+1}, a; \theta); \theta^-\right) \tag{15}$$

In each iteration, the agent will select an action a_t based on the current state S_t, then execute this action and receive new states S_{t+1} and reward r_t from the environment. This information will be collected and used to update the parameters of the evaluation network to improve future decisions. The states, actions, and reward in the algorithm design are defined as follows.

(1) Status: The operational status of each microservice is composed of information such as the number of its replicas, the computing node it is located on, and its security configuration. For the *i-th* microservice, the number of its replicas is n_i. The computing node where the microservice replica is located can be represented as $RS_i = [id_1, id_2, \ldots, id_{n_i}]$, where *id* represents the serial number of the computing node where the microservice replica is scheduled to the cluster. Therefore, the running state of the entire application can be represented as $R_t = \{RS_1, RS_2, \ldots, RS_n\}$, that is, the collection of the running states of all microservices. At t time, the security configuration consists of the current number of replicas $N_t = \{n_1, n_2, \ldots, n_n\}$ and the cleaning cycle $T_t = \{T_1, T_2, \ldots, T_n\}$, combined with the request arrival rate $\lambda_t = \{\lambda_1, \lambda_2, \ldots, \lambda_n\}$. The three together form the security configuration $H_t = \{T_t, N_t, \lambda_t\}$ at the current moment. All this information can be obtained in real time through the API interface by the status monitoring module. After processing the initial state vector by the state monitoring module, the state data is a combination of the running state and the security defense configuration, that is, $S_t = \{RS_t, H_t\}$.

(2) Action: The policy deployment module needs to determine each microservice replica vector $T_t = \{T_1, T_2, \ldots, T_n\}$ and the cleaning cycle vector $T_t = \{T_1, T_2, \ldots, T_n\}$ under the current request arrival rate $\lambda_t = \{\lambda_1, \lambda_2, \ldots, \lambda_l\}$. This is a high-dimensional decision for the policy deployment module. The high-dimensional action space leads to excessive computational complexity in estimating the Q value between actions in different states, resulting in the problem that DQN has difficulty converging. To address this issue, this section decomposes high-dimensional single-step decisions into multiple low-dimensional decisions. Low-dimensional decisions include selecting a microservice node to be updated, changing the number of replicas of the selected microservice, and

altering the cleaning cycle of the selected microservice. The action space is represented as: $A_t = \{i^+, i^-, \Delta N^+, \Delta N^-, \Delta T^+, \Delta T^-, 0\}$. i^+, i^- indicates that the index of the selected microservice increases or decreases by 1, $\Delta N^+, \Delta N^-$ indicates that the selected microservice increases or decreases by ΔN replica, $\Delta T^+, \Delta T^-$ indicates that the cleaning cycle of the selected microservice increases or decreases by ΔT, and 0 indicates that no action is taken. Ultimately, the maximum reward is gradually optimized in the action space to obtain the optimal security defense configuration under the current operating state.

(3) Reward: When calculating the current reward, the defense effectiveness DE defined in Equation (13) is used as the target of the DRL algorithm. By obtaining the operating state and security defense configuration of microservices, current state data is generated; combined with action selection, the defense effectiveness is calculated as the reward.

Then, based on the above model, the DQN Network in this section uses the Convolutional Neural Network (CNN) to extract the s_t state information of $3 \times n$, and then maps the output tensor of the CNN network to the action space through the fully connected network.

The solution process of the optimal security defense strategy optimization algorithm based on DQN is shown in Algorithm 1. In order to reduce the training time of the DQN model and improve the training accuracy of the model, a training strategy combining offline and online is proposed for the microservice scenario. In the first stage, the offline training phase, the model is trained through a large number of randomly generated datasets, which simulate different microservice operation states and security defense configurations. In the second stage, by using the basic parameters obtained in the first stage, the online training stage can start the optimization process more quickly. The advantage of this two-stage training strategy lies the fact in that the initial point of the online training stage is based on the already converged offline model. This not only reduces the overall training time but also improves the performance and response speed of the model in actual deployment.

Algorithm 1. Optimal Security Defense Strategy Optimization Algorithm Based on DQN.

Input: $MAGM$, User request arrival rate
Output: Neural network parameters
1. Initialization: neural network parameter θ, θ^-, the experience replay pool D, discount factor γ, learning rate α, greed coefficient ε, the number of empirical samples L, target network update step C.
2. for episode in range (STEPS) do
3. Under the initial conditions, the microservice security defense configuration $H_t = \{T_t, N_t, \lambda_t\}$ is randomly generated
4. Generate the initial microservice state vector in combination with the runtime status $S_t = \{RS_t, H_t\}$
5. $\varepsilon = \varepsilon - (\varepsilon_{\max} - \varepsilon_{\min})/M$
6. if $\varepsilon < \varepsilon_{\min}$ do
7. $\varepsilon = \varepsilon_{\min}$
8. end if
9. Randomly select an action a_t with a probability of ε, otherwise select $a_t = \max_a Q(S_t, a; \theta)$
10. Modify the defense configuration vector based on action a_t to reach the next state S_{t+1}
11. Calculate the reward r_t based on Equation (13)

Algorithm 1. *Cont.*

Store the obtained sample experience $\langle S_t, a_t, r_t, S_{t+1} \rangle$ in D
  if $|D| > batch$ do
    Collect *batch* samples from D, calculate the loss function based on Formula (14), and update the network parameters θ using the gradient descent method
  end if
  Use Equations (14) and (15) to perform the gradient descent method and update the network parameters θ
  if $|D|\%L == 0$ do
    Update the parameters of the target network $\theta^- \leftarrow \theta$
  end if
 Update the parameters of the target network $\theta^- \leftarrow \theta$
end for
Obtain the optimal microservice security defense configuration

5. Experimental Results and Evaluation

This paper first introduces the experimental environment parameters and application scenarios, then presents the strategies compared with AD2S, and finally analyzes the experimental results to verify the effectiveness and scalability of AD2S.

5.1. Experimental Environment and Parameter Settings

In the simulation experiment, the hardware configuration consists of 7 servers, each equipped with 40-core 2.5 GHz CPU, 32 GB memory, and 1 TB hard disk, among which 1 serves as the management node. The DQN algorithm is implemented based on Pytorch 1.12.1, and the language version is Python 3.9. The neural network consists of two two-dimensional convolutional layers (CNN) and two fully connected layers (FC), following the standard architecture of "convolutional feature extraction, flattening, and fully connected classification." Data are sequentially propagated through the network via feedforward connections. The detailed architecture and parameter configurations are as follows:

First convolutional layer (Conv1): The input channel is 1, and the output channel is 16. 5×5 convolutional kernel is used with padding set to 2 to preserve spatial dimensions, and a stride of 1 is applied. Given an input feature map of size $3 \times n$, the output retains the same spatial resolution, resulting in a feature map of size $16 \times 3 \times n$. This layer is followed by a Batch Normalization (BN) layer and the Rectified Linear Units (ReLU) activation function, which collectively accelerate training and enhance nonlinear representational capacity.

Second convolutional layer (Conv2): The input channel is 16, and the output channel is 3. The convolutional kernel remains 5×5, with stride 1 and padding 2, ensuring that spatial dimensions are preserved. The output feature map is of size $3 \times 3 \times n$. Similarly, this layer is followed by BN and the ReLU activation function.

The output of Conv2 is then flattened into a one-dimensional vector of length $3 \times 3 \times n$, which serves as the input to the fully connected layers.

First fully connected layer (FC1): This layer maps $3 \times 3 \times n$ input neurons to 64 output neurons. A ReLU activation function is applied after this layer to maintain nonlinearity and mitigate the risk of vanishing gradients.

Second fully connected layer (FC2): This layer takes 64 input neurons and produces 7 output neurons, corresponding to the 7 available actions. As the output layer, it applies no activation function, directly outputting the Q-values for each action.

The ReLU activation function is applied after each convolutional layer and after the first fully connected layer, and always after BN, following the standard sequence: Conv/FC,

BN, ReLU. This design normalizes feature distributions prior to nonlinear transformation, thereby improving training stability. Moreover, ReLU sets all negative values to zero, inducing sparse activations that encourage the model to focus on salient features and enhance generalization capability.

On the container cloud cluster, the microservice graph is generated based on the NetworkX 2.8.4 toolkit, and the invocation relationships between microservices are shown in Figure 1. In the simulation experiment, the traffic dataset from electronic retailers [36] was used as the request arrival rate of the microservice, and the dataset was normalized to ensure the consistency of the data range, which is conducive to the learning and prediction of the model. Define request arrival rate greater than 0.7 as high request traffic and request arrival rate less than 0.4 as low request traffic. When the request traffic arrives, the AD2S strategy optimizes and updates the microservice system configuration based on the current state. In each optimization solution, AD2S iterates multiple times to obtain effective decisions.

In order to reasonably set the model parameters of DQN, key parameter settings are made for the scheduling model. Among them, the reward discount factor γ is 0.9, the sampling *batch* is 32, the network update step L is 200, the experience replay pool D is 30,000, the greedy coefficient update step M is 10,000, the learning rate is 0.0001, $\varepsilon_{\max}$ is 1, $\varepsilon_{\min}$ is 0.1, and the risk adjustment coefficient ξ is 0.8. The target weight of microservices ω_1 and ω_2 can be dynamically adjusted according to actual business requirements. For instance, in the financial system where security is given more emphasis, it can be set to $\omega_1 > \omega_2$. In the video service system where QOS of users is given more importance, it can be set to $\omega_1 < \omega_2$. To simplify the problem, the target weights of microservices ω_1 and ω_2 are both 0.5. The set parameters can enable the AD2S algorithm to converge rapidly and obtain a relatively high reward.

5.2. Comparative Strategies

In the experiment, the proposed AD2S is compared with the Unified Configuration Strategy [9], DSEOM Strategy [20], and OADSF Strategy [21] to highlight the advantages of AD2S. Detailed information about the comparative strategies is as follows:

(1) Unified configuration strategy. This strategy utilizes a dynamic cleaning mechanism to configure security protection for microservices. However, this strategy assumes that the cleaning cycle of all microservices is the same, simplifies the optimization problem of security defense configuration, reduces the complexity of optimization, and uses the particle swarm algorithm to obtain the optimal result.

(2) DSEOM strategy. This strategy characterizes the attack difficulty of the system through the attack graph models at the application level and container level and uses the dynamic cleaning strategy to configure the security defense of microservices. However, this strategy utilizes the intermediary centrality characteristic of the graph to solve the key microservice nodes in the attack graph, and only performs security defense configuration on the key microservice nodes, while no security defense configuration is performed on other non-key microservices.

(3) OADSF strategy. This strategy characterizes the attack difficulty of microservices in the container cloud environment through MAGM and configures the security defense for all microservice nodes. Finally, the system defense effectiveness is dynamically optimized by using the DRL method. However, this strategy only considers the security after deploying the dynamic cleaning strategy, lacking the synergy of multi-dimensional defense strategies such as resource allocation strategies and the number of container replicas.

Since DSEOM and SmartSCR only update the security configuration of microservices, in the simulation experiment, the default Horizon Pod Autoscaler (HPA) method of the

Kubernetes orchestration platform is simulated to dynamically expand the number of replicas. The number of independent replicas is updated based on the resource consumption under dynamic traffic requests. Since AD2S not only takes into account the security defense strategies of microservices, but also considers the impact of different cleaning cycles on the number of replicas, which in turn affects the QOS. In the simulation experiment, the number of replicas is updated based on the coordination of multi-dimensional defense strategies such as security defense and resource allocation strategies, which is also one of the main innovation points of this paper.

5.3. Result Analysis

5.3.1. Convergence Evaluation of the DQN Algorithm

As shown in Figure 5, it is the influence of the size of the experience replay pool on the convergence of the AD2S algorithm. Among them, for each training step, DON will interact with the environment 100 times. It can be seen that the AD2S algorithm can converge at 80 steps. The size of the experience replay pool D affects the convergence stability of the AD2S algorithm. When the experience replay pool is set too small, the quintuples of the obtained experience samples are not sufficient, and some important experience samples may be lost. After convergence, it may cause fluctuations in the reward. For instance, when D = 10,000, the neural network is trained and learned based on the empirical samples obtained. When the training reached about 85 and 108 steps, the reward value after convergence fluctuated. When D = 30,000 and D = 50,000, the empirical samples filled in the neural network experience replay pool are more sufficient, and the reward after convergence tends to be stable without obvious fluctuations. When D = 30,000, the convergence speed of the algorithm is faster. Therefore, D = 30,000 is set in this paper.

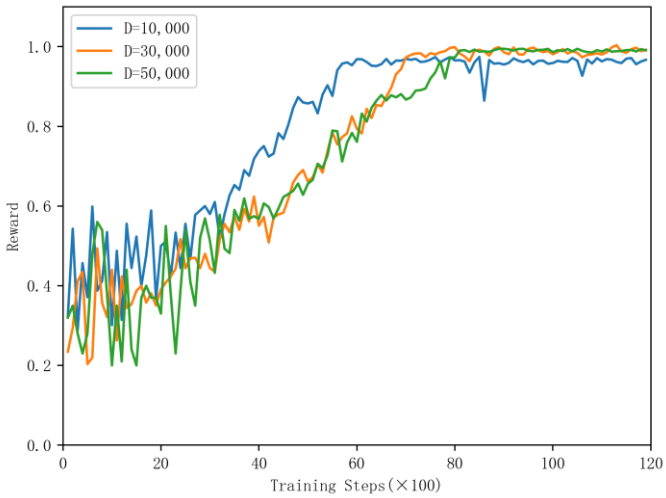

Figure 5. Impact of Experience Replay Pool on Reward.

Figure 6 shows the impact of the update step M when ε drops from ε_{max} to ε_{min} on the convergence of the algorithm. Among them, for each training step, DON will interact with the environment 100 times. It can be seen that as the greedy coefficient decreases, the neural network can converge more quickly. However, the reward M = 5000 is lower than that M = 10,000 and M = 20,000. The main reason for this is that M reflects the speed at which the neural network explores the unknown space during the training process. The faster this speed is, the higher the degree to which the algorithm utilizes the optimal action learned from empirical samples, and the faster the convergence speed, but it may converge to a suboptimal solution. The lower this speed is, the more emphasis the algorithm pays on exploring the unknown action space, the slower the convergence speed, and the higher the

reward obtained. Therefore, based on the experimental results in Figure 5, when M = 10,000 is set in this section, the convergence speed is faster, the reward value is more stable, and there is no significant difference in the reward compared with the setting of M = 20,000.

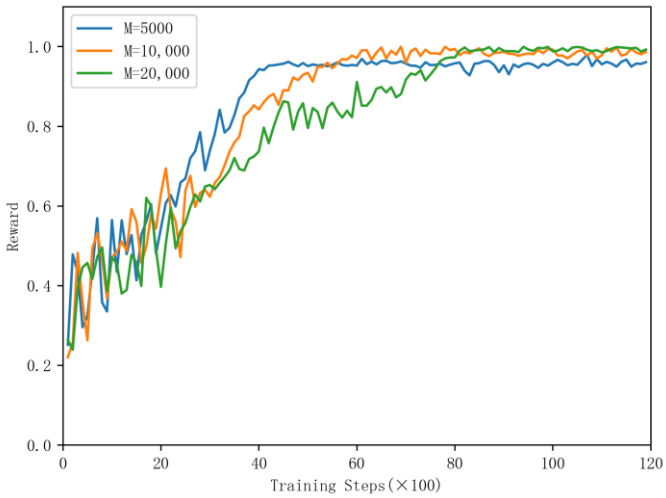

Figure 6. Impact of Descending Steps ε on Reward.

5.3.2. Defense Effectiveness Evaluation of AD2S

To measure the effectiveness of AD2S on the dynamic security protection strategy for microservices, Figure 7 shows the security performance of the system when configuring the dynamic security defense for microservices under different strategies. Each group of experiments was independently repeated 10 times, with a confidence interval set at 0.95. The horizontal axis represents the arrival rates of dynamic traffic requests, and the vertical axis represents the relative security. By comparison, it can be seen that AD2S improves the overall security performance by an average of 13.7% and 38.2%, with standard deviations of 2.1% and 4.1%, respectively under low traffic requests and high traffic requests compared with other security protection strategies, indicating that AD2S can effectively enhance the security defense capabilities of microservices. As can be seen from Figure 6, first of all, under low traffic requests, several strategies can all demonstrate relatively good security performance. This is because under low traffic requests, microservices can reduce the dynamic cleaning cycle, improve the security performance, and at the same time, it is easier to meet the delay constraints. At this time, the system has relatively good security performance.

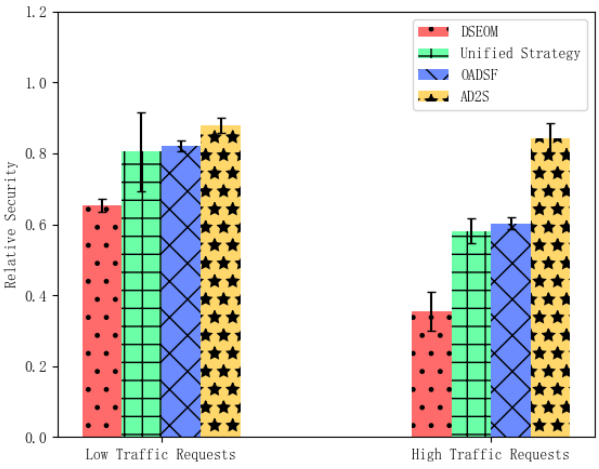

Figure 7. Comparison of Relative Security Under Different Strategies.

Under high traffic requests, the security defense capabilities of strategies such as OADSF, unified configuration, and DSEOM for microservices have significantly declined. Among them, the DSEOM strategy has the lowest security performance. The main reason is that the DSEOM strategy determines the key microservices by calculating the centrality of the mediation and only conducts dynamic defense for the key microservices. Compared with the DSEOM strategy, OADSF has higher security performance. The main reason is that this strategy depicts the attack path and difficulty of microservices from the application level and container level through the attack graph model and configures security defense for all microservice nodes in the attack graph. Finally, the effective defense of the system was dynamically optimized by using the DRL method, which improved the overall security performance of the system. AD2S can still maintain good security performance. This is because this strategy makes comprehensive decisions on the resource allocation, the number of replicas, and dynamic cleaning strategies, etc., to achieve the synergy of security defense strategies and resource allocation strategies. Therefore, under the condition of dynamic traffic requests, this strategy can combine resource allocation and security defense strategies to enhance the security defense capabilities of the system to the greatest extent while ensuring QOS.

To measure the impact of AD2S on QOS, Figure 8 shows the comparison of response delays of system microservices under different strategies. Each group of experiments was independently repeated 10 times, with a confidence interval set at 0.95. The horizontal axis represents the arrival rates of dynamic traffic requests, and the vertical axis represents the response delays of microservices. By comparison, it can be seen that AD2S reduces the response delay by an average of 7.6% and 41.6%, with standard deviations of 5.5% and 4.3%, respectively, under low traffic requests and high traffic requests compared with other security strategies. This indicates that AD2S can enhance the active security capabilities of microservices while effectively reducing the impact on the QOS.

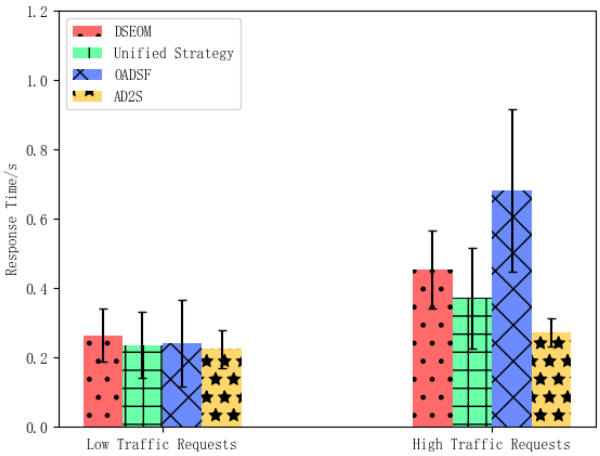

Figure 8. Comparison of Response Time Under Different Strategies.

As can be seen from Figure 8, the response delays of all strategies are relatively similar under low traffic requests. This is because when the traffic requests are small, the user request can be completed under the condition of response constraints, and at this time, the system has a relatively small response delay. Under high traffic requests, the response delay of OADSF is relatively high. The main reason is that when deploying security defense strategies and the operational status of microservices changes, the system uses the default scaling method of the Kubernetes platform to update the number of microservice replicas. However, when the high traffic requests change dynamically, the number of available replicas is insufficient. The response time exceeds the service constraint

conditions, resulting in the current microservice resource configuration and security defense strategy not being effectively matched, thus the response delay is relatively higher. AD2S can still maintain a relatively low response latency. This is because when this strategy allocates resources for microservices, it comprehensively considers the impact of changes in the number of replicas and dynamic cleaning strategies on system security performance and QOS. Therefore, under dynamic traffic requests, by coordinating security strategies with resource allocation strategies, the security performance can be improved simultaneously. Effectively ensure the QOS of the system.

To measure the real-time defense effectiveness of AD2S on the microservice system, Figure 9 shows the overall defense effectiveness of the system when microservices are dynamically configured under different strategies. Each group of experiments was independently repeated 10 times, with a confidence interval set at 0.95. The horizontal axis represents the arrival rates of dynamic traffic requests, and the vertical axis represents the defense effectiveness of the system. By comparison, it can be seen that the defense effectiveness of DSEOM is the lowest. This is mainly because it selects key microservices based on the centrality of the intermediary in the graph. Attackers may select non-key services for attack, resulting in a lower defense effectiveness of the system. OADSF protects all microservices, but it ignores the impact of the cleaning cycle and the number of replicas on QOS. With the arrival of high traffic requests, the defense effectiveness declines to a certain extent. The advantages of AD2S are more obvious. The main reason lies the fact in that when AD2S strategies quantify security performance, they fully consider the attack paths at multiple levels such as the application layer and container layer and combine the probability relationship of successful intrusion by attackers. At the same time, when quantifying the QOS considering that the dynamic cleaning strategy will affect the number of available microservice replicas, the defined defense effectiveness is made more precise. Finally, by combining the traffic requests of microservices and using the DRL strategy to solve the optimal strategy in real time, it can better deal with the actual attack scenarios. Since the unified configuration strategy assumes that all microservices have the same cleaning cycle, when dealing with large-scale microservices, there is a possibility of falling into a local optimum when using particle swarm to solve for the optimal defense configuration.

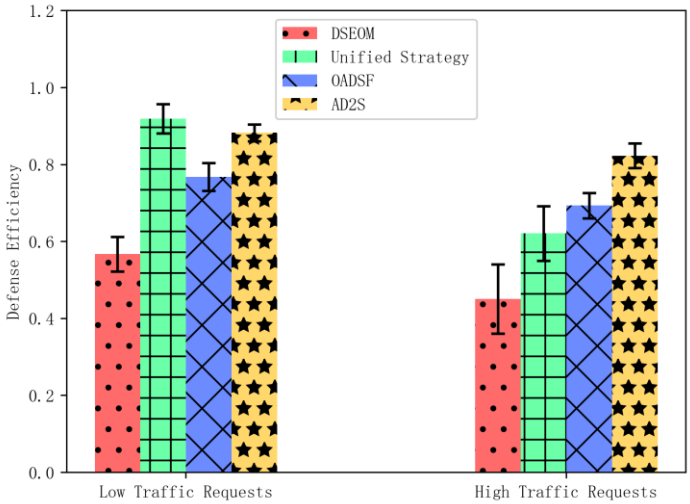

Figure 9. Comparison of Defense Effectiveness Under Different Strategies.

Furthermore, under high traffic requests, the defense effectiveness of the unified configuration strategy is inferior to that of the AD2S strategy. The latter can interact with the agent during the training process and more fully the experience decision samples, thereby demonstrating better defense effects in practical applications. The AD2S strategy

proposed in this paper takes into account both the security gain of microservices and the QOS, quantitatively and precisely describing the security of microservice systems. After convergence, the average defense effectiveness is already very close to the optimal strategy, and it can achieve a stable security defense effect. Compared with the DSEOM strategy and the OADSF strategy, the defensive effectiveness of AD2S has increased by 34.38% and 10.28%, with standard deviations of 2.4% and 3.2%, respectively.

Figure 10 shows the impact of ω_1 and ω_2 on the system's defense effectiveness in the AD2S strategy. Each group of experiments was independently repeated 10 times, with a confidence interval set at 0.95. The influence of parameters ω_1 and ω_2 on the system's defense effectiveness was analyzed. When ω_1 is 0.2, the system's requirements for security performance are relatively low, the dynamic cleaning cycle T_i can be adjusted to a larger value, and QOS can be completed under the constraint conditions. When ω_1 is 0.5, the system's security performance and QOS are relatively balanced, with a certain improvement in security performance, while the QOS remains basically unchanged. When ω_1 is 0.8, the system has relatively high requirements for security performance. The dynamic cleaning cycle T_i is set to a smaller value. However, the dynamic cleaning cycle will directly reduce the number of available microservice replicas, and QOS violates the constraints. The AD2S strategy fully considers this factor and dynamically expands or shrinks the number of replicas, thus slightly improving QOS. The defense effectiveness has not changed significantly. In practical application scenarios, the value of security performance weights can be adjusted according to QOS requirements to maximize security performance while meeting delay constraints.

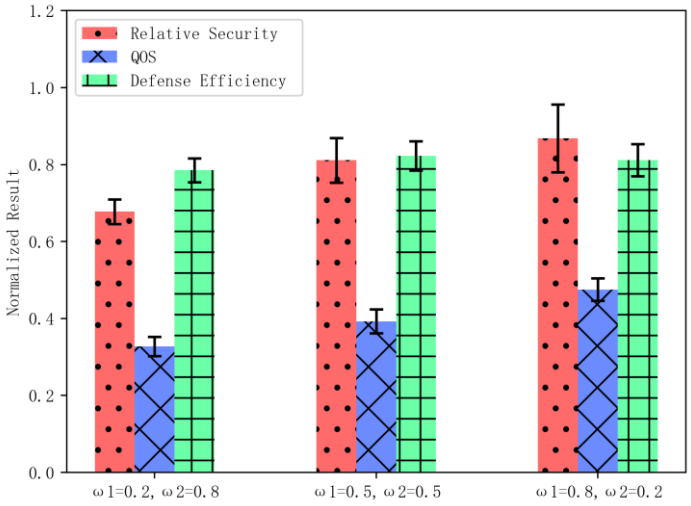

Figure 10. Comparison of Defense Effectiveness Under Different ω_1 and ω_2.

5.3.3. Usability and Scalability Evaluation of AD2S

Figure 11 depicts the usability and scalability of the strategy. By increasing the number of replicas and simulating dynamic traffic requests, the decision time for different strategies to generate system configurations under each dynamic traffic requests are recorded during the simulation process. The horizontal axis represents the number of replicas in the application, and the vertical axis represents the time consumption to solve the security defense configuration. In the experimental setup, the scale of the entire application is changed by altering the number of microservices in the application. By comparison, it can be seen that the unified configuration strategy takes the most time. When the application scale increases to 300, the solution time increases significantly. Therefore, it is suitable for small-scale application scenarios where the traffic requests are relatively small. The time consumption of DSEOM is the shortest. As the application scale increases, the

solution time does not increase significantly. This is because it only needs to execute the dynamic cleaning strategy on the key nodes in the attack graph, with a small computational scale and a fast solution speed. OADSF implements a dynamic cleaning strategy for all microservices, which slightly increases the solution time compared to DSEOM. Finally, the AD2S strategy slightly increases the time. The main reason is that when AD2S solves the security defense strategy, it simultaneously considers the cleaning cycle and QOS, making the problem solution more complex. The AD2S strategy reduces the dimension of the action space, optimizes the training model, and effectively reduces the problem solution time. As the traffic requests increases, the solution time shows a linear growth trend compared with the DSEOM and OADSF strategies and is capable of handling larger-scale application scenarios.

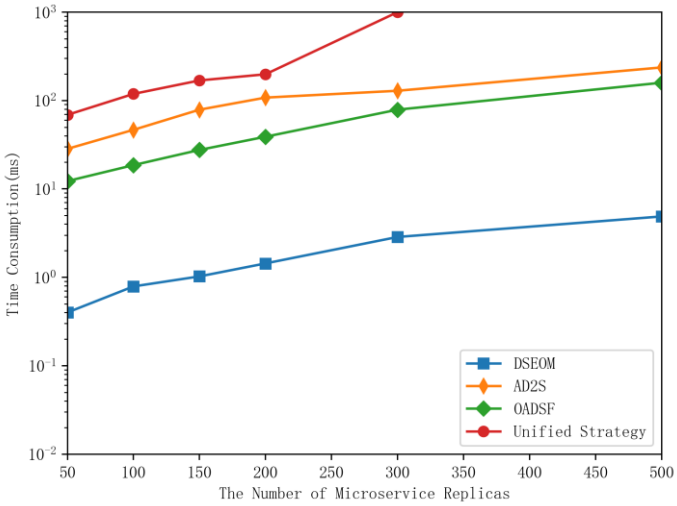

Figure 11. Comparison of Usability and Scalability.

6. Conclusions

This paper proposes the AD2S strategy to address the issue that it is difficult to balance security defense and QOS in a dynamic cloud-native environment. Firstly, the complex attack process of microservices was analyzed, MAGM was constructed, and security threats were extracted from multiple dimensions. By using the attack graph and queuing theory, the relationship among security performance, QOS, cleaning cycle, and the number of replicas was constructed, taking into account both security performance and QOS, and the defense effectiveness was quantitatively described. Then, in order to achieve the dynamic optimization of the microservice defense strategy, an adaptive defense framework was designed, which includes a state monitoring module, a security policy deployment module, and a dynamic defense strategy optimization algorithm based on DRL. Through collaborative work, the optimal cleaning cycle and the number of replicas of the microservices at the current moment are updated in real time. Finally, the experimental results show that, compared with the existing security defense strategies, AD2S can provide an effective system security policy update solution in dynamic traffic requests scenarios. While taking into account the QOS, the defense effectiveness of the AD2S strategy increased by 34.38% and 10.29%.

However, this strategy framework still has the following problems: (1) considering the security configuration under different cleaning cycles, it will inevitably introduce time consumption; (2) this excludes some threats such as insider attacks and supply chain compromises; and (3) integration with the existing system. For future work, consider the following: (1) use different MTD strategies for security configuration for different attack types to further enhance security and reduce time consumption; (2) extend the attack graph

to include privileged internal entities, enhance state representations with software supply chain integrity metrics; and (3) verify in the actual production environment and further visualize the MAGM model.

Author Contributions: Conceptualization, Y.L. (Yuanbo Li) and Y.L. (Yuanmou Li); methodology, Y.L. (Yuanbo Li) and G.W.; software, Y.L. (Yuanbo Li) and H.H.; validation, Y.L. (Yuanbo Li) and H.H.; formal analysis, Y.L. (Yuanbo Li) and Y.L. (Yuanmou Li); investigation, Y.L. (Yuanbo Li) and G.W.; data curation, G.W.; writing—original draft preparation, Y.L. (Yuanbo Li); writing—review and editing, Y.L. (Yuanmou Li) and G.W.; supervision, Y.L. (Yuanmou Li), G.W. and H.H.; funding acquisition, H.H. All authors have read and agreed to the published version of the manuscript.

Funding: This work has been partly supported by the Science and Technology Research Project of Henan Province (Grant Nos. 242102210127 and 252102231010), the National Natural Science Foundation of China (Grant No. 62176113), and the Key Scientific Research Project of Higher Education Institutions in Henan Province (Grant No. 25A520045).

Data Availability Statement: The data presented in this study are available upon request from the corresponding author. The data are not publicly available due to privacy concerns.

Acknowledgments: The authors thank the reviewers for their valuable comments and suggestions.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Zhou, X.; Peng, X.; Xie, T.; Sun, J.; Ji, C.; Li, W.H.; Ding, D. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *IEEE Trans. Softw. Eng.* **2021**, *47*, 243–260. [CrossRef]
2. Abgaz, Y.; McCarren, A.; Elger, P.; Solan, D.; Lapuz, N.; Bivol, M.; Jackson, G.; Yilmaz, M.; Buckley, J.; Clarke, P. Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review. *IEEE Trans. Softw. Eng.* **2023**, *49*, 4213–4242. [CrossRef]
3. Al-Doghman, F.; Moustafa, N.; Khalil, I.; Sohrabi, N.; Tari, Z.; Zomaya, A.Y. AI-Enabled Secure Microservices in Edge Computing: Opportunities and Challenges. *IEEE Trans. Serv. Comput.* **2023**, *16*, 1485–1504. [CrossRef]
4. Al Qassem, L.M.; Stouraitis, T.; Damiani, E.; Elfadel, I.M. Containerized Microservices: A Survey of Resource Management Frameworks. *IEEE Trans. Netw. Serv. Manag.* **2024**, *21*, 3775–3796. [CrossRef]
5. Gao, X.; Steenkamer, B.; Gu, Z.S.; Kayaalp, M.; Pendarakis, D.; Wang, H.N. A Study on the Security Implications of Information Leakages in Container Clouds. *IEEE Trans. Dependable Secur. Comput.* **2021**, *18*, 174–191. [CrossRef]
6. Khan, M.G.; Taheri, J.; Al-Dulaimy, A.; Kassler, A. PerfSim: A Performance Simulator for Cloud Native Microservice Chains. *IEEE Trans. Cloud Comput.* **2023**, *11*, 1395–1413. [CrossRef]
7. Kumar, M.; Dubey, K.; Pandey, R. Evolution of Emerging Computing paradigm Cloud to Fog: Applications, Limitations and Research Challenges. In Proceedings of the 2021 11th International Conference on Cloud Computing, Data Science & Engineering (Confluence), Noida, India, 28–29 January 2021; pp. 257–261.
8. Arouk, O.; Nikaein, N. Kube5G: A Cloud-Native 5G Service Platform. In Proceedings of the GLOBECOM 2020—2020 IEEE Global Communications Conference, Taipei, Taiwan, 7–11 December 2020; pp. 1–6.
9. Bardas, A.; Sundaramurthy, S.C.; Ou, X.; Deloach, S. MTD CBITS: Moving Target Defense for Cloud-Based IT Systems. In Proceedings of the 22nd European Symposium on Research in Computer Security, Oslo, Norway, 1–3 December 2017; pp. 167–186.
10. Nife, F.N.; Kotulski, Z. Application-Aware Firewall Mechanism for Software Defined Networks. *J. Netw. Syst. Manag.* **2020**, *28*, 605–626. [CrossRef]
11. Li, J.; Zhou, H.; Wu, S.; Luo, X.; Wang, T.; Zhan, X.; Ma, X. {FOAP}:{Fine-Grained}{Open-World} android app fingerprinting. In Proceedings of the 31st USENIX Security Symposium (USENIX Security 22), Boston, MA, USA, 10–12 August 2022; pp. 1579–1596.
12. Ni, T.; Lan, G.; Wang, J.; Zhao, Q.; Xu, W. Eavesdropping mobile app activity via {Radio-Frequency} energy harvesting. In Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23), Anaheim, CA, USA, 9–11 August 2023; pp. 3511–3528.
13. Yin, S.; Morvan, F.; Martinez-Gil, J.; Hameurlain, A. MTD-DS: An SLA-Aware Decision Support Benchmark for Multi-Tenant Parallel DBMSs. *IEEE Trans. Knowl. Data Eng.* **2025**, *37*, 2743–2755. [CrossRef]
14. Soussi, W.; Gur, G.; Stiller, B. Moving Target Defense (MTD) for 6G Edge-to-Cloud Continuum: A Cognitive Perspective. *IEEE Netw.* **2025**, *39*, 149–156. [CrossRef]

15. Meier, R.; Tsankov, P.; Lenders, V.; Vanbever, L.; Vechev, M. NetHide: Secure and practical network topology obfuscation. In Proceedings of the 27th USENIX Conference on Security Symposium, Baltimore, MD, USA, 15–17 August 2018; pp. 693–709.
16. Tunde-onadele, O.; Lin, Y.; Gu, X.; He, J.; Latapie, H. Self-Supervised Machine Learning Framework for Online Container Security Attack Detection. *Acm Trans. Auton. Adapt. Syst.* **2024**, *19*, 1–28. [CrossRef]
17. Zhang, T.; Kong, F.; Deng, D.; Tang, X.; Wu, X.; Xu, C.; Zhu, L.; Liu, J.; Ai, B.; Han, Z.; et al. Moving Target Defense Meets Artificial-Intelligence-Driven Network: A Comprehensive Survey. *IEEE Internet Things J.* **2025**, *12*, 13384–13397. [CrossRef]
18. Santos, L.; Brito, C.; Fe, I.; Carvalho, J.; Torquato, M.; Choi, E.; Min, D.; Lee, J.-W.; Nguyen, T.A.; Silva, F.A. Event-Based Moving Target Defense in Cloud Computing With VM Migration: A Performance Modeling Approach. *IEEE Access* **2024**, *12*, 165539–165554. [CrossRef]
19. Carroll, T.E.; Crouse, M.; Fulp, E.W.; Berenhaut, K.S. Analysis of network address shuffling as a moving target defense. In Proceedings of the 2014 IEEE International Conference on Communications (ICC), Sydney, NSW, Australia, 10–14 June 2014; pp. 701–706.
20. Jin, H.; Li, Z.; Zou, D.; Yuan, B. DSEOM: A Framework for Dynamic Security Evaluation and Optimization of MTD in Container-Based Cloud. *IEEE Trans. Dependable Secur. Comput.* **2021**, *18*, 1125–1136. [CrossRef]
21. Li, Y.; Hu, H.; Liu, W.; Yang, X. An Optimal Active Defensive Security Framework for the Container-Based Cloud with Deep Reinforcement Learning. *Electronics* **2023**, *12*, 1598. [CrossRef]
22. Yadav, T.; Mallari, R.A. Technical Aspects of Cyber Kill Chain. In *International Symposium on Security in Computing and Communication*; Springer International Publishing: Cham, Switzerland, 2015.
23. Wang, S.; Ding, Z.; Jiang, C. Elastic Scheduling for Microservice Applications in Clouds. *IEEE Trans. Parallel Distrib. Syst.* **2021**, *32*, 98–115. [CrossRef]
24. Zdun, U.; Queval, P.-J.; Simhandl, G.; Scandariato, R.; Chakravarty, S.; Jelic, M.; Jovanovic, A. Microservice Security Metrics for Secure Communication, Identity Management, and Observability. *Acm Trans. Softw. Eng. Methodol.* **2023**, *32*, 1–34. [CrossRef]
25. Li, H.; Guo, Y.; Sun, P.; Wang, Y.; Huo, S. An optimal defensive deception framework for the container-based cloud with deep reinforcement learning. *Iet Inf. Secur.* **2022**, *16*, 178–192. [CrossRef]
26. Zdun, U.; Queval, P.-J.; Simhandl, G.; Scandariato, R.; Chakravarty, S.; Jelic, M.; Jovanovic, A. Detection Strategies for Microservice Security Tactics. *IEEE Trans. Dependable Secur. Comput.* **2024**, *21*, 1257–1273. [CrossRef]
27. Casola, V.; De Benedictis, A.; Mazzocca, C.; Montanari, R. Designing Secure and Resilient Cyber-Physical Systems: A Model-Based Moving Target Defense Approach. *IEEE Trans. Emerg. Top. Comput.* **2024**, *12*, 631–642. [CrossRef]
28. Li, Z.; Yu, H.; Fan, G.; Zhang, J. Cost-Efficient Fault-Tolerant Workflow Scheduling for Deadline-Constrained Microservice-Based Applications in Clouds. *IEEE Trans. Netw. Serv. Manag.* **2023**, *20*, 3220–3232. [CrossRef]
29. Chen, L.; Xia, Y.; Ma, Z.; Zhao, R.; Wang, Y.; Liu, Y.; Sun, W.; Xue, Z. SEAF: A Scalable, Efficient, and Application-independent Framework for container security detection. *J. Inf. Secur. Appl.* **2022**, *71*, 103351. [CrossRef]
30. Casalicchio, E.; Iannucci, S. The state-of-the-art in container technologies: Application, orchestration and security. *Concurr. Comput.-Pract. Exp.* **2020**, *32*, e5668. [CrossRef]
31. Priya, V.S.D.; Sethuraman, S.C.; Khan, M.K. Container security: Precaution levels, mitigation strategies, and research perspectives. *Comput. Secur.* **2023**, *135*, 103490. [CrossRef]
32. Sultan, S.; Ahmad, I.; Dimitriou, T. Container Security: Issues, Challenge and the Road Ahead. *IEEE Access* **2019**, *7*, 52976–52996. [CrossRef]
33. Zhang, S.; Guo, Y.; Sun, P.; Cheng, G.; Hu, H. Deep reinforcement learning based moving target defense strategy optimization scheme for cloud native environment. *J. Electron. Inf. Technol.* **2022**, *44*, 608–616. [CrossRef]
34. Peng, W.; Li, F.; Huang, C.-T.; Zou, X. A moving-target defense strategy for cloud-based services with heterogeneous and dynamic attack surfaces. In Proceedings of the 2014 IEEE International Conference on Communications (ICC), Sydney, NSW, Australia, 10–14 June 2014; pp. 804–809.
35. Iraqi, O.; Bakkali, H.E. Immunizer: A Scalable Loosely-Coupled Self-Protecting Software Framework using Adaptive Microagents and Parallelized Microservices. In Proceedings of the 2020 IEEE 29th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Bayonne, France, 10–13 September 2020; pp. 24–27.
36. Akamai. Akamai's 2014 Online Holiday Shopping Trends and Traffic Report. 2023. Available online: https://content.akamai.com/PG2112-Holiday-Recap-Report.html (accessed on 15 September 2025).