Red Team BERT: Reliable Automated Penetration Testing with Multi-BERT Architecture

Christophe Genevey-Metat
R&D AI / Cyber Team
NEVERHACK
Rennes, France
cgeneveymetat@neverhack.com
0000-0002-1901-626X

Guillaume Nollet
R&D AI / Cyber Team
NEVERHACK
Rennes, France
gnollet@neverhack.com
0009-0006-7101-8680

Pierre-Marie Satre
R&D AI / Cyber Team
NEVERHACK
Rennes, France
pmsatre@neverhack.com
0000-0002-8010-1885

Loïc Scotto Di Perrotolo
R&D AI / Cyber Team
NEVERHACK
Rennes, France
lscottodiperrotolo@neverhack.com
0009-0001-8614-7008

Olivier Gesny
R&D AI / Cyber Team
NEVERHACK
Rennes, France
ogesny@neverhack.com
0000-0003-2132-4875

***Abstract*—Artificial intelligence has become particularly crucial in its integration with cybersecurity applications. Numerous studies have demonstrated that reinforcement learning agents can identify optimal sequences of actions to attack a network. However, these agents are often overfitted to specific environments and struggle to generalize or adapt to networks that differ from those encountered during training. Moreover, most of these agents rely on complex abstract architectures which lack interpretability in their decision-making processes. In this work, we propose a novel agent architecture that integrates four transformer-based BERT models. This architecture can adapt to any given network, is robust to changes in parameters and objectives, and requires no additional training phase to attack unseen networks. Additionally, our model is interpretable using SHAP values. We introduce a specific context tailored to this architecture and reuse an evaluation metric we developed in our previous work that more accurately reflects an agent's ability to attack a network without prior knowledge. Finally, we discuss the limitations of our approach and outline future directions to facilitate its deployment in real-world applications.**

***Index Terms*—pentesting automation, attack simulation, zero-shot classification, transformers, large language model, adaptability, robustness, explainability**

I. Introduction

Pentesting consists of discovering and exploiting weaknesses within a network, in order to help a company identify vulnerabilities present in their information system. In the context of research, pentesting is always studied as an optimization problem whose objective is to compromise a target with a minimum sequence of actions. Most agents are trained using reinforcement learning, and they choose unit attacks to reach and compromise the target. Many researchers [1], [4]–[7], [9]–[11] have studied RL agents to solve pentesting problems, and several simulators have been developped to help the community train their own RL agents. However, in a real-world application, pentesting is usually more complex than an optimization problem, because network topologies differ across companies: a sequence of actions that works for one organization may fail for another due to specific configurations. Thus an AI agent must explore and discover information before exploiting it. This discovery of information results in extra actions being performed instead of blindly learning the path towards targets.

In our previous work [2], we presented a zero-shot LLM-based agent for automated network penetration testing, offering strong adaptability without retraining. We introduced a novel evaluation metric that rewards informed decision-making over trial-and-error. The model outperformed classical RL agents (DQN, PPO, CLAP) in unseen NASim [8] environments of small and moderate sizes using textual observations. Preliminary SHAP-based analysis provided insight into the agent's decision-making process. Overall, this served as a promising step towards explainable and robust AI for red teaming.

In this paper, we extend our approach by combining several BERT models to build a complete pipeline that determines, at each point in time, the machine one must target and the action that needs to be taken. First, we construct various datasets derived from the NASim environment, with dynamically generated network topologies. We train our BERT models separately: one to predict the correct machine given a textual representation of the network; one to predict the appropriate action based on the selected machine; and two to predict eventual parameters of the action. Each textual representation is distinct and tailored to its respective decision-making task. After training the four BERT models, we develop a sequential pipeline to execute them in inference mode. This pipeline is then used to predict a full sequence of actions on unseen NASim topologies. To improve generalization, we also implement a rollout mechanism and a padding system, allowing the pipeline to adapt to new topologies that may be smaller or larger than those encountered during training.

Finally, we compare our approach to another reinforcement learning-based method called CLAP [10], and our previous work on zero-shot decision making. We demonstrate that our pipeline outperforms these two RL agents and shows much stronger generalization capabilities. We also provide explainability results on our BERT-based architecture and compare them with our initial explainability work on the zero-shot model.

The paper is organized as follows: Section II presents related works on RL agents for pentesting. Section III introduces the environment and data used in our approach. Section IV presents our four BERT architectures for machine, action, and parameter selections, and the specific contexts of each of them. Section V presents our experiment on the explainability obtained after training. Section VI details the performance and results of our pipeline in an RL environment and compares them to other RL agents. Finally, Section VII concludes with future improvements and works to extend our BERT-based system.

II. Related works

The research community has focused on two main areas: generating environments for scenario simulation and emulation, and designing models for efficient real-life deployment.

Famous environments for automating red team exercises include NASim [8] and NASimEmu [5]. NASim (literally "Network Attack Simulator") simulates an information system that is composed of one or multiple subnetworks, each containing a specific number of host machines. Each host has an operating system and exposes services and processes that may be scanned and targeted to gain privileged access to the host. The main goal of NASim scenarios is to gain full access on specific hosts, which are called targets. Firewalls impose restrictions on both inter-subnet traffic and Internet connectivity. NASimEmu builds on the NASim environment to support emulation, and may also generate scenarios with a given level of diversity by changing the path required to reach the target. Other works for environment simulation include the GAP model [11] introduced by Zhou et al. in 2025, which explores domain randomization for scalable scenario generation and meta-reinforcement learning to enable few-shot adaptation. A Large Language Model is used to perform domain randomization and generate diverse environments that are closer to real-world network configurations.

Most of the literature adapts various reinforcement learning methods to the problem of automated pentesting, but most of them show weaknesses regarding generalization difficulties. For instance, in 2022, Yan et al. presented CLAP [10], a variant of the PPO algorithm that can handle multi-objective reinforcement learning in a pentesting context. CLAP uses a cover mask mechanism that allows the model to keep track of previous actions performed in the past. Though the authors show that their model quickly converges to the optimal sequence over the three scenarios during the training phase compared to other algorithms (DQN, Improved-DQN, HA-DQN), they do not show the performance during an evaluation phase. The same goes for AutoRed [3], presented in 2024 by Hasegawa et al., which utilizes Graph Neural Networks (GNNs) to encode the dynamic structure of the target network. This approach was not evaluated on previously unseen topologies, making it unclear whether the agent can generalize effectively to new network structures.

To tackle this challenge of generalization, Yan et al. introduced SetTron [9], which uses randomly-rearranged state representations that include information about each host as well as actions performed on them. SetTron is more flexible than CLAP in that it maintains good performance if one changes the location of the target in the network on which SetTron has been trained. However, the adaptability of the model for truly unseen network topologies has not been demonstrated by the authors. Another approach [7], presented by Nyber et al. in 2024, goes a bit further: a message-passing neural network (MPNN) is trained with multiple agents using reinforcement learning. Evaluated in the CAGE 2 environment, it achieves non-trivial performance on scenarios which it has not been trained on, although this performance remains inferior to Multi-Layered Perceptrons (MLP) trained specifically on these scenarios.

Finally, a certain amount of papers defend the use of Large Language Models (LLMs). On the defensive side, Huan et al. presented in 2024 a two-step LLM called PenHeal [4], that first identifies multiple vulnerabilities in a system, and then suggests optimal remediation strategies. In 2025, Kim et al. introduced a model called CyberAlly [6], which aims to provide real-time, context-aware support for cybersecurity professionnals through textual clues. On the more offensive side, in 2024, Deng et al. introduced PentestGPT [1], a multi-agent LLM planner that automates three key red team processes: planning, execution, and result interpretation, using one LLM for each. The three modules work together to guide the user during the penetration testing process. Finally, in our previous work [2], we introduced a zero-shot LLM-based approach for automated pentesting, which outperformed state-of-the-art RL methods like DQN, PPO, and CLAP on NASim benchmarks and also required no retraining to generalize across unseen topologies. A new evaluation method rewards the discovery of useful information for exploits and privilege escalations, thus promoting better decision-making. Early explainability results using SHAP highlight its potential for interpretable red teaming.

III. Original data, featuring and actions

A. General concept

Selecting an attack target requires enumerating available machines with their properties and predicting the most urgent one. The same principle applies to action selection: given the properties of all available machines and the list of available actions, the most critical action must be predicted. Note that a third choice may be required for actions that require specific parameters (such as exploiting a particular service or escalating privileges on a specific process). In that case, the context of the machine, the chosen action, and the list of

available parameters is given for that final choice. These three choices lead to the definition and training of three separate BERT models.

NASim is ideal for handling some real-world complexities (e.g. firewall restrictions, non-deterministic actions), even though its overall complexity remains much lower than that of an actual real-world application. In the case of NASim, the information required to determine which machine should be attacked encompasses the IP address, role, current access, observed OS, observed processes and observed services of every machine. It is easy to extract this information from the NASim environment and print it sequentially.

To be more precise, the NASim environment stores at each point in time a list of machines, each identified by IP address and characterized by the following: operating system, available services and processes, and status indicators for discovery, reachability, sensitivity classification, and our current level of compromise. We have augmented it with indicators showing whether subnet scans have been launched from that machine.

Once a machine has been selected, the appropriate action to perform on the target machine must also be selected from the various action types that NASim supports:

- Simple actions to collect more information: OS scan, process scan, service scan, subnet scan
- Complex actions aimed at gaining restricted access on a machine: vulnerability exploit, privilege escalation

Complex actions need parameters in order to specify what type of attack we want to execute. For vulnerability exploit-type attacks, these parameters consist of a required service, an optionally required OS, as well as the access granted by the attack. For privilege escalation type attacks, the parameters consist of a process, the given access, and optionally an operating system.

B. Abstraction methods

Pentesters are primarily concerned with actionable vulnerabilities rather than exhaustive system analysis, which implies cross-referencing the vulnerabilities that stem from the processes, services and operation systems of the machine. For this reason, our transformer focuses on linking relevant concepts rather than understanding domain-specific details. For instance, let's imagine an exploit that uses the FTP service in a Linux OS to grant the User access to the attacker. Though this representation appears meaningful, a model only needs to understand that an exploit that uses a service called X, an OS called Y to grant access Z is relevant to be used if the target machine uses service X on OS Y, and access Z has not yet been granted on it.

For this reason, and in order to get results that may be more generalizable and less domain-specific, a conversion is applied to most domain-specific words in the source text at random. "Linux" and "Windows" may be converted to "OS1" and "OS2", respectively, or the other way around. Similarly, services "FTP", "SSH" and "HTTP" may become any of "SERVICE1", "SERVICE2" and "SERVICE3" in any order. The key requirement is that within any given sentence, each service is represented consistently and distinctly from other services.

Sentence	Meaning
IP2 STATUS1 ROLE1 OS1 SERVICE2 SERVICE3 PROCESS0	The machine with IP n°2 is not critical (ROLE1) and has not been attacked yet (STATUS1). It uses operating system 1, houses services 2 and 3, and its processes are not yet known. Machine selection consists of many such sentences separated by commas.
ROLE1 ACCESS0 OS0 SERVICE2 PROCESS0 | PRIVILEGE, OSSCAN, EXPLOIT, DISCOVERY, PROCESSSCAN	The machine we target is not critical (ROLE1), does not have any privileged access (ACCESS0), and we don't know its operating system or processes. We do know that it houses service 2. The five actions listed after the pipe may be launched. This sentence is used for action selection.
ACCESS1 OS1 SERVICE5 SERVICE2 PROCESS2 | PRIVILEGE | PARAM4 PROCESS3 ACCESS2, PARAM1 PROCESS2 ACCESS2	Considering the machine with given characteristics and the fact that a privilege escalation action is about to be launched, two parameters are available: an escalation using process 3, another using process 2. Both grant access 2. This sentence is used for parameter selection.

TABLE I
INPUT SENTENCE EXAMPLES FOR MACHINE, ACTION AND PARAMETER SELECTION.

Table I describes the three types of sentences or propositions that are used in our classification processes.

Of the three classification problems, the one whose inputs tend to be the longest is the machine classification problem. Indeed, if one needs to differentiate between four different machines, one needs to describe all four machines fully. Note that, with this kind of convention, it is possible to identify a machine using either its position in the list (select the fourth machine in the list), or its identifier (select the machine with IP16 - we use the term "IP" loosely here to refer to the machine identifier). In the following section, when defining the transformer architecture for machine selection, we compare the results of our architecture in both cases and point out the advantages and drawbacks of each of them. The same is done for parameter selection.

C. Dataset generation

In our study, we generated both synthetic datasets (generated using hard-coded rules) and simulated datasets (generated by running NASim on different networks and saving the observed environment at each step). The results shown in sections IV and VI only use simulated datasets, and no synthetic dataset has been used to train nor test the models we present in this paper.

In order to generate a labeled dataset for the different parts, we design a heuristic that assigns the labels corresponding to the 'best machine', the 'best action', and the 'best parameter'.

We define the "best machine" to select using the following rules:

- first select any machine that is among the target machines of the network,

- then select any machine which has been compromised but not fully exploited (a machine is considered fully exploited once a subnet scan has been executed from it. For target machines in the network, root access needs to have been obtained as well),
- then select any machine which has not yet been compromised.

Once a machine has been selected, we define the "best action" using the following rules:

- scan the machine OS and services if the machine is not yet compromised,
- launch an exploit if all the information needed to launch one is discovered,
- launch a process scan if the machine is a target and is compromised,
- launch a privilege escalation if the machine is an already compromised target machine and processes have been scanned,
- launch a subnet discovery if the machine is fully compromised (for targets) or compromised in any way (for non-targets).

The selection of parameters ensures consistency by choosing exploits that match the target's operating system and use the services available on the machine. When multiple compatible exploits are available, the selection prioritizes the one that grants the highest privilege level. A similar reasoning is used for privilege escalation actions.

We execute this heuristic on random NASim scenarios that have a number of hosts equal to 4, and a number of services equal to 4.

In Table II, we summarize the number of challenges executed using the heuristic in order to generate the datasets. We also summarize the amount of labels inside each dataset.

TABLE II
NUMBER OF CHALLENGES USED TO MAKE DATASETS

Dataset For	Train	Valid	Test	Labels
Machine selection	100K	50K	5K	10
Actions selection	100K	50K	5K	6
Param selection (exploit)	50K	50K	5K	4
Param selection (privilege)	100K	50K	5K	2

IV. TRANSFORMER DEFINITION AND RESULTS

Since all three problems (machine selection, action selection, and parameter selection) rely on textual input representations, transformer architectures are particularly well-suited to solve them. We train four distinct transformer models for this purpose:

- one for machine selection
- one for action selection
- one for parameter selection in the case of an exploit
- one for parameter selection in the case of a privilege escalation

Each transformer is trained using the labeled data from the aforementioned databases. During training, we constrain our dataset to randomly generated NASim networks with 4 hosts. Generalization to larger networks is explored in section VI. This training proves highly effective, with every transformer achieving 100.0% accuracy on both validation and test datasets.

Table I details the inputs for each transformer. The final entry regarding exploit and privilege escalation parameters applies to both parameter selection transformers: one processes examples containing "EXPLOIT" keywords, while the other handles examples with "PRIVILEGE" keywords.

TABLE III
HYPERPARAMETER COMPARISON BETWEEN THE FOUR BERT MODELS AND THE ZERO-SHOT ARCHITECTURE.

Parameter	BERT	Zero-shot
Epochs	[40, 10, 40, 40]	None
Batch Size	[32, 32, 32, 32]	None
Vocab size	[37, 45, 29, 35]	50,265
Parameters	[486K, 486K, 495K, 495K] = 1.97M	407M

In Table III, we summarize some of the hyperparameters of our four BERT models (machine selection, action selection, "EXPLOIT" parameter selection and "PRIVILEGE" parameter selection respectively), and compare them to our previous work based on a zero-shot model [2]. When comparing the two approaches, we observe that our four BERT models combined have a much smaller amount of parameters than that of the zero-shot model, resulting in a more lightweight architecture.

In the next section, we demonstrate that our model provides more consistent interpretability compared to the zero-shot model.

V. EXPLAINABILITY OF THE RESULTS

Hereafter, we present two methods for model explainability. The first is SHAP values, a game-theoretic approach to explain the influence of certain features using do-calculus. The second approach involves systematically modifying inputs in domain-relevant ways to observe our model's responses.

SHAP values quantify each input feature's contribution to the prediction using game theory. The SHAP value of a model relative to a given input feature corresponds to the expected value of the output of that model as we intervene on the input feature, and represents how much that feature influences the model's output compared to the baseline prediction. There is a set of SHAP values for each predicted class : one may both analyze the reasons why a certain output has been chosen, and why another has been rejected.

A. Machine selection

In this section, we demonstrate how role and status keywords influence our model's decision-making. To illustrate this, we created two different network contexts with four potential IPs to choose from.

In the first context, three machines share the same information (ROLE1 and STATUS1), while one has a different status

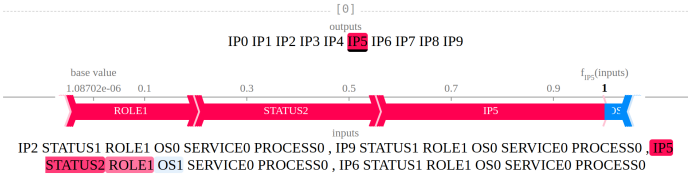

Fig. 1. Shap value of label "IP5"

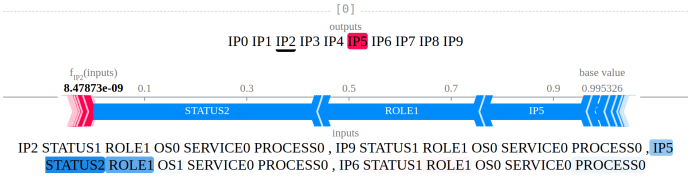

Fig. 2. Shap value of label "IP2"

(STATUS2), indicating that it is already under attack. That machine is correctly predicted (IP5) by the model. In each figure in this section, one can observe possible outputs at the top of the picture and the model input at the bottom. For a selected output (which is underlined), we can observe the contributions of each input token. Figure 1 shows that the words IP5, STATUS2 and ROLE1 from the target machine provide the strongest positive contributions to this prediction, which is consistent with human reasoning.

When examining the other predicted labels, we see little to no influence overall, except for label IP2 (Figure 2), where STATUS2 and ROLE1 from the third machine (IP5) contribute negatively. This negative contribution is also meaningful, as it reduces the confidence in IP2 when machines with higher-priority statuses are present in the input.

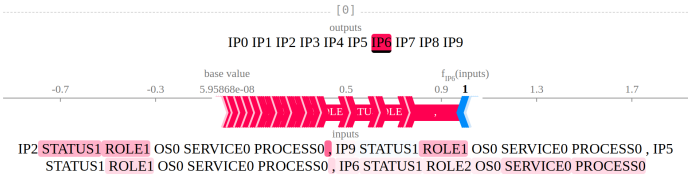

Fig. 3. Shap value of label "IP6"

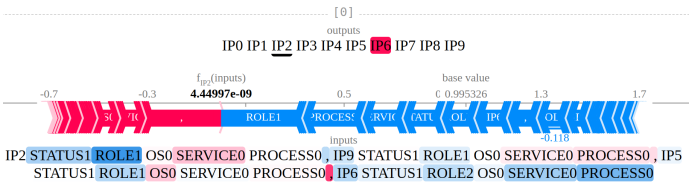

Fig. 4. Shap value of label "IP2"

The second context is quite similar: three machines share the same attributes (ROLE1 and STATUS1), and one machine also has STATUS1 (indicating no attack) but is associated with ROLE2 (indicating a potential target). The predicted label is IP6, which corresponds to the target machine. Interestingly, the SHAP values from Figure 3 indicate that the model's positive attribution is not primarily based on ROLE2, but instead on the STATUS1 and ROLE1 from the other machines in the input. This result can be interpreted as follows : if all others machines have ROLE1 but IP6 has ROLE2, then it is IP6 that must be selected. Further insights are provided in Figure 4, which presents the SHAP values for the alternative prediction IP2. We observe several negative contributions, notably from ROLE1 and STATUS1, which appropriately reduce the model's confidence in selecting IP2. This behavior is logical: when a more relevant role (ROLE2) is present elsewhere in the input, candidates with less relevant roles should receive lower confidence scores. Additionally, ROLE2 associated with the IP6 machine also contributes negatively to the prediction of IP2, reinforcing the model's preference for IP6.

We note, however, that in both Figure 3 and Figure 4, words other than role and status turn out to be important during the classification process. In particular, we observe in this context that the word ',' is one of the main positive factors in the prediction of the IP. This specific case shows us that our model's learning is not perfect, and that we must remain cautious regarding its interpretation. We verified these observations using direct intervention on inputs generated using live scenarios and looking at the response. In doing so, we noticed that downgrading the role and status of the selected machine to role 1 (unimportant) and status 1 (unattacked), when there are other machines with a better role-status combination, changes the prediction of the model 69.4% of the time. Had the model only used role and status to make its decision, we would have expected a 100% score instead. The difference lies in the fact that other parts of the input data are used, most probably because of the important correlation that happens in practice between the status and the information we have access to (OS, services and processes).

B. Action selection

This section demonstrates how operating system and services features influence the action selection process. For this demonstration, we constructed a context with a maximum of six selectable actions.

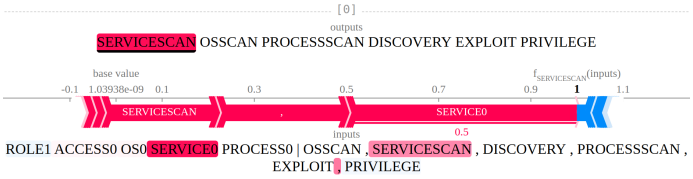

Fig. 5. Shap value of label "SERVICESCAN"

In this scenario, our model does not have any type of information regarding the host: OS0, SERVICE0 and PROCESS0 indicate that the OS, services and processes are unknown. For this reason, the predicted action is a SERVICESCAN. In Figures 5 and 6, we observe that the most significant

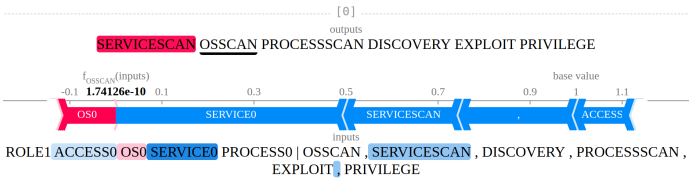

Fig. 6. Shap value of label ”OSSCAN”

contributions come from SERVICE0 and the action name SERVICESCAN itself. Since it is both feasible and necessary to execute a service scan, the model appropriately selects SERVICESCAN over other available actions. Note that, in Figure 6, the OS0 word contributes positively to the OSSCAN prediction, which makes sense since OS0 indicates that the operating system is not known. Still, the model does not predict the OSSCAN action because it is strongly negatively influenced by SERVICE0 and SERVICESSCAN, which reduce its confidence in selecting this action.

In a context different from the one previously presented, manual intervention analyses confirm that unsetting the OS or service information (turning it back to OS0 or SERVICE0 respectively) changes the chosen action 100% of the time (as getting that information is now back to being a priority).

C. Param selection

In this section, we analyze how discovered service names influence our model’s parameter selection. To illustrate this, we constructed a specific scenario where the host uses OS2 and SERVICE2, with one available exploit specifically requiring these two features.

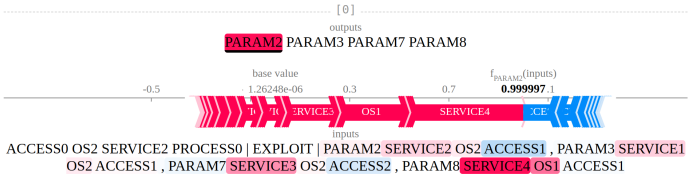

Fig. 7. Shap value of label ”PARAM2”

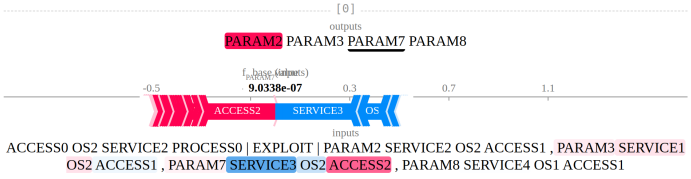

Fig. 8. Shap value of label ”PARAM7”

When examining the SHAP values for the PARAM2 prediction in Figure 7, we observe that contributions come from various features. Several unrelated elements (e.g., SERVICE4, OS1, SERVICE3) contribute positively to the prediction, whereas SERVICE2, which actually matches the service running on the target machine, is only the fourth most positively weighted feature.

To understand how the model still selects the correct parameter, we examine how it evaluates alternative parameters. Each alternative receives strong negative contributions from incompatible services. For example, Figure 8 illustrates a negative contribution from SERVICE3 when predicting PARAM7, which is expected since this parameter does not include that service. Similar patterns are observed for PARAM3 and PARAM8. We hypothesize that rather than directly selecting compatible features, the model eliminates incorrect options by identifying contradictions, ultimately choosing the only parameter that does not contradict the machine context.

D. Discussing BERT vs zero-shot

In our previous work on the zero-shot model [2], we computed SHAP values and observed that the model was often influenced by irrelevant or nonsensical tokens (such as ‘the’, ‘.’, etc.). This observation was one of the motivations for restricting the input context in our current work, where we provide only direct information that could meaningfully support the model’s decision-making process. This approach enables more logical reasoning that aligns with the input context.

VI. Live performance measure

In this section, we present the pipeline used for decision-making in the RL environment, the evaluation metric used to compare different RL agents, and the performance of our RedTeamBERT model compared to the zero-shot and CLAP models across four scenarios.

A. Sequential BERTs for decision-making

Figure 9 depicts the complete RedTeamBERT architecture during the evaluation phase, which combines the three specialized models described in the previous section. The decision-making pipeline operates sequentially: it first selects the target machine to attack, then chooses the appropriate action to perform on it, and finally, if the action requires parameters, it selects the most suitable ones to execute the action on the chosen machine. The contextual information used at each decision level is detailed in the previous section. Additionally, the pipeline includes two mechanisms designed to handle various network topologies: the ”padding system” and the ”roll system”.

1) Padding system: The padding system was developed after we observed that our RedTeamBERT model could be influenced by the length of the input context (either the machine context or the parameter context). To mitigate this issue, we introduced a padding system for both machine and parameter selection. The padding system is generally used to handle topologies that are smaller than those encountered during the training phase. It is triggered whenever the number of machines or parameters available at inference time is smaller than the number observed during training. For

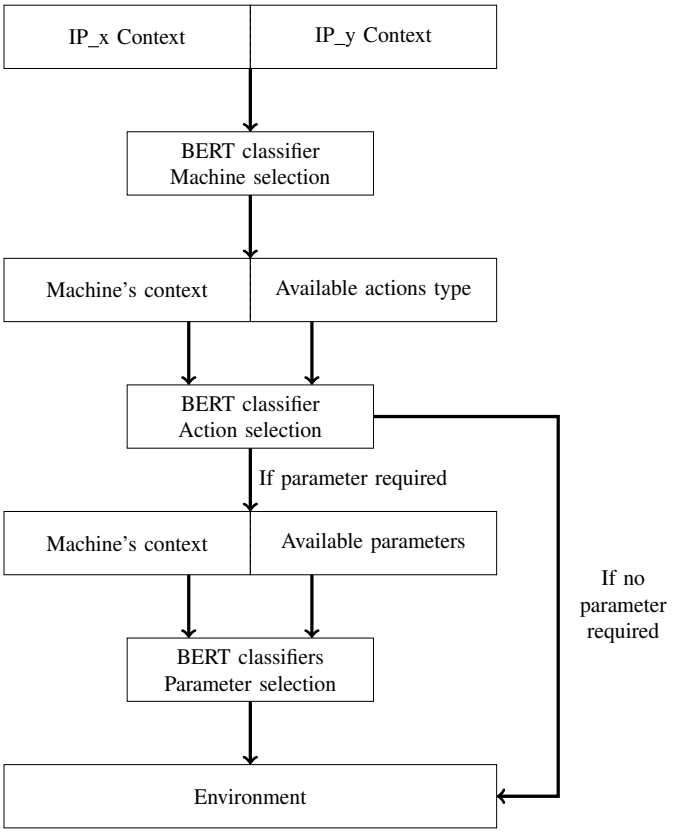

Fig. 9. Decision path taken by our model in inference mode

example, if the model was consistently trained on 4 parameters but encounters only 3 during inference, we automatically add dummy parameters to preserve the expected total of 4. This guarantees fixed-length input sequences, enabling stable decision-making by RedTeamBERT. We chose to fix the context size to 4 machines for machine selection, and 4 parameters for parameter selection. These values correspond to the maximum context sizes observed during training. This mechanism prevents the model from making suboptimal decisions due to context length mismatches between training and evaluation.

2) Roll system: The roll system was developed to enable the BERT model to handle topologies larger than those encountered during the training phase. It is triggered whenever the number of machines or parameters available at inference time exceeds the number seen during training. The roll system operates by selecting a pool of N machines (or parameters) and making an initial prediction. It then creates a new pool by retaining the top prediction and replacing the remaining N-1 items with new candidates. The model processes this new pool to generate another prediction, and this rolling process continues until all machines (or parameters) have been evaluated. In the end, we obtain a prediction that reflects the best machine (or parameter) across the entire set. In theory, this approach creates an unfair advantage for parameters appearing later in the list, since those evaluated early must survive multiple rounds where they could be incorrectly eliminated because of repeated model predictions. However, that is not a problem in our case, because we demonstrated that our BERT-based models for both machine and parameter selection achieve 100% accuracy.

The padding system and roll system can be complementary and may be triggered simultaneously. This can occur, for instance, when the topology is smaller than in the training examples but requires more exploit parameters than expected. Conversely, it may also happen when the topology is larger, but fewer parameters are available compared to those observed during training.

B. Evaluation metric

The standard performance metric for evaluating RL agents on the NASim environment is typically based on the number of actions required to compromise the target machine. However, we argue that this metric is suboptimal for evaluation purposes, as it penalizes exploratory actions aimed at gathering information about network hosts, treating them as unnecessary overhead, and instead favors direct exploitation attempts. Such behavior encourages rote learning and diverges significantly from that of a human pentester, who generally seeks to collect relevant information about a target before initiating any attack, especially in unfamiliar network environments. To address this limitation, we reuse the evaluation metric developed in our previous work [2] that explicitly encourages information-gathering behavior prior to exploitation, thereby aligning the agent's decision-making process more closely with real-world pentesting practices.

$$R_{final} = R_{standard} + \alpha * \sum_{t=1}^{n} \mathbf{1}_{\text{coh}}(a_t, c_t) \tag{1}$$

Our evaluation metric returns the classic reward plus a bonus at the end of the challenge. Equation 1 formalizes this metric, which provides a more accurate assessment of RL agent performance during the evaluation phase. The classic reward, denoted as $R_{standard}$, corresponds to the cumulative cost of all actions performed by the agent. The bonus term is defined as the amount of all actions a_t that are coherent when executed within their respective context c_t, weighted by a configurable factor α. The coherence function $\mathbf{1}_{\text{coh}} : A \times C \rightarrow \{0, 1\}$ returns 1 if all the information required for normally executing action a_t is displayed within context c_t, and returns 0 otherwise. The coefficient α compensates for the cost of information-gathering actions (e.g., os_scan, service_scan, process_scan). Agents that leverage discovery actions to guide their exploits receive a bonus reward. This factor must be set higher than 3, as compromising a machine typically requires discovering at least three attributes: operating system, services, and processes, each of which incurs a cost of -1. The metric is applied consistently across all evaluated agents (CLAP, zero-shot, and RedTeamBERT), and includes a safeguard to prevent multiple bonuses for repeated discovery of the same information.

C. Results

To evaluate our RedTeamBERT model against baselines, we used four scenarios: the tiny and small-linear scenarios already studied in our previous work with the zero-shot model, and two new scenarios, medium-test and large-test, inspired by the original medium and large scenarios. Table IV summarizes the network topology used in each scenario. We do not compare our work against PentestGPT, as the latter is not fully autonomous.

TABLE IV
NETWORK SCENARIOS USED BY OUR MODEL.

	Subnets	Hosts	OS	Services	Processes
Tiny	3	3	1	1	1
Small-linear	6	8	2	3	2
Medium-test	5	15	2	5	3
Large-test	10	27	2	5	3

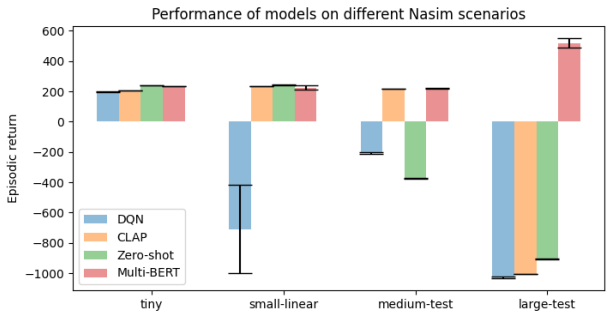

Fig. 10. Performance comparison of RL agents (DQN, CLAP, zero-shot, and RedTeamBERT) trained and evaluated with the new evaluation metric

Figure 10 shows the performance of our RedTeamBERT model compared to other RL agents (DQN, CLAP and zero-shot). We compare them across the four scenarios, using the mean episodic return computed over five runs. The episodic return represents the total reward accumulated by the agent throughout an episode, computed using the new evaluation metric introduced in the previous section. Note that the DQN and CLAP models are always trained on the scenarios on which they are tested, our model was trained exclusively on separate scenarios with four machines, and the zero-shot model is not trained on any scenario.

As shown in Figure 10, the DQN only figures out the path for the tiny scenario, the zero-shot model for the first two scenarios, and CLAP demonstrates strong performance on the tiny, small and medium scenarios only. Each time, these architectures fail when the scenario gets larger.

In addition, both DQN and CLAP are non-adaptive and are tested on the same network they were trained on. Therefore, we actually show the results of different DQN and CLAP models, each one trained on the exact scenario it is tested on. By contrast, our RedTeamBERT approach is trained on a database extracted from networks that are distinct from those used during testing. Therefore, it is the only architecture that is generalizable.

Performance-wise, our model's performance is comparable to that of CLAP and zero-shot for tiny and small scenarios, it is slightly better than CLAP for the medium scenario, and it is much better than the both of them for the large one, as it is the only one that manages to finish it.

TABLE V
RESULTS ON BIGGER SCENARIOS (5 TO 10 SUBNETS).

	DQN	CLAP	Zero-shot	Multi-BERT
Success ratio (at least once)	60%	80%	9%	100%
Success ratio (total)	39%	63%	5.5%	93.5%

We also conducted additional experiments on random scenario batches to compare the generalization abilities of each algorithm. We evaluate models on 100 random medium scenarios (5 subnets, 15 to 20 machines) and 100 random large scenarios (10 subnets, 20 to 40 machines). We can observe in table V that our RedTeamBERT model achieves a much better success rate compared to DQN, CLAP and zero-shot. It does not succeed every time however. The reason behind this is that NASim exploits are made to fail randomly with a certain percentage. Therefore, an attack may either fail because the firewall between the machines makes them impossible, or because we were unlucky when attacking. Both situations are indistinguishable during simulation, which explains the 6.5% of runs that fail for our system.

VII. CONCLUSION

In this paper, we presented a robust and adaptive approach based on four BERT classifiers that outperforms another RL agent (CLAP) as well as our previous work (zero-shot) across different scenarios in the NASim environment, according to our evaluation metrics. In some case, our BERT doesn't reach the maximum episodic reward, however it generally obtains a better episodic reward compared to the other RL agents. We demonstrated a certain level of explainability, which we consider to be superior to that of the previous zero-shot approach.

During some experiments, we observed that our BERT-based pipeline was not always able to complete the challenge, as it failed to learn certain conditions present in random scenarios. In future works, we plan to improve our pipeline to address this limitation and enable it to handle all random scenarios. We also aim to extend our BERT architecture to allow direct training within the RL environment, removing the need to generate a static dataset beforehand. Additionally, we plan to test our pipeline on a real cyber range to assess its performance in practical settings. Finally, we will continue to study the explainability of our model, as we consider this aspect crucial for deployment in real-world applications.

Our version of NASim, our dataset, and our model will be available at the following URL: https://github.com/silicom-hub/bert-pentesting-paper.

REFERENCES

[1] Gelei Deng, Yi Liu, Víctor Mayoral-Vilches, Peng Liu, Yuekang Li, Yuan Xu, Tianwei Zhang, Yang Liu, Martin Pinzger, and Stefan Rass. PentestGPT: Evaluating and harnessing large language models for automated penetration testing. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 847–864, Philadelphia, PA, August 2024. USENIX Association.

[2] Christophe Genevey-Metat, Dorian Bachelot, Tudy Gourmelen, Adrien Quemat, Pierre-Marie Satre, Loïc Scotto, Di Perrotolo, Maximilien Chaux, Pierre Delesques, and Olivier Gesny. Red Team LLM: towards an adaptive and robust automation solution. In *Conference on Artificial Intelligence for Defense*, Rennes, France, November 2023. DGA Maîtrise de l'Information.

[3] Kento Hasegawa, Seira Hidano, and Kazuhide Fukushima. Autored: Automating red team assessment via strategic thinking using reinforcement learning. In *Proceedings of the Fourteenth ACM Conference on Data and Application Security and Privacy*, CODASPY '24, page 325–336, New York, NY, USA, 2024. Association for Computing Machinery.

[4] Junjie Huang and Quanyan Zhu. Penheal: A two-stage llm framework for automated pentesting and optimal remediation, 2024.

[5] Jaromír Janisch, Tomáš Pevný, and Viliam Lisý. Nasimemu: Network attack simulator & emulator for training agents generalizing to novel scenarios, 2023.

[6] Minjune Kim, Jeff Wang, Kristen Moore, Diksha Goel, Derui Wang, Ahmad Mohsin, Ahmed Ibrahim, Robin Doss, Seyit Camtepe, and Helge Janicke. Cyberally: Leveraging llms and knowledge graphs to empower cyber defenders, 2025.

[7] Jakob Nyberg and Pontus Johnson. Structural generalization in autonomous cyber incident response with message-passing neural networks and reinforcement learning, 2024.

[8] Jonathon Schwartz and Hanna Kurniawati. Autonomous penetration testing using reinforcement learning. *CoRR*, abs/1905.05965, 2019.

[9] Yizhou Yang, Mengxuan Chen, Haohuan Fu, and Xin Liu. Settron: Towards better generalisation in penetration testing with reinforcement learning. In *IEEE Global Communications Conference, GLOBECOM 2023, Kuala Lumpur, Malaysia, December 4-8, 2023*, pages 4662–4667. IEEE, 2023.

[10] Yizhou Yang and Xin Liu. Behaviour-diverse automatic penetration testing: A curiosity-driven multi-objective deep reinforcement learning approach, 2022.

[11] Shicheng Zhou, Jingju Liu, Yuliang Lu, Jiahai Yang, Yue Zhang, and Jie Chen. Mind the gap: Towards generalizable autonomous penetration testing via domain randomization and meta-reinforcement learning, 2025.