Vulnet: Learning Navigation in an Attack Graph

Enzo d'Andréa
Inria - LORIA
Nancy, France
enzo.d-andrea@inria.fr

Jérôme François
SnT - University of Luxembourg & Inria - LORIA
Luxembourg, Luxembourg
jerome.francois@uni.lu

Abdelkader Lahmadi,Olivier Festor
Université de Lorraine - LORIA
Nancy, France
firstname.lastname@loria.fr

***Abstract*—Nowadays, new flaws or vulnerabilities are frequently discovered. Analyzing how these vulnerabilities can be used by attackers to gain access to different parts of a network allows to provide better protection and defense. Amongst the diverse analysis techniques, simulations do not necessitate a full infrastructure deployment and recently benefited from advances in reinforcement learning to better mimic an attacker's behavior. However, such simulations are resource consuming. By representing the interconnected hosts of a network and their vulnerabilities as attack graphs and leveraging machine learning, our method, Vulnet, is capable to generalize knowledge generated by simulation and gives insight about attacker capabilities. It can predict instantaneously the overall performance of an attacker to compromise a system with a mean error of 0.07.**

I. Introduction

Thousands of vulnerabilities are discovered each year[1], leading to dynamic attack surfaces. Vulnerability scanners such as OpenVAS[2] allow to scan remote devices and identify vulnerabilities along with their intrinsic criticality. However, evaluating the overall capabilities of an attacker is not a simple aggregate of all the individual vulnerabilities. As an attack graph represents the potential paths of an attacker, analysing them serves to better understand the capabilities of a potential attackers and indirectly the overall risk [1]. Rather than a static analysis, the advantage of using dynamic models is to mimic the behavior of an attacker or a pentester without exposing the real environment as it is modeled by an attack graph.

With recent progress in Machine Learning (ML), dynamic models using high level simulations can be achieved using Reinforcement Learning (RL) [2] where the agents interact with a specially crafted environment representing vulnerable hosts. The agent's objective is to have a maximum reward expressed in terms of exploited vulnerabilities or compromised hosts. Recent tools such as CyberBattleSim [3] have been developed for this purpose.

However, simulations are usually time-consuming since they rely on an iterative execution of attacker's actions and consequences, and on multiple runs to have representative results. On one hand, this is a common characteristic of all simulation-based methods. On the other hand, static methods based on graph analysis applied to attack graphs do not consider the behavior of the attackers. Vulnet lies in between by learning a model applicable on any attack graph without simulation.

[1]https://www.cvedetails.com/
[2]https://www.openvas.org/

In this work, we show that such an existing time-consuming approach can be approximated with a ML model capable to infer risk metric instantaneously such as the probability of a vulnerability to be used by the attacker. Unlike a simulation approach, our model can be applied to any new environment (e.g. attack graphs) even if it was not present in the learning set. Simulation is only used as a pre-requisite to generate learning data. To summarize, our objective is to capitalize and generalize over previously executed simulations on attack graphs. Although Vulnet is independent of the simulation method, this paper considers RL, especially DRL (Deep Reinforcement Learning), as demonstrated to be relevant in CyberBattleSim [3].

The rest of this paper is structured as follows: Section II presents related work. Section III refines the problem. Section IV gives details about Vulnet. The results are presented in Section V. Section VI concludes this paper.

II. Related work

Vulnerability scoring is used to measure the severity and criticality of single vulnerabilities such as Common Vulnerability Scoring System (CVSS) [4]. In [5], a game theory approach is used, guided by CVSS scores to rank vulnerabilities. Bullough et al. [6] and Feutrill et al. [7] also used open-source data and CVSS scores to predict the exploitation of vulnerabilities. Similarly, we aim at evaluating vulnerabilities by considering their implications in attack paths.

Indeed, attack graphs represent the relationship between vulnerabilities or hosts [8]. Sawilla et al. [9] used MulVAL [10] to generate an attack graph and introduced a ranking algorithm. Duan et al. [11] used another type of attack graph where vulnerabilities are modeled as nodes and edges represent the dependencies between them.

To better enhance the analysis of vulnerabilities and anticipate attackers' behaviors, Machine Learning (ML) techniques can be employed, such as Reinforcement Learning (RL) [12]. These techniques can also be applied on attack graphs such as in [13] where Graph Neural Networks (GNN) are used to rank attack graphs. Yousefi et al. [14] used RL to generate an attack graph through the help of MulVAL and inferred a transition graph. Using the CVSS scores as rewards, an agent is trained with a Q-learning strategy. Our work is complementary to these works as Vulnet takes as input RL simulation results to learn a model capable to be applied to another environment without executing the learned agent.

III. Problem definition

Our method uses topological information from the attack graphs to evaluate if and when a vulnerability will be exploited or a host compromised.

The environment where the attacker (or the agent) evolves is an attack graph aggregating all possible transitions between hosts. A transition represents an exploitable vulnerability. Assuming multiple environments as a set of attack graphs $\mathcal{E}$ and a graph $e = (H, V) \in \mathcal{E}$, H is the a set of nodes representing M hosts $H = \{h_1, h_2, \ldots, h_M\}$ with $h_m = \langle interest, breach \rangle$. $interest$ is an integer value representing the interest of an attacker for this host. In practice, it could reflect the critical value of a host (*e.g.* hosting confidential files or private keys). $breach$ is a binary value, true for the host where the attacker starts from, false for the others.

The set of the L directed edges is denoted $V = \{v_1, v_2, \ldots, v_L\}$. An edge of the graph is a vulnerability exploit, later referred to as *exploit* for simplicity, on one host ($source$) to gain further knowledge (credentials or existence) of any $target$ host or privileges on the current host, $source = target$ in that particular case. Otherwise, an attacker can also exploit a non-critical vulnerability on a host ($source = target$ also). We also define $C = \{c_1, c_2, \ldots, c_N\}$, the set of N credentials that allows an attacker to compromise hosts. v_l is represented as the tuple $v_l = \langle source, target, cost, type, outcome \rangle$ with:

- $source \in H$, $target \in H$,
- $cost$, an integer value representing the complexity of the exploit, *i.e.* the difficulty for an attacker to exploit the vulnerability,
- $type \in \{Local, Remote\}$, denoting whether a vulnerability is exploited locally or remotely,
- $outcome$ is the consequence of the exploit and allows the attacker to:
  - discover $target$ host – *Leaked host ID*,
  - discover $target$ host and its credential from C to compromise it – *Leaked credentials*,
  - compromise $target$ host – *Escalation* (only for a local exploit),
  - exploit a non-critical vulnerability (*i.e.* without take-over) on $target$ host – *Generic exploit*.

Because CyberBattleSim is used as the RL-based simulation tool in this paper, this modeling is inspired by its descriptive capabilities, in particular the four types of exploits: *Leaked host ID*, *Leaked credentials*, *Escalation* and *Generic exploit*. Our goal is to define $\mathcal{L} : e \in \mathcal{E} \rightarrow \{S, P_V, T_V, P_H, T_H\}$:

- S is the sum of the rewards a simulated attacker would acquire
- $P_V(v)$ and $T_V(v)$ are the probability and the average timestamp that an exploit will be used for all $v \in V$
- $P_H(h)$ and $T_H(h)$ are the probability and the average timestamp that a host will be compromised for all $h \in H$.

As an example, the *ToyCTF* attack graph from CyberBattleSim is depicted in Figure 1. Exploits have been named according to their outcomes, and those having multiple $target$s in CyberBattleSim like *Cred5* are duplicated for each $target$.

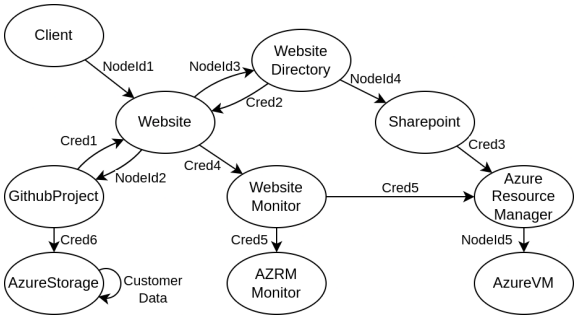

Fig. 1: Attack Graph for the ToyCTF example [3]

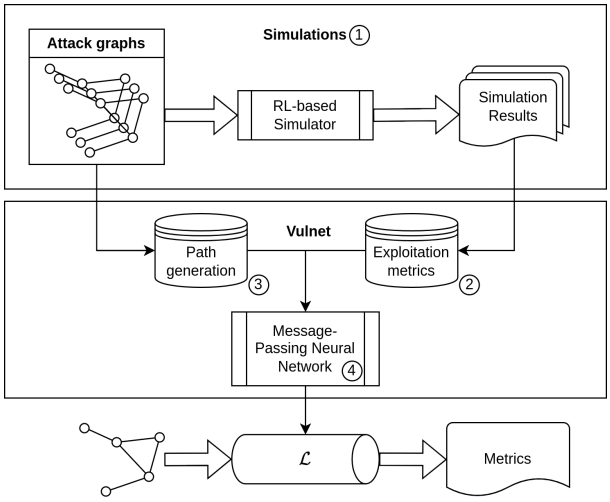

Fig. 2: Overview of the different steps of our method.

IV. Vulnet

Vulnet relies on a learning set (in our case given by RL simulations) and on a Message-Passing Neural Network (MPNN) to rapidly infer the function $\mathcal{L}$ matching an attack graph to the different attacker metrics S, $P_V(v)$, $T_V(v)$, $P_H(h)$ and $T_H(h)$ described above.

As show in Figure 2, our method is divided into 4 steps:

1) *Simulations* based on RL are applied to multiple attack graphs. The agent explores the graphs and the simulation results (metrics) are stored.
2) The *Exploitation metrics* step computes the metrics $\{S, P_V, T_V, P_H, T_H\}$.
3) *Path generation* extracts all the possible sequences of actions of an attacker.
4) *MPNN* is the core of Vulnet. It is a neural network architecture designed to learn the function $\mathcal{L}$ from the extracted metrics and computed paths.

Once learned, the function $\mathcal{L}$ is directly applied to a new attack graph for inferring the different metrics.

A. Building training data by simulation

To create the dataset used for learning $\mathcal{L}$, multiple graphs are randomly generated by creating a set of hosts linked with

vulnerabilities to create attack paths. The exact setup used to generate hosts and exploits is detailed in Section V-A.

In RL, the decisions of the agent are driven by a policy which is learned and improved through a sequence of episodes (its default length is 20). To gather statistical properties about hosts and exploits, a simulation on a given environment is a collection of $K_{run} = 50$ runs, each one starting from scratch to learn an independent sequence of episodes.

Assuming a given attack graph and a given run, the agent can either exploit a vulnerability v or attempt to access a host using one credential from C obtained earlier. The reward for each action a in run δ is computed using the default CyberBattleSim formula:

$$reward_\delta(a) = \sum_{h_m \in H_{comp}(a)} interest_m + K_{host} \times n_{host} + K_{cred} \times n_{cred} + K_{vuln} \times n_{vuln} - cost \quad (1)$$

with $H_{comp}(a) \subset H$, the set of newly compromised hosts thanks to a; $interest_m$, the interest value of the m^{th} one; n_{host}, the number of newly discovered hosts; n_{cred}, the number of newly discovered credentials; $n_{vuln} = 1$ for the first exploitation of the vulnerability, 0 otherwise in order to account positively only the first time it is exploited (in order to avoid giving a reward for a vulnerability that has already been exploited) and $cost$, the difficulty of the exploit of the vulnerability v. $K_{host} = 5$, $K_{cred} = 3$ and $K_{vuln} = 7$ are the default CyberBattleSim coefficient values.

Different strategies are available. Based on the benchmarks provided in CyberBattleSim, we used Deep Q-learning along with the following parameters:

- γ = 0.015
- Replay memory size = 10000
- Target update = 10
- Batch size = 512
- Learning rate = 0.01
- ϵ = 0.90
- ϵ exponential decay = 5000
- $\epsilon_{minimum}$ = 0.10
- Number of episodes = 20

B. Exploitation metrics

The results from the runs of a simulation are aggregated together to extract the exploitation metrics (or attacker metrics), allowing the quantitative characterization of the compromised hosts and the exploitation of vulnerabilities when the represented environment is the target of an attacker. Several metrics are computed at three different levels: global-, host- and vulnerability-level.

The global metric S is a cumulative reward of each action a in all previous actions taken (defined as A) according to equation 1. In addition, it is averaged according to the number of runs K_{run} defined in Section IV-A and normalised according to the theoretically maximum value M_R. The latter is reached if all hosts are discovered, all vulnerabilities are exploited and all credentials are discovered:

$$S = \frac{\sum_{\delta=1}^{K_{run}} \sum_{a \in A} reward_\delta(a)}{M_R \times K_{run}} \quad (2)$$

$$M_r = \sum_{h_m \in |H|} interest_m + K_{host} \times |H| + K_{cred} \times |C| + K_{vuln} \times |V|$$

The probability that a host is compromised is computed according to the number of runs where it happens and the total number of runs:

$$P_H(h) = \sum_{\delta=1}^{K_{run}} \frac{k^\delta(h)}{K_{run}}, h \in H$$

with $k^\delta(h)$ being 1 if h was compromised during the δth run of the agent in the simulation, 0 otherwise. $P_V(v)$ is computed similarly for the probability of the exploit of vulnerability v.

For $T_H(h)$ and $T_V(v)$, the timestamps when the agent compromises a host or exploits a vulnerability are recorded and are normalised between 0 and 1:

$$time_{norm} = \frac{time}{time_{max}}$$

with $time_{max}$, the maximal number of iterations. The normalized timestamps are averaged over the K_{run} runs to obtain a single timestamp per host/exploit and per graph. We use $time = time_{max}$ when a host/vulnerability is not compromised/exploited in the run.

C. Attack paths

To model the different sequences of possible attacker's actions, we consider all possible sequences of exploits $p_k \in \mathcal{P}$ that the attacker can exploit. Although a path is defined as a sequence of exploited vulnerabilities ($p_k = \{p_k^i, \forall i, p_k^i \in V\}$), two sequential vulnerabilities can be exploited from different parts of the graphs. This allows to mimic an attacker in a realistic manner such that, at a given time, any vulnerability from any already compromised host (not only the last one) can be exploited. For example, an attacker may have compromised two hosts and alternate trying to continue its progression from them, *i.e.* hopping between two parallel paths. In Figure 1, a path from the host *Website* to *Website Monitor* could be the sequence [*HostId2*, *Cred4*]. In this path, *Cred4* was used even though the last host of the path was *GithubProject*. We thus need to generate attack paths representing all possible sequences of potentially used exploits even if two successive exploits are not chained in the original graph.

The computation of such refined attack paths are done recursively through a specific depth-first graph traversal but where new starting points are generated on the fly to explore multiple paths in parallel.

D. Message-Passing Neural Network

In order to handle various graph sizes of the previously described extended attack graphs, we opted for a Message-Passing Neural Network (MPNN) architecture as the core of our ML model, inspired from [15], [16]. MPNN is a type of Graph Neural Network that allows the information related to the nodes, edges and whole-graph to be represented as fixed-size embeddings. These embeddings are used as input

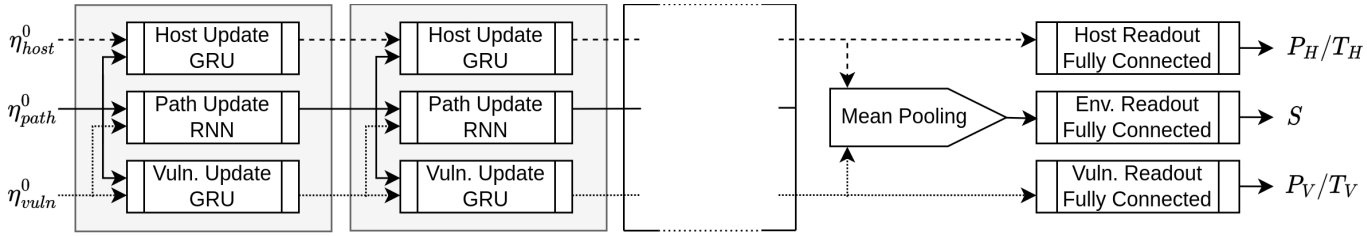

Fig. 3: MPNN-based architecture of Vulnet

and spread over neighbors through an *Update* function in an iterative manner. Intuitively the message passing process represents an attacker exploiting vulnerabilities in a sequential manner. We consider a host as *on the path* if it is the $source$ or $target$ host of any vulnerability exploit in the path. We further denote $T_h \subset \mathcal{P}$ as the paths that have host h as target of their last exploit.

With this formalism, our MPNN architecture is composed of three hidden states for vulnerability-, path- and host-level encodings. These states are inter-dependent according to the following mechanisms:

1) The state of a host h depends on the state of all attack-paths leading to the host (T_h), each intermediate vulnerability exploit of a path also being the final one of another path by design.
2) The state of an attack path depends on the states of all the vulnerabilities exploited in the path.
3) The state of a vulnerability depends on the states of all attack-paths exploiting this vulnerability.

These are formally expressed by denoting the state of a vulnerability v by $\eta_{vuln}(v)$, the state of path p by $\eta_{path}(p)$ and the state of host h by $\eta_{host}(h)$, all are unknown hidden vectors. The above dependencies are expressed as follows:

$$\eta_{vuln}(v) = f(\eta_{path}(\{p_k\}), p_k \in \mathcal{P}, v \in p_k) \quad (3)$$

$$\eta_{path}(p) = g(\eta_{vuln}(\{v_k\}), v_k \in p) \quad (4)$$

$$\eta_{host}(h) = q(\eta_{path}(\{p_k\}), p_k \in T_h) \quad (5)$$

where f, g and q are unknown functions.

Vulnet receives as inputs the initial states:

- $\eta^0_{host}(h)$, containing the value $interest$, normalized by dividing by the maximal theoretical value, and $breach$ of $h \in H$;
- $\eta^0_{vuln}(v)$, containing the value $cost$, also normalized using the maximal theoretical value, a binary for the $type$ and a one-hot encoding of the $outcome$ of $v \in V$
- $\eta^0_{path}(p)$ being 0s for the path $p \in \mathcal{P}$

All states are padded with 0s until the state sizes (configurable hyper-parameter) are achieved. As shown in Figure 3, one *message-passing* operation is made of three *update* functions that use as inputs the states of hosts, paths and vulnerabilities from the previous *message-passing* operations. This operation is repeated τ times, τ being a configurable hyper-parameter. This repetitive process allows us to address the circular dependencies of equations (3) (4) and (5).

Each update operation is applied individually to each single related entity (hosts, vulnerabilities or paths). As shown in the figure, any state $\eta^i_*(*)$ is updated from its previous value, $\eta^{i-1}_*(*)$. Actually, Figure 3 is a general overview whereas, in practice, all the *update* functions are duplicated and linked among them based on the underlying graph structure derived from $\mathcal{P}$. Given that the order of the paths does not have an influence on the state of vulnerabilities or hosts, the states of paths are aggregated. Hence, the state of a host is updated using a single GRU (*Gated Recurrent Unit*) cell and the sum of the states of the paths leading to this host:

$$\eta^i_{host}(h) = GRU(\eta^{i-1}_{host}(h), \sum_{p \in T_h} \eta^{i-1}_{path}(p))$$

The *update* of the state of the vulnerability v is defined similarly:

$$\eta^i_{vuln}(v) = GRU(\eta^{i-1}_{vuln}(v), \sum_{p \in \mathcal{P}, \exists p^i, p^i = v} \eta^{i-1}_{path}(p))$$

However the state of a vulnerability has an impact on the vulnerabilities further down the paths traversing it. For this reason, a Recurrent Neural Network (RNN) is leveraged to aggregate the states of the vulnerabilities in the path p:

$$\eta^i_{path}(p) = RNN(\eta^{i-1}_{path}(p), [\eta^{i-i}_{vuln}(v), v \in p])$$

Because a RNN can handle arbitrary input lengths, a unique RNN can thus be learned and applied to all the different paths of all the graphs independently from the size of the paths.

3 readout functions consisting of fully connected layers transform the host states into the timestamps and probabilities of being compromised for each host and the vulnerability states into their timestamps and probabilities of exploitation. As input to the graph readout function, the states are aggregated through a mean pooling layer and then concatenated to get the overall score S.

V. Results

A. Graph generation

To test our approach, Vulnet needs to be trained on multiple attack graphs. As our objective is to show how Vulnet can generalize from and to any graph, we relied on a randomized graph generation. 3 different sizes are considered: 10, 20 and 30 hosts with the starting point for the simulated attacker randomly chosen. The interest value of each host is randomly uniformly set as an integer between 0 and 10, and the cost

TABLE I: Probability of creation of vulnerability exploits

Outcome	Type	Number of hosts		
		10	**20**	**30**
Leaked credentials	Local or Remote	0.1	0.025	0.011
Leaked host ID	Local or Remote	0.5	0.125	0.055
Generic exploit	Local or Remote	0.1	0.05	0.033
Escalation	Local	0.2	0.1	0.066

of each vulnerability exploit between 0 and 5. Intuitively, the objective is to create enough heterogeneous data to learn how the attacker behaves when facing different environments.

As stated in Section IV-A, the hosts are linked to each other through *Local* or *Remote* exploits having different outcomes. The exploits are created according to a vulnerability probability value given in Table I based on the following process:

- *Leaked credentials*: For each ordered pair of hosts, up to 4 attempts are made to create a vulnerability exploit (edge). The value in Table I refers to the probability of success of each one. *Local* and *Remote* exploits are handled separately, so between 0 and 2 edges are added.
- *Leaked host ID*: The given probability is the probability to create a single vulnerability exploit. *Local* and *Remote* exploits are also considered separately, so between 0 and 2 vulnerabilities are created for each ordered pair of hosts.
- *Generic exploit*: Created for each host as a self-loop, as $source = target$, with the given probability. Because *Local* and *Remote* are independent of each other, between 0 and 2 vulnerability exploits are created for each host.
- *Escalation*: Also created as self-loop. Two exploits can be created with the same probability and they are only *Local* by nature, so up to 2 exploits per host are created.

To be more representative, the density of vulnerability exploits in a graph has been varied. The generation of *high-density* graphs are based on the probabilities of Table I, which are halved for *low-density* graphs to create a lower number of exploits. Our dataset contains 238 graphs: 121 high-density graphs (35 with 10 hosts, 41 with 20 hosts, 45 with 30 hosts) and 117 low-density graphs (47 with 10 hosts, 28 with 20 hosts and 42 with 30 hosts)

Based on these probabilities, the number of iterations, or actions, per episode for the RL agent was tuned to 500 but the agent stops earlier when all hosts are compromised.

B. Hyper-parameters

The configurable hyper-parameters are the sizes of the hidden states of the hosts, vulnerabilities and paths, the number of iterations τ for the MPNN and the number of fully connected layers and their size for the readout part. A grid-search was applied leading to the configuration highlighted in Table II. The model is trained using the Adam optimizer, an initial learning rate of 0.001 and 10 epochs.

C. Experimental setup and running time

The dataset is divided in a 60/20/20 ratio for training, validation and testing. With 2 Intel Xeon Silver 4116 totaling 48 logical cores and a Nvidia Geforce GTX 1080Ti, training takes 14 hours 30 minutes for all hyper-parameters configurations. For comparison, running the RL simulation on a single graph using the same hardware needs 245 minutes on average. In other words, taking into account the hyper-parameters optimization phase, our method is roughly 13 times faster than simulating all graphs from the testing dataset.

TABLE II: Grid-search on hyper-parameters (bold values are the selected ones)

State size			τ	Readout	
Host	Edge	Path		# of layers	Layer size
8/16/**32**	8/16/**32**	8/**16**/32	2/3/**4**	**1**/2/3	2/**3**/4

In inference mode, computing the possible paths takes on average 161ms in addition to the 12ms inference time for the neural network, totaling 173ms per graph on average, while CyberBattleSim needs about 914s per graph. Our method is therefore roughly 5200 times faster than the RL method.

D. Exploitability prediction

In the first experiment, we assess the ability of Vulnet to predict the metrics P_H, T_H, P_V, T_V and S, using the RL simulations as ground truth. We compute the cumulative distribution function of the absolute error. We voluntary place ourselves in the worst case scenario where the DRL agents are each trained on one specific attack graph.

The global score, S, is predicted with a mean error of 0.07, a global error range of ±0.21, and 80% of the scores lies within the ±0.15 error range as shown in Figure 4. Vulnet is thus capable of giving an overall indicator of an attacker performance. Predicting fine-grained metrics is more difficult. In particular the most difficult to predict is the probability for a host to be compromised (P_H), with only 65% of the predicted probabilities having an error lower than 0.4. However, Vulnet is able to predict when a host will be compromised (T_H) with an error lower than 0.3 in 80% of cases. In general, Vulnet is better at inferring vulnerability metrics rather than host metrics. Vulnet is also capable to predict when a vulnerability will be exploited with an error lower than 0.25, *i.e.* within the right quarter of the simulation time, in 80% of cases.

E. Generalization capabilities

For sake of space, we focus here on generalizing over larger graphs or graphs with a higher density of vulnerabilities which actually presents more complex graph to learn on or run simulations on. The model was thus trained with 10-hosts and 20-hosts graphs and underwent a new hyper-parameters optimization. The model is then applied to the 30-hosts graphs.

To highlight how much the prediction capabilities differ from the version trained with all sizes of graphs, we compute the difference between the cumulative distribution of the errors from the fully trained model in previous section. In Figure 5, the difference for the global score S quickly rises to over 20 points before dropping back to near 0. Hence, even though the score is not as accurately predicted, the majority of errors are shifted to a range of 0.1/0.3 compared to 0.0./0.2 in the

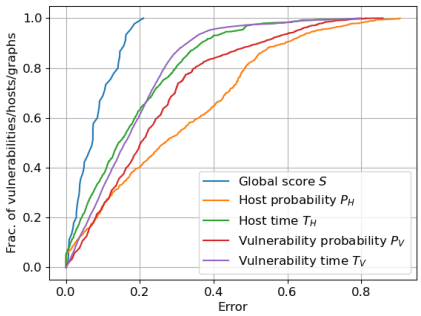

Fig. 4: Cumulative distribution of the errors independently of the type of graph

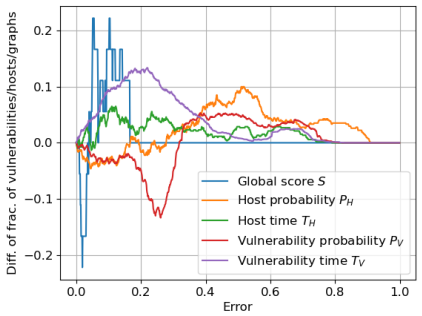

Fig. 5: Performance degradation when excluding 30-hosts graphs from learning

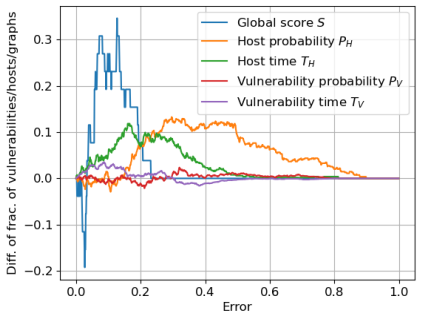

Fig. 6: Performance degradation when excluding high-density graphs from learning

original experiment. For the other metrics (P_H, T_H, P_V, T_V), the difference is lower than 0.15 in all cases, indicating that even though the performances are not as good, the model is still able to predict the metrics of graphs with unseen sizes.

We also evaluated the performance of the models with the same approach as before by only learning the model low-density graphs and applying it to high-density graphs. In Figure 6, the global score S is partially shifted to the range 0.1/0.3 like in the experiment on size-generalization. The differences in the distributions of the vulnerability metrics P_V and T_V remain in the range ± 0.04, indicating the model can generalize these metrics with a good confidence. For the host metrics P_H and T_H, the differences in the distributions rise up to +0.12. The model is therefore also capable of generalizing for the host metrics but less than the vulnerability metrics.

VI. Conclusion

Our main goal was to alleviate the need of time-consuming simulations to anticipate the potential attacks' consequences on targeted hosts. Actually, an attackable environment is represented as attack graphs composed of hosts alongside their vulnerabilities. While (D)RL-based simulations can be applied to such an environment, Vulnet provides a direct inference of multiple security metrics relying on this graph. Although we observed an acceptable overall degradation of the the metrics' accuracy in comparison with a DRL approach, with an error never exceeding 0.23 in the case of the global score, Vulnet is thousands times faster. Our future plan is to adapt Vulnet to integrate the defensive strategies.

Acknowledgment

This work has been partially supported by the French National Research Agency under the France 2030 label (Superviz ANR-22-PECY-0008). The views reflected herein do not necessarily reflect the opinion of the French government. This research was funded in part, by the Luxembourg National Research Fund (FNR), grant reference INTER/ANR/20/14783140/GLADIS.

References

[1] N. Poolsappasit, R. Dewri, and I. Ray, "Dynamic security risk management using bayesian attack graphs," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 1, pp. 61–74, 2012.

[2] J. Nyberg, P. Johnson, and A. Méhes, "Cyber threat response using reinforcement learning in graph-based attack simulations," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2022.

[3] M. D. R. Team., "Cyberbattlesim," https://github.com/microsoft/cyberbattlesim, 2021, created by Christian Seifert, Michael Betser, William Blum, James Bono, Kate Farris, Emily Goren, Justin Grana, Kristian Holsheimer, Brandon Marken, Joshua Neil, Nicole Nichols, Jugal Parikh, Haoran Wei.

[4] FIRST, "CVSS v3.1 Specification Document." [Online]. Available: https://www.first.org/cvss/specification-document

[5] L. Maghrabi, E. Pfluegel, L. Al-Fagih, R. Graf, G. Settanni, and F. Skopik, "Improved software vulnerability patching techniques using CVSS and game theory," in *International Conference on Cyber Security And Protection Of Digital Services (Cyber Security)*, 2017.

[6] B. L. Bullough, A. K. Yanchenko, C. L. Smith, and J. R. Zipkin, "Predicting Exploitation of Disclosed Software Vulnerabilities Using Open-source Data," in *International Workshop on Security And Privacy Analytics (IWSPA)*. ACM, 2017.

[7] A. Feutrill, D. Ranathunga, Y. Yarom, and M. Roughan, "The Effect of Common Vulnerability Scoring System Metrics on Vulnerability Exploit Delay," in *Sixth International Symposium on Computing and Networking (CANDAR)*, 2018.

[8] W. He, H. Li, and J. Li, "Unknown Vulnerability Risk Assessment Based on Directed Graph Models: A Survey," *IEEE Access*, vol. 7, 2019.

[9] R. E. Sawilla and X. Ou, "Identifying Critical Attack Assets in Dependency Attack Graphs," in *Computer Security - ESORICS 2008*, S. Jajodia and J. Lopez, Eds. Springer, 2008.

[10] X. Ou, S. Govindavajhala, and A. W. Appel, "MulVAL: a logic-based network security analyzer," in *Security Symposium*. USENIX, 2005.

[11] C. Duan, Z. Wang, H. Ding, M. Jiang, Y. Ren, and T. Wu, "A Vulnerability Assessment Method for Network System Based on Cooperative Game Theory," in *Algorithms and Architectures for Parallel Processing*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020.

[12] T. Cody, P. Beling, and L. Freeman, "Towards Continuous Cyber Testing with Reinforcement Learning for Whole Campaign Emulation," in *2022 IEEE AUTOTESTCON*, Aug. 2022, pp. 1–5.

[13] L. Lu, R. Safavi-Naini, M. Hagenbuchner, W. Susilo, J. Horton, S. Yong, and A. Tsoi, "Ranking Attack Graphs with Graph Neural Networks," vol. 5451, Apr. 2009, pp. 345–359.

[14] M. Yousefi, N. Mtetwa, Y. Zhang, and H. Tianfield, "A Reinforcement Learning Approach for Attack Graph Analysis," in *17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, 2018.

[15] K. Rusek, J. Suárez-Varela, P. Almasan, P. Barlet-Ros, and A. Cabellos-Aparicio, "RouteNet: Leveraging Graph Neural Networks for Network Modeling and Optimization in SDN," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 10, Oct. 2020.

[16] Y. Xue, J. Guo, L. Zhang, and H. Song, "Message Passing Graph Neural Networks for Software Security Vulnerability Detection," in *International Conference on Computer Network, Electronic and Automation (ICCNEA)*, 2022.