Discovering How to Attack a System

Fabrizio Baiardi[1][a], Daria Maggi[2] and Mauro Passacantando[1][b]
[1]*Dipartimento di informatica, Universita di Pisa, Largo B. Pontecorvo 3, Pisa, Italy*
[2] *Dipartimento di Ingegneria Informatica, Universita of Pisa, Pisa, Italy/ Consulting Engineer at Cisco Systems Portugal*
{*fabrizio.baiardi, mauro.passacantando*}*@unipi.it, d.maggi1@studenti.unipi.it*

Keywords: genetic algorithm, rule based system, attack strategy, digital twin

Abstract: We evaluate the performance of a genetic algorithm to discover the best set of rules to implement an intrusion against an ICT network. The rules determine how the attacker selects and sequentializes its actions to implement an intrusion. The fitness of a set of rules is assigned after exploiting it in an intrusion. The evaluation of the distinct sets of rules in the populations the algorithm considers requires multiple intrusions. To avoid the resulting noise on the ICT network, the intrusions target a digital twin of the network. We present a preliminary experimental results that supports the feasibility of the proposed solution.

1 INTRODUCTION

This paper discusses the adoption of a genetic algorithm, GA, to discover a set of rules that implements the best attack strategy against an ICT network. Each rule implements an attacker action; the GA returns the best ones to implement an intrusion and control some network modules. To evaluate each rule set, we use it to drive an intrusion and the fitness of a set decreases with the time to reach the goal. Intrusions target a digital twin of the network, i.e. a labeled graph. The first experimental results confirm that a GA can discover the best strategy. This work is organized as follows. Sect.2 briefly discusses the emulation of an intelligent adversary. Sect.3 describes the kernel ideas of the proposed solution and the modeling of both the attacker and target network. It also introduces the GA design and how it generates and refine the set of rules. Sect.4 presents the rule-based system and details the prototype implementation. The last section presents first experimental results.

2 STATE OF THE ART

We discuss strategies for adversary emulation and denote by intrusion the sequence of attacker actions to control some network modules.

2.1 Adversary emulation

Attackers are intelligent and alternate attacks and collection of information on the vulnerabilities of network modules to discover the shortest sequence of actions to reach a goal, the control of some module. The knowledge of the attacker strategy to select actions is fundamental to detecting and stopping intrusions. This requires accurate information on both the selection and the actions.

A penetration test is an emulation exercise where a red team implements an intrusion. Proper countermeasures are adopted if the team is successful. The team lacks information on the module vulnerabilities to mimic that an attacker collects it as the intrusion goes on. The accuracy of this test is low because, due to the network size, the time to consider all the vulnerabilities might as well encompass the exercise time frame. Hence, the red team cannot discover all the intrusions. Furthermore, real attackers may apply strategies that differ from those of the red team because, while they are interested in the quickest intrusion they also prefer to use the same exploits against distinct targets (Allodi et al., 2021). Lastly, information collection is as important as attack execution (Applebaum et al., 2017). Hence, the same red team should attack a network at most twice: to discover critical vulnerabilities and then to verify they have been removed. Further exercises to discover other intrusions require a new team, because the previous ones have already collected information on the modules and their vulnerabilities. Tools to automate

[a] https://orcid.org/0000-0000-0000-0000
[b] https://orcid.org/0000-0003-2098-8362

the intrusion and emulate the attackers can solve the shortcomings of human teams and time constraints but multiple tests results in a large noise in a network. To avoid the noise, we introduce a digital twin, i.e., a digital version of the network (Baiardi and Tonelli, 2022). A twin abstracts most of the target details but it represents system assets, i.e. nodes with their modules, vulnerabilities and connections. Furthermore, it describes the attacks these vulnerabilities enable in terms of preexploitation and postexploitation states, as in (Inokuchi et al., 2019). When using a twin, intrusions are automated through a software platform that emulates the red team actions. Distinct emulations return information on the effectiveness of distinct strategies.

2.2 Neural networks

The abstraction capabilities (LeCun et al., 2015) of deep learning favor its adoption in several domains. However, deep learning - and in general, machine learning - is yet to gain momentum in offensive security. (Aiyanyo et al., 2020) reviews its performance but there are some reasons why deep learning could not be the best solution in this field (Marcus, 2018):

1. an intrusion is a hierarchical composition of activities, and the current performance of neural networks is not satisfactory for similar problems (Lake and Baroni, 2018);
2. while an attacker collects information during the intrusion, deep learning requires structured information on the target;
3. emulation is much more a common sense problem than a classification one.

Hence, we have investigated a GA as an alternative.

3 The Proposed Solution

This section outlines the proposed solution and discusses how a twin models the target network, the attacker, and their interactions. Then, it outlines the definition of the GA.

3.1 The Network and Its Twin

A target network is defined by its modules, hardware and software ones, their connections and their vulnerabilities. An attacker exploits these vulnerabilities to acquire access rights on some modules and take action on other ones.

A twin represents a network as a labeled graph where each node n(m) represents a module m and is labeled by the vulnerabilities of m. An edge from n(m1) to n(m2) denotes that an attacker can exploit from m1 a vulnerability in m2 and gain access rights on the latter. Each edge is labeled by an attack and by attributes that quantify the attack complexity, its execution time, success probability, and the largest number of repetitions. Since a module m1 can take an action on module m2 if it can interact with m2, the edges also take into account the network topology.

3.2 Attackers

We describe how attackers behave and how the GA models them and evaluates their fitness.

3.2.1 Modelling Attacker

In an intrusion, an attacker chains actions and attacks to gain some rights on modules and reach its final goal, the control of some modules. Before attacking a module, an attacker scans it to discover the module vulnerabilities. Thus, at each step of an intrusion, the attacker decides whether to attack a module it has already scanned, or to look for a new potential target among the modules it can reach from those it has already attacked. Attacking is usually preferred in the earlier steps of an intrusion, to increase the number of sources of a scan. Moreover, by acquiring information on promising nodes, an attacker can avoid wasting time and attempts.

We describe an attacker as a rule-based system where a rule consists of a premise and a conclusion, according to *modus ponens*. Each rule models one attacker action and the firing of a rule corresponds to the action execution. The whole set of rules codifies the attacker strategy. A rule may include distinct premises, each a condition on some fact in a domain. In our domain, facts convey information on the system assets and the current attack status or on the nodes the action can target. As an example, a precondition may check if the attacker has previously scanned a module or if the module is affected by a vulnerability. A rule is activated if all its premises are satisfied. The rule conclusion consists of a set of postconditions. The intended meaning is that when a rule fires the corresponding action is executed and, if successful, it affects the environment and/or the attacker so that each rule postconditions becomes true. As an example, a postcondition may change the state of a module to *scanned* or add a new module to those an attacker can target or control. Distinct rules are activated anytime the attacker can execute distinct actions. To solve this conflict, each rule is paired with an importance value, the salience, that determines the probability it is chosen and fired.

3.2.2 Emulating Attackers

To evaluate an individual, i.e. an attacker, the GA executes the corresponding rules. This results in a sequence of steps where at each step, the attacker executes an action according to the fired rule. The emulation stops when the attacker has reached its goal or all options have been exhausted and no rules can fire. Each rule action is associated with a cost and the cost of a sequence is the sum of the cost of the rules it has fired, where each action repetition contributes to the overall cost.

To determine activated rules, an emulation needs an efficient representation of the security status. This status is a set of parameters for each module that may be updated any time a rule fires. The minimal set of parameters to model both information acquisition and attack consists of two Boolean values for each module. They record, respectively whether the module has already been scanned and whether the attacker has acquired some rights on it. A rule precondition may match some of these values. Anytime a rule fires, the success or the failure of the action, its outcome, is determined because a successful action updates the security status. The outcome is decided through the output of a random generator according to the action success probability. If an action may fail, the repetition factor in the twin determines the number of possible repetitions. The only action that is always successful is the scanning of a module so that a module is scanned only once. The security status also records the number of attacks that have used each graph edge.

In general, the structure of a rule is

$$\langle \text{security state} \rangle, \langle \text{vulnerabilities} \rangle, \langle \text{connections} \rangle \rightarrow \langle \text{ action, security state } \rangle$$

where *securitystate*, *vulnerabilities* and *connections* are premises, i.e. conditions on the security status, on module vulnerabilities and connections. If the rule fires and the action is successful, the security status is updated. As an example, if the attacker controls N_1, it has scanned but does not control N_2 (security state premise), there is a vulnerability in N_2 (vulnerabilities premise) and N_1 is connected to N_2 (connection premise), then it can attack N_2 (action). If this rule fires and the attack is successful, the attacker will control N_2 (new security state).

3.2.3 Evaluating Attackers

If we only consider rule premises and post-conditions, each attacker is described by the same rules. As an example, a rule may state that after scanning a module M the module can be attacked or that a successful attack to M enables an attack to a module connected to M.

To pair distinct attacker, i.e. individuals, with distinct rules, the GA pairs each individual with two sets that convey information on the network topology and include, respectively, promising modules and the deadpoints. A promising module leads the attacker to its goal, while a deadpoint has no outbound edge or has resisted all the attack attempts.

The same rule may differ in distinct attackers because its salience depends upon the nodes it matches. The salience of a rule that matches a promising module is larger than when the rule matches another one. Hence, the salience of a rule that scans or attacks a promising module is larger than those of rules targeting not promising ones. Furthermore, a rule where the action targets a deadpoint, has lower salience than rules that do not match a deadpoint.

To discover promising modules and deadpoints, the GA analyses a log that stores the actions of an attacker and the modules it has targeted in an emulation. The GA classifies attackers in three categories. It also pairs an attacker with a priority and a distance from the target module in terms of the number of hops or the success probabilities paired with arcs. Each attacker may be:

- successful if it reaches its goal. Its distance is 0 and the priority decreases with the sum of the costs of its actions;
- close if its distance is lower than a threshold δ. Its priority decreases with this distance;
- unsuccessful if its distance from the goal is larger than δ.

The distance and the priority of an individual determine its fitness.

3.3 Algorithm Set Up

The GA is applied to a population after evaluating each of its individuals, i.e. a set of rules plus the set of promising nodes and the one of the deadpoints. Some parameters of the GA are set *a priori* such as the elite size namely the number of individuals that reproduce. Other parameters will be determined by the GA, such as the probability of choosing an action, that is a function of the salience of the rule, and the sets of, respectively, promising nodes and deadpoints a parent will transmit to its descendants.

The selection operator analyzes each individual in the current population and classifies it as either successful, close, or unsuccessful. Then, it ranks individuals in a listby increasing distance from the goal. The last attackers in the list are the unsuccessful ones.

The crossover operator selects the parents of the next population with a probability that decreases as

the position of an individual in the list increases. Thus, given two individuals, the operator prefers the one paired with:

- the highest priority if at least one is successful;
- the lowest distance if they are promising or unsuccessful candidates.

After choosing the parents, the GA transmits to the offspring the salience, promising nodes, and deadpoints. If both parents are successful, the salience of each action of a descendant inherits is the weighted average of those of the parents. The most influential parent is the most successful one. The individual also inherits a subset of shared deadpoints and its size is randomly generated according to a uniform probability distribution. If only one parent is successful, its promising modules are transmitted only. Otherwise, the sets of promising modules and of deadpoints are those of the one with the shortest distance.

The mutation operator updates the salience of each rule to explore the solution space.

In our solution, the features an offspring receives are either network-specific or network independent. Promising modules and deadpoints are network specific, while rule salience is network independent.

4 Prototype implementation

We have implemented a prototype to evaluate the GA performance and the time to converge to an optimal solution. In the prototype, the action space for an attacker is the minimal one and it includes two actions: to scan and discover the vulnerabilities of a device, and to attack. Actions such as exfiltrating information or evading some defence mechanism are neglected because the attacker executes them only if it is successful. In general, an attack success probability is the sum of a base success probability (inferred from public databases (The MITRE Corporation, 1999)) plus a bias factor, depending on the attack complexity. To simplify the twin, any attack has a success probability equal to 0.7. Furthermore, a twin node corresponds to a network node rather than to a module.

The rule-based system is implemented in CLIPS (Cafasso, 2016) that supports the specialization of rules for promising nodes and deadpoint and it offers *conflict resolution strategies*. CLIPS creates a stack of rules that match distinct facts, nodes in the prototype. A rule position depends on its *salience*. A resolution mechanism use salience to solve conflicts. We have embedded CLIPS in the Python environment because CLIPS lacks granular control on the order of fact presentation in the firing of a rule. A dynamic update of the salience ensures that a rule will be fired before other ones, but there is no control in CLIPS on the order to match rule premises to the security state, vulnerabilities and connections. We have mitigated this through Python-calls that update a rule salience according to the promising or deadpoint nodes it matches. Further criteria to update the salience will be described in the following.

5 Results

In the experiments to evaluate the GA performances, the first parameter is the network size, and, incidentally, its density. We gradually increase this size. In a small network, the number of paths from the source to the target gains too much weight in the evaluation. If there are very few winning options, the population will converge very quickly to them without a detailed evaluation of alternative solutions.

The population size, i.e. the number of distinct attackers the GA considers, is another parameter and the parent pool is tuned according to this size. The larger the population size, the larger the breadth size of the search for the optimal solution. This speeds up the convergence to the optimal path but larger populations are not effective for small networks. Notice that we do not assume each member of a population will actually attack the target system. The population size is interesting only to evaluate how it influences the GA convergence.

The third parameter is the network complexity and the complexity of exploiting each edge. Ideally, the larger the edge weight, the harder the success of the corresponding attack.

5.1 Full Exploration

Here, the Python environment updates the rules so that an agent fully explores the subtree rooted in a promising node before moving to another one. As expected, the convergence slows down and the time to discover the optimal paths decreases with a larger parent pool. Instead, the time to reach the 100% hitmark increases with the network size.

5.1.1 20 nodes, 100 attackers

In these experiments the system graph is not weigthed so that the best path is the shortest one and its length is four. We have tested four distinct elite sizes but all the attackers are tested on the same path. In Fig. 1 x axis is the number of generations, while the y one is the number of successful attackers, i.e. that have

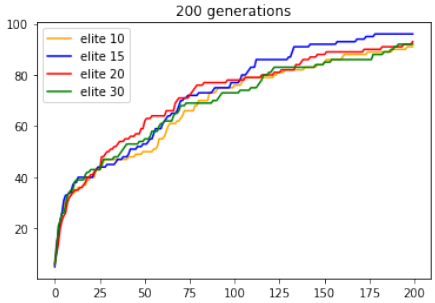


Figure 1: 20 nodes, 100 attackers, 200 generations.

reached the target node. A small elite size results in a steeper growth of successful attackers, with the 80% threshold being hit at around 100 generations. When the size of the mating pool is larger, in the first 10/20 generations at least, some parents are not successful and have reached a 'close-enough' node with a small cost. A larger diversity results in slower but more stable growth. The 80% benchmark is hit after 100 generations. However, the progress in both cases slows down and neither of the populations reaches the 100% mark in the given number of generations. Fig. 1 shows that an elite size of 15 results in the most linear growth before the 80% mark and it is the only case reaching the 100% mark. Furthermore, all the population has learned how to reach the goal.

5.1.2 80 nodes, 100 attackers

The network is larger but with the same edge density as the 20 nodes one. The length of the path to be discovered is six. This strongly increases the problem complexity because of the larger network size. The curve shapes in Fig. 2 are similar to those for the corresponding elite size ratios. Due to the larger solution space, the parent pool will be richer and broader even if we reduce its size.

5.1.3 80 nodes, 1000 attackers

Here we analyze the performance of a large population targeting a large network. We assume the larger population compensates for the larger number of paths so that the initial exploration of the solution space should result in a broader frontier of the search. Then, the genetic operators will focus the search on the cheaper paths. Fig. 3 shows the number of successful attackers per generation for distinct elite sizes. The curves are similar independently of this size.

5.2 Breadth-first Exploration

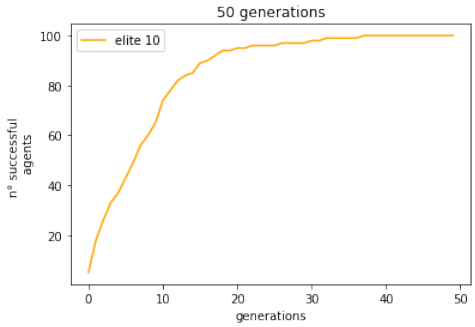
In these experiments, we update rule salience so that an attacker adopts a breadth-first strategy. We expected that the 100 attackers hitmark would have been reached sooner due to both the network size and the breadth-first search. In fact, the method analyzes all promising nodes the attacker can reach almost simultaneously. The 20 nodes network is included to highlight the distinct behaviours for the various sizes.

5.2.1 20 nodes, 100 attackers

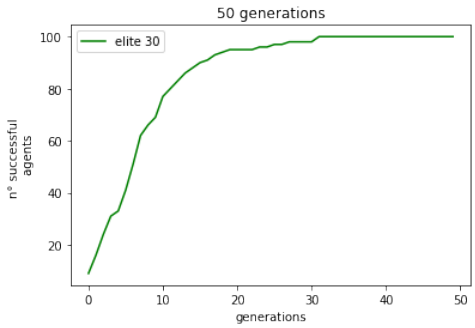
As in the previous experiments, we have made some preliminary observations on the 20 nodes network. Fig. 4 summarizes the curves produced by varying the elite size. It confirms that the time to reach the 100% hit-mark depends upon the elite size rather than upon individual performances.

5.2.2 80 nodes, 100 attackers

The number of paths and of the paths to a goal is much larger than in the 20 nodes case. Moreover, the ratio of the population size to the number of edges implies that a smaller elite size might result in a converge to a local minimum and not to the global one. Fig. 5 compares the various curves and shows how a larger elite size slows down the convergence.



(a) 10 elite size



(b) 30 elite size

Figure 2: 80 nodes, 100 attackers, 50 generations.

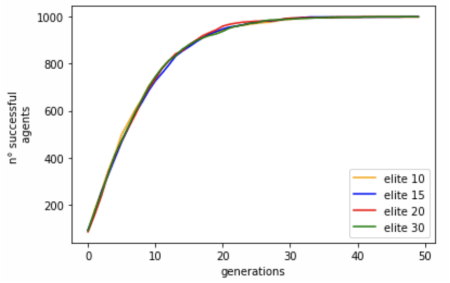


Figure 3: 80 nodes, 1000 attackers, 50 generations.

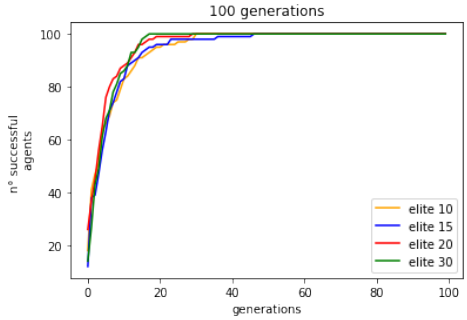


Figure 4: 20 nodes, 100 attackers, 100 generations.

5.2.3 80 nodes, 1000 attackers

The population size may result in a convergence before the 15th generation. Fig. 6 compares the various curves for the 10% to 30% of the population size. Here a larger elite size speeds up the convergence.

6 CONCLUSIONS

The results of our prototype confirm that a GA can discover the best strategy, i.e., the best set of rules, to attack a network. Further experiments will assess the performance of the GA against larger networks and with distinct attack success probabilities. Another interesting point is transfer learning. This is related to

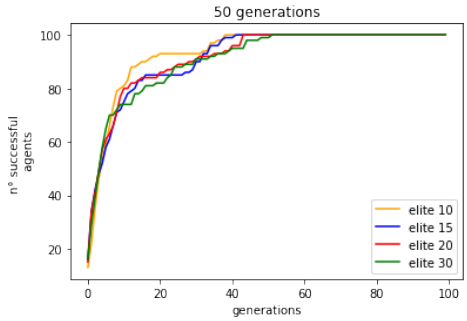


Figure 5: 80 nodes, 100 attackers, 100 generations.

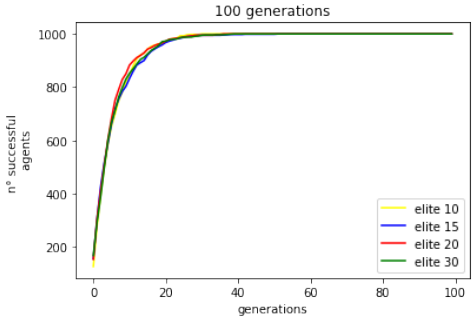


Figure 6: 80 nodes, 1000 attackers, 100 generations.

the learning speed-up when some rules of the population we train to acquire the control of a module M1 have previously learned to acquire the control of a distinct module of the same network.

REFERENCES

Aiyanyo, I., Samuel, H., and Lim, H. (2020). A systematic review of defensive and offensive cybersecurity with machine learning. *Applied Sciences*, 10:5811.

Allodi, L., Massacci, F., and Williams, J. (2021). The work-averse cyberattacker model: Theory and evidence from two million attack signatures. *Risk Analysis*, https://doi.org/10.1111/risa.13732.

Applebaum, A., Miller, D., Strom, B., Foster, H., and Thomas, C. (2017). Analysis of automated adversary emulation techniques. In *Proceedings of the Summer Simulation Multi-Conference*, SummerSim '17, San Diego, CA, USA. Society for Computer Sim. Int.

Baiardi, F. and Tonelli, F. (2022). Twin based continuous patching to minimize cyber risk. *Eur. Journal for Security Research*, pages 1–17.

Cafasso, M. (2016). CLIPS. https://clipspy.readthedocs.io/en/latest/. Accessed: 2020-02-26.

Inokuchi, M., Ohta, Y., Kinoshita, S., Yagyu, T., Stan, O., Bitton, R., Elovici, Y., and Shabtai, A. (2019). Design procedure of knowledge base for practical attack graph generation. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, Asia CCS '19, page 594–601, New York, NY, USA. Association for Computing Machinery.

Lake, B. and Baroni, M. (2018). Still not systematic after all these years: On the compositional skills of sequence-to-sequence recurrent networks. https://openreview.net/forum?id=H18WqugAb.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521:436–44.

Marcus, G. (2018). Deep learning: A critical appraisal. https://doi.org/10.48550/arXiv.1801.00631.

The MITRE Corporation (1999). CVE. https://www.cve.org/. Accessed: 2020-09-09.