Evolving neural networks for threat process mitigation with autoencoder-based fitness functions.

Henry D. Navarro H. [a,b,*], Héctor Bullejos [b], Carmelo Garrido [b] and Elena Naranjo [b]
[a] *Research and Development Lab, Vision Analytics, Avenida de Europa 19, 28224, Pozuelo de Alarcón, Madrid, Spain.*
E-mail: contact@henrynavarro.org
[b] *R&D Department, Capgemini Engineering, Calle Campezo, 1, 28022, Madrid, Spain.*
E-mail: engineering@capgemini.com

Abstract. Threat detection and mitigation is a crucial aspect of cybersecurity research, and researchers have dedicated significant efforts to developing effective methods for identifying and responding to various types of threats, such as viruses and phishing attacks. Once an intruder has gained access to a system, it is essential to detect any malicious processes and prevent them from causing harm. In this paper, it is presented a real-time hybrid method for detecting and mitigating malicious processes on a simulated network system. This approach combines deep neural networks and evolutionary algorithms with an autoencoder-based fitness function and detects in real-time anomalous behavior in each device connected to the network.

Keywords: Neuroevolution, Genetic Algorithms, Threat detection, Autoencoders, Deep Neural Networks

1. Introduction

Deep neuroevolution, autoencoders, and cybersecurity are three interrelated areas that have the potential to significantly impact the field of information security. Deep neuroevolution refers to the use of evolutionary algorithms to optimize artificial neural networks, which can be used to improve the accuracy and efficiency of various machine learning tasks. Autoencoders, on the other hand, are a type of neural network that can be used to learn efficient representations of data, and have been applied to a wide range of problems including dimensionality reduction, anomaly detection, and data compression. In the context of cybersecurity, these techniques can be used to improve the ability of systems to detect and mitigate various cyber threats, such as malware, phishing attacks, and network intrusions.

To study the effects of malicious processes, a virtual network was created consisting of 20 virtual instances using open source software. It was assumed that the malicious process had already infiltrated the system using techniques such as ransomware or social engineering. System data was collected from these virtual instances using open source libraries and used to train neural networks and apply evolutionary algorithms based on deep neuroevolution to detect and terminate the malicious process. This controlled environment facilitated the study of the behavior of malicious processes and the development of effective methods for detecting and responding to them

Evolve neural networks has been used in different tasks but mainly in train neural networks to play video games [17,9,7], although there are other applications as music generation [10] and modelling biological phenomena [15]. These neural network training

[*]Corresponding author. Henry D. Navarro H., E-mail: contact@henrynavarro.org.

methods rely on genetic algorithm operations to obtain the best agent to perform a specific task. However, there is not too much research related to this topic and cybersecurity as it is shown in the Section 2.

Evolutionary algorithms rely on a fitness function to identify the best agents in each generation. In this proposed method, the best agents are those that are able to "kill" the malicious process, returning the virtual instances to their "normal states". To determine the normal state of an instance, an autoencoder neural network is used. This neural network learns the healthy state of each instance and calculates the mean squared error (MSE) between its inputs and outputs. The closer the MSE is to zero, the healthier the state of the instance. The best agents are chosen according to this criterion and the malicious process is terminated using a simple command in any Linux based system (`kill`).

2. Related work

Autoencoders are neural networks that consist of an encoder to generate a vector of features in the latent space (of smaller dimension) and from the input data and a decoder, which seeks to reconstruct the input data from this latent vector [12]. This type of neural networks are used in anomaly detection [23,16,22], natural language processing [12] and dimensionality reduction [20,19].

It has been demonstrated by other researchers that autoencoders can be used to detect other types of threats, such as Denial of Service attacks ([8]). In their paper, an autoencoder is used to classify different types of Denial of Service attacks. The goal is not only to classify the state of a virtual machine as healthy or infected, but also to quantify the level of infection of a virtual instance. To achieve this, the autoencoder neural network is trained solely on healthy state data and the mean squared error (MSE) is used as a fitness function to select the best agents for threat detection. The lower the MSE, the healthier the state of the instance.

As for deep neuroevolution, there are no direct applications in the field of threat mitigation. However, other researchers have used genetic algorithms to improve network security [4,3,13]. In most of these papers, binary genes are used to block or detect certain network attacks, such as Syn-Flood and Smurf. To do so, common operations in genetic algorithms such as selection, crossover, and mutation are applied.

There have been several investigations into malware detection using deep neural networks [18,1,11]. However, these researchers train their models using gradient-based algorithms. In this paper, an alternative method for training a neural network to detect malicious processes that create idle tasks that affect common system values such as CPU, memory RAM, and hard drive is presented. By detecting and terminating these processes, the healthy state of a virtual instance can be restored using a genetic algorithm with an autoencoder-based fitness function.

Therefore, the main contributions of this paper are:

1. In [8], autoencoders are used as a classification neural network to detect and classify Denial of Service attacks. In the current paper, it is trained an autoencoder to detect anomalies caused by malicious processes that affect common parameters of the virtual instance.
2. Genetic algorithms have been used to improve security [4,3,13], but in the proposed method, it is used an autoencoder-based fitness function to evolve neural networks and select the best agents.
3. In [21], a comparison of different autoencoder architectures is conducted, while the proposed model in the current paper can detect any anomaly in each instance.
4. Other hybrid methods have been proposed [5,2], but these methods are only used to detect system threats, while the proposed method generates agents that mitigate system threats and are trained using a non-gradient-based algorithm.

3. Methodology

3.1. Network simulation

The virtual network to emulate the attacks was built and simulated based on the fact that an intruder had already obtained access to the network and already represents an internal threat, that is, the file that executes the malicious process is already found in the virtual instances. A network architecture with 20 instances has been virtualized using a network orchestrator with 4 physical machines.

These instances have their system data sent to a NoSQL database, where the variables explained in the Section 3.2 are monitored and used to develop the proposed models.

3.2. *Dataset*

To create the dataset, the normal traffic was first extracted from a virtualized environment to obtain "healthy" data for the system. A software tool was used to extract 97 features related mainly to CPU usage, memory RAM usage, and hard drive usage, which were divided into 10 main groups. Next, a cyber attack was introduced into each instance of the virtualized network and data was extracted to create an additional class: a malicious process called "Logic Bomb", which was developed by the team. It affects the regular parameters such as CPU, memory, and disk by creating idle processes.

3.2.1. *Central Processing Unit (CPU) features*

They refer to the load characteristics that the CPU of the instance may have. A malicious process acting on the instances is expected to drive CPU usage across the entire instance in order to crash it.

3.2.2. *Core features*

These features are similar to CPU variables, with the difference that they are disaggregated by core. Due to how virtualized instances have been created (1GB of memory, 1 core only), these variables are expected to have a high correlation with the CPU variables.

3.2.3. *Disk Input/Output (Disk I/O) features*

Disk I / O operations include both read and write or Input / Output (usually defined in KB / s) involving a physical disk. In simple words, it is the speed with which the data transfer takes place between the hard disk drive and RAM, or basically it measures the input / output time of the active disk. It is a performance measure and is therefore used to characterize storage disks as HDD, SSD, and SAN. A malicious process is expected to constantly perform read and write operations to cause a saturation of this hardware.

3.2.4. *Entropy*

Entropy available on the system.

3.2.5. *Filesystem*

This set of features refer to file system statistics on disk. Some malicious processes create an infinite loop that creates files indefinitely in order to saturate it.

3.2.6. *Memory Swap*

When the physical memory or RAM in the system becomes full, the *Memory Swap* is used. In this process, the inactive pages of the memory are moved to the swap space, creating more memory resources. This space is especially useful for systems that do not have RAM; however, it is slower to access because it is located on the hard drive and should not be considered as an alternative to RAM. As mentioned previously, a malicious process is expected to crash the RAM memory and then begin consuming the resources of the swap memory.

3.2.7. *Memory hugepages*

Hugepages are useful in managing virtual memory on Linux systems. As the name implies, they help manage large pages in memory that are larger than the default (usually 4KB). Hugepages is useful for both 32-bit and 64-bit configurations. Hugepages sizes can range from 2MB to 256MB, depending on the kernel version and hardware architecture. A malicious process is expected to increase the values of this variable.

3.2.8. *Socket summary*

These variables refer to the summary of open socket metrics in the system. A socket is nothing more than a communication channel between two programs that run on different computers or even on the same computer. Malicious processes seeking to attack the network are expected to affect these variables.

4. Anomaly detection classification for fitness function

An autoencoder neural network is an unsupervised learning algorithm that applies backpropagation, configuring the target values to be equal to the inputs. That is, the response variable that the machine learning algorithm tries to learn is such that $y^{(i)} = x^{(i)}$, where $x^{(i)} = \{x^{(1)}, x^{(2)}, \dots, x^{(n)}\}$ are the input variables detailed in Section 3.2 and the Appendix A.

4.1. *Autoencoder architecture*

An autoencoder consists of two parts: an encoder, denoted as f_ϕ, and a decoder, denoted as g_θ. The entire autoencoder neural network can be represented as $h_{W,b} = g_\theta \circ f_\phi$, where W and b represent the weight matrix and bias vector of the neural network, and h represents the final transformation function or hypothesis.

Next, n_l denotes the number of layers in the network. In this case, a network with three hidden layers was taken, therefore $n_l = 5$. The l-th layer is denoted as L_l, therefore L_1 is the input layer and L_{n_l} denotes the output layer. The parameters $(W, b) = (W^{(i)}, b^{(i)})$, with $i = 1, \ldots, 5$ are also denoted, where $W^{(l)}$ represents the weight matrices (weights) of each layer and $b^{(i)}$ represents the bias vector (bias) associated with the connections between unit (neuron) j in layer l and unit i in layer $l+1$. Similarly, the activation function that "connects" layer $l-1$ with layer l is denoted as $f_{\cdot l}$. Finally, s_i denotes the number of neurons (units) in layer i.

To reduce the dimensionality of the dataset, a selection of variables is made, as explained in the results Subsection 6.2.1. The number of neurons in the encoder is selected such that $s_i = [\![\frac{s_{i-1}}{2}]\!]$ (half the number of neurons in the previous layer), while in the decoder, $s_i = [\![2 \times s_{i-1}]\!]$ (double the number of neurons in the previous layer). The symbol $[\![\cdot]\!]$ represents the integer part of a real number.

Like any neural network, it requires activation functions that allow connecting all the units (neurons) of each layer. For the autoencoder, the activation functions are shown in Eqs. (1)

$$\begin{aligned} f_{\cdot 2} &= \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \\ f_{\cdot 3} &= ReLU(x) = x^+ = \max(0, x), \\ f_{\cdot 4} &= f_{\cdot 2} = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \\ f_{\cdot 5} &= f_{\cdot 3} = ReLU(x) = x^+ = \max(0, x) \end{aligned} \tag{1}$$

$a_i^{(l)}$ denotes the activation (the value at the output) of the unit i in the layer l. For $l = 1$, the notation $a_i^{(1)} = x_i$ is used to denote the i-th input as shown in Eqs (2).

$$\begin{aligned} z^{(l)} &= W^{(l-1)} a^{(l-1)} + b^{(l-1)} \\ a^{(l)} &= f_{\cdot l}(z^{(l)}) \end{aligned} \tag{2}$$

Graphically, the architecture of the autoencoder can be seen in Figure 1.

The autoencoder is trained using the well-known stochastic gradient descent algorithm with 60 epochs. The next step is to determine the decision threshold that will be used to classify instances as infected. That is, any instance with a value greater than this threshold will be considered infected or exhibiting anomalous behavior. To determine this threshold, the mean squared error between the input data and the output of the trained autoencoder is used, as shown in Algorithm 1.

Additionally, to meet the objective, it is necessary to determine the threshold at which an anomaly can be identified. To find this decision threshold, the mean squared errors for the training data (healthy instances) will be tested and the optimal one that maximizes accuracy in the confusion matrix will be determined, as shown in Algorithm 2.

Algorithm 1 Anomaly detection algorithm with the autoencoder

```
Input: Validation dataset $X_{val} = \{x^{(1)}, \ldots, x^{(n)}\}$,
autoencoder $h_{W,b} = g_\theta \circ f_\phi$.
Train dataset with normal behaviour $X$, Anomaly
train data $x^{(i)}$ $i = 1, ..., N$, threshold $\alpha$

1:  $\phi$, $\theta \leftarrow$ Train the autoencoder with normal data $X$
2:  for i=1 to N do
3:      $MSE(i) = \|x^{(i)} - g_\theta(f_\phi(x^{(i)})\|_2^2$    ▷ Mean Square Error (MSE)
4:      if $MSE(i) > \alpha$ then
5:          $x^{(i)}$ is an anomaly
6:          pred←1
7:      else
8:          $x^{(i)}$ is not an anomaly
9:          pred←0
10:     end if
11: end for
Output: Vector of classifications.
```

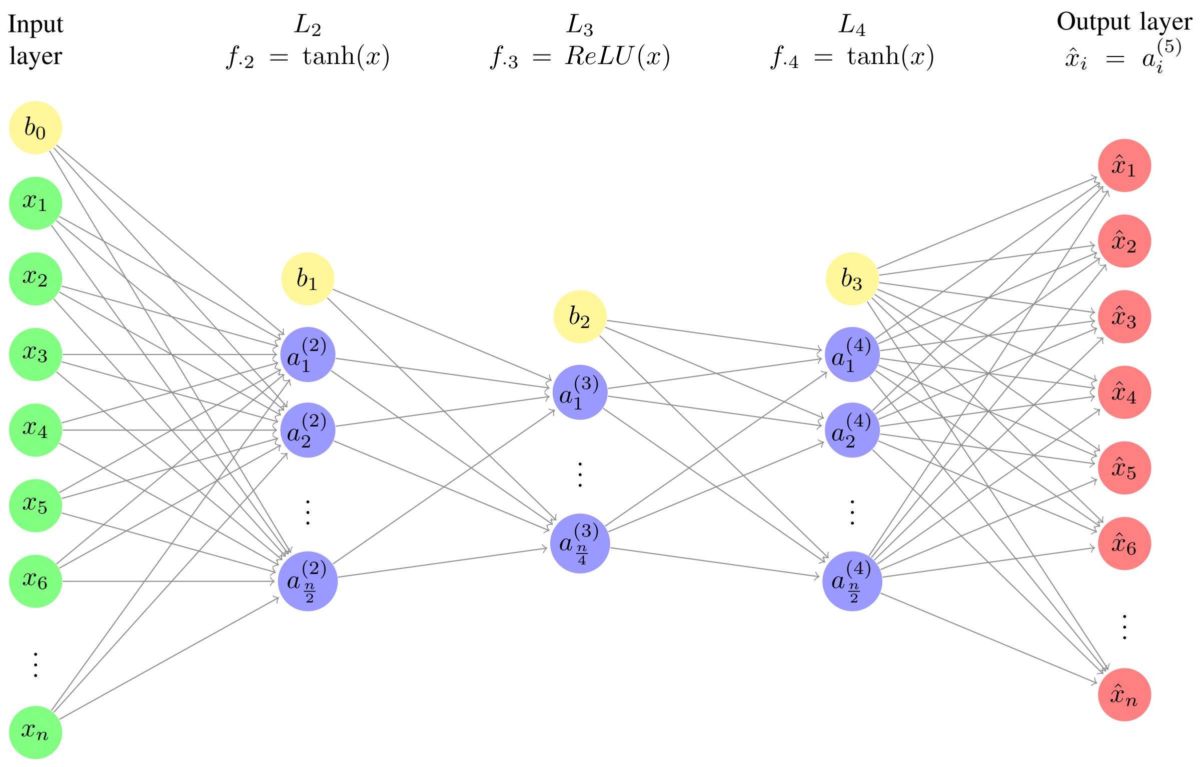
Input layer
L_2 $f_{\cdot 2} = \tanh(x)$
L_3 $f_{\cdot 3} = ReLU(x)$
L_4 $f_{\cdot 4} = \tanh(x)$
Output layer $\hat{x}_i = a_i^{(5)}$
b_0 x_1 x_2 x_3 x_4 x_5 x_6 x_n
b_1 $a_1^{(2)}$ $a_2^{(2)}$ $a_{\frac{n}{2}}^{(2)}$
b_2 $a_1^{(3)}$ $a_{\frac{n}{4}}^{(3)}$
b_3 $a_1^{(4)}$ $a_2^{(4)}$ $a_{\frac{n}{2}}^{(4)}$
$\hat{x}_1$ $\hat{x}_2$ $\hat{x}_3$ $\hat{x}_4$ $\hat{x}_5$ $\hat{x}_6$ $\hat{x}_n$

Fig. 1. Architecture of the neural network of the autoencoder type used for anomaly detection

5. Evolutionary algorithm approach

5.1. *Features for agents*

To identify malicious processes in an instance, we will study certain features that may be indicative of such processes. These variables do not directly describe the instance itself, but rather the processes running on it, since there is no dataset of processes that have been classified as malicious. The hybrid machine learning and genetic algorithms must be able to detect and mitigate these processes, using the correct order (`kill` process or not). In this case, the variables have been grouped into four main categories.

5.1.1. *File descriptors*

The file descriptor is a non-negative integer that uniquely identifies the files opened in a session. Each process is allowed to have up to nine file descriptors open at one time. The bash shell reserves the first three file descriptors (0, 1, and 2) for special purposes [6].

5.1.2. *Central Processing Unit (CPU)*

They refer to the load characteristics that the CPU may have for each process. A malicious process acting on the instances is expected to drive up CPU usage in order to crash it.

5.1.3. *Memory*

This set of variables refer to the RAM memory consumption that each process performs. A malicious process should carry out a high consumption of these type of resources.

5.1.4. *Temporal features*

As the name indicates, in this set are the variables that are related to time.

5.2. *Architecture of the agents (neural networks) of the population*

Following the notation in the Section 4.1, the neural network has parameters $(W, b) = (W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)})$. That is, $W^{(1)} \in \mathbb{R}^{15x8}$, while $W^{(2)} \in \mathbb{R}^{1x15}$. A softmax function is used as a network hypothesis. Thus, the hypothesis will be $h : \mathbb{R}^2 \to [0, 1]^2$ as the softmax

Algorithm 2 Obtaining the optimal α threshold for anomaly detection

```
Input: Train dataset $X_{train} = \{X^{(1)}_{train}, \ldots, X^{(n)}_{train}\}$, Validation dataset $X_{val} = \{X^{(1)}_{val}, \ldots, X^{(n)}_{val}\}$, autoencoder $h_{W,b}$.

  $\hat{X}_{train} \leftarrow h_{W,b}(X_{train})$    ▷ Evaluate the training data in the autoencoder.
  $Max\_MSE\_train \leftarrow \max\left(\sum_{i=1}^{n}\left(X^{(i)}_{train} - \hat{X}^{(i)}_{train}\right)^2\right)$    ▷ From the list above, the largest values of root mean square error are taken
  $\hat{X}_{val} \leftarrow h_{W,b}(X_{val})$    ▷ Same for validation data.
  $MSE_{val} \leftarrow \sum_{i=1}^{n}\left(X^{(i)}_{val} - \hat{X}^{(i)}_{val}\right)^2$
  $partition \leftarrow Max\_MSE\_train/1000$
  Initialize $\alpha$, $Accuracy_{optimal}$, $MSE_{optimal}$ with zero values.

  for i=1 to 1000 do    ▷ 1000 threshold partitions are tested, from 0 to $Max\_MSE\_train$ and the best is saved:
      $y^{(i)}_{val} \leftarrow anomaly(MSE_{val}, \alpha)$    ▷ Apply the Algoritmh 1 (vector of 1's y 0's)
      $Accuracy_{current} \leftarrow$ Get the accuracy from $(y_{val}, y^{(i)}_{val})$
      if $Accuracy_{current} > Accuracy_{optimal}$ then
          $Accuracy_{optimal} \leftarrow Accuracy_{current}$
          $\alpha \leftarrow MSE_{current}$
      end if
      $MSE_{current} \leftarrow MSE_{current} + partition$
  end for

Output: Optimal threshold $\alpha$.
```

function of the last layer of each neural network:

$$h_{W,b}(x) = \left(\frac{e^{a_1^{(3)}}}{\sum_{j=1}^{2} e^{a_j^{(3)}}}, \frac{e^{a_2^{(3)}}}{\sum_{j=1}^{2} e^{a_j^{(3)}}}\right) \quad (3)$$

Therefore, for the logical bomb attack, $n^{[2]} = 2$. That is, a vector as it is shown in Eq. (4):

$$\begin{aligned}(P_{\texttt{no}}, P_{\texttt{kill}}) = (&P(\texttt{no kill process} \mid \\ &X = x_1, \ldots, x_n), \\ &P(\texttt{kill process} \mid \\ &X = x_1, \ldots, x_n))\end{aligned} \quad (4)$$

such that $P_{\texttt{no}} + P_{\texttt{kill}} = 1$, where x_i are the monitored variables for that process, defined in Section 5.1. So the activation function in the last hidden layer will be given by the function softmax in Eq. (3).

The chosen neural network for this study can be seen in Figure 1.

5.3. *Evolving Neural Networks*

Therefore, each agent is a neural network known as a multilayer perceptron (MLP) with $n^{[0]} = 8$ *input units* (neurons in the input layer), due to the variables of the processes to analyze (See Section 5.1), $n^{[1]} = 15$ *hidden units* (neurons in the hidden layer) This design intention is to have twice as many neurons in the hidden layer as in the input layer, with $n^{[2]} = 2$ *output units* (neurons in the output layer). An illustration about the architecture of these agents is shown in Figure 2.

5.4. *Initialization*

There will be λ neural networks (initial population size) per generation, where λ is the maximum number of instances that can be virtualized in the environment according to computer resources. The weights $W^{[l]}_{ij}$ of the λ neural networks are initialized using the general rule for establishing weights in a neural network, which is to set them close to zero without being too small. This is done using the HEWEIGHTS defined in Algorithm 3.

Algorithm 3 He Initialization

```
1: function HEWEIGHTS(agent)
2:     n ← number of nodes of layer $l$    ▷ agent is the neural network
3:     y ← $\frac{1.0}{\sqrt{n}}$
4:     agent.weights ← $U(-y, y)$    ▷ random numbers with distribution $U(-y, y)$
5:     return agent
6: end function
```

Therefore, the initialization operation of the deep neuroevolution algorithm is a configuration of λ multi-

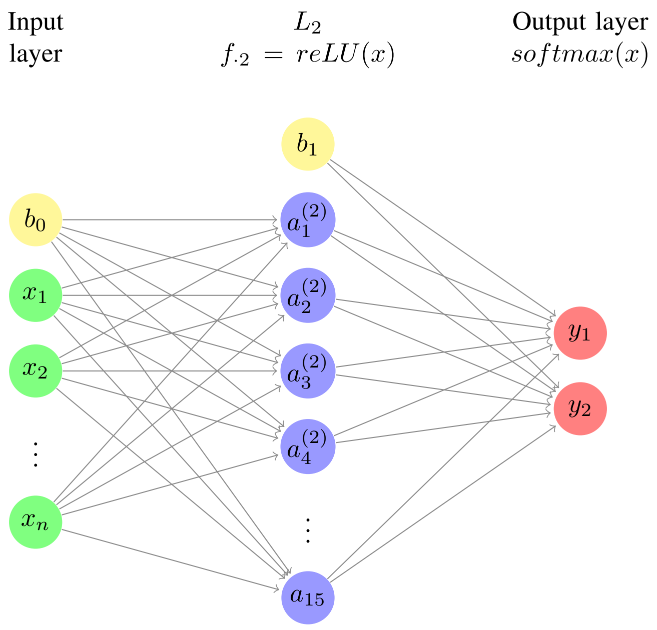

Fig. 2. Neural network architecture used for attack mitigation.

layer perceptrons with weights initialization $W_{ij}^{[l]}$ with uniform distribution as seen in the Algorithm 4.

Algorithm 4 Initialization

Input: population size or number of agents λ, number of neurons in input each layer: $n^{[0]}, n^{[1]}, n^{[2]}$, weights initialization function ϕ.

```
1: agents ← empty list
2: for $i = 1, \ldots, \lambda$ do
3:     init_agent ← create a MLP with $n^{[0]}, n^{[1]}, n^{[2]}$ neurons in each layer and ReLU as activation function.
4:     init_agent ← HEWEIGHTS(init_agent)    ▷ function that returns weights using He's initialization.
5:     agents[i] ← init_agent    ▷ add to list the agent initialized
6: end for
```

Output: λ neural networks with He's initialization.

5.4.1. *Evaluation*

This operation determines the quality of the agents that result from the initialization or after each operation (selection, crossing, mutation, elitism). To do this, we use the pretrained autoencoder with healthy data. The autoencoder provides a measure through mean square error, as described in Section 4. This allows to assess the fitness of the agents.

The goal of this model is to provide a measure that allows for the detection of anomalies in the data of virtual instances. To do this, the evaluation operation is based on the mean square error (MSE) between the input data $x_1, x_2, x_3, \ldots$ (instance data) and the output data $\hat{x}_1, \hat{x}_2, \hat{x}_3, \ldots$ of the autoencoder.

Thus, the evaluation operator or *fitness function* of the Deep Neuroevolution algorithm will be given by Eq. (5):

$$fitness = MSE = \frac{1}{n^{[0]}} \sum_{i=1}^{n^{[0]}} (x_i - \hat{x}_i)^2 \qquad (5)$$

where $x_1, x_2, x_3, \ldots$ are the input data of the autoencoder (instance data) and $\hat{x}_1, \hat{x}_2, \hat{x}_3, \ldots$ is the data in output the output of the autoencoder. **The closer the fitness function is to zero, the better the agent will be.**

5.4.2. *Selection and Crossing*

After the evaluation, those individuals/agents (from the n neural networks) with the best fitness (Eq. (5)) should be selected, in the case of study, a fitness value closer to zero will be an indicator of better agent.

Now, within the operations that will defined are two of the most important: Selection and Crossing. For the first operation, the name is quite intuitive and it seeks to select the best agents according to the fitness function (the closer to zero, the better). The best n_{top} will be selected, where

$$n_{top} = \left[\!\left[\frac{-1+\sqrt{1+4\cdot 2\cdot \lambda}}{2} \right]\!\right] + 1 \tag{6}$$

and $[\![x]\!]$ is the *floor* function. This is because the crossover operation requires reproducing $\lambda - 1$ agents from the best ones, which requires a number n_{top} such that $\sum_{i=1}^{n_{top}} i = \lambda$. Therefore, Eq. (6) is the solution to the Eq. (7)

$$\frac{n_{top}(n_{top}+1)}{2} = \lambda, \tag{7}$$

where λ is the number of agents in the initial population P_0

The best n_{top} agents are used to define the crossover operation, which aims to create a population of size $\lambda - 1$ by combining the weights W_{ij} of a "father" agent with those of several "mother" agents with a probability of 0.5. This is a Bernoulli event, in which a coin is flipped and, if the result is heads, the offspring will inherit the weight W_{ij} of the father; if the result is tails, it will inherit the weight W_{ij} of the mother. The detailed algorithm can be seen in Algorithm 5.

Algorithm 5 Selection and Crossing

```
1: $n_{top} = \left[\!\left[ \frac{-1+\sqrt{1+4\cdot 2\cdot \lambda}}{2} \right]\!\right] + 1$
2: function SELECTION AND CROSSOVER(agents,$n_{top}$)
3:     agents_top= select best $n_{top}$ agents from the list agents.
4:     children= empty list
5:     max_id = 1        ▷ Initialize a loop end parameter
6:     while max_id<=n-1 do
7:         for i = 1,...,max_id do
8:             mother=agents_top[i]        ▷ Neural network mother
9:             father=agents_top[max_id]        ▷ Neural Network father
10:            for each unit of neural network $W_{ij}^{[parent]}$ do
11:                coin=$U(0,1)$        ▷ Get random number with uniform distribution
12:                If coin$\leq 0.5$ then
13:                    $W_{ij}^{[children]}$=$W_{ij}^{[father]}$        ▷ Update each weight of each unit
14:                Else
15:                    $W_{ij}^{[children]}$=$W_{ij}^{[mother]}$        ▷ Update each weight of each unit
16:            end forchildren[i]=new_children
17:        end for
18:    end while
19:    Return children
20: end function
```

5.4.3. Mutation

In evolutionary algorithms it is convenient that for the new descendants formed by selection and crossing, some of their genes (in this case the weights of the neurons) can be subjected to a mutation with a low random probability of a change. This implies that some weights of the neurons of the initial λ neural networks, will be modified with a "small" change that depends on a value known as *mutation power* [17]. This hyperparameter is called σ.

The mutation occurs to maintain diversity within the population, without this operation, the values of the weights W_{ij} of the neural networks (agents) would only maintain the values obtained from initialization.

The operation of adding a value $\sigma \cdot \mathcal{N}(0,1)$ with $\sigma = 0.02$ to the weights W_{ij} of the units (neurons) of each neural network is performed in order to allow these weights to vary beyond the values they have taken in the initialization operation (Algorithm 4). This value σ, which is very small and has a distribution of $\mathcal{N}(0, 0.02^2) = \mathcal{N}(0, 0.0004)$, prevents the weights from having extreme changes while allowing them to vary. This value can be considered analogous to the learning rate in the case of stochastic gradient descent and is defined in Algorithm 6.

5.4.4. Elitism

The best agent mitigating the attack is selected as just one neural network, referred to as elitism. From the top n_{top} (Eq. (6)) agents are selected, a process known as *tournament elitism* is performed. This involves testing each agent (neural network) a total of m times in the virtualized environment with the infected instances, and calculating the average *fitness* (Eq. (5)) of these m tests. This average determines which agent is the best after the m executions. Finally, the elite agent (neural network) is added to the list of child agents (mutated_children), preserving the best of all **unmutated** agents without any alteration, allowing the agent with the best attack mitigation characteristics to

Algorithm 6 Mutation

```
1: $\sigma = 0.02$    ▷ mutation power
2: mutated_children= empty list
3: function MUTATION(children)    ▷ As input it needs the list returned by selection and crossover operation (Algorithm 5)
4:     for i = 1,...,total of children do
5:         for each weight $W_{ij}$ of unit of neural network do
6:             $r_{norm}$= get random number $\mathcal{N}(0,1)$
7:             $W_{ij}$=$W_{ij} + \sigma \cdot r_{norm}$    ▷ Update each weight of each unit
8:             new_children=assign to this neural network the weights $W_{ij}$.
9:         end for
10:
11:         mutated_children[i]=new_children
12:     end for
13:     Return mutated_children
14: end function
```

be kept. The Algorithm 7 explains how this operation is defined.

Algorithm 7 Elitism

```
1: $n_{top} = \left[\!\left[ \frac{-1+\sqrt{1+4\cdot 2\cdot \lambda}}{2} \right]\!\right] + 1$
2: function ELITISM(mutated_children,agents,$n_{top}$)
3:     agents_top= select best $n_{top}$ agents from the list agents.
4:     for i = 1,..., $n_{top}$ do
5:         agent_test = agents[i]
6:         fitness_agent= empty list
7:         j=1
8:         repeat
9:             test the agent in the virtualized environment
10:             fitness_agent[j]=calculate the fitness using Eq. (5).
11:             j=j+1
12:         until j=$m$
13:     end for
14:     fitness_mean[i]=$\frac{1}{m}\sum_{j=1}^{m}$fitness_agent[j]
15:     elite=select agent with best fitness_mean
16:     new_generation=add elite to mutated_children
17:     Return new_generation
18: end function
```

On the other hand, agents that kill all processes are penalized, as the neural network will learn to reduce the MSE given by the autoencoder based on killing all processes of the instance. When the agent tries to apply the `kill` command to all processes, the command is changed to not kill any processes. The neuroevolution algorithm is illustrated in Figure 3.

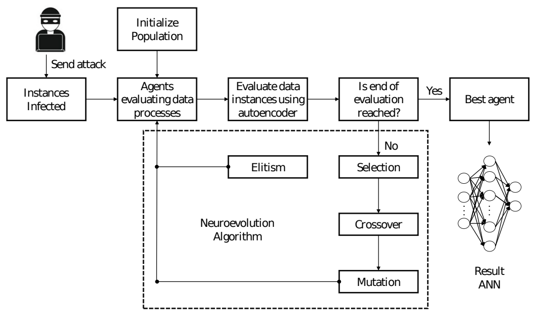

Fig. 3. Neuroevolution algorithm flow.

6. Experiments and Results

6.1. Feature selection

Due to the high dimensionality of the data used to train the autoencoder, it is necessary to discard certain features in order to improve the performance of the model. To reduce the number of variables, a preliminary study of each variable was conducted, comparing the results between the different states of each instance. The first variables to be deleted were those that remained unchanged throughout the data extraction, regardless of the state of the virtual instance. Through this process, 26 of the 97 initial variables were eliminated. A short list of some of these variables can be seen in Table 1.

It is well known that multicollinearity in data can significantly reduce the predictive power of many machine learning models. While an autoencoder is not a linear model, it is still beneficial to reduce the high dimensionality of the data. One effective way to do this is by considering the correlation between variables.

To reduce multicollinearity in the data, we first identified variables that had a strong correlation with each other (a Pearson's correlation coefficient higher than 0.7). After examining the relationships between these variables, we conducted a study on the variables themselves to determine which ones were redundant. In order to have a more comprehensive understanding of the system, we also tried to retain at least one representa-

Variable 1	Variable 2	Correlation
load 15	load norm 15	1.0
memory used bytes	memory free	1.0
filesystem free	filesystem available	0.99
entropy available bits	entropy pct	0.99
core idle pct	cpu idle pct	0.99
cpu iowait pct	core iowait pct	0.99
diskio io time	diskio write count	0.99
process summary sleeping	process summary total	0.98
cpu total pct	core idle pct	0.98
fsstat total size used	fsstat total size free	0.987
diskio iostat await	diskio iostat write await	0.92
socket summary all count	socket summary udp all count	0.91
diskio iostat queue avg size	diskio iostat busy	0.90

Table 1

Pearson correlation (ρ). List of some variables with correlation coefficient $\rho > 0.90$.

tive variable for each module. Through this process, 58 out of the 97 total variables were discarded."

6.2. *Results*

The results show that it is possible to determine the state of a instance using autoencoders and also detect malicious processes that affect the normal behaviour of a computer. So the results are presented in two steps: anomaly detection results and evolutionary algorithms for neural networks.

6.2.1. *Autoencoder model results*

The autoencoder model was trained with only the healthy state of the data using the stochastic gradient descent algorithm with 60 epochs. The mean square error was monitored during the training process, the results are shown in Figure 4.

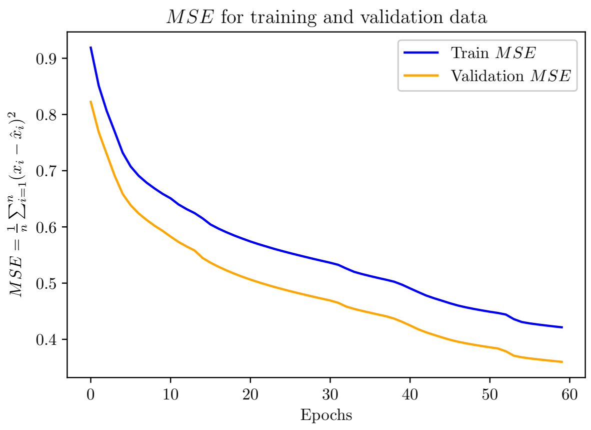

Fig. 4. Epochs vs. MSE. Mean square error for a trained (Train) and validated (Validation) data for the autoencoder neural network with data from healthy machines after feature selection.

After the model is trained, it is necessary to determine the decision threshold from which the state of the machine is considered an anomaly, for this, the Algorithm 2 was used. Which gave as a result an optimal threshold $\alpha \approx 1.1945\ldots$, the closer value to this the state of the instance is, the greater the consideration will be as a healthy state of the instance. Therefore, the best "parents" will be those that after killing the processes, lead to the state of the instances at values closer to this α. After the data extraction of anomaly values, the results of this analysis can be seen in Figure 5.

After the model is trained, it is necessary to determine the decision threshold at which the state of the machine is considered anomalous. To do this, we used Algorithm 2, which resulted in an optimal threshold $\alpha \approx 1.1945\ldots$. The closer the value of the state of the instance is to this threshold, the greater the consideration of it as a healthy state of the instance. Therefore, the best "parents" will be those that, after killing the processes, lead to the state of the instances with values that are below this α. The results of this analysis can be seen in Figure 5.

With this decision threshold, a confusion matrix is obtained in order to consider the autoencoder not only as a anomaly detector but also as a classifier and to provide results about classification metrics. The results are shown in Table 2.

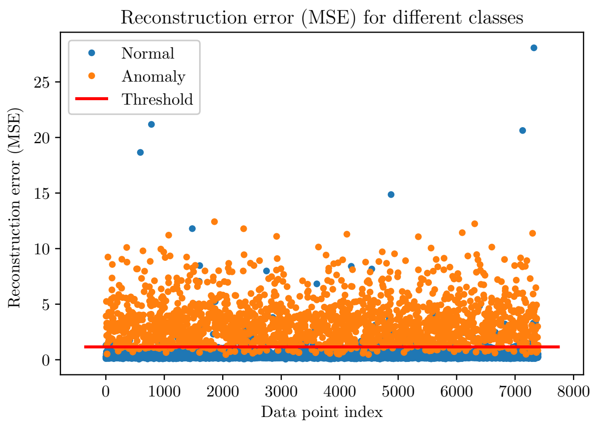

Fig. 5. Threshold decision from the mean square error (MSE). The red line represents the threshold decision $\alpha \approx 1.1945\ldots$ to classify the anomaly.

		Real	
		L. Bomb	Healthy
Predicted	L. Bomb	1629	174
	Healthy	135	5457

Table 2

Classification considering $MSE > \alpha$ is infected.

A 10-fold cross validation is performed, and the performance measures (mean of the 10-fold results) shown in Eqs. (8) are obtained.

$$\begin{aligned} Accuracy &= \frac{\text{Correctly classified}}{\text{Total of samples}} = 0.9632 \\ Precision &= \frac{\text{Correctly classified as infected}}{\text{Samples predicted as infected}} = 0.9335 \qquad (8) \\ Recall &= \frac{\text{Correctly classified as infected}}{\text{Samples actually are infected}} = 0.9136 \end{aligned}$$

These results demonstrate that the autoencoder model is capable of classifying states of the virtual machines and can be considered a good anomaly detector. In this training, only the healthy state of the instance was used.

6.2.2. *Neuroevolution results*

The most important results of this paper are presented in this section. It will be shown how a neural network trained with a hybrid machine learning algorithm incorporating evolutionary algorithms is able to reduce the impact of these attacks on virtual instances.

In Figure 6, it can be seen how the Mean Square Error between the input data and the output provided by the autoencoder is almost brought to zero, demonstrating the desired result of the best agent successfully eliminating the malicious process and reducing the attack's impact after just a few generations.

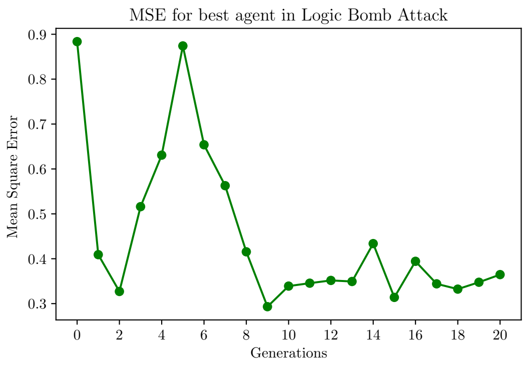

Fig. 6. MSE for the best agent in each generation using an algorithm to evolve neural networks

On the other hand, an agent that kills all the processes will also reduce the effect of the threat, but the instance will be unusable which is not the goal of this study. Figure 7 shows how the number of processes killed by the best agent are reduced in each generation, which means it is reducing the effect of the threat while killing the right process (malicious).

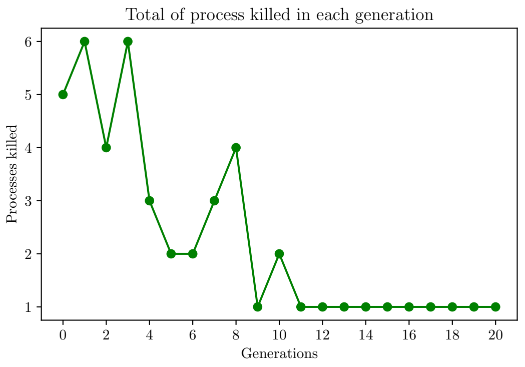

Fig. 7. MSE for the best agent in each generation using an algorithm to evolve neural networks

7. Conclusions

In this paper, a new method for reducing the impact of cyber attacks, as well as the application of neuroevolution algorithms in cyber security, has been presented. The results of other researchers [8,14,21] using autoencoders to classify cyber attacks have been confirmed, although a different methodology and type of cyber attack were used in this case. Autoencoders can provide a measure of how close a virtual instance is to a healthy state, allowing the combination of this measure with a neuroevolution algorithm to select the best agents, resulting in an innovative hybrid algorithm capable of detecting malicious processes running in a shell. A hybrid method for detecting and mitigating malicious processes on simulated network systems in real-time is presented in this paper. By combining deep neural networks and evolutionary algorithms with an autoencoder-based fitness function, anomalous behavior in each device connected to the network is able to be detected. Using a virtual network and open source libraries, the behavior of malicious processes was studied and effective methods for detecting and responding to them were developed. Results show that the proposed approach is successful in detecting and terminating malicious processes, returning the virtual instances to their normal states as determined by an autoencoder-based fitness function. This work represents a promising direction for future research in the field of cybersecurity and deep neuroevolution.

Appendix

A. Appendix: Description of each variable

A.1. Central Processing Unit (CPU) Features

cpu_cores: The number of CPU cores present on the host. The non-normalized percentages will have a maximum value of 100% · cores. The normalized percentages already take this value into account and have a maximum value of 100%.

cpu_user_pct: The percentage of CPU time spent in user space. On multi-core systems, you can have percentages that are greater than 100%. For example, if 3 cores are at 60% use, then the system.cpu.user.pct will be 180%.

cpu_system_pct: The percentage of CPU time spent in kernel space.

cpu_nice_pct: The percentage of CPU time spent on low-priority processes.

cpu_idle_pct: The percentage of CPU time spent idle.

cpu_iowait_pct: The percentage of CPU time spent in wait (on disk).

cpu_irq_pct: The percentage of CPU time spent servicing and handling hardware interrupts.

cpu_softirq_pct: The percentage of CPU time spent servicing and handling software interrupts.

cpu_steal_pct: The percentage of CPU time spent in involuntary wait by the virtual CPU while the hypervisor was servicing another processor.

cpu_total_pct: The percentage of CPU time spent in states other than Idle and IOWait.

A.2. Core features

core_user_pct: The percentage of CPU time spent in user space.

core_system_pct: The percentage of CPU time spent in kernel space.

core_nice_pct: The percentage of CPU time spent on low-priority processes.

core_idle_pct: The percentage of CPU time spent idle..

core_iowait_pct: The percentage of CPU time spent in wait (on disk).

core_irq_pct: The percentage of CPU time spent servicing and handling hardware interrupts.

core_softirq_pct: The percentage of CPU time spent servicing and handling software interrupts.

core_steal_pct: The percentage of CPU time spent in involuntary wait by the virtual CPU while the hypervisor was servicing another processor. Available only on Unix.

A.3. Disk Input/Output (Disk I/O) features

diskio_read_count: The total number of reads completed successfully.

diskio_write_count: The total number of writes completed successfully.

diskio_read_bytes: The total number of bytes read successfully. On Linux this is the number of sectors read multiplied by an assumed sector size of 512.

diskio_write_bytes: The total number of bytes written successfully. On Linux this is the number of sectors written multiplied by an assumed sector size of 512.

diskio_read_time: The total number of milliseconds spent by all reads.

diskio_write_time: The total number of milliseconds spent by all writes.

diskio_io_time: The total number of of milliseconds spent doing I/Os.

diskio_iostat_read_request_merges_per_sec: The number of read requests merged per second that were queued to the device.

diskio_iostat_write_request_merges_per_sec: The number of write requests merged per second that were queued to the device.

diskio_iostat_read_request_per_sec: The number of read requests that were issued to the device per second.

diskio_iostat_wirte_request_per_sec: The number of write requests that were issued to the device per second.

diskio_iostat_read_per_sec_bytes: The number of Bytes read from the device per second.

diskio_iostat_read_await: The average time spent for read requests issued to the device to be served.

diskio_iostat_write_per_sec_bytes: The number of Bytes write from the device per second.

diskio_iostat_write_await: The average time spent for write requests issued to the device to be served.

diskio_iostat_request_avg_size: The average size (in bytes) of the requests that were issued to the device.

diskio_iostat_queue_avg_size: The average queue length of the requests that were issued to the device.

diskio_iostat_await: The average time spent for requests issued to the device to be served.

diskio_iostat_service_time: The average service time (in milliseconds) for I/O requests that were issued to the device.

diskio_iostat_busy: Percentage of CPU time during which I/O requests were issued to the device (bandwidth utilization for the device). Device saturation occurs when this value is close to 100%.

A.4. Entropy

entropy_available_bits: The available bits of entropy.

entropy_pct: The percentage of available entropy, relative to the pool size of 4096.

A.5. Filesystem

filesystem_available: The disk space available to an unprivileged user in bytes.

filesystem_files: The total number of file nodes in the file system.

filesystem_free: The disk space available in bytes.

filesystem_free_files: The number of free file nodes in the file system.

filesystem_total: The total disk space in bytes.

filesytem_used_bytes: The used disk space in bytes.

filesystem_used_pct: The percentage of used disk space.

fsstat_count: Number of file systems found.

fsstat_total_files: Total number of files.

fsstat_total_size_free: Total free space.

fsstat_total_size_used: Total used space.

fsstat_total_size_total: Total space (used plus free).

A.6. Memory Swap

memory_swap_pct: Total swap memory.

memory_swap_used_bytes: Used swap memory in bytes.

memory_swap_free: Available swap memory.

memory_swap_used_pct: The percentage of used swap memory.

A.7. Memory hugepages

memory_hugepages_total: Number of huge pages in the pool.

memory_hugepages_used_bytes: Memory used in allocated huge pages in bytes.

memory_hugepages_used_pct: Percentage of huge pages used.

memory_hugepages_free: Number of available huge pages in the pool.

memory_hugepages_reserved: Number of reserved but not allocated huge pages in the pool.

memory_hugepages_surplus: Number of overcommited huge pages.

memory_hugepages_default_size: Default size for huge pages.

A.8. Socket summary

socket_summary_all_count: All open connections.

socket_summary_all_listening: All listening ports.

socket_summary_tcp_memory: Memory used by *Transmission Control Protocol* (TCP) sockets in bytes, based on number of allocated pages and system page size.

socket_summary_tcp_all_orphan: A count of all orphaned tcp sockets.

socket_summary_tcp_all_count: All open TCP connections.

socket_summary_tcp_all_listening: All TCP listening ports.

socket_summary_tcp_all_established: Number of established TCP connections.

socket_summary_tcp_all_close_wait: Number of TCP connections in *close_wait* state.

socket_summary_tcp_all_time_wait: Number of TCP connections in *time_wait* state.

socket_summary_udp_memory: Memory used by UDP sockets in bytes, based on number of allocated pages and system page size.

socket_summary_udp_all_count: All open UDP connections.

B. Appendix: Description of each variable for the agents.

B.0.1. File descriptors

process_fd_limit_hard: limit hard on the number of file descriptors opened by the process. The hard limit can only be increased by root.

process_fd_limit_soft: limit soft on the number of file descriptors opened by the process. The process can change the limit soft at any time.

process_fd_open: the number of file descriptors opened by the process.

B.0.2. Central Processing Unit (CPU)

process_cpu_total_norm_pct: Percentage of CPU time that the process consumes since the last event. This value is normalized by the number of CPU cores and ranges from 0% to 100%.

B.0.3. Memory

process_memory_size: The total in bytes of virtual memory that the process has.

process_memory_rss_bytes: The Resident Set Size (RSS) in bytes. The proportion of memory used by a process that is held in main memory (RAM), that is, the memory that the process occupied in main memory or RAM.

process_memory_share: the shared memory in bytes that the process uses.

B.0.4. Temporal features

process_cpu_start_time_seconds: The time (in seconds) since the process started.

References

[1] I. Ahmad, A. B. Abdullah, and A. S Alghamdi. Application of artificial neural network in detection of probing attacks. In *2009 IEEE Symposium on Industrial Electronics Applications*, volume 2, pages 557–562, 2009.

[2] Abdullah Shawan Alotaibi. A hybrid attack detection strategy for cybersecurity using moth elephant herding optimisation-based stacked autoencoder. *IET Circuits, Devices & Systems*, 15(3):224–236, 2021.

[3] Ehab Bader and Hebah H. O. Nasereddin. Using genetic algorithm in network security. *International Journal of Research and Reviews in Applied Sciences*, 5:148–154, 11 2010.

[4] Zorana Banković, Dušan Stepanović, Slobodan Bojanić, and Octavio Nieto-Taladriz. Improving network security using genetic algorithm approach. *Computers & Electrical Engineering*, 33(5):438 – 451, 2007. Security of Computers & Networks.

[5] Rajasekhar Batchu and Hari Seetha. A hybrid detection system for ddos attacks based on deep sparse autoencoder and light gradient boost machine. *Journal of Information &amp; Knowledge Management*, 2022.

[6] Richard Blum and Christine Bresnahan. *Linux Command Line and Shell Scripting Bible*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2015.

[7] Bobby D. Bryant and Risto Miikkulainen. Neuroevolution for adaptive teams. In *In Proceedings of the 2003 Congress on Evolutionary Computation (CEC 2003*, pages 2194–2201, 2003.

[8] Ferhat Ozgur Catak and Ahmet Mustacoglu. Distributed denial of service attack detection using autoencoder and deep neural networks. *Journal of Intelligent & Fuzzy Systems*, pages 1–11, 07 2019.

[9] Matthew Hausknecht, Joel Lehman, Risto Miikkulainen, and Peter Stone. A neuroevolution approach to general atari game playing. *IEEE Trans. Comput. Intell. AI Games*, 6(4):355–366, December 2014.

[10] Amy K. Hoover, Paul A. Szerlip, Marie E. Norton, Trevor A. Brindle, Zachary Merritt, and Kenneth O. Stanley. Generating a complete multipart musical composition from a single monophonic melody with functional scaffolding. In Mary Lou Maher, Kristian J. Hammond, Alison Pease, Rafael Pérez y Pérez, Dan Ventura, and Geraint A. Wiggins, editors, *ICCC*, pages 111–118. computationalcreativity.net, 2012.

[11] Rajesh Kumar. Malicious code detection based on image processing using deep learning. 11 2018.

[12] Jiwei Li, Thang Luong, and Dan Jurafsky. A hierarchical neural autoencoder for paragraphs and documents. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1106–1115, Beijing, China, July 2015. Association for Computational Linguistics.

[13] Louis Lobo and Suhas Chavan. Use of genetic algorithm in network security. *International Journal of Computer Applications*, 53:1–7, 09 2012.

[14] Vukosi Marivate, Fulufhelo Nelwamondo, and Tshilidzi Marwala. Autoencoder, principal component analysis and support vector regression for data imputation. *CoRR*, abs/0709.2506, 09 2007.

[15] Stefano Nolfi and D. Floreano. Evolutionary robotics: The biology, intelligence, and technology of self-organizing machines. 01 2001.

[16] Mayu Sakurada and Takehisa Yairi. Anomaly detection using autoencoders with nonlinear dimensionality reduction. pages 4–11, 12 2014.

[17] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning, 2019.

[18] Shun Tobiyama, Yukiko Yamaguchi, Hajime Shimada, Tomonori Ikuse, and Takeshi Yagi. Malware detection with deep neural network using process behavior. pages 577–582, 06 2016.

[19] Wei Wang, Ying Huang, Yizhou Wang, and Liang Wang. Generalized autoencoder: A neural network framework for dimensionality reduction. *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 496–503, 2014.

[20] Yasi Wang, Hongxun Yao, and Sicheng Zhao. Auto-encoder based dimensionality reduction. *Neurocomputing*, 184, 11 2015.

[21] Ahmed Latif Yaser, Hamdy M. Mousa, and Mahmoud Hussein. Improved ddos detection utilizing deep neural networks and feedforward neural networks as autoencoder. *Future Internet*, 14(8), 2022.

[22] Chong Zhou and Randy C. Paffenroth. Anomaly detection with robust deep autoencoders. *KDD 2017*.

[23] Bo Zong, Qi Song, Martin Renqiang Min, Wei Cheng, Cristian Lumezanu, Daeki Cho, and Haifeng Chen. Deep autoencoding gaussian mixture model for unsupervised anomaly detection. In *International Conference on Learning Representations*, 2018.