https://doi.org/10.2352/ISSN.2470-1173.2020.3.MOBMU-331

AI-based anomaly detection for cyberattacks on Windows systems - Creation of a prototype for automated monitoring of the process environment

Benjamin Yüksel, Klaus Schwarz, Reiner Creutzburg

Technische Hochschule Brandenburg, Department of Informatics and Media, IT- and Media Forensics Lab, Magdeburger Str. 50, D-14770 Brandenburg, Germany

Email: benjamin.yueksel@th-brandenburg.de, klaus.schwarz@th-brandenburg.de, creutzburg@th-brandenburg.de

Abstract

Cyber security has become an increasingly important topic in recent years. The increasing popularity of systems and devices such as computers, servers, smartphones, tablets and smart home devices is causing a rapidly increasing attack surface. In addition, there are a variety of security vulnerabilities in software and hardware that make the security situation more complex and unclear. Many of these systems and devices also process personal or secret data and control critical processes in the industry. The need for security is tremendously high.

The owners and administrators of modern computer systems are often overwhelmed with the task of securing their systems as the systems become more complex and the attack methods increasingly intelligent. In these days a there are a lot of encryption and hiding techniques available. They are used to make the detection of malicious software with signature based scanning methods very difficult. Therefore, novel methods for the detection of such threats are necessary.

This paper examines whether cyber threats can be detected using modern artificial intelligence methods. We develop, describe and test a prototype for windows systems based on neural networks. In particular, an anomaly detection based on autoencoders is used. As this approach has shown, it is possible to detect a wide range of threats using artificial intelligence. Based on the approach in this work, this research topic should be continued to be investigated. Especially cloud-based solutions based on this principle seem to be very promising to protect against modern threats in the world of cyber security.

Introduction

The protection and monitoring of computer systems has played a very important role for many years. Authenticity, integrity, confidentiality as well as the availability of different systems are to be ensured. The increasing spread of IT systems such as computers, servers, smartphones, tablets and smart home devices, however, has led to a rapidly increasing attack surface [1], [2], [3]. Often the owners or administrators of these systems are overstrained with the protection due to the complexity and variety of the devices. In addition, there are numerous vulnerabilities in software and hardware, which make the security situation unclear and make it even more difficult to achieve a good level of security. The number of threats from software and hardware security gaps has also been increasing steadily for several years. While there were 1020 known security vulnerabilities in 2000, this number will rise to 7946 by 2014. In 2018 there were 16,555 known vulnerabilities [8].

Additionally, the risk of known vulnerabilities is not significantly reduced. The severity of a vulnerability is indicated by a value of the so-called general vulnerability rating system. This ranges from zero to ten and the risk increases with an increasing value. This value had an average of 6.6 in 2000, and in the period from 2014 to 2018 it was still 6.2 [8]. Due to the increasing number of threats, this means that the threat level will continue to rise. Many of today's computer systems also store and distribute sensitive and personal information or control important processes. As a result, attacks are becoming more and more attractive to hackers and the need for computer security is constantly increasing. For these reasons, the investigation of new approaches, such as AI-based methods for the detection of IT security incidents, is a very important research area.

Autoencoder

Autoencoders are basically artificial neural networks that are designed to learn how to efficiently encode input data and then decode the generated code. The encoder of the autoencoder has a high number of input neurons and a significantly lower number of output neurons. It therefore compresses the data by mapping a usually very large input to a smaller number of output neurons. The generated code contains the essential characteristics of the input data. The decoding is performed by the decoder. The decoder expands the generated code and generates an output as close as possible to the input.

Autoencoders usually work lossy. The degree of loss depends on how similar the input data is to the training data. Input data with similar characteristics to the training data leads to a small loss while other data can lead to larger losses. By calculating the loss, the similarity of the input data to the training data can be determined. The loss can be determined by comparing the input and output data. In this way, an autoencoder can be used to detect anomalies within the data. Non-trivial anomalies within the essential characteristics of the data can also be detected. Autoencoders can thus detect deviations within the data that are not perceptible to humans [13]. Another advantage of autoencoders is that only positive examples of the data to be learned are needed

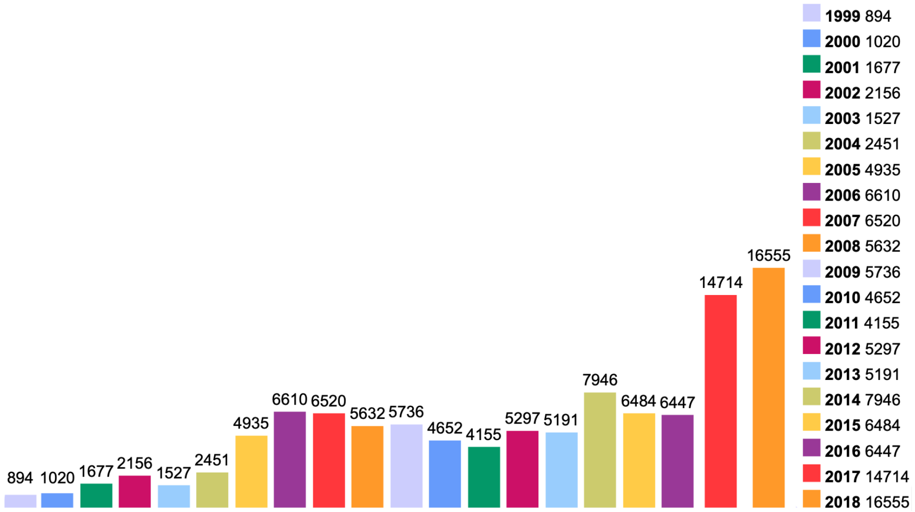


Figure 1. *History of known vulnerabilities in the period from 1999 to 2018 Source: https://www.cvedetails.com/browse-by-date.php*

for training. Related to the topic of this thesis these are data representing the normal operation of the system. It is not necessary to collect data from infected systems. So the training of autoencoders is basically done by creating input data at input and output. It is therefore supervised learning. During training, the weightings between the neurons are adjusted to produce an output for input that is as similar as possible [13].

Approach, Components and Programm Sequence

For this work, five test networks with different parameters were created. In the evaluation phase of this work, they will compete in five different scenarios to prove whether cyber threats can be detected with modern methods of artificial intelligence. The artificial neural networks developed for this work were implemented in the form of an autoencoder. Autoencoders consist of two components, an encoder and a decoder. The encoder maps a large amount of data onto a smaller amount. It therefore performs a compression. During compression, features are lost for which the encoder is not trained. The decoder then performs an opposite operation. It expands the data and produces the closest possible output to the original input. This operation also leads to greater losses if the data does not match the learned pattern. The whole process is generally a lossy procedure. However, there is a particularly large deviation between the input and output data if the input data deviates from the learned pattern. This effect can be used for anomaly detection and is applied to the developed prototype. The autoencoder was trained to take the usual snapshots of the system. Afterwards snapshots of the system were transferred to the Autoencoder. The smaller the deviation between input and output of the network, the closer the system state is to the desired normal state. Since malicious software usually makes changes to a system that do not match the normal behavior, this leads to an altered deviation between input and output values of the autoencoder. Such a change can be detected, depending on the size of the deviation.

The software basically consists of three components. These are a user interface, an export function for the running processes and a learning component. The user interface is the controlling component of the program. It coordinates and controls the flow of data for the process exporter and the learning and evaluation scripts. The export of running processes is used to read as much useful information about the running system as possible. It also provides export functions for json files and a proprietary binary format. The learning scripts are used for initial learning of the artificial neural network based on the collected data about the computer system. The evaluation scripts are used to evaluate the current system state. They access the learned neural network and use it to estimate the current threat situation of the system.

The execution of the entire program can be divided into two phases. The first phase is the learning phase. This serves to initialize the entire system. During the learning phase, a large amount of information about the system is first collected. This is done when the user interface executes the process export. This information is in the form of snapshots, each containing a list of all processes and associated information. These snapshots are taken over a long period of time and at short intervals. This creates the required large amount of data. They represent the normal state of the system. The artificial neural network is then trained on the basis of this data. As soon as the process of data collection is completed, the process exporter sends a signal to the user interface. This now starts the actual learning process. The training takes place by executing the learning scripts with the corresponding parameters. The neural network is now gradually trained to the normal state of the system in many epochs. After all learning epochs have been completed, the neural network is saved and the training data is removed from the system. The learning script now also sends a signal to the user interface that the process is complete. The user interface then calculates the threshold values for when a threat should be displayed. The threshold values are calculated by executing the evaluation script once on a small number of snapshots. The thresholds are then saved and the initialization process of the system is complete. The user interface now switches to Surveillance Mode.

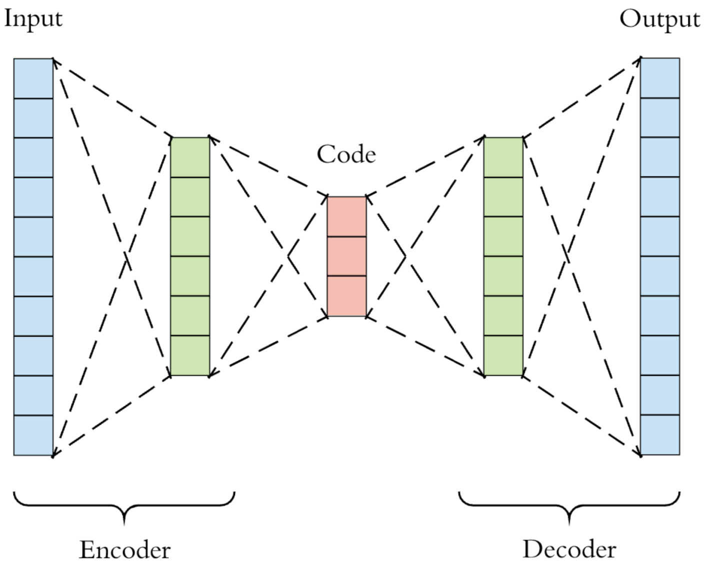
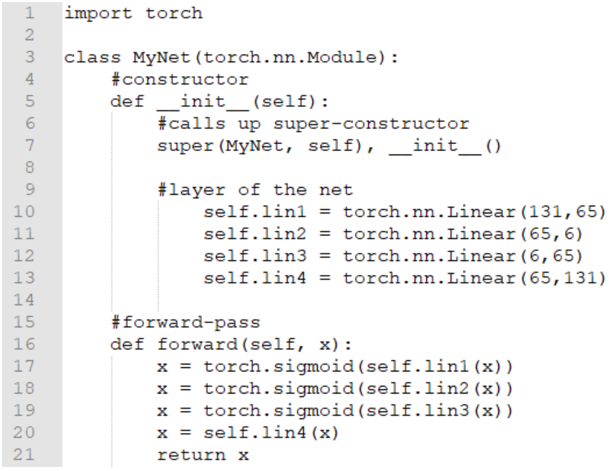


Figure 2. *Schematic representation of an autoencoder Source: https://towardsdatascience.com/generating-images-with-autoencoders-77fd3a8dd368*

In Surveillance Mode, the system is scanned for threats in real-time. This process represents an infinite loop. In this loop, the process exporter is started first to collect a small number of snapshots of the system. When the process is complete, another signal is sent to the user interface. The user interface then starts the evaluation of the collected snapshots by the evaluation script. Once the evaluation is complete, the threat values are transmitted back to the user interface. The user interface evaluates the results and notifies the user if the threshold values are exceeded or a threat is detected.

Prototype Development

For the development of the neural networks in this work, mainly the Python library PyTorch was used. PyTorch is a library for working with tensors. A tensor can be simply described as a vector of n-dimensional vectors. Tensors are used to represent data in deep learning. PyTorch was developed especially for deep learning tasks under the programming language Python and is based on the library Torch written in Lua [14] [15]. The first step in creating an artificial neural network with PyTorch is to create a class that describes the neural network itself. This class is called a model here. PyTorch already provides the base class torch.nn.Module for this purpose. The created model class must inherit from this base class. In the constructor of the model class the super constructor is called first. This super constructor performs all necessary initializations. Immediately afterwards, the layers of the artificial neural network are created. Various functions are available for this in torch.nn. For example, the function Linear creates a linear layer. The first parameter is the number of inputs and the second parameter is the number of outputs. The next layer must always have as many inputs as the previous layer has outputs. Once the structure of the network has been defined, another method called forward is required. This describes the forward pass and thus the calculations to be performed in the individual layers. An example of such a class is given below. It represents an autoencoder with five layers. The sigmoid function is used as activation function.

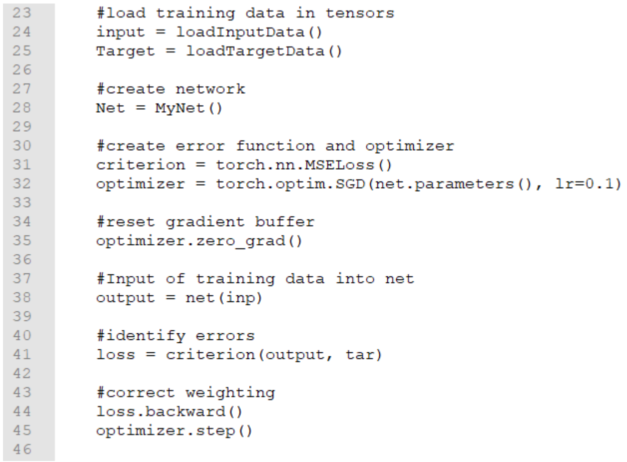


```
import torch

class MyNet(torch.nn.Module):
    #constructor
    def __init__(self):
        #calls up super-constructor
        super(MyNet, self), __init__()

        #layer of the net
            self.lin1 = torch.nn.Linear(131,65)
            self.lin2 = torch.nn.Linear(65,6)
            self.lin3 = torch.nn.Linear(6,65)
            self.lin4 = torch.nn.Linear(65,131)

    #forward-pass
    def forward(self, x):
        x = torch.sigmoid(self.lin1(x))
        x = torch.sigmoid(self.lin2(x))
        x = torch.sigmoid(self.lin3(x))
        x = self.lin4(x)
        return x
```

Figure 3. *Code extract of class with autoencoder*

Now that the structure of the network has been defined, an object of the network can be created. The network can then be trained. For training with PyTorch an optimizer from torch.optim and an error function from torch.nn is required. For the optimizer different parameters are expected, depending on the method. These must be taken from the PyTorch documentation. Basically the training runs after the following steps. First, optimizer and error function are created. Then the gradient buffers of the optimizer are reset with the function zero_grad. Afterwards a part of the training data is put into the network and the output is calculated. The error function is now used to determine the error between the output of the network and the expected value. To adjust the weights in the network, the backward and step functions of the error function or optimizer are called. The weights are now corrected and you can continue with the next training data.

In the following example the optimizer torch.optim.SGD and the error function torch.nn.MSELoss are used.



```
23      #load training data in tensors
24      input = loadInputData()
25      Target = loadTargetData()
26
27      #create network
28      Net = MyNet()
29
30      #create error function and optimizer
31      criterion = torch.nn.MSELoss()
32      optimizer = torch.optim.SGD(net.parameters(), lr=0.1)
33
34      #reset gradient buffer
35      optimizer.zero_grad()
36
37      #Input of training data into net
38      output = net(inp)
39
40      #identify errors
41      loss = criterion(output, tar)
42
43      #correct weighting
44      loss.backward()
45      optimizer.step()
46
```

Figure 4. *Code extract of class with autoencoder*

Manual loading of the training data is required in PyTorch. There is no standardized way. The only requirement is that the data is afterwards stored in a PyTorch tensor. The library offers various conversion functions to convert lists and tensors, for example. Tensors themselves can be described as multidimensional, nested vectors. After the net has been trained, it can be used to evaluate data. This corresponds to line 38 in the code example above and will not be discussed again. In practice, the code shown above would still have to be extended by two loops. Learning usually takes place in so-called epochs. Epochs represent repetition cycles. Furthermore, in a real application example, the data will be divided into several so-called batches. The batches are then fed into the network one after the other and the weightings are adjusted after each processed batch. Once all batches have been processed, the process starts again in the next epoch. This is repeated until the desired number of epochs is reached. After the basics of PyTorch have been clarified, the concrete implementation is now discussed. The implementation of the artificial neural network was divided into five Python scripts. These are NeuroNet.py, BinaryLoad.py, HelperFunctions.py, Train.py and Evaluate.py. The script NeuroNet.py contains a class for defining the neural network. The definition is done as in the example given above. The definitions of the networks in the individual experiments will be discussed later at the appropriate point. Furthermore the script contains functions for training and querying the net. The training is done with the function train. This function expects as parameters the created net of the class MyNet, a tensor with input data, a tensor with target data, the learning rate and the number of epochs. The net is queried using the ”ask” function. This function expects a created network of the class MyNet and the input data. The difference between input and output is returned. The difference is calculated by summing the absolute difference between the respective values of the input and output neurons:

$$\sum_{i=0}^{n} |x_i - y_i|.$$

In this formula n is the number of input and output elements. Since the neural network is an autoencoder, the number of input and output elements is identical. The variable x stands for an element of the input vector X with the index i. The variable y stands for an element of the output vector Y with the index i.

The script BinaryLoader.py contains a class that takes care of loading the process data in a binary format. The process data is read out, then stored in a list and returned. The script HelperFunctions.py contains helper functions. These are trivial and will not be discussed in detail. The scripts Train.py and Evaluate.py are ultimately the scripts to be executed for training or evaluating snapshots. These scripts are configured using the configuration file config.txt in the same directory. The parameters are specified line by line. Train.py expects as parameters the file name of the training data, the learning rate, the number of epochs, the file name of the network and a Boolean. The Boolean indicates whether the specified network is to be recreated or trained further. If the learning script is executed, these configuration parameters are loaded first and all output is redirected to the file logfile.txt. Then the function loadData is called. This function uses the BinaryLoader class to load the training data into a list. Afterwards a normalization of the data is performed. The normalization is necessary because the accepted input range is usually between zero and one. This is due to the activation functions used. Normalization is therefore performed according to the following formula:

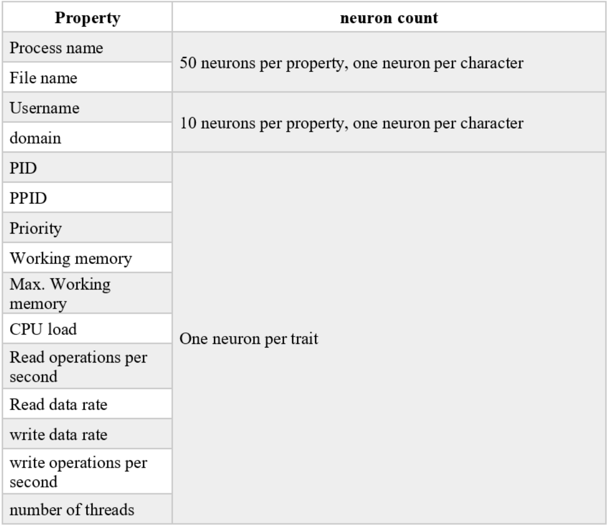
$$x = \frac{x}{\max(X)}.$$

The variable x represents the input value while X is the vector of all values of the same type. After the training data is normalized, a transformation of the data is performed. The data is divided into an adjustable number of batches and then converted into PyTorch tensors. The loading process is now complete. Now the neural network is created from the NeuroNet class and loaded if necessary. The training is done by calling the train function. Finally, the new neural net is saved and the script ends. The script Evaluate.py works in the same way as the training script. The loading of the data to be evaluated is done exactly as described above. After loading the data, the neural network is created and loaded based on the NeuroNet class. The normalized and transformed data are then applied to the neural network for each snapshot. The difference is then determined with the function ask. The difference is stored for each individual snapshot and the minimum, maximum and average of the values is then determined. These are used by the user interface to evaluate the threat level.

The neural network was developed with the help of the scripts explained above. As usual in this field, the development was done experimentally. In this case this means that different network structures, activation functions and parameters were tested. For each experiment a neural network was trained with the corresponding parameters. Then the error rate was determined based on the test data. The neural network with the lowest error rate was then used. The error rate is determined using the Mean Square Error function of PyTorch. The training and testing was always done with the same data so that the results are comparable. The data was collected from a Windows 10 virtual machine with Windows 10 version 1903/18362 dated 2019-05-21. The training data includes a total of 95,000 snapshots, with about 70 running processes. Due to hardware limitations, they were split into five binary files. Three of these files contain 15,000 snapshots each and the remaining two files contain 25,000 snapshots each. The test data consists of 15,000 system snapshots. In order to be able to display the learning curves of the networks graphically, the current error rates on the training data, for each learning epoch, were stored in a file. The learning curves are additionally indicated in the description of the tests.

Since a large number of attempts have been made, only the five most successful and therefore most relevant attempts are described below. For these experiments, a number of epochs of 1,000 and a learning rate of 0.08 have already been defined. These values were determined on the basis of previous experiments. A number of epochs of more than 1000 apparently did not lead to any significant improvement in the error rates. Moreover, the learning process at this number, with a duration of about 18 hours per experiment, is already very time-consuming. Therefore a further increase of the epochs, with the available means, was not reasonable. In combination with this number of epochs, the learning rate of 0.08 proved to be suitable. It led to a relatively fast reduction of the error rates. Lower values led to a very slow descent. Higher learning rates, on the other hand, always led to poor results, as so-called overfitting occurs or some minima are skipped. This means that the network only learns the training data and cannot process other data well. In addition, the optimum is often missed.

The number of input and output neurons is already determined by the training data and the type of net. As it is an autoencoder, the number of input and output neurons is identical. A number of 131 neurons is required in the first and last layer. The composition of this number can be seen in the following table.



Property	neuron count
Process name	50 neurons per property, one neuron per character
File name	
Username	10 neurons per property, one neuron per character
domain	
PID	One neuron per trait
PPID	
Priority	
Working memory	
Max. Working memory	
CPU load	
Read operations per second	
Read data rate	
write data rate	
write operations per second	
number of threads	

Figure 5. Composition of neuronnumber

For the properties process name, file name, user name and domain, all neurons that remain unused due to their length are set to zero. All other properties are numerical values with constant length. Accordingly, one neuron is provided for each of these properties.

Test Network 1

For the first test network an error rate of 0.0033 was achieved. The artificial neural network has 131 input and output neurons. The first hidden layer has 65 neurons. With the second hidden layer the number of neurons is reduced to ten. This layer represents the representation code. The data is therefore compressed to ten neurons and represented by them. With the third hidden layer the expansion of the data begins. It also has a number of 65 to create symmetry to the encoder. The output layer finally has 131 neurons. All layers are fully connected and use the sigmoid activation function. The following diagram shows the learning curve of the network. The error rate at the beginning of the learning process was 0.0302. The trained net reaches an error rate of 0.0033. This applies to both training and test data. When the test snapshot was analyzed during normal operation, an average hazard index of 278 was determined for the system.

Test Network 2

For the second test network an error rate of 0.0035 was achieved. This is slightly higher than in the first attempt. On the other hand, the small number of neurons led to a faster decline in the error rate. The artificial neural network has 131 input and output neurons. The first hidden layer has 65 neurons. The second hidden layer reduces the number of neurons to eight. This layer represents the representation code. With the third hidden layer the expansion of the data begins. It also has a number of 65 to create a symmetry to the coder. The output layer finally has 131 neurons. All layers are fully connected and use the sigmoid activation function. The following diagram shows the learning curve of the network. The error rate at the beginning of the learning process was 0.0302. The trained net reaches an error rate of 0.0035. When analyzing the test snapshot, during normal operation, an average hazard index of 322 was determined for the system.

Test Network 3

For the third test network, the number of neurons was further reduced. Instead of eight neurons in the second hidden layer, only six were used to test the further reduction of the code length. The input layer still has a number of 131 neurons, followed by the first hidden layer with 65 neurons. To maintain the symmetry of the network, the third hidden layer also has 65 neurons and the output layer has 131 neurons. All layers are fully connected and use the sigmoid activation function. The following diagram shows the learning process of the net. The error rate at the beginning of the learning process was also 0.0302. The trained network reaches an error rate of 0.0034. When analyzing the test snapshot, during normal operation, an average hazard index of 302 was determined for the system.

Test Network 4

For the fourth test network, the number of neurons was again reduced. Instead of six neurons in the second hidden layer, three were used to test a further reduction in code length. The input layer still has a number of 131 neurons, followed by the first hidden layer with 65 neurons. To maintain the symmetry of the network, the third hidden layer also has 65 neurons and the output layer has 131 neurons. All layers are fully connected and use the sigmoid activation function. The following diagram shows the learning process of the net. The error rate at the beginning of the learning process was 0.0302. The trained net reaches an error rate of 0.0038. When analyzing the test snapshot, during normal operation, an average hazard index of 287 was determined for the

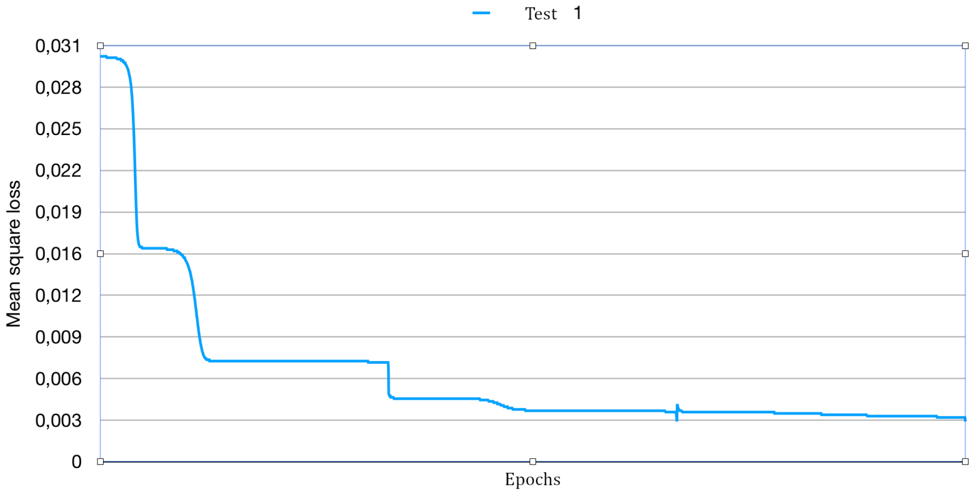


Figure 6. *Learning curve of Test Network 1*

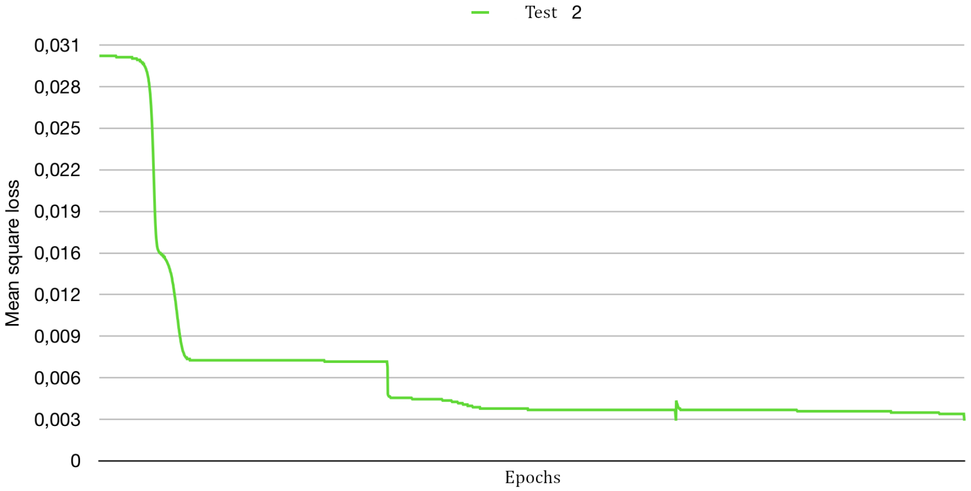


Figure 7. *Learning curve of Test Network 2*

system.

Test Network 5

For the fifth test network, two additional hidden layers were added to the network. Since the number of neurons and links is now much larger, the number of epochs for this experiment was doubled from 1,000 to 2,000. This proved to be useful for the larger net, as it requires more training. The input layer still has a number of 131 neurons, followed by the first hidden layer with 65 neurons. The second hidden layer has a number of 30 neurons. The third and thus middle layer has a number of 10 neurons. To maintain the symmetry of the network, the fourth hidden layer also has 30 neurons and the fifth has 65 neurons. The output layer again has 131 neurons. All layers are fully connected and use the sigmoid activation function. The following diagram shows the learning process of the network. The error rate at the beginning of the learning process was 0.0308. The trained net reaches an error rate of 0.0020. When analyzing the test snapshot, during normal operation, a hazard index of 234 was determined for the system.

Test Environment and Test Execution

The now following applicability tests serve to evaluate and analyze the developed software, the neural networks and their results. It shall be determined to what extent the detection of anomalies or malware within the system is possible with the developed prototype. Since the tests are carried out with real malware, some safety precautions must be taken beforehand. It must be ensured that the malware cannot leave the test system and possibly infect other systems. The following tests are therefore carried out in a shielded environment. The test system is run inside a virtual machine without network connection. The host system also has no network connection and was set up specifically for this purpose.

The test system is a Windows 10 operating system in the version 1903/18362 of 21.05.2019. All dependencies necessary for the execution of the developed software were installed on this system. In addition, various types of user software were installed to create an environment that is as realistic as possible. These include Google Chrome, Mozilla Firefox, LibreOffice, Adobe Flash Player, Steam, GIMP, Notepad++, Skype and all dependencies necessary for these applications. The host system is also Win-

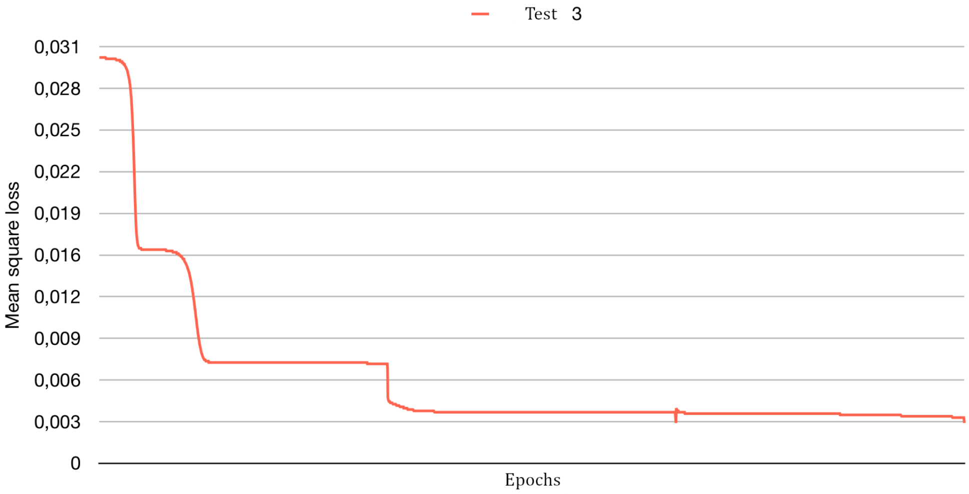


Figure 8. *Learning curve of Test Network 3*

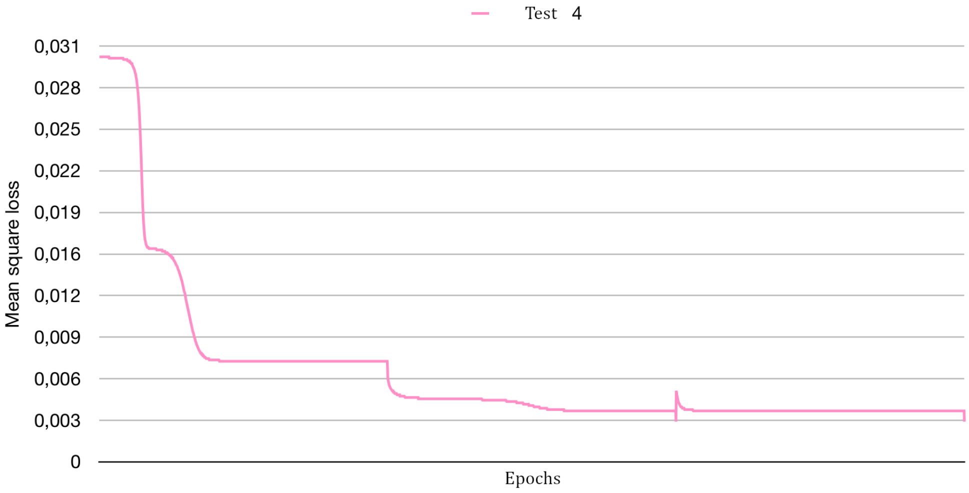


Figure 9. *Learning curve of Test Network 4*

dows 10 version 1903/18362.

The execution of the applicability tests was carried out according to the following procedure. The first step was already done in chapter three. This is the training of the neural network for each test. The five best networks were selected. The detailed description of these nets was also done in chapter three. First, one of the five learned networks is imported into the software. Then a backup copy is created by the test system. This is necessary because the test system is changed or destroyed when the malware is executed. Before each test to be performed, the virtual machine is reset to its original state. This ensures that each test scenario starts from the same initial situation and that the tests are performed independently of each other. After the test system is restored, it is prepared for the execution of the test. The corresponding malware is copied onto the system. The native Windows antivirus system is deactivated during the entire test. This should not influence the course of the test. Afterwards the monitoring of the system by the developed software and the network under test is started. During this process, the current threat value is always logged in the host system. The malware is then executed shortly afterwards. Then the evaluation and graphic representation of the threat values is carried out. Based on these values, it can be seen which of the selected networks best completes the application test.

Scenario 1 – WannaCry

In the first scenario the malware WannaCry is used. This is a ransomware that encrypts a variety of accessible content on all connected data carriers and prompts the user to pay a ransom. If the amount is not paid, it is first increased. If the amount is still not paid, the data can no longer be decrypted and is lost. In addition, WannaCry spreads through a security hole in the SMB protocol in version one and installs the DoubelPulsarBackdoor stolen by the NSA. WannaCry was first discovered in May 2017. Finally, on May 12, 2017, a major cyber attack was launched, infecting 230,000 computers in over 150 countries. WannaCry caused enormous financial and also economic damage due to its high spread [18], [9].

The following five graphs show the course of the threat as determined by the respective neural network. As can be seen, the threat value was relatively constant in the first part. This area rep-

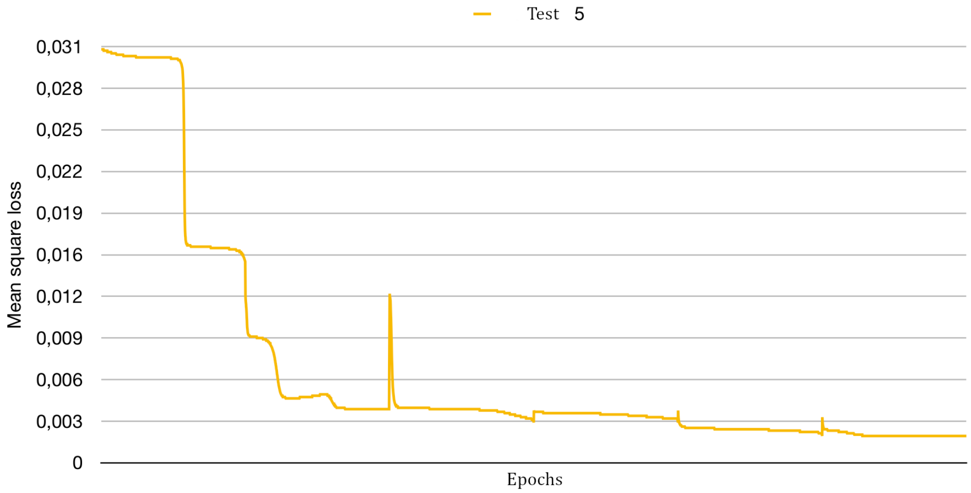


Figure 10. *Learning curve of Test Network 5*

resents the learned normal operation of the system. After about 30 seconds the Malware was started. For all five networks a significant increase in the threat values can be observed. The higher the difference between the normal operation and the threat value after the malware was executed, the more reliable the detection is. The first network achieved a difference of 31.1, the second network achieved a difference of 33.50 and the third network 30.41. Nets four and five achieved a maximum difference of 43.01 and 53.57. The largest difference was thus achieved by nets four and five. A detection of WannaCry is possible with all trained nets.

Scenario 2 – Emotet

In the second scenario the neural networks are tested with the malware Emotet. This is a banking Trojan that specializes in intercepting online banking login information. Additional modules can also be loaded, as is common for other Trojans. This malware has so far mainly affected public authorities and companies. Germany is in first place, followed by North America. The spreading is done by spam mails. These are automatically sent by infected systems to known contacts. Since the spam mails are then received by known and trustworthy persons, they appear particularly authentic. In addition, this software can also exploit gaps in the SMB protocol with the help of subsequently loaded modules. The BSI classifies emotet as a major threat [18], [19].

The following five graphs again show the identified threat level for the five networks. After about 30 seconds, the Emotet malware was started. The first network reached a difference of 9.09. The second network showed a difference of 19.80 and the third network 50.71. Nets four and five achieved a maximum difference of 22.82 and 51.29. The largest difference was thus achieved by nets three and five, with net five again achieving the best results. Network one did not allow the malware to be reliably detected. Networks two and four would allow detection, but the differences between normal operation and the execution of the malware were small.

Scenario 3 – Zeus

In the third scenario the neural networks are tested with the malware Zeus. Zeus is a Trojan that is specially designed to steal money. It is also known as Zbot. It was first discovered in 2007 and has become one of the most successful botnet software. Zeus also served as the basis for other derived malware. Zeus' main goal is to create a botnet as large as possible and thus spread as widely as possible. It nests itself in Windows systems and then receives commands from a command and control server. This server can then be used to retrieve information from the infected systems or perform actions on the systems. In the past, it was mainly used to retrieve login information for online banking websites. It is usually spread by spam messages or downloads from the Internet [18].

In the following, the threat values determined are again presented chronologically. Again, normal operation was first recorded for 30 seconds and then the malware was started. For networks three and five, a significant increase in the detected threat level is also visible for the Zeus malware. Networks two and four also allow the Malware to be detected. With network one, no reliable detection is guaranteed here either.

The first network achieved a difference of 10.98. The second network detected a difference of 18.25 and the third network 34.99. Nets four and five achieved a maximum difference of 24.40 and 50.90. The greatest difference was thus achieved by nets three and five, with net five also achieving the best results here.

Scenario 4 – NanoCore

In this scenario the NanoCore Remote Access Trojan is tested. This software is a malware for remote access to the infected computer system. It was first sold in 2013 in underground forums. The software offers a variety of functions like a keylogger, different functionality to read passwords, remote control of the system, file transfer, a command line and a variety of reloadable modules. Current versions of NanoCore are distributed via e-mail and vulnerable file formats such as pdf and other documents [20].

In the following the threat values for the NanoCore malware are shown. The first part of the graphs shows the normal operation of the system. After 30 seconds the malware was executed. As the graphs show, NanoCore leads to a significant deflection in all networks. The detection of NanoCore is however possible with

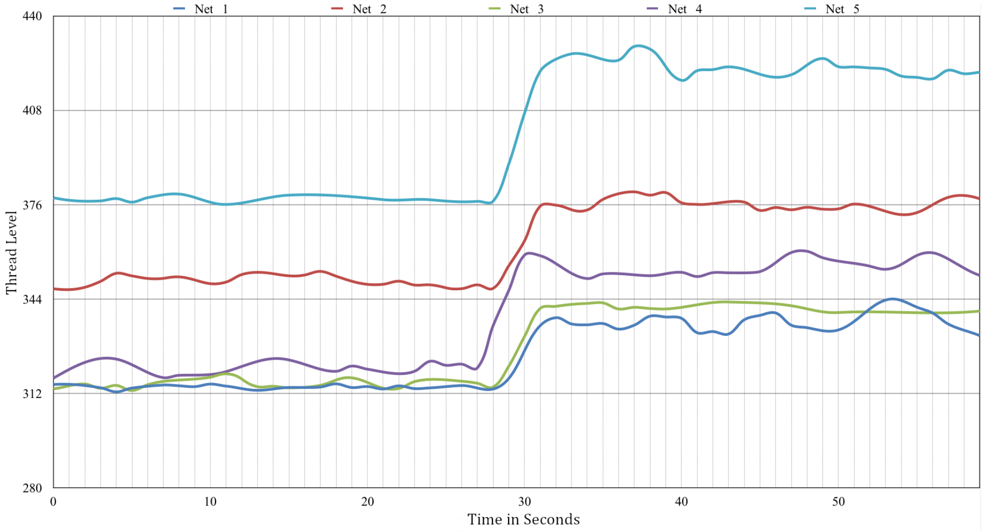


Figure 11. *IHS. "Internet of Things (IoT) Connected Devices Installed Base Worldwide from 2015 to 2025 (in Billions)". (2019): Statista. Web. Aug 20, 2019*

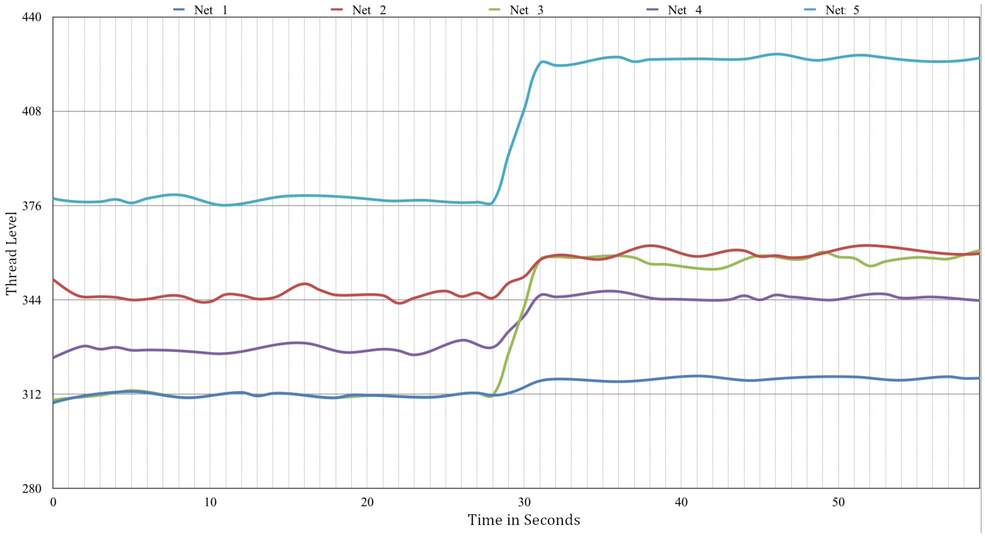


Figure 12. *IHS. "Internet of Things (IoT) Connected Devices Installed Base Worldwide from 2015 to 2025 (in Billions)". (2019): Statista. Web. Aug 20, 2019*

all tested nets. The first net reached a difference of 45.25. The second net showed a difference of 18.52 and the third net 32.74. The nets four and five achieved a maximum difference of 19.19 and 63.25. The largest difference was thus achieved by nets one and five. Again, network five achieved the best results.

Scenario 5 – In-house development

In the last scenario, the developed software is tested against self-developed malware. More precisely, this is a Trojan horse, with the associated control software. The actual malware can be generated from the control software and can then be sent to infect other computer systems. The software automatically hides itself in the Windows system after the first start, is then executed invisibly and will then attempt to establish a connection to the control software. If the connection was successful, the software offers a variety of operations. These include the reading of system information, remote control of the infected computer, a keylogger, an invisible command line, the exchange of files between systems, and various functions to blackmail the victim. Access to webcam and microphone is also implemented in loadable modules. If the software was executed via an administrator account, it is also possible to extend the rights to the system account ”NT authority” via psexecute. The software is not detected by any antivirus software at the time of writing this paper. This makes the test particularly interesting.

The following figure shows the five graphs with the threat values for the self-developed malware. The curve of network two is particularly interesting in this scenario. First of all, a significant increase in the threat value can be seen after the Malware has been started. After the Remote Access Trojan has then started transmitting the data packets, a clear wave pattern can be seen. As it turns out, the network is able to determine the recording and transmission of the screen contents, which takes place at fixed intervals. Furthermore, the threat level did not decrease after the initial con-

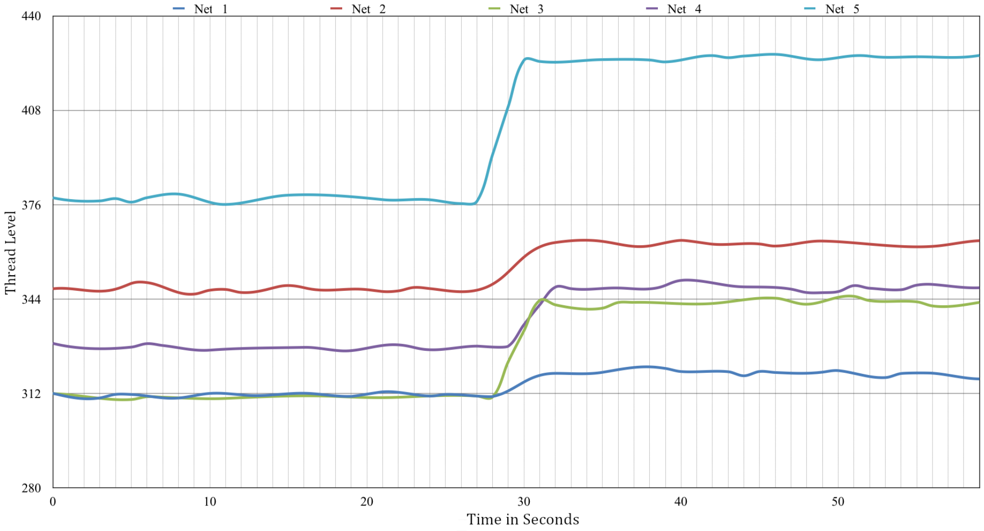


Figure 13. *IHS. "Internet of Things (IoT) Connected Devices Installed Base Worldwide from 2015 to 2025 (in Billions)". (2019): Statista. Web. Aug 20, 2019*

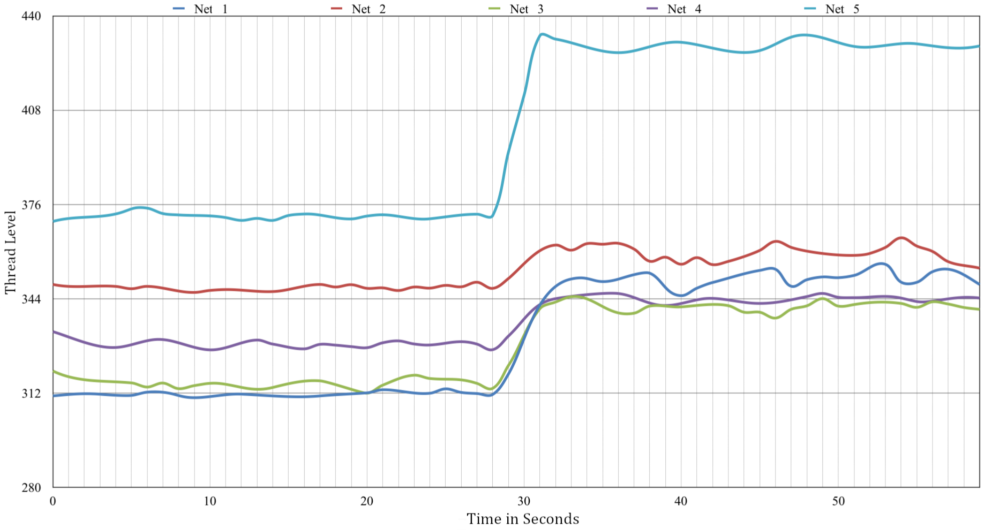


Figure 14. *IHS. "Internet of Things (IoT) Connected Devices Installed Base Worldwide from 2015 to 2025 (in Billions)". (2019): Statista. Web. Aug 20, 2019*

nection to the command and control server was established, as it was the case with the other networks. Network five also allows the malware to be detected again. A clear deflection can be seen after the software was started. Networks one, three and four delivered a threat value with strong fluctuations in this scenario. The reason for this could not be explained. It is suspected that these were triggered by an unnoticed background process that is still unknown to the networks. Nevertheless, it should have been possible to detect the malware in this case, as an increase in the threat values is still apparent.

The first network achieved a difference of 45.02. The second network showed a difference of 47.72 and the third network 24.60. Nets four and five achieved a maximum difference of 47.68 and 32.05. The greatest difference in this test was thus achieved by nets two and three.

Summary and Outlook

It has been shown in this work, that the detection of anomalies and thus also of malware within a computer system is possible by means of artificial neural networks. With the approach developed here, malware was clearly detected in 23 of the 25 tested scenarios. This corresponds to a detection rate of 92% in the tests performed here. False positive detection did not occur during a two-week test period. The two false-negative cases are based on network one and three. The correct detection rate of all other networks was 100% for the malware used here. In total, the network five scored best.

The prototypically implemented application was able to warn the user of potential threats in most cases. This shows that the use of artificial neural networks is very promising in the area of IT security. It is possible to detect threats during an ongoing attack, but also after successful infection or while accessing the computer system. In addition, systems with artificial neural net-

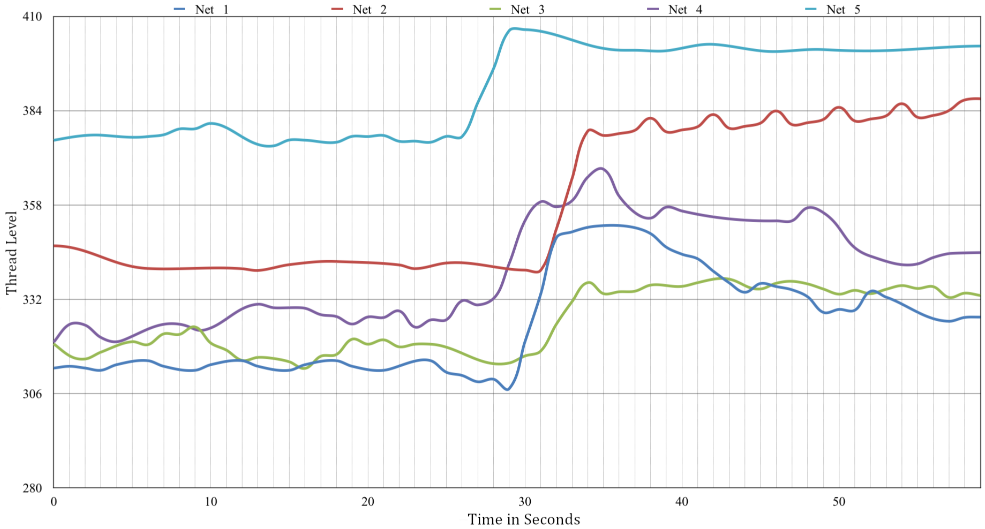


Figure 15. *IHS. "Internet of Things (IoT) Connected Devices Installed Base Worldwide from 2015 to 2025 (in Billions)". (2019): Statista. Web. Aug 20, 2019*

works can detect threats that are not detectable by other systems based on signatures and pattern recognition. As an example, a specially developed malware was tested that was not detected by conventional virus scanners. However, the neural networks developed here were always able to detect this software. Even threats that are not immediately or not at all perceptible to humans can be detected. Since this is only a prototype, this is still a very relevant topic for future research work. It is assumed that with appropriate resources even better results in the detection of threats can be achieved. Due to the constantly increasing computing power of hardware, the use of artificial neural networks is also becoming more and more attractive. The appearance of the first powerful quantum computers may also open up completely new possibilities for detecting anomalies using artificial intelligence. In the future, even more powerful artificial intelligence will be possible, enabling even more reliable and accurate detection of threats.

References

[1] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas: *DDoS in the IoT: Mirai and Other Botnets.* vol. 50, no. 7, pp. 80–84, 2017.

[2] E. Bertino & N. Islam, *Botnets and Internet of Things Security.* 2017.

[3] R. Hallman, J. Bryan, G. Palavicini, J. Divita & J. Romero-Mariona: *IoDDoS The Internet of Distributed Denial of Service Attacks - A Case Study of the Mirai Malware and IoT-Based Botnets*. Proceed. 2nd Internat. Conference on Internet of Things, Big Data and Security - Volume 1: IoTBDS. SciTePress, 92017, pp. 47–58.

[4] A. Tuor, S. Kaplan, B. Hutchinson, N. Nichols & S. Robinson: *Deep learning for unsupervised insider threat detection in structured cybersecurity data streams.* Workshop 31st AAAI Conference on Artificial Intelligence 2017.

[5] Y. Meidan, M. Bohadana, Y. Mathov, Y. Mirsky, A. Shabtai, D. Breitenbacher & Y. Elovici: *N-BaIoT - Network-based detection of IoT botnet attacks using deep autoencoders.* IEEE Pervasive Computing, 17(3), 2018, pp. 12-22.

[6] I. Arnaldo, A. Cuesta-Infante, A. Arun, M. Lam, C. Bassias & K. Veeramachaneni: *Learning representations for log data in cybersecurity.* International Conference on Cyber Security Cryptography and Machine Learning (pp. 250-268). Springer: Cham 2017.

[7] IEEE 1044-1993: *Standard Classification for Software Anomalies.* IEEE Inc.: New York 1994.

[8] *CVE details Website (2019)*, `www.cvedetails.com`, (last access Jan 20, 2020).

[9] *Lagebericht von 2018 des Bundesamt für Sicherheit in der Informationstechnik.* `https://www.bsi.bund.de/DE/Publikationen/Lageberichte/lageberichte_node.html`, (last access Jan 20, 2020).

[10] *Microsoft Windows Development Center Website 2019.* `https://docs.microsoft.com/en-us/windows/desktop/api/`, (last access Jan 20, 2020).

[11] G. Hinton, T. J. Sejnowski: *Unsupervised Learning: Foundations of Neural Computation.* MIT Press 1999.

[12] A. Rostamizadeh, A. Talwalkar: *Foundations of machine learning.* MIT Press, Cambridge, MA 2012.

[13] H. Soma, O. Sinan: *Hands-On Machine Learning for Cybersecurity: Safeguard your system by making your machines intelligent using the Python ecosystem.* Packt, 2018.

[14] *Pytorch Website (2019).* `https://pytorch.org`, (last access Jan 20, 2020).

[15] *InfoWorld Website (2018).* `https://infoworld.com` Serdar Yegulalp: Facebook brings GPU-powered machine learning to Python, (last access Jan 20, 2020).

[16] *Embarcadero Website (2019).* `http://docwiki.embarcadero.com/RADStudio/Tokyo/en/`, COFF2OMF.EXE the Import Library Conversion Tool, (last access Jan 20, 2020).

[17] *Official Python repository on GitHub Website (2019)*, `https://github.com/python/cpython`, (last access Jan 20, 2020).

[18] *Kaspersky Website (2019)*, `https://www.kaspersky.de`, (last access Jan 20, 2020).

[19] *BSI Website (2019)*, `https://www.bsi-fuerbuerger.de`, (last access Jan 20, 2020).

[20] *Malwarebytes Labs Website (2019)*, `https://blog.malwarebytes.com`, (last access Jan 20, 2020).

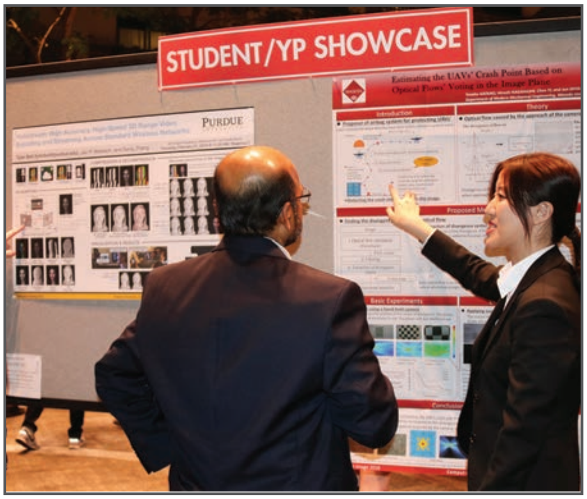
[21] B. Yüksel: *KI-basierte Anomalieerkennung bei Cyberangriffen auf Windows-Systeme - Erstellung eines Prototypen zur automatisierten Überwachung der Prozeßumgebung.* Master Thesis, Technische Hochschule Brandenburg, Department of Informatics and Media, 2019 (last access Jan 20, 2020).

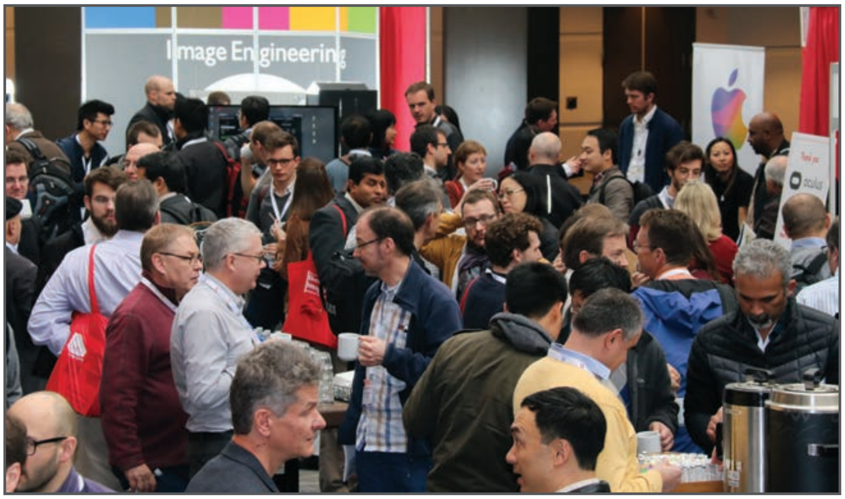
Author Biography

Benjamin Yüksel studied Computer Science at Technische Hochschule Brandenburg and received his M.Sc. in 2019. His research interests include Cybersecurity, Ethical Hacking and Machine Learning.

Klaus Schwarz received his B.Sc. in Computer Science in 2017. He is finishing his Master Thesis in 2020 ....

Reiner Creutzburg is a retired professor for Applied Informatics at the Technische Hochschule Brandenburg in Brandenburg, Germany. He is a member of the IEEE and SPIE and chairman of the Multimedia on Mobile Device (MOBMU) Conference at the Electronic Imaging conferences since 2005. His research interest is focused on Cybersecurity, Digital Forensics, Open Source Intelligence, Multimedia Signal Processing, eLearning, Parallel Memory Architectures, and Modern Digital Media and Imaging Applications.





- AR is a fundamental and transformative interface and platform.
- 20+ years from my initial working demo to today's AR displays
- Optimistic that it won't take another 20 to reach ubiquitous consumer AR!