Artificial Intelligence for detecting IoT data intrusions

Daniel Nilsson
Abstract

Artificial Intelligence for detecting IoT data intrusions

Daniel Nilsson

The importance of the Internet of Things (IoT) continues to grow in modern society, yet it remains relatively vulnerable to data intrusion attacks. These attacks can compromise both the IoT networks themselves and the services that rely on them. This work explores a number of artificial intelligence solutions for their viability in intrusion detection within the IoT. The Bayesian Network, Artificial Neural Network and Support Vector Machine are judged to have the most potential. A traditional three-layer Artificial Neural Network is tested in practice, using roundtrip time and power consumption data to explore different parameters. An overall accuracy of 94% is achieved with only 1% false positive ratio, with a data window of 9 or more readings and a data ratio of a few times more non-attack data than attack data. It is concluded that Neural Networks are viable for use in the field of IoT intrusion detection.
# Contents

1 Introduction ............................................................................. 8  
  1.1 Motivation and problem statement ........................................... 8  
  1.2 Approach alternatives ............................................................ 9  
    1.2.1 Chosen approach ............................................................ 9  
    1.2.2 Other possible approaches ................................................. 9  
  1.3 Results ................................................................................. 9  
  1.4 Outline ................................................................................. 10  

2 Background .............................................................................. 11  
  2.1 The Internet of Things .......................................................... 11  
  2.2 AI methods ........................................................................... 12  
    2.2.1 Overview ..................................................................... 12  
    2.2.2 Suitability .................................................................... 21  
  2.3 Topologies ........................................................................... 21  
    2.3.1 Centralized ................................................................. 21  
    2.3.2 Distributed ................................................................. 22  
    2.3.3 Decentralized (Hybrid) ................................................. 22  
  2.4 Attack Methods ................................................................... 22  
    2.4.1 Denial of Service ......................................................... 22  
    2.4.2 Eavesdropping ............................................................ 23  
    2.4.3 Impersonation .............................................................. 23  
    2.4.4 Software Exploitation .................................................... 23  
    2.4.5 Sinkhole/Wormhole ....................................................... 23  
  2.5 Related work ........................................................................ 23  

3 Design and Implementation .................................................. 25  
  3.1 Design ................................................................................ 25  
    3.1.1 IoT scenario ............................................................... 25  
    3.1.2 Data Models: Roundtrip time and power consumption ........ 27  
    3.1.3 Choice of AI model: Neural Network .......................... 32  
    3.1.4 Choice of topology: Distributed ................................. 32  
  3.2 Implementation ................................................................... 32
### 4 Evaluation

4.1 Experiments ................................. 34
   4.1.1 Experiments on the roundtrip time data .......... 34
   4.1.2 Experiments on the power consumption data with preprocessing 39
4.2 Summatory experiment overview ............................... 45

### 5 Conclusions and Future work

5.1 Conclusions .................................. 49
5.2 Future work .................................. 50
# List of Figures

2.1 An illustration of the K-Nearest Neighbors algorithm .......................... 13
2.2 A graphical illustration of linear regression ................................. 14
2.3 An illustration of the K-Means algorithm ................................. 14
2.4 An illustration of the DBSCAN algorithm ................................. 15
2.5 An illustrative example of principal component analysis ................. 16
2.6 A simple illustration of Classification/Regression Trees ................. 17
2.7 An illustration of a neural network ................................. 18
2.8 A simple illustration of a Bayesian Network ................................. 19
2.9 An illustration of a support vector machine ................................. 20

3.1 An illustration of the wireless testbed ................................. 26
3.2 An illustration of the learning data ................................. 28
3.3 An illustration of the new power consumption data ......................... 29
3.4 An illustration of the power consumption data difference ................. 30
3.5 An illustration of the used preprocessing method ................................. 31

4.1 Experiment: Data Ratios ................................. 35
4.2 Experiment: Iterations ................................. 36
4.3 Experiment: Segment Size ................................. 37
4.4 Experiment: Training Data Amount ................................. 38
4.5 A naive comparison ................................. 40
4.6 An illustration of the model’s accuracy when trained on different singular nodes’ RTT data ................................. 40
4.7 An illustration of the model’s accuracy when learning solely from the power consumption of different singular nodes ................................. 41
4.8 Illustrations of how accuracy changes when the preprocessing batch margin is altered ................................. 43
4.9 Illustrations of the accuracy of the three cases in 4.8 when not preprocessed ................................. 44
4.10 An illustration of the model’s accuracy when predicting preprocessed data from a single node ................................. 45
4.11 An illustration of how the results change in the preprocessed data when more events are included in an examined point ................................. 46
4.12 An illustration showcasing the effect of adding extra no-attack data to even out the otherwise-imbalanced preprocessed data ................................. 47
Chapter 1

Introduction

1.1 Motivation and problem statement

In this modern age, the Internet of Things (IoT) is rapidly becoming a critical part of our society[10]. Miniaturized sensors and computers are everywhere, working with improved artificial intelligence algorithms and cloud computing to create smart devices of all sorts. Digitalization and connectivity helps the IoT become an increasingly critical part and valued optimizer in everything from transportation and resource management to health care.

However, it does not come without its risks. The billions of connected devices are seldom designed to work securely together (a survey[35] in an enterprise environment found that 83% of inter-IoT transactions happen over plaintext, with 41% not even using SSL or other forms of encrypted communication at all), struggle with limited computation resources and yet, being networked devices, they are exposed and vulnerable to attack - especially wirelessly. Both communication channels and component interfaces are exposed, and failures in one place often propagate elsewhere, especially in large systems. Access control and encryption can ultimately only provide so much protection - attacks are usually directed at availability, control system latency or inter-component exchanges. Although the attacks are typically quite simple, exploiting well-known flaws often borne from rushed development, the high reliance on and quantity of IoT applications means that even simpler attacks can significantly impact the reliability and availability of IoT services and applications.

Attack detection and mitigation are basic resilience building blocks, but their implementation and effectiveness both highly depend on the assumed context and attack models. This work aims to explore the field of attack detection using AI for a subset of the possible attack models, within an assumed context similar to that of an IoT network between embedded systems. The intention of this is to discover what AI methods are and are not viable for this role, and to discover how well different approaches work in practice.
The goals of this thesis are twofold:
(I) Identify AI-based attack detection models applicable in embedded systems.
(II) Implement a testable attack detection model and test its ability to detect attacks under different conditions.

1.2 Approach alternatives

1.2.1 Chosen approach

To identify suitable AI models, I chose a literature study. A total of nine different approaches were examined this way.

As for the choice of model, I ultimately elected to implement a neural network, a machine learning-based AI model. This was done partly due to ease of implementation (owing to time constraints) and partly due to seeming viability as determined in the aforementioned study.

The testing of the model was carried out using two different datasets, one containing roundtrip times and the other containing raw power consumption data. Attacks were emulated by using a jammer to disturb signals. Numerous experiments were performed using this data, testing (amongst other things) changes caused by a differing amount of learning iterations, the effects of preprocessing and the consequences of an unbalanced learning set.

1.2.2 Other possible approaches

Had I possessed more time, a more test-driven examination of the more promising-looking AI options may have been a viable approach. Whilst this may have allowed for a more certain foundation than relying on others’ research, it was not an option due to time constraints.

Another way to approach things is naturally to use another AI model. For example, a Bayesian network could have been a viable alternative as it too is quite suitable for our problem (as mentioned in Section 2.2.1). The change to another model could have altered the end result of this work a significant amount due to testing viability, ease of implementation, the model’s detection capabilities and more.

1.3 Results

Research suggested that of the explored AI models, Bayesian Networks, Principal Component Analysis, Support Vector Machines and Neural Networks were the more viable. An implementation using the latter method, with compiled roundtrip time data, showed not only that Neural Networks are indeed a viable choice (with a false positive ratio of 1% and a true positive ratio of 60%), but also that a Neural Network does not
have to be implemented as a Deep Neural Network in order to work fairly well - shallow approaches are viable and less resource-demanding. Using power readings gave somewhat worse results, with the lowest false positive rating being at 8%, but it does have a lot of room for improvement. Table 4.1 just above the conclusion contains a more detailed summary.

Preprocessing was also explored, and was found to have mixed results. It allowed the model to perform somewhat better against many of the worse cases but also worsened the accuracy on data sets the model performed better on.

1.4 Outline

The work first briefly explains the Internet of Things before examining a number of different AI models. This thesis then examines several topology options and attack methods before looking at the data that is used and how to use it. After that, it goes over which attacks to model, which AI model to use and which topology to employ, as well as the general points on the model implementation.

Then, the experiments used to evaluate the implemented model are presented. Finally, it concludes with a discussion of what conclusions can be drawn and suggestions for potential future work.
Chapter 2

Background

This chapter contains information on background information necessary to understand the rest of the work, as well as information on numerous AI models explored as part of this work’s objectives.

The Internet of Things is explained, following which the various AI methods explored in this work are brought up and examined. Following this, different topologies and attack methods are explored.

2.1 The Internet of Things

The Internet of Things (IoT) is a network of numerous (typically embedded) sensors and actuators collectively responsible for the control and/or monitoring of some process or environment[31].

The Internet of Things as a whole is formed by the so-called “perception layer” (being the aforementioned sensors and other data-creating hardware connected in an IoT network) connected in the network layer (the internet connection) which communicates to the middle-ware layer (cloud services etc) in order to get the final data to the application layer (the user)[8]. Most embedded systems that make up the Internet of Things on the perception layer are limited in their capabilities, having limited power supplies, memory size and processing speed[8].

There is no upper limit on the size of an IoT system, so a solution involving such a system must be scalable. Due to the limitations of the embedded systems that the IoT connects, its software must be efficient in terms of the perception layer’s limitations. The amount of network traffic generated must also be limited as the connections between the IoT devices have limited throughput and range[8].

Software operating in the IoT must fulfill these criteria by using limited memory, processing power and energy.
2.2 AI methods

AI methods are used for various purposes within IoT, allowing these systems to act with seeming intelligence. One of these purposes is attack detection.

A survey[20] presented information regarding numerous different AI techniques, most of which typically use machine learning in the process of their creation. I examined many of these in an attempt to determine which were sensible for usage in resource-constrained systems.

The most important criteria here is that the attack detection software must have relatively small computational cost, that it must not use a lot of memory, and that it must be suited for the actual task of detecting attacks.

2.2.1 Overview

A number of AI methods are examined below. Advantages and disadvantages, as well as how the methods work and how they may in theory be used for attack detection is presented. The suitability is discussed in the final paragraph of each examined approach.

K-Nearest Neighbors

The K-nearest neighbors algorithm is an algorithm that relies on classifying new data points based on the k "closest" data points in the training set. For example, if k equals four, the four closest data points from the training set will determine how to classify the input data point[27]. An attack detection implementation would simply have given attack- and non-attack data points to use for classification.

Machine learning algorithms may themselves classify the training data based on some data (or perhaps learn the optimal value for k), although typical AI approaches have labelled data with set k-values allowing them to be immediately deployed - no learning time is necessary in those cases.

This is showcased in Figure 2.1, where the red- and blue points are learning data, and the black point is the data to be classified. In this case, k=4 (though in practice you may want to choose something else as k=4 is prone to ties) - a red circle showcases the four closest points. As three of them are blue, the black point would then be classified as a blue point.

Whilst it has good accuracy and computation speed, it was quickly deemed inappropriate as a candidate for this work’s implementation due to requiring that the entire training set be stored during active operation[20], making it unfit for memory-constrained environments such as embedded systems.
Figure 2.1: An illustration of the K-Nearest Neighbors algorithm in a case with K=4, red and blue classes in the training data and a black input data point. The input data point would be classified as a blue point here.

**Linear Regression**

Linear regression is a method relying around the learning of a function $f(x,w)$ that allows one variable to describe another. More specifically, the function $f$ maps $φ(x) → y$ with a fixed, linearly combined set of functions of the input variable $x$ that are either linear or nonlinear[20]. In a manner of speaking, the function $f$ is equal to $g(x)w_1 + h(x)w_2 + ... + i(x)w_n$ for some set of $n$ linear functions $g...i$ and a set of weights $w$, where $n$ is any non-negative number.

An intrusion detection algorithm could, for example, be done by having $f(x)$ be the odds of an ongoing attack, and having it consist of other functions $g(x)$, $h(x)$,... each of which attempt to assess the risk of there being an attack from a particular direction or of a certain sort. How effective this would be is another question entirely.

Machine learning approaches learn the desired function, whereas conventional AI approaches can be given them directly - sometimes, the relationship is known and can thus be directly applied. Figure 2.2 is a simple 2d illustration of how a linear regression model may emulate data.

I discarded this approach as unviable for this work due to the fact that the use of this algorithm requires an output that can be modelled linearly. One can note that linear regression is not unseen within AI[22], but we cannot ensure that the patterns we are trying to model can be easily modelled linearly - if not, linear regression is cumbersome at best. There are better options.
Figure 2.2: A graphical illustration of linear regression. The red line $f(x,w)$ is created based on the black data points and so would then likely describe the underlying behavior of the data being analysed. Note that this is a very simple example.

Figure 2.3: An illustration of the K-Means algorithm in a case where $K=3$. The black points are training data, the colored stars are the placed centerpoints and the colored points are examples of how new data would be categorised.

**K-Means**

The K-Means algorithm is a clustering algorithm which works by categorizing data points into one of a number of clusters. The K-Means algorithm learns by optimizing the position of $K$ (initially randomly placed) centerpoints within the data space such that the sum of the squared distance between the training data piece and their closest centerpoints is as small as possible. During runtime, data is then categorized based on which of these $K$ points it is closest to[21]. You can see an illustration of this in Figure 2.3.

An attack detection algorithm could do this by getting some $K$ points to place for both attack- and no-attack scenarios, then at runtime it would categorize incoming data as attack and no-attack events based on which of these two sets had a point in the closest proximity.

However, as this algorithm typically has a fairly high computational cost (both to train
Density-Based Spatial Clustering of Applications with Noise (DBSCAN)

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is, much like K-Means, a clustering algorithm. The algorithm requires a set of training points, a value $\epsilon$ and an integer $N \geq 3$.

DBSCAN, conceptually, works as follows: For each data point, if the data point has at least N-1 other data points within the distance $\epsilon$ or less of itself it is classified as a 'core' point. Core points within a distance of $\epsilon$ or less to each other are then grouped together, with the grouping being transitive (eg. if A groups with B and B groups with C then A groups with C). All non-core data points are then assigned to any particular cluster if they are at a distance of $\epsilon$ or less to one or more of the cluster’s member data points (though this cannot connect clusters), and are classified as noise if no such close-by cluster exists[25]. Figure 2.4 provides an example of the algorithm.

An attack detection algorithm could use this method to cluster its learning data. These clusters could then be marked as attack- or no-attack clusters based on the data that makes them up, with outliers being discarded as anomalies. This would produce labelled clusters which would cover certain areas of the variable space. Readings that then fall within these areas could be classified based on the cluster they fall within, allowing for attack detection.

Since DBSCAN is an approach more efficient on large datasets[20] than embedded systems, used more for analysing datasets than processing real-time data, it was deemed inappropriate for use in this work.
Principal Component Analysis

Principal component analysis, which is a method revolving around feature extraction[20] and data compression[5], was deemed unviable for attack detection due to its nature. Its key function is to distill a large dataset of possibly correlated values to a smaller set of uncorrelated variables known as principal components[5].

Each principal component the algorithm creates during runtime accounts for as much of the data variance not yet accounted for by prior principal components as possible[5]. The first component thus attempts to have it so that the values being dealt with diverge as little as possible from it. Subsequent components try to do the same to reduce the remaining variance further given the presence of earlier components.

In a way, the principal components form a new coordinate system which best aligns with the direction the data varies[7], as seen in Figure 2.5. This can then allow for the extraction of the data points which best correlate with the principal component set, or to represent the dataset more compactly.

If someone wanted to implement an attack detection algorithm using this method, one approach could be to produce two principal component sets -one for attacks and one
for non-attacks- and detect attacks based on which aligned more closely with readings.

**Classification/Regression Trees**

Classification/Regression Trees is a method where the algorithm builds up a tree structure during learning. The resulting tree has a number of concluding ends and a number of criteria. When new data is inserted into the system, it then interacts with these criteria until it reaches a leaf node in the tree that draws some conclusion about the data. An example of such a tree is illustrated in Figure 2.6.

Implementing an attack detection algorithm using this tree method would mean finding a number of true/false values (for example whether a reading is much larger than the one that came before) and then building up a tree where navigation is determined by these values in a way that accurately leads to a leaf (tree endpoint) representing the correct conclusion.

It quickly became apparent upon further study that the tree-based approach was not one commonly seen for attack detection within resource-starved environments such as embedded systems, as sources on the matter weren’t nearly as prevalent as those for the other more closely examined options. This, I believe, was sufficient reason to turn focus towards the other options under examination to the exclusion of a tree approach, as the lack of sources could be indicative of the inappropriate nature of the alternative.
Figure 2.7: An illustration of a neural network. The dark-yellow input layer feeds its output into the first of the N black hidden layers (between which input/output arrows are omitted), the last of which feeds its output into the purple output layer. Input/output arrows are colored for ease of separation only.

As such, this approach was considered unfit for this work.

**Artificial Neural Networks**

Artificial neural networks (ANNs) are a popular form of AI which rely on machine learning in order to be created. It consists of numerous artificial neurons which each take some weighted set of inputs and then emits a particular output. A typical ANN has a set of neurons called the input layer, a set of neurons making up the output layer, and then one or more hidden layers connecting the two[23]. Each layer feeds its output into the next layer as input. You can see an illustration of this structure in Figure 2.7.

The learning process of an ANN consists of determining the optimal set of weights for the network of neurons such that a cost function (for example the mean square distance between the network’s guesses and the training data) is minimized[23]. These weights are placed on the edges that form connections between the neurons, increasing or reducing the effect of the value sent through said edge.

Neural Networks cannot be effectively created manually. A way to use it for attack detection would thus be to employ machine learning on a training data set of attack- and non-attack data, fine-tuning the model as necessary until its predictions become sufficiently accurate. Care would have to be taken to not make the network too large or complex during this process in order to respect IoT resource constraints.

Whilst methods exist to improve the efficiency of neural networks[16][12], the fact remains that a neural network is slow to train and, unless created carefully, often rather resource-demanding[12]. At the very least its use demands that numerical approximations be used[16]. Nevertheless, neural networks have been successfully employed in
Figure 2.8: A simple illustration of a small Bayesian Network. Each node is marked by a letter, and each edge is signified with an arrow. For example, node D depends on node A, with nodes C and E depending on it.

Bayesian Networks

A Bayesian Network is best described as a probability machine. In a Bayesian model, the program seeks to discover the value of some hidden variable based on certain known variables. A Bayesian Network is a Directed Acyclic Graph, a 'DAG', which consists of a set of nodes, representing values, and a set of directed edges. An edge from node A to node B indicates that the value of node B is statistically dependent on that of node A[13].

In the case of a machine learning implementation the model will, during the learning process, learn how this dependence is arranged. If A and B are truth values, it will learn the probability of B being true if A is true, B being true if A is false, et cetera based on the statistics of the learning data, though A and B may very well be continuous variables. Depending on the implementation the learning process may also involve (partially) determining dependences, or that task may be handled by the programmer[13]. Bayesian Network AI implementations that do not rely on machine learning may instead have these probabilities (or estimates of them) given to it directly.

When later deployed, the Bayesian model can then reason about the probability of an unknown value having a particular value based on the values of what values it depends on and/or depends on it. A simple Bayesian Network is illustrated in Figure 2.8.

One way to employ a Bayesian network for attack detection is to first construct a set of nodes with relations, each point in the Bayesian network corresponding to some at-
Naïve Bayes, a less costly but limited variant, has been used in embedded systems in the past[33] and Bayesian systems have been used in embedded intrusion detection[14], so it is a viable option.

**Support Vector Machines**

A support vector machine is a partitioning algorithm. It uses a (set of) multidimensional hyperplane(s) to separate the areas containing data of different categories. Given an N-dimensional space of labeled training data (two-dimensional in case of data (x,y), for example), the support vector machine attempts to find a plane of N-1 dimensions such that it separates the data of a particular label from any data labeled differently in a way that leaves the most space between the plane and any differently-labeled data[20] (though some practical implementations tolerate some misclassified points[26]). During deployed runtime, the algorithm then determines how a new data point should be labeled based on which side of the hyperplane(s) the data is placed[20].

An illustration of the algorithm can be seen in Figure 2.9.

Using this model for attack detection would be a matter of generating one or more of the lines used in the model based on known attack- and non-attack data (which would
then allow for the necessary labelled points). In deployment, the model would then label new data based on which side of these lines it is.

Support vector machines have seen use within embedded systems[4][24] and can utilize the so-called kernel trick (a mathematical trick to avoid the need for a nonlinear hyperplane by effectively mapping the data into a higher dimension) for performing nonlinear classification[26].

### 2.2.2 Suitability

Most approaches were, for one reason or another, excluded relatively quickly from my considerations. The reason why they were deemed unsuitable is found as the final paragraph of them.

Feed-forward neural networks, classification/regression trees, support vector machines and a naive Bayesian approach required closer examination. Neural networks are a popular method in the field of artificial intelligence. Though deep neural networks can be quite resource-intensive, methods such as pruning exist to mitigate these problems[12] and more importantly, a neural network does not necessarily have to be deep, so it deserved a closer look. As for the tree- and Bayesian approaches, a study[3] showed them to have good and quite similar performance, and support vector machines have attracted some interest within the field of embedded solutions[4] as well as being memory-efficient[20], so I found closer examination to be prudent.

In the end, Support Vector Machines, Bayesian Networks and Artificial Neural Networks were deemed the most suitable options. Principal Component Analysis could possibly also warrant some further exploration.

### 2.3 Topologies

Every attack detection solution (as well as most other systems) meant to interact with an Internet of Things network at large requires the implementation to be done in the way of a network topology. A topology determines how the detection system is laid out across the IoT network. For our purposes, this means choosing whether to implement the software as centralized, decentralized (sometimes called hybrid) or distributed - three popular options that cover the IoT system in different ways.

#### 2.3.1 Centralized

A centralized implementation would mean having a singular node responsible for monitoring the network at large[20]. Whilst this allows for a dedicated node with more resources, allowing it to achieve higher accuracy through performing more advanced analysis, a singular node is a single point of failure[28]. Furthermore, it is inherently unscalable[28] as a larger IoT network will lead to a matching increase in network traf-
fic (unless data is only passively read, which limits the options in terms of what to use for analysis) as well as computation load.

2.3.2 Distributed

The opposite approach is a distributed implementation. In such an approach, each node in the IoT network has its own local AI process responsible for that node alone[20]. This approach is scalable as the number of processes scale linearly with the number of nodes being inspected. There are no problems in finding a sufficiently strong node to hold a central process, nor with network congestion caused by reports sent to said central node, as there is no central node at all. On the other hand, this approach lacks a global overview of the network, and so can miss things that the centralized and decentralized versions may catch. Furthermore, as stated in Section 2.1, these smaller nodes are usually very limited in capability. Collaborative intrusion detection commonly sees the nodes networking in a peer-to-peer model for best effect[28].

2.3.3 Decentralized (Hybrid)

Decentralized implementation is a hybrid of the aforementioned two options, attempting to get the best of both worlds. In this approach, there is a central node, but there are also a number of child-nodes of this node acting as intermediaries between the top node and the examined nodes. These intermediaries may preprocess, filter and otherwise handle some data, lessening the amount of work the core node needs to perform without sacrificing the ability to see the entire system[20]. This tree structure allows it to overcome the performance bottleneck of centralized implementation, though the single-point-of-failure problem remains[28].

2.4 Attack Methods

There are many ways to attack an IoT system. Attacks may come from outside the system, or they may emerge internally from corrupted nodes. Some attack types will now be described.

2.4.1 Denial of Service

A Denial of Service (DoS) attack is an attack which aims to compromise the system by interfering with accessibility, thus preventing legitimate users from contacting or interacting with the hardware[18]. It does this by flooding the network or target with messages (a jamming attack is one example) or by transmitting specific information that triggers a crash in the appropriate location, and is one of the most popular network security attacks[18]. IoT networks are especially vulnerable as even low-rate DoS can overwhelm the weak processing power of typical IoT nodes[9]. DDoS, Distributed Denial of Service, is a variant with multiple attack sources[9].
2.4.2 Eavesdropping

Eavesdropping is exactly what it says on the tin. An adversary, in this type of attack, will try to listen in on traffic coursing through the IoT network which they are not meant to overhear, allowing them to learn what they otherwise should not know[9]. This can pose a threat to both privacy and confidentiality.

2.4.3 Impersonation

An impersonation attack is an attack in which an attacker pretends to be a valid user. Approaches include utilizing known user credentials or access information, or by replaying a message intercepted or eavesdropped on earlier (known as a replay attack)[9]. Attackers can also (in what is called a fabrication attack) attempt to generate their own malicious messages.

2.4.4 Software Exploitation

IoT vulnerabilities do not lie solely in the networking. The devices themselves are likewise vulnerable and can be infected by malicious software (malware) which may cause them to behave erroneously or maliciously[9]. Malware may be introduced by the introduction of an infected device into the network, through an introduced firmware upgrade or from whichever app or service the user(s) use to manage the network[9], to name some examples. Some approaches manage this through brute force using known weak passwords, whereas others exploit known vulnerabilities in the devices or software[29]. Impersonation attacks are also a viable method to gain access.

2.4.5 Sinkhole/Wormhole

A sinkhole attack is the disruption of the communications network such that some messages disappear instead of arriving at their intended destination[11]. A wormhole attack involves making some messages arrive at the wrong destination instead of the intended one[11]. Both are a form of routing attack.

2.5 Related work

Various machine learning algorithms were brought up in [20] and the work discussed their use within IoT. It also brought up the importance of choosing appropriate data for a model. Various intrusion detection approaches within the IoT were surveyed in [34]. The implementation of an anomaly-based intrusion detection system for use with mobile ad-hoc networks, which is a form of dynamic IoT network, was handled in [17] to generally good success. A Naïve Bayesian Network intrusion detection algorithm is also discussed in [14]. It most importantly explored the importance of preprocessing IoT system events to improve classification efforts and explored the effect of using a so-called hidden naïve Bayes multiclassifier, finding it to produce better results than the typical model.
Crypto-ransomware (a form of software exploitation attack) was detected within the Internet of Things by using the power consumption levels of the networked devices in [6]. Contrasting the anomaly-based detection methods used in this work (which aims to detect attacks more in general), they utilized a signature-based model to detect the specific signatures of crypto-ransomware power consumption and detect attacks that way.

Problems in security and privacy within a specific IoT environment, the smart home, were discussed in [9]. The main point of interest is that it serves as a very concrete example of an Internet of Things situation and the vulnerabilities therein. A Denial of Service attack method specifically targeted at IoT environments was brought up in [18]. It found, amongst other things, that systems which transmit larger packets are more vulnerable to such an attack, but more importantly provides a perspective from the attack side of IoT security.

Various security risks for IoT networks were classified in [8]. The work explored several layers of the IoT network structure, ranging from individual sensor networks to the cloud systems and applications used by the owner. Some attack models and security measures within the IoT were both identified in [30]. Problems with trying to use unsupervised learning within IoT systems, such as the issue of early learning processes exploring approaches that could cause network disaster, were also outlined.
Chapter 3

Design and Implementation

3.1 Design

3.1.1 IoT scenario

The scenario used for model testing in this work is that of a set of sensor nodes wirelessly connected to form an IoT network. The majority of the nodes regularly send wireless packets with their collected information to a node responsible for data collection. An attack occurs to disturb this network, interfering with communication. We do not know whether this attacker is a foreign agent or a compromised network node, only that we acquire no usable data directly from the attacker.

IoT testbed

The testbed employed for data collection consists of twenty nodes, one of which serves as the collector node. These nodes are on the same floor of the same building, but are placed in different rooms and locations. Test readings are gathered via ethernet cable in order to avoid interfering with the wireless traffic.

A map of the testbed layout is illustrated in Figure 3.1. The nodes are numbered 1-20.

Application

Wenqing Yan, Christofer Flinta and Andreas Johnson gathered data from this testbed for the paper Machine-Learning Based Active Measurement Proxy for IoT Systems[31]. I was given access to this data, which contains data such as timestamps, node IDs, sensor data and most importantly roundtrip time data.

I also used this testbed to generate my own data, gathering power consumption readings using the same testbed.
Figure 3.1: An illustration of the wireless testbed. Each dot on the map corresponds to a device, with the nearby boxes containing some information about the associated node. Orange boxes signify that the node has a GPS connection, as opposed to the normal grey nodes which lack them. Credit to Weng Qing Yan, Christofer Flinta and Andreas Johnson[31] for the map, though I have modified it slightly for the sake of clarity.
Attack(s) to model: Denial of Service, Sinkhole, Software Exploitation

Some attacks that threaten IoT systems are outlined in Section 2.4. Given the data available to me, the attack types seemingly fit for modelling to some extent are Denial of Service and Sinkhole/Wormhole attacks. The reason for DoS is obvious since jamming is in fact a form of DoS attack. A sinkhole or wormhole attack can often be detected based on similar principles - sinkholes in particular will cause message time-outs much like a very delayed message. Software Exploitation may also be suitable, notably in cases where infected nodes try to interfere with others or, in the case of power consumption data, where power consumption is noticeably escalated.

Jamming attack simulation

To simulate attacks, a signal jammer was used, disturbing the network. This jammer disturbs the communication channel used by the nodes to communicate wirelessly, causing interference which requires extra work to overcome.

The effect of a jammer node on the system has, for our purposes, two important effects. The jammer occupies the transmission channel, forcing those nodes that detect the disturbance to wait for a better opportunity as the channel appears occupied. This increases both message delays and node power consumption as the nodes spend active time waiting.

Aside from this, when other nodes do transmit, the jammer may interfere with the transmitted data packages. Packages may be damaged or lost entirely. This creates a necessity to retransmit these packages, delaying the complete messages and increasing node power consumption.

In the case of power consumption data, this is very flexible, as abnormally increased power consumption (in this case caused by having to resend messages or wait for a better time) could be a sign of any number of unusual behaviors. Roundtrip time alterations can likewise show signs, but may be less broad in detection scope.

The jammer used for the power consumption experiments was placed in the same room as node 16.

3.1.2 Data Models: Roundtrip time and power consumption

The work was performed using two distinct datasets: roundtrip time data and power consumption data. In both cases, viable nodes were selected from the full set of data in order to carry out the learning- and testing processes for the system.

Given roundtrip time data

This dataset was created by Wenqing Yan, Christofer Flinta and Andreas Johnson in the paper Machine-Learning Based Active Measurement Proxy for IoT Systems[31]. The work in question revolved around examining the effect of jamming which I consider
Figure 3.2: An illustration of the learning data, being the roundtrip time (RTT) of messages sent by numerous nodes. Each RTT is the average of five messages, thus limiting severe outliers. On the left is the control data. On the right is the same nodes but with a jammer disrupting signals (the “attack”). Note that the non-attack data has a lot more measurements overall, at five measurements per every one attack data point. Nodes 126, 127 and 133 were excluded as they had no readings.
The orange lines are medians; the box-shapes are the width of 50% of the data and the lines extend to everything else other than outlier data (being the small percentage of data sometimes present which diverged from the general behavior of the node, unrepresentative of typical behavior), which was excluded for the sake of readability.

to be a good enough proxy to an actual attack to be usable for my first experiments. The data overall is a set of files, one for each of their experiments, where each line corresponds to a roundtrip time value and the other data at that point. As previously mentioned, each line includes many fields, but the ones of primary interest in this work are node number and message roundtrip time. In this particular dataset, node 6 in the testbed served as a receiver.

As can be seen in Figure 3.2, most nodes were (mostly) unaffected by the supposed attack, but a few nodes suffered noticeably from it.

**Gathered power consumption data**

Power consumption data was collected from the testbed by measuring voltage and current consumption. The data takes the form of a set of files for power consumption, where each file contains the data for that particular variable (voltage, miliAmpere and overall/current joule) where miliAmpere was the one used in this work. Each data line consists of a timestamp and a set of readings consisting of the values from all twenty
Figure 3.3: An illustration of the new power consumption data. The illustrations show the distributions of the readings within the data in both attack- and no-attack scenarios. Some nodes (particularly node 5 which would surely make for good training data) have a visibly strong reaction to the data, whereas others are barely affected at all. Nodes 6 and 12 were omitted as they were consistently busy nodes with regular power consumptions as high as 20 mA (0.02 Ampere), making the rest of the data hard to read. Notably, a higher ratio of power spikes creates a taller column in this illustration.

Two testbed experiments were performed, one with a jammer 'attack' and one with normal operation only, both of which had a duration of twelve hours. This resulted in raw power consumption timeline data both of normal operation data and ongoing attack data with a jammer, each totalling nearly 70 000 readings with measurements taken at even intervals roughly every 600 milliseconds. One major advantage of this data is that it is something that could easily be gathered in deployment (for example by having a detection device reading the power usage to determine whether there are attacks), making it more usable for actual systems. An illustration of the various readouts can be seen in Figure 3.3.

Only a small fraction of the data is related to activity when messages were received or sent, with the rest being idle operation data with some smaller amount of noise in the measurements.

**Preprocessing**

For typical AI methods, the idle periods are not particularly relevant as they do not truly differ between attacks and non-attacks. As the idle operation data forms the majority
of the power consumption readings, I chose to apply preprocessing on that data in an attempt to improve the model’s accuracy.

Some ways to preprocess timeline data was brought up by [2]. The time series differencing approach, found by taking the backwards difference (e.g., current minus previous) of each value, may have some good merit especially if the algorithm lacks greater overview as it is both fairly simple and would easily adapt to differing base power consumptions. Their work focused on finding good data for predictive purposes, not classification purposes, however, so their relevance here is somewhat lessened.

Seemingly more relevant are methods such as outlier detection as mentioned in [15]. In this specific instance the outliers are of interest - the points that stand out from the normal power consumption. These typically occur when a message is sent or received. The difference between an undisturbed message’s power consumption and one which was disturbed by an attack can be seen in Figure 3.4.

More concretely, one notable thing to do is to filter out idle operation data from the

![Figure 3.4: An illustration of the power consumption data difference between where a message is sent in an attack situation and where there is no attack ongoing, showcased using a small piece of each experiment’s mA power readings. Note that the axes are subtly different.](image)
ongoing attack set, as it is undesirable for the model to treat 'nothing is happening' as a potential attack. It is, however, important to still give the model a sense of distance between the remaining events as what seems to make the data distinct is how close together the spikes are.

Feature construction and feature transformation is another method described in [15], where more useful data is fabricated from what is available. The main challenge in such an approach would be to discover what features are both useful and viable for construction - a problem in its own merit. One example of something that could be done is to convert each value into a "time since last spike value" number, which would solve the aforementioned distance problem.

Ultimately, I ended up using the following preprocessing solution: Detect when a sudden power spike occurs. At that point, count how many measurements occur before some number n of "normal" values have been seen in a row. Save the length of this sequence and the time duration since the start of the last sequence. These values become the new dual-value data points.

The preprocessing method in question is illustrated in Figure 3.5. It showcases the process described above.
3.1.3 Choice of AI model: Neural Network

The AI methods most viable for our purposes are described in Section 2.2.2. For my initial implementation, I chose the neural network solution. Given the presence of several viable models, this choice was primarily due to the ease of implementation. Tensorflow is a neural network library for Python. It allows for rapid development and prototyping whilst being viable for deployment[1], and can even be deployed on embedded systems with Tensorflow Lite. This makes it viable for testing without potentially spoiling the validity of the tests by using a model unfit for the Internet of Things.

3.1.4 Choice of topology: Distributed

Different options for topology choice are described in Section 2.3. In the implementation, the choice was made to train the model on data from several nodes (though data from different nodes does not emerge at once) and then test it on a similar test set. In a way, this means the testing implementation is a centralized one. However, it makes no assumptions as to where each reading originates from, meaning that it could just as well be implemented as a distributed system.

Since the system does not require the perks of a centralized approach to operate well, it is ultimately a distributed implementation.

3.2 Implementation

I implemented the models using the neural network library known as Tensorflow[1]. Nodes 4, 5, 11 and 15 were chosen as learning and evaluation data as they were the ones significantly affected by the attack (see Figure 3.2) and thus were most likely to provide good learning data.

Data was preprocessed by being separated into two equally-sized sets - a training set and an evaluation set each in turn shuffled then split into readings and attack/no-attack labels. Care was taken to ensure that both sets had an equal attack-no-attack ratio. The data was then fed into a tensorflow neural network trainer with binary loss functions and success metrics, the system allowed to train over 16 iterations over the training data before the model was evaluated on the evaluation dataset.

When the data was separated into single readings during the preprocessing stage, the neural network achieved a correctness rating of roughly 88-91% over the course of multiple training attempts. Following some experimentation, I found that preprocessing the data such that each data point processed also contained the next few data points allowed the model to achieve greater accuracy. The model’s success rating reached 92-93% when the next two data points from the same node were included, and got as high as 94-95% when the next nine were included. Throughout all of these tests, the model falsely labelled circa 1% of the non-attack data as attacks.
The power consumption data proved somewhat harder for the model to accurately predict. When a single node’s data was fed into the base model without preprocessing, the overall accuracy was only 60%, with a higher accuracy (circa 70%) on attack than no-attack data (circa 50%). Feeding all twenty nodes’ un-preprocessed data in at once resulted in a much more polarized result, with an accuracy just over half where one side (varying between training attempts) achieved an accuracy over 80% whereas the other was only accurate a quarter of the time. It seems very likely that this is a consequence of how the majority of the raw data is power consumption levels of when no messages are being sent or received.

The main reason for the low detection accuracy when using all nodes is likely the fact that the effect of the jammer is not seen equally in all nodes. It has primarily local impact (see Figures 3.2 and 3.3), making the data from many nodes misleading to the learning algorithm as there is little difference between attack- and non-attack situations. Performing complete experiments like the above using all nodes in the power consumption data was also unfeasible due to the size of the data and the time limits on the project.

For these reasons, experiments on this data were limited to a few more well-chosen nodes: one case with the seemingly most promising nodes that seemed to react strongly to the attack (5, 17 and 20), one with just node 5 and one with three more normal nodes in terms of attack effect (4,10,16) that better represents the entire dataset’s average behavior.

I evaluated a few preprocessing techniques before deciding on the one mentioned in Section 3.1.2. Time series differencing turned out to be quite useless overall, as it did not notably improve accuracy. Simply filtering out all the points where no consumption spikes occurred only marginally improved accuracy. The method ultimately chosen performed better than these two, though it had the side effect of creating a lot more post-preprocessing data that was an attack than that wasn’t. This was probably because of the presence of more power spikes creating more data overall for the attack data than the non-attack data post-preprocessing. This could be balanced out by utilizing the non-attack data of additional nodes to balance the dataset, which can be seen in Figure 4.12.
Chapter 4

Evaluation

4.1 Experiments

A number of experiments were performed to evaluate various aspects of the implementation, which are shown and described here. For all these experiments, training data and validation data were always distinct halves of the same dataset.

Each experiment graph herein consists of three displays: the overall accuracy, the false positive percentage and the true positive percentage. These show the model’s ability to correctly classify readings, to not generate more false alarms than necessary and its ability to detect attacks, respectively.

4.1.1 Experiments on the roundtrip time data

Each of the following experiments generate a collection of twenty resulting data points. These data points are generated with the same values excepting the examined value, and are based on eleven distinct training sessions each. In total, an experiment is thus the result of 220 distinct runs.

As a reminder, only nodes 4, 5, 11 and 15 were employed as training data, both in order to reduce the time the experiments took to run and to limit the extent to which bad learning data could enter the data set.

Attack/no-attack data ratio

One experiment examined the relationship between the ratios of attack- and no-attack data in terms of the model’s accuracy - important background information. Crude expectations would be for true positive ratings to benefit from more attack data, whereas false positives benefit from more non-attack data.

A graph illustrates the results in Figure 4.1. Both true- and false positives are wildly
Figure 4.1: A graph displaying the results of training the model with differing ratios between attack and no-attack data. By default (in state 1/1), there are roughly five no-attack points for every attack point. We can see that the false positive rating flattens out at state 0.8/1, and the true positive rating worsens the more the no-attack data dominates, so the most appropriate data ratio appears to be four non-attack points for every attack point. Note the differences in y-axis.

The false positive ratings stabilize surprisingly quickly, whereas the changes in true positive ratings turned out to be more gradual, better matching the naïve assumption of a linear relationship.

This may hint at the conclusion that the best ratio is one where true- and false positive data occur in the same quantity. However, it is much worse to get a false positive than a false negative. Many false negatives will cause attacks to be less reliably detected, but more false positives will reduce trust in the system as a whole as an administrator will be falsely alerted of attacks. If attacks do not occur often, this is likely to reduce their trust in the detection system and may cause them to ignore true attack alerts entirely.

At equal ratios, true positive and true negative readings both reach 90%. However, using the full set with all data results in a false negative ratio of less than two percent (something more easily observed in later experiments’ figures such as in Figure 4.2) in exchange for a 25-30% reduction in true positive accuracy. This is worse when seen as percentual units, but in terms of percent it is a much better exchange as false positives are reduced by a factor of five. Thus, in order to limit the false positive ratio, I chose to accept the loss in true positive accuracy that using the full original dataset with its five-to-one no-attack bias brings.
Figure 4.2: A graph displaying the results of training the model over a varying number of iterations over the full dataset. Note the y-axes - the false positive ratings in particular experience very little variance. As it turns out, the model with this dataset requires only a few iterations. Additional ones afford little extra gain beyond the first few.

**Iterations**

Over the course of the learning process, a neural network in training will typically iterate over the full learning data several times, something that is true for this model too. In this experiment, the objective is to validate that there are enough iterations for the model to properly absorb the knowledge in the data. Given the size of the model, the number of iterations used in the other tests is quite likely excessive - the necessary amount is expected to be smaller. This is because a larger model with more layers contains more complex internal interactions and thus takes longer to reach global equilibrium.

Figure 4.2 showcases the resulting experiments. As it turned out, the system quickly reached mostly normal behavior after just a few iterations over the data, converging between two and four iterations for all metrics. 16 iterations is evidently excessive given the situation, but by all appearances the additional iterations appear to cause no notable changes in overall accuracy once the initial convergence occurs. To ensure that no unexpected slow convergence polluted the trustworthiness of the subsequent experiments and given the generally quite short runtimes allowing for some extra iterations to be used, 16 iterations were maintained.

**Window width**

The software being experimented on takes in several chronologically adjacent readings at once (using what is known as a sliding window) in order to make its predictions. Experiments were performed on how the size of the data window influenced the accuracy of the model at large. One would expect the predictions to improve as the model
Figure 4.3: A graph displaying the results of training the model with a differing amounts of subsequent data points. For a data segment of size \( n \), each data point read also includes the next \( n-1 \) data points (which in practice is not at all dissimilar to including the \( n-1 \) previous ones here and is done with the next ones here for convenience’s sake only). From this, we can see that it is most appropriate to include circa the ten subsequent/previous data points when a point is processed. Note the differences in y-axis.

The results are in Figure 4.3. Whilst it is true that including only a few data points causes worse results on all fronts (as the model has a bad overview of the scene), increasing the window size beyond eight seemed to have little discernible effect. Notable is that the false positive ratings seem to dip lower at a segment size of 7 to 12, something not observed in other tests. A reason for this may be that a window that is too large has a larger chance to include a misleading data point which may occasionally confuse the model. Furthermore, a larger window width implies a greater likelihood of the detection result being dependent on the result of the last sequential data point, so increasing the width comes with greater downsides.

The size of 3 generally used in the other experiments overall seems fairly reasonable. Though it does perform marginally worse than many higher values (particularly in the ‘dip’), it is ultimately a very minor difference, and limiting the effects of data dependency is also important.

**Data quantity**

Another point of interest is how the amount of data available affects the model’s accuracy, done by removing a certain portion of the data. This is done by gradually increasing the amount of data randomly picked from the data half used for learning, though the size of the validation data remains the same. All in all, the full learning set
Figure 4.4: A graph displaying the results of training the model on different amounts of training data with eleven runs per point. As can be seen, the accuracy, true positive and false positive ratings all roughly even out at circa 40% of the full training data set, meaning that there is sufficient data available for the model to have some merit. Note the differences in y-axis.

contains 240 hours of measurements, totalling 61652 values. The same is true for the validation data.

An obvious expectation here is to see the model’s accuracy sink whenever the data amount is insufficient, with the model generally improving with a larger wealth of data to use. On the other hand, more data means a longer learning process and, if data is taken from the validation set to boost the learning set, validation tests may become inaccurate.

Figure 4.4 shows the results. Using only a small percentage of the training data led to notably worse results; ones gradually improving as more data was included until the model eventually mostly converged at roughly 30% of the full learning set size. Due to this, it is an entirely reasonable assumption that the dataset is of sufficient size to facilitate good training for the model.

Various experiments with nodes and layers
A similar experiment was also performed with different layer setups in the model, though it proved ill-suited for display in a graph. As it turned out, most models containing at least two additional layers (aside from the default softmax one that formats output for binary prediction), with one of those being the input layer, performed well enough that any further increase in model complexity made little difference.

Notably, removing node 15 from the learning data resulted in a zero percent false positive rating and an eighty percent true positive rating. Of course, this is less useful
information than one may think, as it removes any real semblance of difficulty for the model by removing the source of high RTTs in no-attack scenarios, meaning that the model may have been less viable for deployment though it is hard to say such for sure.

Another experiment performed was the addition of other data points than just the round trip times. Adding loss ratio alone had little effect save for making the model lean a bit more towards classifying points as attacks, causing a higher true positive rating but also a false positive increase.

An attempt to include various different node ratings for the file, bringing each reading up to close to twenty points, merely caused the model to fail to detect any attacks at all. The consequences of polluting the learning data with unimportant data is as such quite clear - irrelevant data should be filtered out if possible. Most likely, the model could have surmounted this obstacle given a sufficient quantity of learning data, but I only have access to so much and the data in question was evidently insufficient for the model to learn things proper in this regard.

**Naive comparison**

As a comparison, a naïve AI algorithm based around a simple cut-off point was implemented, and can be seen in Figure 4.5. As one may expect, it performs notably worse than the neural network introduced earlier, with true positive percentages generally matching false positive percentages - both rapidly sinking to 30% and gradually descending to zero from there. The AI solution is evidently superior.

**Tests on singular nodes**

Given what was learned, testing the model on each individual node is of interest. For each node’s data illustrated in Figure 3.2, that is to say all with viable data, a test was performed to see how well the model predicted its behavior when learning from only that node’s data.

Figure 4.6 shows the results. Nodes 4, 5 and 11 -being the only ones with a high true positive rating- all exceed 95% accuracy whereas most others float around the 75% range. Node 18 performed especially poorly with 65% accuracy, managing to have a high false positive rating and a low true positive rating. In general, nodes that lack sufficiently distinct attack values seem to default to assuming that there is no attack, matching the data bias. This further highlights the importance of good training data.

### 4.1.2 Experiments on the power consumption data with preprocessing

The next point to be examined here is the experiments that were performed on the power consumption time series data gathered from the wireless sensor testbed specifically for this work. Each individual element examined was run 11 times, and most of the graphs below contain twenty elements, totalling 220 runs per experiment. Due
Figure 4.5: A graph displaying the results of a naïve attack detection algorithm based on RTTs. Recall that any message with an RTT exceeding ten timed out in the used data. The algorithm naïvely classifies segments based on the length of the round trip times in the segment, naïvely assuming that an attack occurs if there are more values above than below the metaphorical line. This example has a segment size of 3 as is default in the other graphs; brief tests showed that an increase in segment size similarly increased the naïve model’s tendency to conclude a non-attack, but did not increase overall accuracy.

As can be seen, the higher the limit, the more the model assumes that attacks are not occurring, lowering both true positive and false positive ratings, but even at its best it fails to reach an accuracy above 0.85. As each run uses a randomly picked 50% of the data, some variance is still visible, though results ultimately do not differ much.

Figure 4.6: An illustration of the model’s accuracy when trained on different singular nodes’ RTT data. Nodes 4, 5 and 11 exceed their peers by being able to correctly identify attacks, whereas node 18 performs especially poorly.
Figure 4.7: An illustration of the model’s accuracy when learning solely from the power consumption of different singular nodes. Some nodes are obviously easier for the model to learn from accurately, with nodes 5, 16, 17 and 20 in particular being the most accurate whereas node 14 is impressively inaccurate. Keep in mind that the learning- and validation data are non-overlapping subsets of the same dataset in all cases.

to time constraints, all of the gathered data could not be used at once for these experiments - some nodes were selected each time. For the sake of expediency, the number of iterations over the training data was reduced to 4, as opposed to the 16 used in Section 4.1.1 - this number proved decently viable in an earlier experiment as seen in Figure 4.2.

**Single-node accuracy test without preprocessing**

The model was initially tested without preprocessing, with each node’s data getting its own test. This shows which data was more viable for allowing the model to learn accurate predictions. Figure 3.3 already gave some clues as to how this result would turn out, with certain nodes (such as node 5) being more obviously viable than others (such as node 10).

Figure 4.7 showcases the result. Some nodes, such as 5 and 17, achieve very good accuracy close to 90%, with node 5 in particular exceeding 95% accuracy. Others such as node 10 perform no better than random guessing, and node 14 even managed to perform worse than that, misleading the model’s predictions entirely. Some nodes are clearly more appropriate learning targets than others.
Preprocessing margin experiments

I examined the impact of changing the length of the safety margin (how many ‘normal’ readings needed to occur in a row before the event was considered to be concluded) in the preprocessing method. I also examined how using different subsets of data influenced the overall accuracy. As a reminder, the data in question is summarized in Figure 3.3. These tests were combined by testing every length from 1 to 20 in the margins for three different data sets ((5,17,20),(5) and (4,10,16) - one set of three good ones, one set of the best one, and one mixed set of good and bad ones), as well as by comparing the accuracy of these sets taken ‘raw’.

The former case was examined in Figure 4.8. The naïve expectation is to see the curve reach a peak at some certain size - before that, it misclassifies an attack sequence as multiple smaller events, and after that it starts misclassifying distinct sequences as a single sequence.

In the (5,17,20) instance, this is the case. On the other hand, in the (5) instance, the accuracy only degrades with a larger limiter, presumably because the attacks are so blatantly obvious. In the (4,10,16) case, there is the expected initial rapid rise, but the accuracy doesn’t degrade again afterwards - though the overall accuracy of that instance is generally lower than with the other two.

The overall accuracy decrease is expected given that the experiment where no preprocessing took place was taken into account. As expected and seen in Figure 4.9, the (4,10,16) instance has a much-lessened accuracy compared to the other two. Most notably, the (5,17,20) case seems to be the one that benefits the most from the preprocessing, whereas the (5) instance is only minorly improved and the (4,10,16) case even seems to get slightly worse. The preprocessing thus seems to have some downsides in the more ambiguous cases, though the added advantage of the result taking up a lot less space than the raw data cannot be denied.

Single-node accuracy test with preprocessing

Now that the effects of preprocessing are better known, it is viable to once again examine single-node accuracy - this time with preprocessing enabled. Given the behavior of the curves in Figure 4.8, it would appear that easier sets suffer from higher preprocessing limiters, whereas the harder ones struggle more with low ones. Naïvely, one would expect the result of this experiment to resemble, though not imitate, the results in Figure 4.7.

Figure 4.10 showcases the result of the experiment. A few nodes, such as nodes 8 and 14, have improved notably, with node 20 emerging to nearly rival the accuracy of node 5. Some others, such as nodes 1, 3 and 17 experience drops in accuracy, however, with node 4 sinking below 40% accuracy not unlike what node 14 did in the un-preprocessed set. Given the behaviors seen in Figure 4.8, it is likely that the preprocessing method is ill-suited for handling cases where only a single node’s data is involved in the learning
Figure 4.8: Illustrations of how accuracy changes when the preprocessing batch margin is altered. Notable is that some of the changes in accuracy is due to there being a more even ratio of attack to no-attack data post-preprocessing as the limiter size increases. Nevertheless, the difference in behavior between the three sets is quite clear, though the less carefully chosen case (4,10,16) performs a lot worse overall compared to its compatriots.
Using only node 5, the most obviously affected node, creates the best accuracy, but this is an observation that may be better taken with a grain of salt as the verification data then also only emerged from set 5. The more "average" elements clearly perform a lot worse and are a lot more varied compared to the more deliberately chosen node case, however.

Ideally, another experiment would be performed here examining how the preprocessing as well as the normal runs work with combinations of nodes. However, due to time constraints, this experiment was not performed.

**Value count/margin limit interaction experiment**

Another point examined was how the changes in the preprocessing margin limit interacted with alterations to how many resulting values were included in every point. Considering that the post-preprocess values are rather small, it is not all too unreasonable for an implementation to save a number of prior preprocessed results for its analysis. For this reason, the limiter size tests were performed with widths of 1, 3 and 10 for the sake of comparison - all using the same dataset of nodes 5, 17 and 20. One would expect the result to improve with a larger value count. As can be seen in Figure 4.11, a higher number of values included seemed to stabilize the results and, particularly in the 10-case, worked to reduce the number of false positives by roughly twenty percentual units without exacting a notable toll on true positive ratings as the system had a better overview.

**Data padding experiment**

A lot of the experiments thus far suffer from high false positive rates due to the preprocessing method’s propensity for creating more attack data than no-attack data out of a previously equal-size pair of sets. An approach examined here shifts the natural bias of the learning process by adding additional non-attack data to compensate for
Figure 4.10: An illustration of the model’s accuracy when predicting preprocessed data from a single node. When compared to Figure 4.7, some nodes perform better whereas others perform worse. The benefits of preprocessing in this particular use case are thus somewhat dubious.

the preprocessing method’s inclination to produce more attack data than no-attack data given originally equally sized sets.

Adding additional nodes into the no-attack data seemed to resolve this issue, as is illustrated in Figure 4.12. The false positive rating is reduced to below 10%, though it comes at the cost of true positive ratings which degrade to only 60% accuracy.

4.2 Summatory experiment overview

Notable data from the experiments performed in Section 4.1 are gathered in table 4.1. This shows (amongst other things) that the best overall accuracy of 94% was achieved with the roundtrip time data in Figure 4.3, that the best preprocessing limiter size is 8 and that the model should iterate over the full data at least four times.
Figure 4.11: An illustration of how the results change in the preprocessed data when more events are included in an examined point, similar to what’s seen in Figure 4.3, though here only width 1, 3 and 10 are examined. Having only one event clearly creates a lot of unpredictability in the model’s behavior, whereas a larger width makes it more consistent. The effects of changing the limiter size are also visibly different between the three cases.
Figure 4.12: An illustration showcasing the effect of adding extra no-attack data to even out the otherwise-imbalanced preprocessed data. As can be seen, this has a very beneficial result in the false positive rate, though true positive rates suffer as a consequence.
### Raw data results

<table>
<thead>
<tr>
<th>Best raw data parameters</th>
<th>Iterations</th>
<th>Window size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best raw data attack accuracy parameters</td>
<td>4+</td>
<td>8+</td>
</tr>
<tr>
<td>Best raw data non-attack accuracy parameters</td>
<td>4+</td>
<td>7-12</td>
</tr>
<tr>
<td>Worst raw data parameters</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Preprocessing results

<table>
<thead>
<tr>
<th>Best preprocessing parameters - overall</th>
<th>Limiter size</th>
<th>Data set</th>
<th>Data width</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best preprocessing parameters - attack accuracy</td>
<td>1-9</td>
<td>Singular (5)</td>
<td>Small (1)</td>
</tr>
<tr>
<td>Best preprocessing parameters - non-attack accuracy</td>
<td>15-20</td>
<td>Mixed (4,10,16)</td>
<td>Large (10)</td>
</tr>
<tr>
<td>Worst preprocessing parameters</td>
<td>1</td>
<td>Mixed (4,10,16)</td>
<td>Small (1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Best preprocessing parameters - overall</th>
<th>Limiter size</th>
<th>Data set</th>
<th>Data width</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best preprocessing parameters - attack accuracy</td>
<td>1-9</td>
<td>Singular (5)</td>
<td>Small (1)</td>
</tr>
<tr>
<td>Best preprocessing parameters - non-attack accuracy</td>
<td>15-20</td>
<td>Mixed (4,10,16)</td>
<td>Large (10)</td>
</tr>
<tr>
<td>Worst preprocessing parameters</td>
<td>1</td>
<td>Mixed (4,10,16)</td>
<td>Small (1)</td>
</tr>
</tbody>
</table>

### Other notable values

<table>
<thead>
<tr>
<th>Best</th>
<th>Data ratio (no-atk/atk)</th>
<th>Training data percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/1 to 5/1</td>
<td>30+%</td>
<td></td>
</tr>
<tr>
<td>(0-0.5)/1 or 1/(0-0.5)</td>
<td>5%</td>
<td></td>
</tr>
</tbody>
</table>

### Best accuracies

<table>
<thead>
<tr>
<th>Overall</th>
<th>False positive</th>
<th>True positive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Roundtrip data</td>
<td>94% (Figure 4.3)</td>
<td>1% (Figure 4.3)</td>
</tr>
<tr>
<td>Naïve roundtrip data algorithm (Figure 4.5)</td>
<td>Avg. 80%</td>
<td>Avg. 15%</td>
</tr>
<tr>
<td>Preprocessed time data</td>
<td>92% (Figure 4.11)</td>
<td>8% (Figure 4.12)</td>
</tr>
<tr>
<td>Un-preprocessed time data (Figure 4.7)</td>
<td>Best 95% avg. 70%</td>
<td>Best 10% avg. 40%</td>
</tr>
</tbody>
</table>

Table 4.1: A set of tables containing various notable information points about the experiments performed for Section 4.1. Notable points include the best result of 94% overall accuracy and the best parameters for optimizing against overall-, false positive and true positive ratings in several cases.

**Glossary**

- **Iterations**: The number of times the learning process iterates over the full given learning data during the learning process.
- **Window size**: How many chronologically adjacent data points are included in a given data point - this is like a sliding window.
- **Limiter size**: How many 'normal' readings must be found in a row before the preprocessing method determines that the event is over.
- **Data width**: Preprocessed tests’ equivalent to window size.
Chapter 5

Conclusions and Future work

5.1 Conclusions

Given the knowledge gleaned from the experiments in Section 4.1, knowledge partially summarised in Table 4.1, several conclusions can be drawn.

Worthy of note is that a neural network does not necessarily have to be deep to be successful. The neural network herein worked well with only an input layer, an output layer to format the output and a singular hidden layer to connect the two. These layers do not even have to be particularly big. As such, it is not necessary to employ a deep neural network with a large number of layers in order to achieve fair success.

With a best-performance false positive percentage of just 1% and an accompanying, decent true positive accuracy of 60%, neural networks are a viable choice for attack detection algorithms. Given the existence of Tensorflow Lite (as mentioned in Section 3.1.3) which exemplifies the viability of resource-constrained neural networks, it is likewise viable for deployment on resource-constrained systems such as typical IoT devices. Thus, neural networks are viable for use in attack detection scenarios within the Internet of Things.

The most important lesson learned in regards to preprocessing is that it can have a very mixed effect. The preprocessing method employed in the experiments does mitigate the flatline issue that some nodes suffered (where a lot of the produced data windows contained no notable events and so only served to confuse the network’s learning process) and -perhaps more importantly- produces something that takes up a lot less memory than the raw data. On the other hand, some nodes suffered from the loss of information that the preprocessing brought. All in all, however, it appeared that preprocessing had a greater benefit when applied on data where attacks were less obvious.
5.2 Future work

One of the more obvious continuations on this work is to, based on the information presented herein, implement an Internet of Things intrusion detection system viable for deployment on a live system.

Though this work examines some AI methods, there are many more that go unexplored. A potential future work would thus be to explore various models not covered here in order to provide a more comprehensive view of the alternatives as a whole. Bayesian Networks and Support Vector Machines seemed to have particularly good potential here.

Another aspect to examine is the preprocessing methods. Section 4.1.2 utilized one preprocessing model and a few other possibilities were brought up in Section 3.1.2, but it is fully possible -very likely, even- that there is a better method than the one employed here. A future work could seek to determine what preprocessing methods work best for timelines where the data of interest is sparse, as well as what works best for detecting attacks.
Bibliography


