Aggregation of Heterogeneous Anomaly Detectors for Cyber-Physical Systems

by

Murray Dunne

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2018

Statement of Contributions

Portions of this thesis have been adapted from other works currently submitted for peer-review.

- Section 3.2 is co-authored by Sean Kaufmann and myself.
- The symptom examples in Section 3 were selected by Dr. Giovani Gracioli.
- The eleven anomaly scenarios in Section 7.3 were selected in a cooperation between Shefali Sharma and myself.
- The Palisade printed circuit boards were developed by Dr. Carlos Moreno. The Palisade firmware is entirely my own work.
- The autoencoder layout (Figure 4.2) was selected by Shailja Thakur.
- Figure 6.2 is co-authored by Dr. Carlos Moreno and myself.

All of the work in this thesis was completed under the guidance of Dr. Sebastian Fischmeister at the University of Waterloo. He provided ideas, discussion, and reviewed this thesis.

Abstract

Distributed, life-critical systems that bridge the gap between software and hardware are becoming an integral part of our everyday lives. From autonomous cars to smart electrical grids, such cyber-physical systems will soon be omnipresent. With this comes a corresponding increase in our vulnerability to cyber-attacks. Monitoring such systems to detect malicious actions is of critical importance.

One method of monitoring cyber-physical systems is anomaly detection: the process of detecting when the target system is deviating from expected normal behavior. Anomaly detection is a vibrant research area with many different viable approaches. The literature suggests many different anomaly detection methods for the diversity and volume of data from cyber-physical systems. We focus on aggregating the result of multiple anomaly detection methods into a final anomalous or non-anomalous verdict.

In this thesis, we present Palisade, a distributed data collection, anomaly detection, and aggregation framework for cyber-physical systems. We discuss various methods of anomaly detection and aggregation and include a case study of anomaly aggregation on a cyber-physical treadmill driving demonstrator. We conclude with a discussion of lessons learned from the construction of Palisade, and recommendations for future research.

Acknowledgements

Thanks to Dr. Sebastian Fischmeister, my supervisor, for all the help, support, and the opportunity to pursue this research.

Thanks to Dr. Carlos Moreno for helping with all my questions about hardware design.

Dedication

This thesis is dedicated to Kathryn, Tim, and Allie. Thank you for all your support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Cyber-physical systems are distributed, large-scale, and often life-critical systems such as autonomous vehicles, unmanned aircraft, and smart electrical grids [1]. They are composed of sensors, actuators, and various networking and processing equipment. From driverless cars to medical devices, our critical infrastructure is becoming distributed as fast as such systems can be integrated. With the rise of these systems comes a commensurate increase in cyber attacks [1].

We can monitor the condition of cyber-physical systems by examining live data traces and comparing them against a predetermined characterization of normal behavior. Such a process is called anomaly detection [2]. The anomaly detection problem can be stated as: given a stream of data representing a property of some system, determine if that data represents normal behavior or not. This result is often given as an estimate of the probability (or confidence) the system is performing anomalously. The source data stream could be an explicitly measured metric, such as the current speed of a car as collected by a speedometer, or side-channel data, such as the power consumption of a computer processor.

In this thesis, we consider only the detection of cyber attacks on cyber-physical systems. This approach is called intrusion detection. Intrusion detection is divided into methods that identify specific data patterns that are presumed to be malicious (called signature detection), or approaches that recognize deviations from normal system behavior (called anomaly detection) [3]. When we discuss intrusion detection in this thesis, we are referring to the second type: anomaly detection.

In physical systems, evidence of an attack as it occurs is sufficient to activate automated or manual safeguards. The objective of an anomaly detection system (or intrusion detection system) is to detect cyber-attacks on the instrumented system before they are capable of rendering harm. This adds a requirement to practical anomaly detection algorithms: they must work on system data as it is collected, that is, they must be *online* algorithms.

As cyber-physical systems grow and gain components, the number of data streams available to anomaly detection systems increases. An anomaly system must be able to ingest all these streams and produce a decision on the condition of the monitored system. Thus the anomaly aggregation problem for cyber-physical systems can be stated as:

Given a set of confidence levels from anomaly detection algorithms on multiple data streams at a given time, determine if they monitored system as a whole is behaving normally or anomalously at that time.

This decision may be informed by previous system state and the history of anomaly detector confidence levels.

A complex event processing system is a distributed system that aggregates event notifications to identify scenarios of interest [4]. A collaborative intrusion detection system is a set of individual intrusion detection systems that cooperate to detect coordinated attacks [5]. Collaborative intrusion detection systems are built to tackle the anomaly aggregation problem. We can consider a collaborative intrusion detection system to be a special case of an complex event processing system where the scenarios of interest are anomalous system behaviour. In this thesis we present Palisade, a complex event processing system that we use in this work for anomaly detection.

This thesis is split into three contributions. The first contribution is a description of *symptoms* of anomalies that may manifest in data being ingested by an anomaly detection system. We define symptoms of anomalies as perturbations in the data being considered for anomaly detection that may indicate the presence of an anomaly (see Chapter 3). The second contribution is Palisade, a complex event processing system that we use for anomaly detection in cyber-physical systems (Chapter 6). Palisade uses Redis [6] as a central publish-subscribe broker, which adjudicates between data sources, anomaly detectors, and aggregators. The final contribution is an evaluation of published aggregation algorithms on the anomaly aggregation problem (Chapter 7). We use data from a cyber-physical demonstrator platform and perform realistic cyber attacks taken from the Common Attack Pattern Enumeration and Classification (CAPEC) database.

We begin with a background review of ensemble learning, anomaly detection, and applications for intrusion detection (Chapter 2). We then present a mathematical setup for perturbations in data leading to anomalies, which we call symptoms (Chapter 3) which is followed by a discussion of anomaly detection algorithms (Chapter 4) and ensemble aggregation algorithms (Chapter 5). We discuss the construction of Palisade (Chapter

6) and then present a case study using an Advanced Driver-Assistance Systems (ADAS) demonstrator (Chapter 7) and its results and lessons learned (Chapter 8). We conclude in Chapter 9 and discuss future work.

Chapter 2

Background

2.1 Anomaly Detection

We have defined anomaly detection in the introduction (Section 1) based on Chandola et al.'s [2] definition. Anomaly detection is the process of comparing data representing the current state of the target system to a predetermined characterization of normal system behavior. Deviations from normal behavior are considered anomalies.

Anomaly detection can be thought of as a binary classification problem. Given a sample of data from the system, classify it as normal or anomalous. In an online environment, the definition of what constitutes a sample that can then be classified gets murky. Online classification techniques can be separated into stationary or non-stationary stream classifiers [7]. Stationary classifiers operate under the assumption that the data stream is stable. Non-stationary classifiers can account for *concept drift*. Concept drift is the notion that the underlying distribution of the data will change naturally with time, and this is not an anomaly. Consequently, machine learning techniques that apply in a non-stationary setting must be able to train (or adjust) online to not miss-classify data that has changed as a result of concept drift [7]. Concept drift can be mitigated by continuously updating the model as the underlying distribution of the data changes [8] or by trigger mechanisms based on statistic change detectors [9].

In this thesis, we consider only stationary classifiers in an anomaly detection context. If the system is varying from its expected operating conditions, we consider it an anomaly, not a natural process. We leave an investigation of non-stationary classifiers in an anomaly detection context for future work (see Section 9.2).

Krawczyk et al. also make a distinction between chunk-based and online classifiers [7]. Chunk-based techniques build up a set of samples in a fixed-size buffer before evaluating them as a batch, which might include several iterations over the chunk. This is distinct from Krawczyk's definition of an online detector, which processes each data instance separately at arrival time [7]. This definition is difficult to distinguish in practice: it may be that some "online" classifiers maintain a buffer of previously seen samples as internal state, even if they produce a classification for each input. Alternatively, it could be that a chunk-based technique keeps a fixed-size buffer internally that may be iterated over several times and produce a classification for each sample in that buffer. As such, we do not distinguish classifiers by this metric.

Anomaly detection techniques for intrusion detection are broken down by Hoang et al. into three categories: statistical methods, data-mining methods, and machine learning methods [10]. They consider statistical methods to be those that measure a statistic about a system property, such as processor usage, or network packet count, and check that statistic maintains a distribution observed during normal system behavior. Data-mining methods extract additional features from underlying metrics. Methods such as outlier detection, system-call sequence analysis, Bayesian belief networks, and hidden Markov models are data-mining based methods. Finally, machine learning methods consist of more opaque designs such as artificial neural networks or random forests.

2.2 Ensemble Learning

Ensemble learning is the process of making a classification decision as a composite of multiple classifiers [11]. Ensemble learning systems may also be called: ensemble classifiers, composite classifier systems, classifier fusion, committees of neural networks, voting pool of classifiers, and various other names.

2.2.1 Diversification

The primary benefit of ensemble classifiers comes from mitigating the weaknesses of the member classifiers such that the composition improves on the accuracy of any one member. For such improvement to be possible, the classifiers must be *diverse* [11]. Specifically: as no classifier is perfect, the member classifiers must make different errors. This can be accomplished by training the member classifiers on (possibly disjoint) subsets of the training data, varying the training parameters of the classifiers, or combining entirely different classification methods (the approach we take this thesis) [11].

There exist a multitude of statistics for quantifying diversity. Polikar describes six of them in his 2006 paper [11]: correlation, q-statistic, disagreement and double fault measure, entropy, Kohavi-Wolpert variance, and measure of difficulty.

2.2.2 Diversifying Algorithms

Algorithms for ensemble learning are often structured around inducing diversity in a population of classifiers built with the same algorithm [12, 13, 14, 15]. Here we discuss several algorithms that induce diversity to construct an ensemble.

Boosting

Schapire proved in 1990 that the existence of a "weak" learner (a learner with some error α and a lower bound on such error) can constructively be used to build a "strong" learner (with error at most $3\alpha^2 - 2\alpha^3$) [12]. That is, the error can be made arbitrarily small, rather than having a fixed bound. He begins training a classifier h_1 on some subset of the training set normally. He then trains a second classifier h_2 on a subset of the training set composed half of samples correctly classified by h_1 and half of samples incorrectly classified by h_1. A third classifier h_3 is trained on instances where h_1 and h_2 disagree, and the final classification is made by majority vote of h_1, h_2, and h_3. This algorithm can be repeated arbitrarily to boost the accuracy of any given classifier, so long as the classifier can produce results better than random guessing for all inputs.

AdaBoost

Freund and Schapire improve on boosting with a multiplicative weight update algorithm called AdaBoost [13]. Beginning with each training sample having equal weight, the chance that a given sample is drawn into the training set for the next classifier decreases multiplicatively for each classifier that has previously correctly classified it. The multiplicative factor is proportional to the error of the classifier on that iteration. That is, samples that are "easy" to classify are selected for training less often as later classifiers are trained. The final classification is produced by tallying the number of times a sample was classified as a given class across all iterations, and weighting them by the error of the classifier at the iteration that classification was made.

There exist several variants of this algorithm [11, 16], including versions capable of boosting accuracy for regression problems.

Bagging

Breiman describes a simple ensemble learning algorithm using a procedure called bootstrap aggregating (shortened to *bagging*) [14]. Bagging takes the subset approach to diversification and requires only one classification algorithm. For a training set T of size n, bagging begins by drawing k replicate data sets by sampling n samples from T with replacement. This means that each replicate set is the same size as T, but may contain multiple copies of some samples, and omit others. Breiman later adds a variant of bagging using smaller subsets that selects samples that are important, similarly to Schapire [17]. Chawla et al. show this is superior to random selection [18].

Random Forests

In 2001 Breiman developed the popular Random Forest approach to ensemble classification [15]. He begins by drawing a training set for each tree using bagging, and then feature selection and ordering is randomized for each tree. The trees are not pruned. A plurality vote among the trees determines the final classification. The combination of random test set selection, random feature selection for each tree, and random ordering of feature decision within each tree produces significant diversity between trees. Breiman shows that random forests compare well to AdaBoost on his data sets [15].

Haeusler et al. user random forests along with seven confidence measurement functions to stereo vision problems [19]. They begin by sampling the training sets using bagging [14], and then train each tree by greedily selecting a confidence measure and associated binary decision test with the lowest error according to the chosen measure. Final results are decided by majority vote (they only have two classes) as described by Breiman [15].

Stacked Generalization

Wolpert suggests a method where the predictions of the ensemble classifiers are used to create a meta-classifier [20]. Similarly to cross-validation, the component classifiers are trained on most of the data, omitting a different subset for each classifier. The classifiers are evaluated on these previously omitted subsets and the resulting predictions, along with the true labels, act as input to the meta-classifier [11]. Wolpert states that this can be viewed as a more sophisticated version of cross-validation, and a generalization of a winner-takes-all ensemble method [20].

2.2.3 Combination Techniques

Combination based techniques focus on methods of aggregating the class label results of the member classifiers, rather than methods varying the classifiers themselves [11]. This allows them to operate on an ensemble composed of classifiers based on different algorithms, rather than varying a single algorithm in the case of diversifying techniques.

Voting

Both simple majority and weighted majority voting are reasonably effective under the assumption that each classifiers output is independent and sufficiently accurate [11]. Weighted majority voting may increase the accuracy of the ensemble if some member classifiers are more accurate than others. For many ensembles, the assumption that the classifier's output is independent is either not true (for classifiers trained with bagging or boosting) or extremely difficult to verify.

Mukkamala et al. employ majority voting in their ensemble based intrusion detection platform [21]. They use three independent machine learning algorithms which vote on a class for a sample of network activity.

Mixture of Experts

Similar to stacked generalization, Jacobs et al. suggest training a set of weights that are then fed to a straightforward combination rule [22, 11]. The primary distinction here is that the network that learns the weights for the combinator can consider the input training samples, whereas the meta-classifier in stacked generalization does not know about the underlying training data.

Logistic Regression

Ho et al. use logistic regression to combine the results of multiple classifiers in an ensemble [23]. They convert the probabilities of each class output by a member classifier on a sample into a rank vector. Then for each sample, for each class, the rank of that class in each detectors rank vector becomes a new sample class prediction vector. These vectors train a logistic regressor that produces the ensemble classification.

Bayesian Combination

Predictions from independent member classifiers may be combined in a Bayesian fashion as outlined by Buntine [24, 16]. They assign each classifier a weight equal to its accuracy on a given training set. This is multiplied by the classifiers probability for each class, and these values are summed across all member classifiers. The highest total likelihood is the ensemble prediction. More explicitly: for member classifiers C_i, training set T with samples x_j and labels $y_j \in L$ we let:

$$Acc(C_i) = \big(P(C_i(x_j) = y_j | L(x_j) = y_j)\, \forall x_j \in T\big) \tag{2.1}$$

$$Cla\hat{s}s(x) = \arg\max_{l_k \in L} \sum_i \left[Acc(C_i) P(C_i(x) = l_k | x)\right] \tag{2.2}$$

Dempster-Shafer

Dempster-Shafer can also be used for combining classifiers by giving each possible class $y \in L$ a "basic probability assignment" $b(y, x)$ for any sample instance x [16]. From there a given belief function can be maximized subject to a normalization factor.

Online Approaches

Krayczyk et al. outline the most common structure for online approaches as a voting ensemble with update replacement [7]. For each chunk of streamed data D_i all the classifiers in the pool cast a vote on the class of D_i and the winner is determined by some voting system. Then a new classifier C_i is trained on only D_i and added to the pool. If the pool exceeds a maximum size, some classifier is removed from the pool. This may be simply the oldest classifier, or the poorest classifier according to some performance metric.

Street and Kim's approach matches Krayczyk et al.'s standard framework, but they weight the performance of each member classifier according to how close the vote is [25, 7]. On a correct prediction, the performance metric of that classifier is increased in linear proportion to the difference between the number of votes given to the highest voted class, and the number of votes given to the second-highest voted class. If the highest voted class was incorrect, this difference is instead the difference between the highest voted class and the number of votes for the correct class. If it predicts incorrectly it's value is decreased proportionally to the difference between the number of votes for the class it voted for, and the number of votes for correct class [25]. By this metric, classifiers may have high accuracy classifying simple samples, but their inclusion would not be useful. Classifying the simple samples correctly is only a necessary characteristic for a plurality of the classifiers; a classifier must perform on the remaining samples to be a valuable inclusion in the ensemble. Street and Kim remark that this approach performs comparably to a single classifier for stationary datasets (as expected). However, in the presence of concept drift, they recover from the drift dramatically faster than a single classifier [25].

Scholz and Klinkenberg take a similar approach to Street and Kim, but they assign weights to each training sample and to each classifier [26]. For each batch read, they may choose to train a new classifier to add to the ensemble depending on the performance of existing classifiers on a chunk of input. The member classifiers are re-weighted continuously based on the new data and the weights of the iteration of the classifiers (Scholz and Klinkenberg compare this to logistic regression [26]).

2.2.4 Deep Learning Based Approaches

Deep leaning based ensemble methods are a mix between varying a single classification algorithm [27, 28] and combining classification results [29, 30]. They generally employ neural networks [27, 30] to produce the ensemble classification.

Meta-Classifiers

Qui et al. employ a meta-classifier approach similar to stacked generalization [27]. They train a set of 20 deep belief networks (essentially deep fully-connected networks) with varying hyperparameters. The results of those networks are collected into a matrix which feeds a support vector regressor [31] to produce the predicted values. They compare their results to a standalone support vector regressor, a smaller three-layer neural network, a single deep belief network, and an ensemble technique using multiple copies of the three-layer network. Their ensemble of deep belief networks outperforms the other options across seven distinct datasets.

Deng and Platt employ a mixture-of-experts style linear and log-linear combination technique to speech recognition [29]. They consider the posterior probabilities of each class for each frame of input audio as a set of result vectors from the member classifiers. Each classifier is assigned a weight matrix, which can be solved for analytically over a training set for in both the linear and log-linear cases. They construct an ensemble involving a deep fully-connected network, a convolutional deep network, and a recurrent network. Their results show that log-linear ensembles outperform linear ensembles in the two-member and three-member ensembles they considered. All ensembles beat any network in isolation.

Yin et al. construct an ensemble of stacked autoencoders to classify human emotions based on physiological signals [30]. They build a stacked autoencoder by first training a single autoencoder and then training the next autoencoder with the hidden representation of the first network. The hidden representation of that layer is used as input for the next layer and so on. The network ends with a two-neuron output layer that produces the target emotion ranges. They construct a separate stacked autoencoder for each physiological feature, which are then grouped two-by-two in a tree structure by fully connected layers. Finally the output at the end of the tree is fed into a Bayesian model that computes the final emotional verdict. This is a stacked generalization approach taken to the extreme. Yin et al. report a 5.26% improvement over the then best existing classifier [30].

Xu et al. use the inbuilt variability of extreme learning machines to build an ensemble classifier for real-time security assessment of power systems [28]. Extreme learning machines are similar to traditional feedforward neural networks, except the weights of the hidden layers are selected randomly, and the biases are computed analytically, rather than using various gradient descent methods. This causes extreme learning machines to have significant internal variability over the same dataset and provides the needed diversity to implement an ensemble. Since Xu et al.'s security classification is binary, the implemented as a single result value in $[-1, 1]$. They then split this range into three regions: credibly secure, incredible, and credibly insecure. From there a vote is taken where the larger count among the member classifiers of credibly secure or insecure becomes the classification, as long as the number of incredible classifications does not exceed a threshold. Xu et al. also consider real-valued security measures (rather than binary classification). Here results are deemed incredible if they deviate sufficiently from the median value between all the member predictors.

Bagging Approaches

Yang et al. employ bagging as described by Breiman [14] on an ensemble of stacked denoising autoencoders [32] to predict oil prices. Denoising autoencoders first corrupt the input data intentionally and at random, then the corrupted data is fed through an autoencoder. The hidden representation of the first autoencoder is used as input for the next denoising autoencoder in the stack. The stack ends with a supervised neural network taking input from the final autoencoder's hidden representation. Yang et al. then construct replica training sets drawn randomly with replacement (as described by Breiman) and train a stacked denoising autoencoder on each set. They take the average predicted price as the ensemble prediction.

2.3 Collaborative Intrusion Detection Systems

Zhou et al. define a Collaborative Intrusion Detection System (CIDS) as a set of individual Intrusion Detection Systems (IDSs) that cooperate to detect coordinated attacks [5]. Elshoush and Osman provide a general architecture for this technique in their survey of collaborative intrusion detection systems [33]. They further refine this concept into Collaborative Intelligent Intrusion Detection Systems (CIIDSs), which perform *alarm correlation* on the results of the member intrusion detectors.

Elshoush and Osman's architecture splits a CIIDS into two specific types of subsystem: *detection units*, and *correlation units* [33]. Correlation has a well-defined mathematical definition from statistics. Thus, to avoid confusion, we refer to Elshoush and Osman's correlation units as *aggregation units* in this thesis. A detection unit is a single IDS that monitors a subsection of the target system and produces simple alerts. Detection units are supposed to provide occasional false positives. They can afford to occasionally report perturbations that may not actually be anomalous rather than avoiding reporting things that do represent an underlying anomaly. Aggregation units then take the simple alerts from detection units and use an alarm correlation algorithm to transform them into a final intrusion verdict on a system scale. Elshoush and Osman also note the existence of hierarchical CIIDS structures with multiple layers of aggregation units [33], but they indicate this approach is weakened by intermediate aggregation units abstracting away data that may be useful in producing a final verdict.

Chapter 3

Symptoms of Anomalies

We define anomaly detection for cyber-physical systems (see Section 2.1) as characterizing data representing the state of a cyber-physical system as normal or anomalous. However, small deviations from normal behavior are to be expected on occasion and are not representative of an anomaly alone. For example, an unusually cold reading from a temperature sensor may mean a malicious actor has moved the sensor, or that it is just an unusually cold day. To this end, we construct the idea of an anomaly symptom.

Symptoms represent the realization of a perturbation in an internal, unobserved state machine. They do not prove an anomaly by their mere presence, but an anomaly may cause one or more symptoms, hence the disease-symptom analogy. A system may be behaving anomalously (diseased), but an anomaly detection system can only ever observe the symptoms of that anomaly.

Therefore, this chapter does not address types of anomalies, but rather the symptoms of underlying anomalies. The list in this section is not exhaustive but categorizes common anomaly symptoms. This taxonomy relates to Mitre's Common Attack Pattern Enumeration and Classification (CAPEC) [34], in that both can be used to classify capabilities and behaviors. They differ substantially in that CAPEC describes possible attacks, while this section describes *symptoms* of those attacks or other, non-malicious events.

These symptom definitions allow those developing anomaly detection systems to give formal names to perturbations that may indicate an anomaly in the target cyber-physical system. These identifications can be used to select and implement anomaly detection algorithms that target those anomalies.

In Section 7 we use these symptoms to identify perturbations that may occur as a result of cyber attacks we run on a target cyber-physical system. We use a Advanced

Driver-Assistance Systems (ADAS) demonstrator [35] to model attacks on a car driving on a treadmill. We select anomaly detection algorithms for our case study based on the symptoms identified from the chose cyber attacks.

3.1 Continuous Signal Anomaly Symptoms

For the purposes of anomaly symptoms, we define a continuous signal as a digitally sampled signal with a constant sample rate, represented here as a time series. This signal is expected to be the result of readings from a single sensor, not an amalgamation of many sources. The constant sample rate means that the sample time of each value is known from its index in the time series.

3.1.1 Spikes and S-waves

We define a spike (Figure 3.1(a)) as a subsequence of contiguous samples that lie farther than a given number of standard deviations from the current mean of the signal. To account for signals with means that change over time, we consider the distance to the mean of a *window* of samples prior to the subsequence. More formally, given a time series y, a window size n, and a constant factor c, the subsequence $y_{[p+1,q]}$ is a spike iff

$$\forall y_t : p < t \leq q,\ |y_t - \bar{y}_{[p-n,p]}| > c \cdot stdev(y_{[p-n,p]}) \tag{3.1}$$

We define S-waves (Figure 3.1(b)) as spikes with an additional deviation in the opposite direction immediately following the spike. S-waves can mimic spikes if the counter-spike is sufficiently dampened.

Example: A flooding attack in the vehicle Controller Area Network (CAN) network indicating that the collision prevention system issued a command to engage the brakes can cause a collision [36] and it is an example of a Spike/S-Wave symptom. Such an attack falls under the category of CAPEC-125: Flooding [34].

3.1.2 Drifting

A drift (Figure 3.1(c)) is a slow movement of the signal mean over a period of time. We consider only linear drift here; logarithmic and sub-linear drifts are rare, and higher order drifting encroaches on the definition of level changes or spikes. Mathematically, a continuous signal y is offset by tc, where t is the time index and c is a constant representing the slope of the drift. Formally, given a time series y, a nominal version of that time series $\hat{y}$, and a slope c, a subsequence $y_{[p,q]}$ has linear drift iff

$$\forall y_t : p \leq t \leq q,\ y_t = \hat{y}_t + tc \tag{3.2}$$

Example: An infrared combustible sensor, when functioning over the operational temperature limit, may drift or fail [37].

3.1.3 Noise

Noise (Figure 3.1(d)) is a usual part of any signal. Noise is considered a symptom of an anomaly only when it is more pronounced than is typical. We define noise as a normally distributed offset around the true value of the signal. Given a time series y, some noisiness coefficient n and nominal time series $\hat{y}$, a subsequence $y_{[p,q]}$ is noisy iff

$$\forall y_t : p \leq t \leq q,\ y_t = \hat{y}_t + \mathcal{N}(0, n) \tag{3.3}$$

Where $\mathcal{N}(0, n)$ is a standard normal distribution centered at zero with standard deviation n.

Example: Compressed air in truck brakes may generate acoustical interference and cause metallic friction noise from track vehicles in ultrasonic sensors [38].

3.1.4 Clipping

We define clipping as a loss of data at the extrema of a signal range (Figure 3.1(e)), where a signal is of a higher amplitude than is supported by the sensor or transmission medium. Thus a clipped signal can be represented by a series of identical samples at the maximum or minimum extent of the sample medium.

Example: A partially blinding attack on a camera of a vehicle by emitting light can hide objects [39]. This light can exceed the input range of the camera and would appear as clipped. This attack is an example of CAPEC-607: Obstruction [34].

3.1.5 Loss

While loss (Figure 3.1(f)) may more typically refer to high noise levels making it difficult to decode a signal, here we use loss to indicate a complete loss of a signal. Although trivially an anomaly, a total loss of signal may be a symptom of temporary network disruption without any more dangerous cause. We represent a total loss of signal as a sudden transition to a fixed sample value. This can be observed as a special case of clipping, where the extrema of the signal are identical for a short time.

Example: An attack sending a large volume of request messages over the J1939 protocol increases the computational load of the recipient ECU until it is not able to perform regular activities like transmitting periodic messages [40]. Such an attack is an example of CAPEC-125: Flooding [34].

3.1.6 Smoothing

We define smoothing to be a reduction in the short term variance of a signal compared to recent history. Smoothing (Figure 3.1(g)) is the rarest of the symptoms presented here, with few natural causes. Given a constant k representing how far back the recent historical signal variance should be considered, and the factor threshold τ at which the signal is considered smoothed, we say a subsequence of n samples $y_{[t,t+n]}$ is smoothed iff

$$var(y_{[t,t+n]}) < var(y_{[t-(nk)-1,t-1]})\tau \tag{3.4}$$

Example: In an attack of a control system, the attacker may observe and record sensor readings and then continuously repeat the recorded values during the attack [41]. This is an example where the sensor values are smoothed. Such an attack falls under the category of CAPEC-148: Content Spoofing [34].

3.1.7 Amplification

Amplification (Figure 3.1(h)) is a simple gain on the target signal. For amplification of an original signal we multiply every sample by some factor. Given the magnitude of the amplification α, and an unamplified time series $\hat{y}$, a sample y_t is amplified iff

$$y_t = \alpha\hat{y}_t \tag{3.5}$$

Example: Analog to Digital Converters (ADCs) can be attacked by amplifying analog signals past the dynamic range of the device. These attacks can obscure other malicious behavior and damage hardware [42]. This type of attack is an example of CAPEC-153: Input Data Manipulation [34].

3.1.8 Level Change

A level change (Figure 3.1(i)) symptom is observed when the mean of a signal changes in a short period and then remains consistent at the new level. Slower changes may fall under drifting. Given a time series y, an acceptable minimum level change threshold ℓ, a minimum number of samples the mean change must persist n, a level change has occurred over a window of w samples $y_{[t,t+w-1]}$ iff

$$|\bar{y}_{[t+w,t+w+n]} - \bar{y}_{[t-n-1,t-1]}| > \ell \tag{3.6}$$

Example: An attack that increases the amount of code execution will increase the power consumption of the system, which can be observed as a level change [43]. Such an attack could be an example of CAPEC-175: Code Inclusion, or CAPEC-242: Code Injection [34].

3.1.9 Frequency Change

A frequency change (Figure 3.1(j)) occurs when the primary frequency of a signal changes over a short period. We say a frequency change occurs if the primary frequency in a sliding window moves more than some threshold over some time window. Given a time series y, a function P which extracts the frequency of the highest peak from a Discrete Fourier Transform (DFT) (denoted $\mathcal{F}$), a threshold τ, and a minimum number of samples the frequency change must persist n, a subsequence of w samples $y_{[t,t+w-1]}$ experiences frequency change iff

$$|P(\mathcal{F}(y_{[t+w,t+w+n]})) - P(\mathcal{F}(y_{[t-n-1,t-1]}))| > \tau \tag{3.7}$$

It may be useful to consider more frequencies, but we restrict our definition to only consider the primary frequency for simplicity.

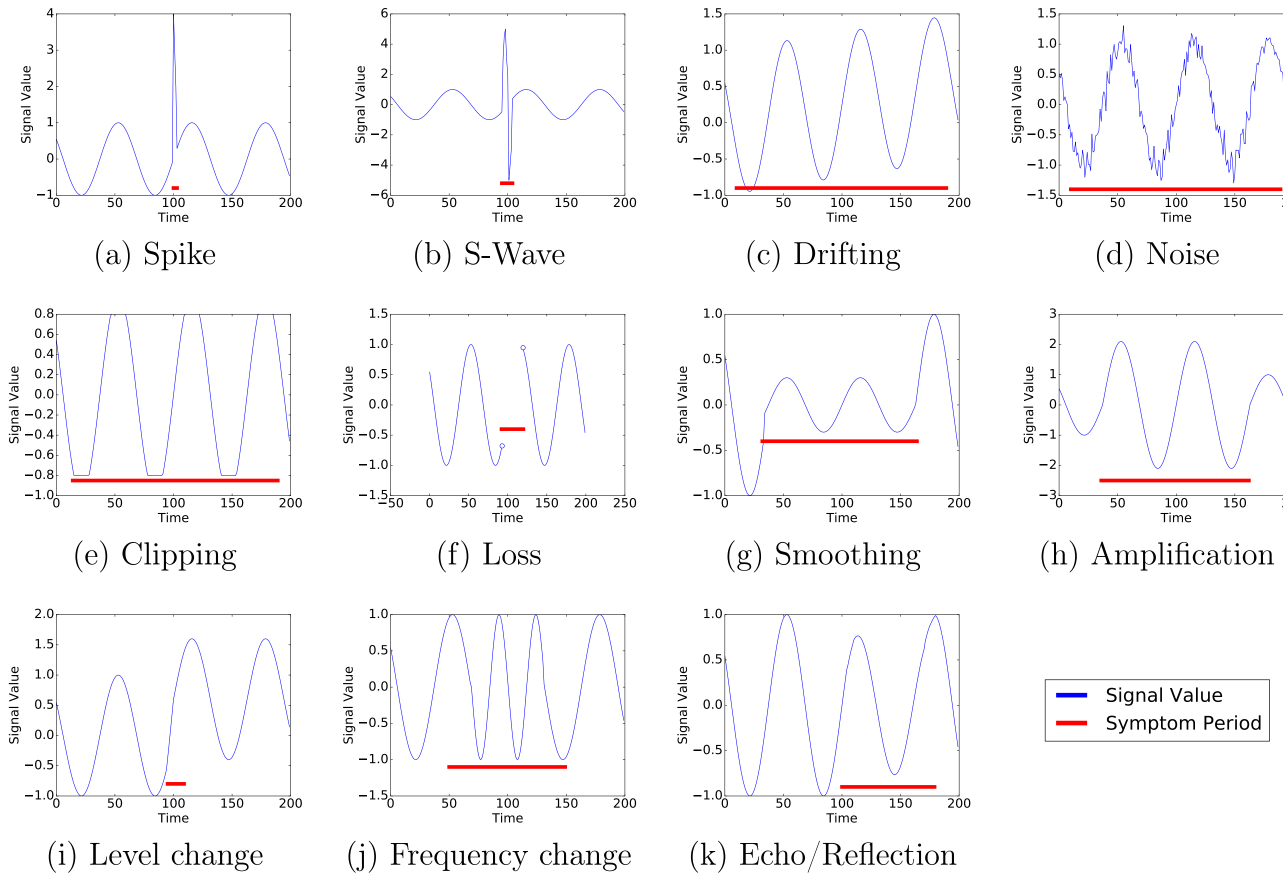
Example: An attack inserting flash of light into the vehicle camera may change the frequency in which the control reacts to new environmental conditions [39]. This attack is an example of CAPEC-607: Obstruction [34].

3.1.10 Echo/Reflection

We consider an echo (Figure 3.1(k)) to be a duplication of a previous series of samples on top of the underlying signal at a later position. A reflection is identical to an echo, excepting that the repeated signal is inverted. Given a time series y, an echo length e, an echo coefficient (the factor at which the echo is played back) q, and the nominal form of the time series $\hat{y}$, we consider the subsequence $y_{[t,t+e]}$ as the origin of the echo, we say that the subsequence $y_{[t',t'+e]}$ has echo iff

$$y_{[t',t'+e]} = \hat{y}_{[t',t'+e]} + y_{[t,t+e]} \times q \tag{3.8}$$

Example: According to Petit et al, a relay attack on the original signal sent from the vehicle LiDAR creates fake echoes and can make real objects appear closer or further than their actual location, thus affecting the mission planning [39]. This attack is an example of CAPEC-586: Object Injection [34].



(a) Spike (b) S-Wave (c) Drifting (d) Noise (e) Clipping (f) Loss (g) Smoothing (h) Amplification (i) Level change (j) Frequency change (k) Echo/Reflection

Figure 3.1: Time Series Anomaly Symptoms

3.2 Event Series Anomaly Symptoms

We define event series as a sequence of discrete samples with no defined frequency. The time between events is neither bounded nor known in advance. While a single sample from a time series carries only a single real value, an event (sample) in an event series may carry more complex, hierarchical information. We define a single event as a tuple $(\mu, t, \cdot)$, where μ is the topic the event occurs on (also called the source, or the channel), t is the timestamp of the event, and $\cdot$ is whatever information the event carries. An event represents the advancement of the execution by a single transition of a finite automaton.

3.2.1 Event Frequency Change

While we define events to have no fixed sampling frequency, they may still exhibit periodic or semi-periodic behaviour. This means for some event series, we can expect a reasonably consistent inter-arrival time (or frequency). When this frequency changes suddenly or unexpectedly, it can be a symptom of an anomaly. It is especially likely to be an anomaly symptom when such a change is observed in multiple traces simultaneously.

We define the inter-arrival time of an event as the difference in clock times of successive events in the same topic (also as the inverse of the event frequency). Specifically, given a trace T, a topic μ, and a non-empty interval defined by the end points $t_1, t_2 : t_1 < t_2$, the inter-arrival time is defined as $\mathit{interArrival}(\mu, (t_1, t_2)) \triangleq ((\max t : (\mu, t, \cdot) \in S) - (\min t : (\mu, t, \cdot) \in S))/(|S| - 1)$ where $S = \{(\mu, t, \cdot) \in T : t_1 \leq t \leq t_2\}$.

Event frequency measures how often events occur for some time span. Given a trace T, an event name μ, and a non-empty interval defined by the end points $t_1, t_2 : t_1 < t_2$, the event frequency of μ is given as $\mathit{eventFreq}(\mu, (t_1, t_2)) \triangleq 1/\mathit{interArrival}(\mu, (t_1, t_2))$.

A rapid change in event frequency can be found by taking the difference between successive time intervals in the trace. If the difference exceeds some threshold, then the change in event frequency may indicate an anomaly. Given a trace T, an event name μ, a window size w, and a threshold ϵ, an event frequency change may be defined as $\exists t_1, t_2, t_3 : \mathit{eventFreq}(\mu, (t_1, t_2)) - \mathit{eventFreq}(\mu, (t_2, t_3)) > \epsilon$.

Example: Lin and Siewiorek introduced their Dispersion Frame Technique (DFT) to predict hardware failures [44]. From analyzing the logs of file servers, they observed that there exists a period of an increasing rate of intermittent errors before most hardware failures.

3.2.2 Unexpected Event

In general, traces contain only a limited vocabulary of event names. While the actual events are made unique by their clock times, the event names are repeated across many events. An event with a never-before-seen name in a trace may be an anomaly symptom.

To remove the problem of the first few events in any given traces counting as anomalous merely by being the first instance of a normal event, we state the definition in terms of the probability of an event occurring. Given a trace T and a threshold ϵ, an unexpected event may be defined as an event $e : \mathcal{P}(e \in T) < \epsilon$.

Example: Bellovin reported receiving broadcast packets meant for local networks, requests to unused ports, and requests to unoccupied addresses over the public Internet at AT&T in his classic whitepaper [45]. These types of requests are examples of CAPEC-169: Footprinting [34].

3.2.3 Periods of Silence

A period of silence is a segment of time within a trace which is entirely or nearly devoid of events. Events generally have differing frequencies during normal system operation, but nominal system behavior usually results in at least some events appearing in any given period of time. Thus a period in the trace without events may be an anomaly symptom.

The time threshold for when an interval is considered a period of silence will vary between systems. Accounting for these differences, we define a minimum length of the interval that is system dependent. Now we can define a period of silence given a trace T, a number of events n, and a minimum length $\ell > 0$. A period of silence is an interval with end points $t_1, t_2 : t_2 - t_1 \geq \ell$ and $|\{(\cdot, t, \cdot) \in T : t_1 \leq t \leq t_2\}| < n$.

Example: Haque et al. found that Markov Chain models performed well at detecting missing message anomalies in High Performance Computing (HPC) logs [46].

3.2.4 Sampled Value Anomaly Symptom

When a trace is composed of events, it may still exhibit anomalies that are found in continuous data streams as defined in Section 3.1. However, as events do not have a defined sampling rate from which to derive a corresponding continuous data stream, we must apply a more clever transformation. There are several well known methods for extracting a continuous data stream from irregularly spaced events the Wiley/Marvasti method [47], the Voronoi method [48], and the Adaptive Weights Method [49].

Example: Wang and Stolfo found that PAYL payload based intrusion detector was able to achieve nearly 100% accuracy in some instances by comparing the byte frequency patterns of network payloads to the same host and port [50]. This is an example of Sampled Value Frequency Change.

Chapter 4

Anomaly Detectors

Anomaly detectors are algorithms that consider the data from a single system metric and consider if that metric is anomalous in isolation. We introduce detectors here to set up the discussion of anomaly aggregators in Section 5. Anomaly aggregators are constructed from the results of multiple anomaly detectors, so we describe anomaly detectors first.

A single metric from a cyber-physical system performing anomalously may not warrant a system-wide anomaly, so an anomaly detector alone is free to produce false positives if necessary. Anomaly detectors consume one or more streams of data (that may have been pre-processed) and produce a new stream: the expected probability of an anomaly. In this section, we discuss the different types of anomaly detectors and review existing example detectors of each type. We do not define any new anomaly detectors.

To clarify the discussion of anomaly detection algorithms, we use the following notation. An anomaly detector is a function

$$f(D) \mapsto P(anomaly) \in [0, 1]$$

were $P(anomaly)$ is the detector's best guess of the probability there is an anomaly within D, a set of contiguous samples from a source data stream. A data stream is either a time series or an event series (see Section 3), represented as an arbitrarily long vector of samples, each with an associated time they were observed. That is, for a given window (sub-vector) in the source data stream the anomaly detector produces a probability $P(anomaly) \in [0, 1]$ that it believes its source data stream to be anomalous within that window. It is expected that $P(anomaly)$ is zero for the vast majority of windows examined by the detector. Therefore, detectors may choose not to produce any output for a window, which will be formally interpreted as producing $P(anomaly) = 0$.

We divide anomaly detectors into three broad groups: watermark detectors, statistic-based detectors, and machine learning detectors. These map to Hoang et al.'s three categorizations [10] excepting that we refer to Hoang et al.'s statistical detectors as watermark detectors, and Hoang et al.'s data-mining based detectors as statistical detectors. Watermark detectors are also sometimes called tripwire detectors [51]. Anomaly detectors vary as a trade-off between false negative rate, false positive rate, and explainability.

Explainability is the measure of how easy it is to determine why a classifier made a decision. It is valuable for system verification, legislative compliance, and human learning [52]. In an anomaly detection context, explainability is valuable for aggregation, in that it makes it easier to justify reporting an anomaly, and post-mortem analysis.

Watermark detectors (also called tripwire detectors) check that a single property maintains a known value or range of values. These properties range from the source address of a network packet, to the hash of an executable file, or the current consumption of a motor. Watermark detectors are often trivial to implement and have a low false positive rate. Although watermark detectors are usually limited to detecting the simplest of anomalies, occasionally they will detect more complex anomalies. Watermark detectors have high explainability; when they detect an anomaly, it is often intuitive why it occurred. For example there was a network packet from an unexpected network source, the executable file did not have the expected contents, or the motor was consuming more power than normal.

Statistic-based detectors evaluate a well-defined statistic over an input window and report an anomaly if the statistic exceeds a given threshold. This statistic may be checking the variance of a window in the source data stream, its slope over an extended period of time, or the time between network packets. Some statistic-based detectors will maintain some additional state to adjust $P(anomaly)$ depending on the state of previous windows. They have moderate to high explainability. When an anomaly occurs, there is a reasonable explanation of why it occurred. Examples include: the variance of the water flow rate was too high, or network packets are arriving with increasingly larger latencies.

Machine learning detectors can be categorized further into library-based and regression-based detectors. Library-based machine learning detectors maintain a collection of patterns that match normal behavior. *Motif* detectors [53, 54] are an example of library-based methods. Regression-based detectors attempt to predict or reconstruct a sample given the system is operating normally, and then compare their prediction to the true sample value. Other machine learning detectors may attempt to predict $P(anomaly)$ directly. Machine learning detectors require significant training. They are typically based on artificial neural networks, and therefore have low explainability. A common approach is to generate an intermediate transformation as the output of the machine learning algorithm, and have a more straightforward statistic produce $P(anomaly)$ from the result of that transform. This can increase the explainability of a machine learning detector. Other improvements are being made in the area of explainable deep learning, with several new techniques showing promise [55].

Different detection algorithms place varying demands on the size of input windows they handle. A simple feed-forward neural network will have a fixed size input vector, and even a recurrent neural network must select a size of input vector, even if it will advance it across samples in a continuous signal. Varying this window size will affect the results of a machine learning detector. Statistic detectors must also make a careful choice of window size. In the spike detector outlined below (see Section 4.2) we measure a spike based on the standard deviation of the signal within the current window. In all these cases the selection of window size effects the result of the detector; some choices of window size will be better than others for different detectors.

As a consequence of different detectors performing better at different window sizes, a single window cannot be attributed as the source of an anomaly detected by multiple detectors running on the same data stream. For more complex anomalies, and for symptoms on event series anomalies such as frequency change (Section 3.2.1) there may not be a fixed time at which the anomaly occurs, but any detectors that report such an anomaly must still select a window (or multiple windows) to report $P(anomaly) > 0$. These factors combined mean that the times at which multiple anomaly detectors report symptoms of the same anomaly cannot be relied on to match the time of the underlying anomaly. This adds uncertainty to the times of anomalies reported to the aggregator. For more information on the consequences of anomaly timing uncertainty see Section 5.

4.1 Watermark Detector Example: Hash Detector

A hash detector is a watermark detector for monitoring the content of files or network packets. Tripwire is a popular hash detector used for monitoring file systems [51]. When checking network packets, hash detectors detect unexpected event anomalies (see Section 3.2.2). A hash detector is aimed at finding simple intrusion cases where the attacker replaces system components with their own programs or naively attempts to alter network packets with a fixed payload, such as a system status report.

To train a hash detector for some expected input string f, the hash detector computes *SHA-512*(f) [56] and stores it, along with the information that identifies what string f'

should match f. For files, this is typically the location of the file on the target filesystem, along with remote connection information if necessary. For network packets, this is usually the sender, receiver, and application-specific identifier of the packet expected to have fixed content.

In operation on a file, a hash detector polls the content of that file at some determined interval. The interval should be selected such that the detector does not overwhelm the file system with continuous reads. If filesystem change notifications (such as inotify [57]) are available on the target system, a hash detector may subscribe to them rather than polling. When working on a network packet, the hash detector waits for a packet matching the expected sender, receiver, and identifier.

Upon receipt of the contents of a file, or a network packet f', the hash detectors computes $\mathit{SHA\text{-}512}(f')$ and compares it to $\mathit{SHA\text{-}512}(f)$. If they are not equal, the hash detector reports an anomaly with $P(\mathit{anomaly}) = 1$.

Like most watermark detectors, a hash detector is not complicated. It checks for the most straightforward attacks only, and exchanges simplicity for accuracy. If the file or packet is not what was expected: there is an anomaly.

4.2 Statistic Detector Example: Spike Detector

A spike detector detects spikes according to the formula in Equation 3.1: given a time series y, a window size n, and a constant factor c, the subsequence $y_{[p+1,q]}$ is a spike iff

$$\forall y_t : p < t \leq q,\ |y_t - \bar{y}_{[p-n,p]}| > c \cdot \mathit{stdev}(y_{[p-n,p]}) \tag{3.1}$$

Here the window size selected for the spike detector is $n+q-p$, a subsequence of the trace that includes a window for computing the standard deviation and then a second window the spike. The parameters of the spike detector are then the choice of n, c, and $q-p$. See Figure 4.1 for an example of a spike detector applied to the throttle control of a model car (see Section 7 for details).

Fortunately we can choose $q - p = 1$. We do this by observing that for sufficiently large n the impact of any one sample $\hat{y}_t \in y_{[p-n,p]}$ on $\bar{y}_{[p-n,p]}$ and $\mathit{stdev}(y_{[p-n,p]})$ is small (because sample mean and standard deviation have a sample count in their denominators). Consider a sample that is an early part of a multi-sample spike. If that sample spills over into the region of the window that is used to compute $\bar{y}_{[p-n,p]}$ and $\mathit{stdev}(y_{[p-n,p]})$, it will have a negligible effect on those measures. Any spike that is small enough that choosing

$q - p = 1$ would not detect it for sufficiently small c is not a spike as defined by the choice of c. Choosing a smaller c will yield spikes as small as desired.

This leaves the choice of c and n, which can be determined during training. Training c is a matter of choosing an acceptable probability that a sample will be a spike based on the sample rate and the observed distribution of the data. A non-zero spike count on the training data is acceptable, as the aggregator makes the final verdict on if an anomaly is reported (see Section 5). Selecting c is more of a selection of spike frequency than a selection of a final anomaly threshold. A larger c will cause only the larger spikes to be detected, a smaller c will detect all the spikes a larger c would detect and more. Once c is determined, the choice of n is merely a case of trading memory for volatility. A larger n will require storing a larger buffer $y_{[p-n,p]}$, but will yield a more stable mean and standard deviation (as a larger sample size will yield a better approximation of the population mean and standard deviation).

4.3 Machine Learning Detector Example: Autoencoder

An autoencoder is a form of unsupervised representation machine learning [58]. The objective of an autoencoder is to learn to represent the input in a smaller feature space. In the case of deep learning autoencoders, this space is arbitrary and develops as a product of training the artificial neural network. There is no human-understandable mapping back from the features of this space back to real features. Autoencoders can also be viewed as a dimensionality reduction technique.

An autoencoder is composed of an encoder function $e(X) \mapsto R$ and a decoder function $d(R) \mapsto X$ [58] where R is the lower-dimension "encoded" representation of the input. The encoder is tasked with learning the mapping from the input to the encoded representation, and the decoder with converting the encoded representation back to the associated input. Together the encoder and decoder are combined into a single neural network $d(e(X)) \mapsto X$ which can then be conveniently trained on its own input. That is, the learning process minimizes the error between the original input and the reproduced input from the result of the network.

While the encoded representation has many uses, in anomaly detection we are only interested in the combined network $d(e(X)) \mapsto X$, and specifically in the reconstruction error of that network. We train the autoencoder to learn how to encode, and then decode, only on normal data. When the autoencoder encounters anomalous data, it will have difficulty reconstructing it because it was not trained on similar samples, and will yield a large reconstruction error.

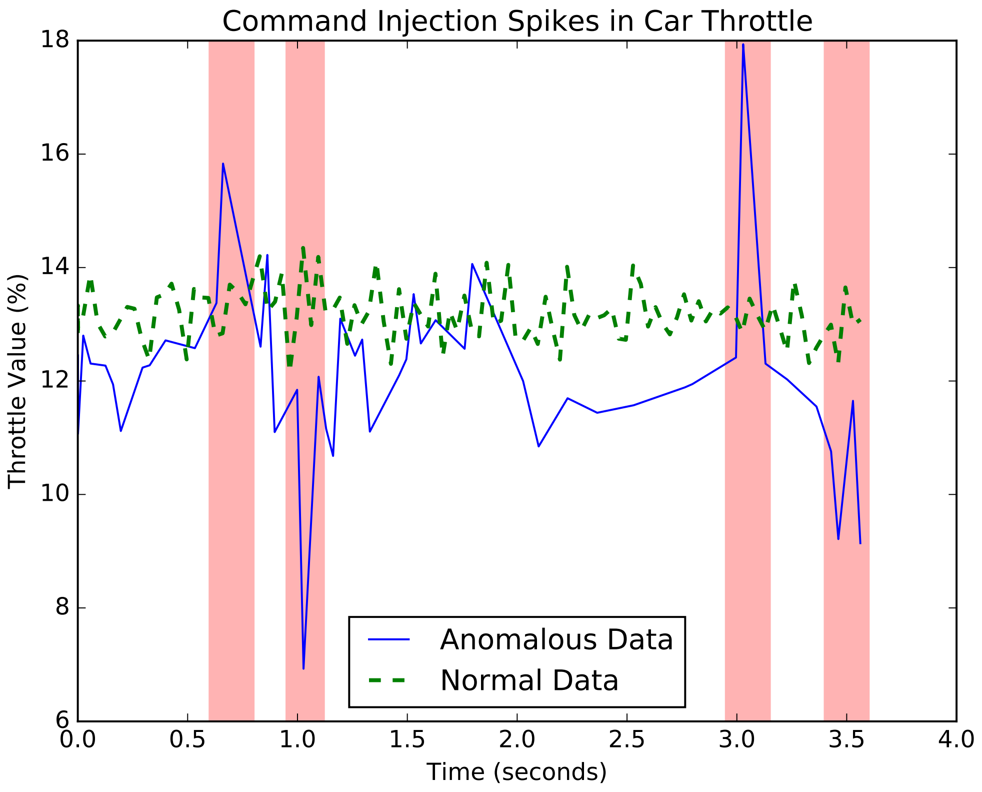


Figure 4.1: Spike Detector Example - Highlighted Regions are Spikes

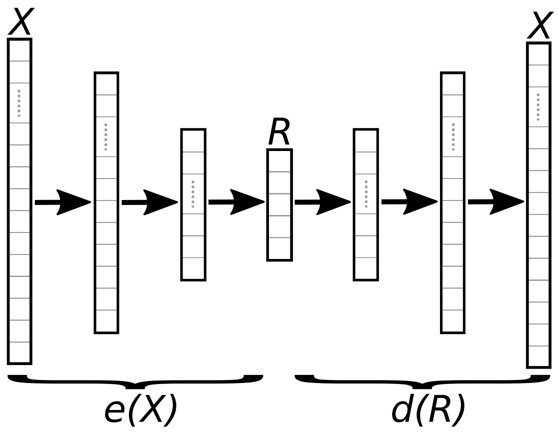


Figure 4.2: Autoencoder Neural Network Layout

Autoencoders for anomaly detection have been well explored in the last year. Chen et al. constructed an ensemble of autoencoders for anomaly detection using randomized connection dropping achieve diversity [59]. Xu et al. use autoencoders to detect anomalies in statistics from web applications [60], and Baur et al. use autoencoders for segment anomalies in brain magnetic resonance images [61].

Out implementation of autoencoders is similar to Chen et al.'s RandNet [59] excepting that we do not randomly disconnect some of the connections, as we are not constructing an ensemble of varying autoencoders specifically. In our structure (see Figure 4.2) each autoencoder begins with a fully connected layer that takes the input and reduces it to a vector of size 20. Another layer reduces that vector to another vector of size 10, and a final encoding layer takes it to the encoded representation R which is a vector of size 5. The process is then reversed for the decoder: a fully connected layer takes the hidden representation and expands it to a vector of size 10, and another layer to a vector of size 20, and then a final layer back to the same dimension as the input.

Chapter 5

Anomaly Aggregators

The job of an anomaly aggregator is to collect the anomaly probabilities output from anomaly detector nodes in the system and produce a final decision: should an anomaly be reported to the system operator. We will consider aggregators as a part of the Palisade software subsystem (see Chapter 6) and evaluate some existing ensemble learning methods as aggregators in a case study (see Chapters 7, and 8). In this section, we expand on the definition of the anomaly aggregation problem outline in Chapter 1 and provide a formal mathematical definition, followed by five example aggregators.

We consider the result of an aggregator as a fundamentally a binary decision; we do not consider reporting different levels of anomalousness. This was an intentional design choice: if the system is in an anomalous state, it is the operator's responsibility to engage relevant safeguards. We do not want to add the additional overhead of determining if the anomaly is severe enough to warrant said safeguards. Alarm fatigue occurs where the system operator grows desensitized to alarms due to an abundance of false alerts [62]. This can cause the operator to disregard alerts that indicate a true system-wide anomaly. Aggregation algorithms must be designed so that if the system reports an anomaly, the anomaly will be taken seriously. We leave a discussion of how much information to show the system operator for future work.

Anomaly aggregators can be viewed as a learning ensemble where the member classifiers are the anomaly detector nodes (see Section 4). However, here we come to a break from existing literature on ensemble learning (see Section 2); we have a heterogeneous collection of member classifiers working on different inputs and may report the same underlying anomaly at different times (as discussed in Section 4). The collection is heterogeneous because different algorithms will perform differently on different data streams from within a cyber physical system (see Section 4). This heterogeneity and the addition of a time domain adds interesting constraints to the selection of aggregation algorithms.

To clarify the discussion of anomaly aggregation algorithms, we use the following notation. An anomaly aggregation algorithm is a function

$$f(\mathcal{V}) \mapsto \{\text{Normal}, \text{Anomalous}\} \tag{5.1}$$

where $\mathcal{V}$ is a matrix of sets where set $\mathcal{V}_{ij}$ contains the results of anomaly detection algorithm i (of n total algorithms) running on input channel j (of m total channels). Each element $(p, t) \in \mathcal{V}_{ij}$ is a tuple where p is the probability $P(anomaly)$ reported by the detector for this set at time t. The result of f is a single value: either the input probabilities represent an anomaly, or they are normal behaviour. When in use in an online (see Chapter 4) anomaly detection environment f is reevaluated every time some anomaly detector produces a new tuple (p, t).

Some ensemble learning methods from the literature (such as majority voting, weighted voting, and logistic regression, see Section 2.2.3) do not explicitly include a time domain, so we apply a time quantization approach to enable these instance-based methods. Let t' be the time component of the last anomaly probability report tuple (p, t) in any $\mathcal{V}_{ij}$. Let t_{now} be the present system time. For some time quantum q (which is typically selected as some fraction of a second) create a new matrix $\mathcal{V}^q$ with dimensions $n \times m \times \lceil (t_{now} - t')/q \rceil$. Each entry $\mathcal{V}^q_{ijk}$ in this matrix is the the probability entry p from a tuple (p, t) in $\mathcal{V}_{ij}$ where $t_{now} - qk \geq t > t_{now} - qk - q$. This makes the submatrix at $k = 0$ (called $\mathcal{V}^q_{ij0}$) the most recent quanta within $\mathcal{V}^q$. If there is more than one such tuple, it is the average of all p from tuples satisfying the condition. If there are no such tuples, $\mathcal{V}^q_{ijk}$ is zero (in Section 4 we discuss that no output is assumed to be $P(anomaly) = 0$). See Figure 5.1 for a visual example of this quantization.

If an anomaly detector reports multiple anomalies within a single quanta, we select the average probability between all such reports. $\mathcal{V}^q_{ijk}$ represents the results of a single anomaly detector i working on channel j during time quanta k. We are treating the anomaly detectors contributing to the aggregators as black-box functions matching the definition of an anomaly detector as defined in Chapter 4. If there are more than one report from such a detector within a quanta, the aggregator has no additional information to choose between those reports. It cannot tell if the first report was more representative of the cyber-physical system state than the second, or if the second was more representative than the first (and so on for more than two reports). An average is a straightforward method of balancing these possibilities when no other information is available. Future work may wish to consider different combination methods, such as selecting the first or last value in the quanta.

We limit the size of $\mathcal{V}^q$ (and $\mathcal{V}$) by selecting a maximum number of quanta r to maintain history for. After r quanta have elapsed, we drop the oldest quanta from the end of $\mathcal{V}^q$, keeping the matrix a constant size. By limiting the number of quanta kept in $\mathcal{V}^q$ we fix the space complexity of $\mathcal{V}^q$ at $O\big(nmr\big)$.

We must also decide when to reevaluate $f(\mathcal{V}^q)$. There are two choices: reevaluate $f(\mathcal{V}^q)$ as soon as some detector reports a new tuple (p,t) or wait one quantum q between evaluations regardless of how many new tuples (p,t) arrive during that time. While the first approach covers all the possible version of $\mathcal{V}^q$ during a quantum (as each new tuple may update t_{now} and cause a full reconstruction of $\mathcal{V}^q$), the second approach is the one we adopt for this experiment. The second approach is made more straightforward than the first approach by the ability to append a new two-dimensional matrix onto $\mathcal{V}^q$ for that quantum without having to reconstruct all the other two-dimensional matrices for all past quanta (as t_{now} has changed by exactly q).

Maintaining $\mathcal{V}^q$ only costs $O\big(nm\big)$ per quanta elapsed, and constant time per anomaly reported by any anomaly detector. Updating an average in one cell of $\mathcal{V}^q$ takes constant time (so long as the number of entries in each cell is maintained, which adds only constant space complexity). Updating $\mathcal{V}^q$ for a new time quanta takes $O\big(nm\big)$ time, as the submatrix at $k = r$ used for the oldest quanta can be filled with zeros and reused as the newest quanta if $\mathcal{V}^q$ is stored as a linked-list of quanta (which adds only a size r overhead for linked-list pointers, not enough to change the space complexity of $\mathcal{V}^q$). We can store $\mathcal{V}^q$ as a linked list of quanta because we wait one quanta every time we reevaluate $f(\mathcal{V}^q)$ (see above).

5.1 Simple Majority Vote Aggregator

We begin with a majority vote aggregator as described by Polikar [11]. The majority vote aggregator uses $\mathcal{V}^q$ as described above. It considers only one two-dimensional sub-matrix of $\mathcal{V}^q$, specifically the sub-matrix where $k = 0$, which will be the results from detectors in the most recent quanta. For each element, we consider it a vote for an anomaly if $\mathcal{V}^q_{ij0} \geq 0.5$ and a vote against an anomaly otherwise. We construct the total anomaly vote X as:

$$X = \sum_{ij \in \mathcal{V}^q} \begin{cases} 1 & \text{if } \mathcal{V}^q_{ij0} \geq 0.5 \\ -1 & \text{otherwise} \end{cases} \tag{5.2}$$

For some threshold T, if $X > T$ then we report (that is, $f(\mathcal{V}^q)$ produces) *Anomalous*, otherwise we report *Normal*. Training the simple majority vote aggregator is a matter of

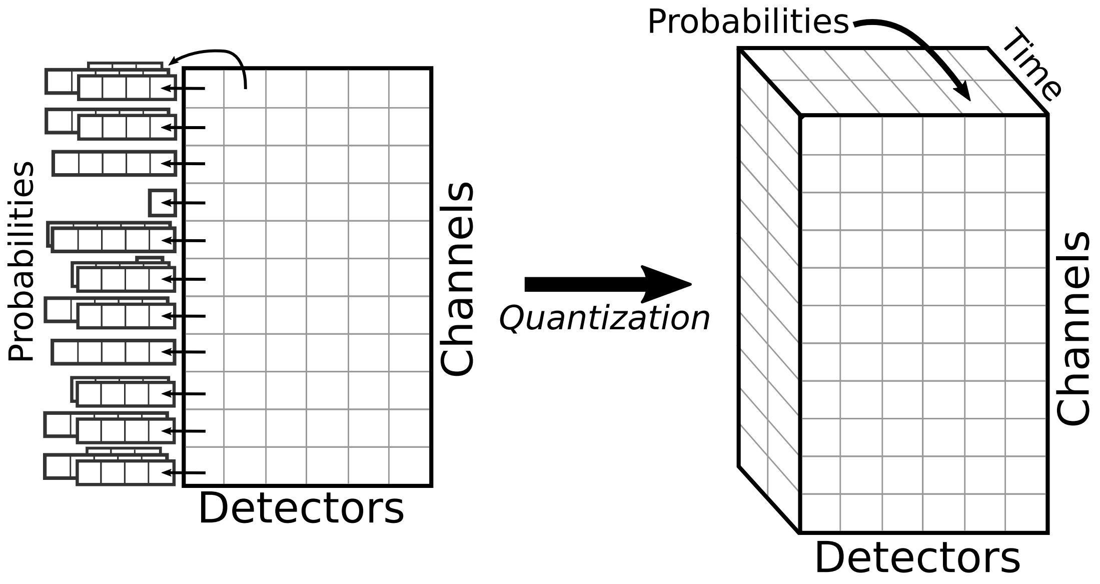


Figure 5.1: Quantization of Aggregator Input in Time Domain

voting on each instance of the training set and selecting T to be the threshold at which no non-anomalous instances are classified as anomalous (the zero-false-positive threshold).

The simple majority vote aggregator is not intended to be accurate (see the results in Section 8), but instead to provide a baseline against which to judge the other aggregation algorithms. Any more sophisticated algorithm should perform better than a simple majority vote that considers no history.

5.2 Weighted Majority Vote Aggregator

The weighted majority vote aggregator works similarly to the simple majority vote aggregator except that a detector's vote is weighted by its accuracy across all channels. This is based on Polikar's description of weighted majority voting [11] were we select the weights to be the accuracy of the detectors on the training set. We use the standard statistical definition of accuracy for an anomaly detector D_i:

$$Acc(D_i) = \frac{\text{true positives} + \text{true negatives}}{\text{true positives} + \text{true negatives} + \text{false positives} + \text{false negatives}} \tag{5.3}$$

We again use only the most recent sub-matrix of $\mathcal{V}^q$ where $k = 0$, and a vote for an anomaly is counted if $\mathcal{V}^q_{ij0} \geq 0.5$, otherwise a vote against and anomaly is counted. We can then construct the total anomaly vote X as:

$$X = \sum_{j \in \mathcal{V}^q} \sum_{D_i} \begin{cases} Acc(D_i) & \text{if } \mathcal{V}^q_{ij0} \geq 0.5 \\ -Acc(D_i) & \text{otherwise} \end{cases} \tag{5.4}$$

Again like the simple majority vote aggregator, we consider some threshold T. If $X > T$ then we report *Anomalous*, otherwise we report *Normal*. To train the weighted majority vote detector, we compute X on each instance of the training set and select T to be the zero-false-positive threshold.

5.3 Logistic Regression Aggregator

Ho et. al. suggest a method of combining decisions from multiple classifiers using logistic regression [23]. We implement their technique here for a binary classification of the sub-matrix of $\mathcal{V}^q$ where $k = 0$ (as used in the voting schemes above). Ho et al. define Y_c as 1 if the c is the correct predicted class for the ensemble on sample Y, and $Y_c = 0$ if it is not. From here the let $P(Y_c = 1|X_c) = \pi(X_c)$ where X_c is the vector of ranks given to class c by the member classifies. If a classifier i ranks class c as the most likely class for a sample, it will have the highest rank value. In binary classification this means $X_{ci} = 1$ if classifier i believes c is the correct class, and $X_{ci} = 0$ otherwise. Now we can define $\pi(X_c)$. Ho et al. let the probability that the ensemble will predict the correct class given the outputs of the member classifiers (here as $\mathcal{V}^q_{ij0}$ to match our notation) be:

$$\pi(\mathcal{V}^q_{ij0}) = \frac{\exp\left(\alpha + \beta \mathcal{V}^q_{ij0}\right)}{1 + \exp\left(\alpha + \beta \mathcal{V}^q_{ij0}\right)} \tag{5.5}$$

where α is a scalar and β is a weight vector (see below) that can be determined by logistic regression. For binary classification, the values of α and β are determined for only one of the classes (in this case *Anomalous*). If the resulting probability $P(anomaly) > 0.5$ then we predict *Anomalous*; otherwise we predict *Normal*.

The logistic regression aggregator is trained by splitting out each two-dimensional sub-matrix varying on k from $\mathcal{V}^q$ in the training set, then flattening this matrix into a vector (so that when multiplied by β it produces a scalar). The values $\mathcal{V}^q_{ijk}$ are bucketed into two buckets: if $\mathcal{V}^q_{ijk} > 0.5$ it is bucketed to 1, otherwise it is bucketed to zero. All the flattened sub-matrices (now vectors) so bucketed are fed as input to a logistic regressor (in this case LIBLINEAR [63]). The resulting α and β are used during testing to produce the probability of the ensemble predicting an anomaly. For some user selected threshold T if the result of the logistic regression $\pi(\mathcal{V}^q_{ij0}) > T$ then we report Anomalous; otherwise, we report Normal. As in voting, T is selected to be the zero-false-positive threshold.

5.4 Probability Sum Aggregator

In Section 4 we conclude that anomaly detectors may report anomalies at a different time than the anomaly actually occurred. However, detectors do report the probability $P(anomaly)$ that they believe an anomaly has occurred at the time they report it. Here we must make an assumption: the time-domain error of a reported anomaly is normally distributed around the time the underlying anomaly occurred.

There are two primary arguments against the normal distribution of time-domain errors: you cannot detect an anomaly before it occurs so the distribution of the time domain error cannot be symmetric, and anomalies are not instantaneous events, and so there is no fixed point against which to measure the time-domain error.

The first argument is correct in that anomalies cannot be detected before they occur. However, due to windowing, a detector may report an anomaly on a window with an underlying anomaly near its end. Some detectors may not be able to ascertain where in the considered window the anomaly occurred, and many detectors will default to the time of the beginning or the middle of the window. This leaves a significant gap between when the anomaly is reported and when it occurs and provides a reasonable basis for selecting a symmetric anomaly time-domain error distribution.

The second argument also has merit; anomalies are not instantaneous events. However, if the underlying anomaly has a start and end time, some detectors will not be able to ascertain when the anomaly started or ended. Consider a watermark detector working on a slowly leaking tank: the leak (underlying anomaly) started long before the water fell below the threshold checked by the detector. A detector looking at the tank pressure may have detected the leak earlier, and one looking at the tank temperature may detect it later. Since the anomaly aggregator does not know underlying physical properties of the anomaly, assuming the time within the anomaly when the report occurs to be normal is a fair assumption.

If we assume the time domain error is normally distributed, then we can sum the probabilities from different detectors as if they were probabilities for the same event: that there is a true anomaly to report. The probability sum aggregator considers all tuples (p, t) in $\mathcal{V}$ equally, regardless of the source detector. To determine if there is an anomaly at time t_{now}, for some sensitivity factor σ the probability sum aggregator evaluates:

$$\mathcal{S} = \sum_{ijk \in \mathcal{V}} \left[\frac{\mathcal{V}_{ijk}[p]}{\sigma\sqrt{2\pi}} e^{-\left(t_{now} - \mathcal{V}_{ijk}[t]\right)^2 / 2\sigma^2}\right] \tag{5.6}$$

and reports an anomaly if $\mathcal{S}$ is greater than or equal to some threshold T selected during training. This is just a sum of the normal probability density function for reported anomaly probability tuple, shortened vertically depending on the probability reported by the source detector (the factor $\mathcal{V}_{ijk}[p]$ in the numerator). The sensitivity factor σ is the standard deviation of the normal distribution that we assume the error in the anomaly time prediction to take. This is a hyperparameter to be selected before training.

The probability sum aggregator is trained by evaluating it on the training data and observing the maximum value of $\mathcal{S}$ over the normal training set and the minimum value of T over the anomalous training set. T can then be selected depending on the desired level of sensitivity as opposed to the number of false positives. Since it is desirable to have as few false positive as possible, selecting T to be equal to the lowest value of $\mathcal{S}$ for which a true anomaly occurs in the training set is a reasonable choice. If training only on normal data, select $\mathcal{S}$ to be slightly larger than the largest value of $\mathcal{S}$ for which no true anomaly is present in the training set (the zero-false-positive threshold).

5.5 Stacked Generalization

Wolpert describes stacked generalization as an approach for ensemble learning where a higher-level classifier is trained on the outputs of the member classifiers [20] (see Section 2.2.2). Here we implement stacked generalization without cross-validation on the inputs to the member classifiers. The primary benefit of this cross-validation is the ability to use the majority fraction (in the case of 10-fold cross-validation, this fraction is $\frac{9}{10}$) to train each of the member classifiers, and the entirety of the training set to train the higher-level classifier. Our implementation of stacked generalization assumes that the member classifiers have already been trained on a different dataset, and thus the cross-validation is unnecessary as we can already use the entire training set on the higher-level classifier.

We implement the higher-level classifier as a fully connected feed-forward neural network (see Figure 5.2). Neural networks are well explored for anomaly detection [64, 65, 66, 67] and are considered viable, so it is fitting to examine them here in our comparison

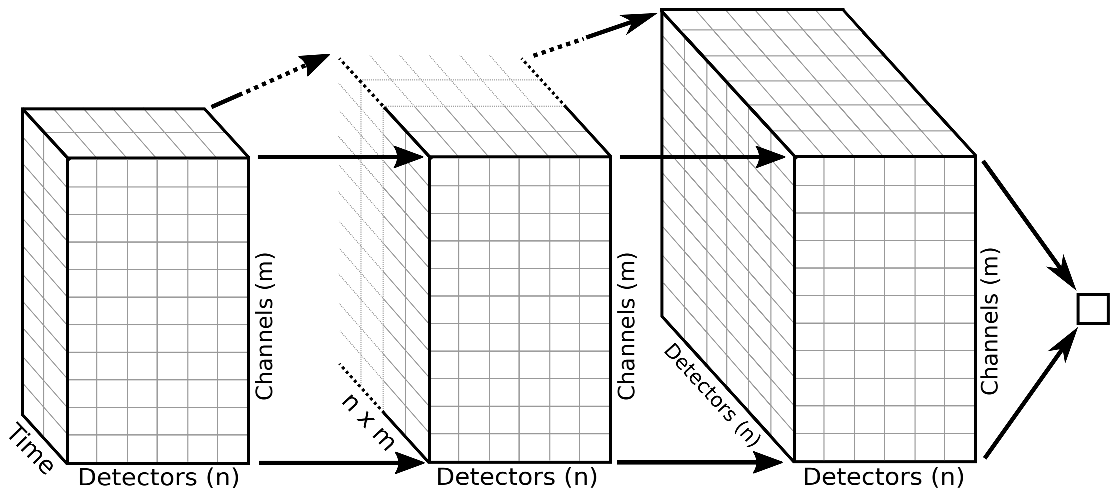


Figure 5.2: Stacked Generalization Neural Network Layout

of aggregators. The network takes the most recent r quanta from $\mathcal{V}^q$ to add additional historical context to the classifications of the algorithm. While Wolpert's method did not consider classifications of member classifiers that vary in the time domain [20], we feel it is a reasonable augmentation to the approach.

The network consists of three fully connected layers that reduce the input from the sub-matrix of $\mathcal{V}^q$ with shape $n \times m \times r$ (recall r is the number of quanta kept in $\mathcal{V}^q$) down to a single output. The first layer increases the size of the hidden representation to the shape $n \times m \times nm$ (because in general $nm >> r$, and Sietsma and Dow conclude that small first hidden layers produce poor generalization performance [68]). Then we reduce the shape to $n \times m \times n$ and then down to 1. The output layer uses sigmoid activation [69], and all the other layers use rectified linear activation.

Our stacked generalization method is trained on the training set using a traditional backpropagation approach. Once the network is trained we select a threshold T to be the zero-false-positive threshold of the aggregator as run on the training set. We report *Anomalous* in testing if the output of the final layer of the neural network is larger than T; otherwise, we produce *Normal*. See Chapter 8 for the results of this implementation of neural network stacked generalization.

Chapter 6

Palisade: Anomaly Detection for Cyber-Physical Systems

Palisade is a complex event processing framework built for anomaly detection in cyber-physical systems. It is divided into software and hardware subsystems. The hardware components collect diagnostics from the target cyber-physical system and pass them to the software components, which perform the anomaly detection and inform the system operators of a verdict.

6.1 Palisade Requirements

The construction of Palisade was motivated by the desire to have a lightweight, efficient, and modular system for anomaly detection in cyber-physical systems. To accomplish this aim, we outline a set of requirements that drive the design of Palisade.

Requirement 1: Palisade shall scale to any size of target cyber-physical system. While presently Palisade has been installed on motor vehicles, we aim to install Palisade on larger and more complex target systems in the future.

Requirement 2: Anomaly detectors in Palisade should support being distributed over a network. To go with Requirement 1, we must be able to scale the computational capability of Palisade to accommodate arbitrary computational requirements from large cyber-physical systems.

Requirement 3: Multiple anomaly detectors shall be able to run on the same data.

Many detection algorithms in Palisade (see Section 4) may work with multiple data streams. Thus, detectors must be able to share data stream for maximum effect.

Requirement 4: Palisade shall be able to collect side-channel data at a sufficiently high sampling rate. The accuracy of anomaly detection algorithms on side-channel data scales with the sampling rate of that data (to a point [70]).

6.2 Palisade Terminology

For the discussion of Palisade, we define actors in the system by specific names defined herein.

The *target system* is the cyber-physical system that Palisade is set up with. This means the Palisade hardware is installed locally along with the components of the target system, and not operating remotely.

The *system operator* is a human in charge of operating or maintaining the target system. This would be the driver/pilot for a motor vehicle or airplane. When Palisade generates an alert, it is the system operator's responsibility to respond to the alert and undertake whichever safeguards are necessary. Palisade (as presented here) is strictly an anomaly detection framework; it does not attempt to counteract malicious activity, only detect it.

A *control unit* is a microprocessor that controls a component of the target system. For a car, examples of control units include the engine control unit (ECU), the infotainment controller, the Anti-lock Braking System (ABS) controller, and many others.

We define a *sensor* as a device that measures a physical property at a fixed point within the target system. Examples of such properties include vibration, temperature, and electric current. At present, the hardware implementation of Palisade uses five types of sensors: microphones, accelerometers, gyroscopes, temperature sensors, and current sensors.

We define a *probe* as a sensor that measures a digital signal travelling on a bus. Examples of such buses might include: a Controller Area Network (CAN), a Serial Peripheral Interface (SPI), or an Inter-Integrated Circuit (I^2C) bus. Palisade currently only has probe that monitors activity on a CAN bus.

A *sensor suite* is a set of sensors, one of each type Palisade includes, that measure the characteristic of one component of the target system. A single sensor suite in Palisade consists of a microphone, an accelerometer, a gyroscope, a temperature sensor, a current sensor, and a CAN bus probe.

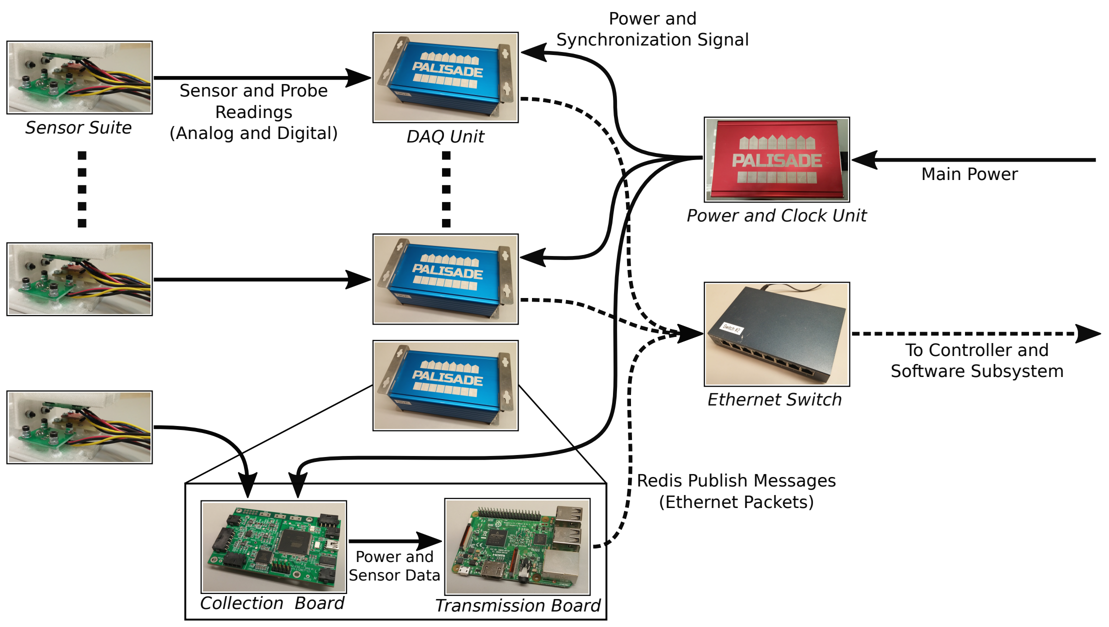


Figure 6.1: Palisade Hardware Subsystem

6.3 Palisade Structure

The hardware subsystem of Palisade (see Figure 6.1) is composed of separate Data Acquisition Units (DAQs). Each unit records data from an attached sensor suite. Each unit instruments a subsystem of the target system. In practical terms, this means that the sensor suit instruments a single device and the control unit for this device. For example, consider the engine control unit in a car. The accelerometer, gyroscope, and temperature sensor are mounted directly to the engine block, the microphone is affixed to the frame near the exhaust, and the current sensor is connected to the power supply of the engine control unit. Together this sensor suite is connected to a single DAQ. In a full installation of Palisade, there are multiple DAQs, each instrumenting a separate subsystem of the target system. Continuing with the car analogy: there would be a DAQ for the engine, transmission, infotainment, ABS, alternator/battery, and possibly more.

Each of these DAQ units is connected to the central *controller*. There is only ever one controller, which adjudicates between the hardware and the software subsystems of Palisade. The controller runs Redis [6], (REmote DIctionary Server) as a publish-subscribe broker. For a discussion on the choice of publish-subscribe broker, see Dunne 2018 [71].

Redis maintains a list of *channels*, to which packets of data can be sent over a network socket. Components of the Palisade software subsystem can subscribe to channels and will receive any data packets sent to the requested channels. Each DAQ publishes data to Redis on the controller, and the software components subscribe to those streams.

The software subsystem of Palisade is composed of *nodes*. Nodes subscribe to a set of channels in Redis and publish to another set. Together, they form a complex event processing system [4]. There are three types of nodes: source, processor, and sink. Source nodes stream input data on a channel into Redis. Processing nodes include pre-processors, that consume a channel and produce a new channel after applying a transformation, and anomaly detector nodes (herein referred to as detectors) which consume a channel and produce a verdict on if the data on the channel is currently anomalous. Sink nodes consume a channel, which is often the result of a detector, and either aggregate results into a final anomalous verdict on the entire target system, or simply display the results to the operator in a graphical user interface.

6.3.1 Data Acquisition Units

Data Acquisition Units (DAQs) are the core building block of the hardware subsystem of Palisade. They aggregate the results from a sensor suite and pass them on to the controller. Each DAQ is composed of two primary components: an *collection board*, and a *transmission board*. The collection board collects data from the sensor suite and passes it to the transmission board, which formats the data and sends it to the controller on a Redis channel (see Figure 6.1).

Collection Boards

A collection board is a custom printed circuit board (PCB) build around an Atmel UC3A3256 microprocessor [72]. We split the sensors into two large groups: analog sensors and digital sensors. The microphone, temperature sensor, and current sensor are analog sensors. They are connected to an Analog to Digital Converter (ADC) on the collection board which forwards the digital form of the signal from these sensors to the Atmel microprocessor. The accelerometer, gyroscope, and CAN bus probe are digital sensors and are connected directly to the Atmel microcontroller over I^2C for the accelerometer and gyroscope, and SPI for the CAN bus probe.

We sample the current sensor using a two channel bandwidth reduction method proposed by Moreno and Fischmeister [73] with permission. Consider a window of the current

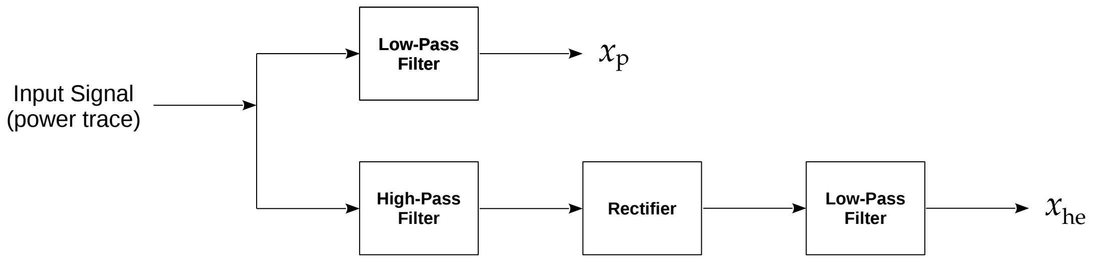


Figure 6.2: Current Sensor Bandwidth Reduction Method

consumption of a control unit as x. Moreno and Fischmeister's method runs a low-pass filter and a high-pass filter on x (see Figure 6.2). The result of the low pass filter x_p forms the first channel of power recorded by the ADC on a DAQ unit. This can be sampled at some sample rate r. The result of the high-pass filter is then passed through a rectifier, essentially removing local periodicity from the signal and reducing it to an envelope. This envelope is passed through a low-pass filter and produces x_{he}, the second channel of power recorded by the ADC. The innovation here is that x_{he} can also be recorded at sample rate r, but it contains information about the magnitude of the high-frequency activity in x. x_{he} would not have sufficient sample rate to record that activity, but it does have sufficient sample rate to record the envelope of that activity. We use this two channel approach to sample current consumption from the control units of the target system.

We choose to sample the current sensor channels at 200 kHz, the microphone at 50 kHz, and the remaining channels at 1 kHz. The CAN probe can receive two messages at a time when it is polled, so it could be considered sampled at 2 kHz. Due to the decision to have a DAQ unit for each of the control units in the target system, for anomaly aggregation to remaining consistent (specifically the construction of $\mathcal{V}$ and $\mathcal{V}^q$, see Section 5), we need samples from channels recorded by each unit to occur at the same time. This means that a detector that is examining a sample A_t from time t on channel A will be observing a sample collected at the exact same time (in as much as is possible) that a detector observing a sample B_t from time t on channel B. Ensuring A_t and B_t are collected at the same time, where A and B are channels originating from separate DAQ units, is the primary challenge we faced in the design of the Palisade hardware subsystem.

We solve this synchronization problem with a *synchronization signal*, which is a digital clock signal that shared among all the DAQ units. We chose a shared hardware clock because it can be used to make a hardware trigger for analog to digital converters to sample, and therefore guarantee the samples from different DAQ units are taken at the

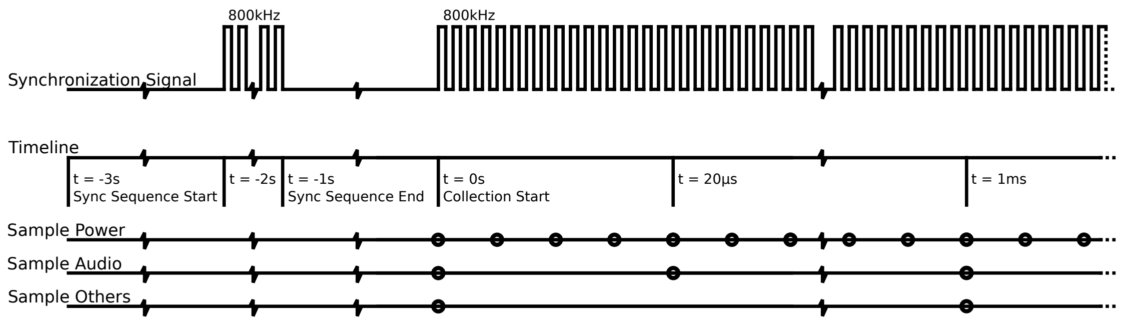


Figure 6.3: DAQ Synchronization Signal

same time. A disadvantage of this technique is that it requires an additional wire to carry the signal to all the DAQ units.

The synchronization signal is generated at the *power and clock unit*, a separate hub of the Palisade hardware subsystem. The power and clock unit is tasked with supplying constant DC power to all the DAQ units and generating the synchronization signal. The synchronization signal dictates when the collection board polls the sensors for samples (see Figure 6.3). Due to the internal limits of the hardware clocking in the UC3A3 microcontroller on the collection board and the ADC, the synchronization signal must be supplied at least four times the rate we wish to sample the highest rate sample (the current sensor). Thus we generate a synchronization signal at 800 kHz. Since each unit shares this signal and uses it as input to a hardware timer, they all will poll their sensors at the same time.

Unfortunately, a simple square wave clock is insufficient for complete synchronization. While it can ensure the sensors are polled at the same time, the separate collection units do not know at which time a given sample was polled (it knows that it was a multiple of 5 μs, because we are polling at 200 kHz). Thus the units need a method of keeping track of exactly which sample they are collecting.

Since the power and clock unit supplies power to all the DAQ units, it knows when the collection boards in the DAQs are beginning their power-up sequence. The power and clock unit contains a small 8-bit microcontroller (the Atmel ATtiny25 [74]) that powers up at roughly the same time as the collection boards and generates the synchronization signal. However, due to the initialization process on the collection boards taking a variable amount of time depending on attached sensors, relying solely on power-up time to synchronize is insufficient.

To remedy this we add a startup sequence (called the *sync sequence*) to the beginning of the synchronization signal that occurs immediately on power-up (see Figure 6.3). The sync sequence begins by holding a value of low on the synchronization signal for one second, this allows for the variable startup time of the collection boards. Once the collection boards are initialized, they wait for the synchronization signal to begin oscillating at 800 kHz, which it does for another second. Then the synchronization signal goes low again, and the collection boards initialize their hardware timers, so when the synchronization signal begins oscillating at 800 kHz indefinitely they all count time $t = 0$ exactly at the same time.

The reason the sync sequence has two periods of no oscillation is so that collection boards that are powered on late can distinguish between a disconnected clock signal and the startup sequence. The second period of silence also allows collection boards to differentiate between starting late and seeing the permanent oscillation only, and starting at the same time as the other DAQ units (and therefore seeing the period of silence). If a collection board detects it has started later than the other units, it reports this error to the controller, which forwards the fault directly to the system operator. This indicates a fault in the Palisade system and is separate from detecting anomalies.

The downside to having a startup sequence of such a duration is that there is a minimum of three seconds before the Palisade system can begin collecting data from the target system. This can be mitigated by the system operator powering on Palisade before starting up the target system. Future work on Palisade may wish to look at a system that encodes the current time as a digital value in the synchronization signal (not supported by the current Palisade hardware), as this would remove the need for the startup sequence.

Once every millisecond, the collection board collects all the samples polled during that millisecond and sends them in Universal Serial Bus (USB) packets to the transmission board.

Transmission Boards

A transmission board is a Raspberry Pi 3 Model B [75], a small single-board computer running Linux (specifically Raspbian, a Debian Linux derivative [76]). The transmission board receives power from the collection board and is also connected to the collection board over USB.

The collection board listens on for USB packets from the collection board once a millisecond. Those packets contain exactly one millisecond of sensor readings from the attached sensor suite (see Figure 6.1) along with the exact timestamp of the millisecond in which that data was collected (relative to time $t = 0$, see Figure 6.3). The collection

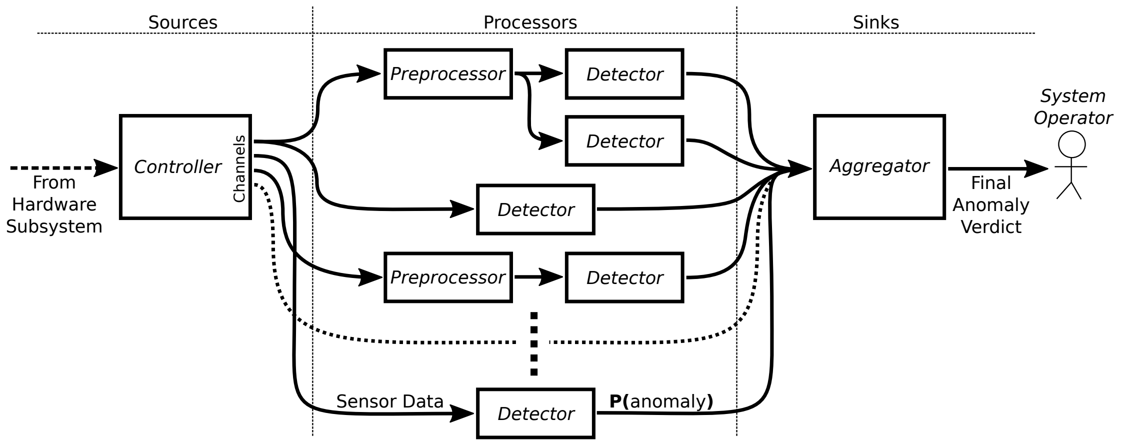


Figure 6.4: Palisade Software Subsystem

unit separates this data into individual channels, one for each sensor, and sends a publish command containing this data and its timestamp to the Redis server running on the controller.

We separate the collection board from the transmission board to the unreliability of communications over IP networks. The collection board has strict timing requirements to maintain, and dispatching data over high-speed USB has very low latency when you have complete control over the receiver and sufficient buffer space allocated. This separation of concerns allows the transmission board to maintain a buffer of several seconds of data in case of temporary network outages. Since the collection board timestamps each millisecond's worth of data, additional latency on the transmission board will not affect detector accuracy.

6.3.2 Software Subsystem

The Palisade software subsystem is composed of source nodes, processor nodes, and sink nodes (see Figure 6.4). Source nodes are responsible for providing input data for the processor nodes. Processor nodes either perform a pre-processing task and pass on data to other processors, or are anomaly detector nodes (see Section 4). The results $P(anomaly)$ from the detector nodes are forwarded to the sink nodes, which are either aggregators (see Section 5) or a graphical user interface.

Source nodes provide input to Redis. The transmission boards from the hardware subsystem of Palisade are also source nodes in the software subsystem, and the controller itself could also be considered a source. Other source nodes may read input from recorded log files and play it back into Redis, simulating a live environment with DAQ units without necessitating a full hardware setup. Source nodes may also be bridges to other systems that supply data that Palisade nodes may wish to consider. Many target systems also use a central publish-subscribe broker similar to Redis, and converting data from that system to Redis channels may provide more valuable input for detector nodes.

Processor nodes are primarily composed of anomaly detectors (see Section 4), which consume an input channel from Redis and produce a probability $P(anomaly)$ that the target channel contains an anomaly in a given window. There are also a few processor nodes that pre-process a Redis channel and republish it to Redis under a new channel name, where a detector can then consume it. Such processors perform tasks such as taking the derivative of a channel or combining multiple channels into one channel (such as combining the X, Y, and Z axes of the accelerometer or gyroscope into one magnitude value). This allows detectors to work on different inputs from the same channel without requiring any changes to the coder of the detector itself.

Sink nodes are generally composed of anomaly aggregators. At any one time in Palisade, one anomaly aggregator is operational at a time. For a breakdown of anomaly aggregators, see Section 5. Aggregators consume the output of detector nodes and produce a final anomaly verdict that can then be passed on to the system operator. Other types of sink nodes include graphical user interfaces for monitoring the statuses of individual detectors and logging nodes that keep track of the state of various detectors and data channels for postmortem analysis.

Chapter 7

Case Study

To validate the claims made about the performance of Palisade for cyber-physical systems we test the Palisade detectors and aggregators on an Advanced Driver-Assistance Systems (ADAS) demonstrator. While the data used here was not collected by the hardware components of Palisade, we still consider this a test of a representative set of the anomaly detectors and aggregators in Palisade.

7.1 ADAS Demonstrator

Donghyun Shin constructed the ADAS demonstrator as a test environment for researching anomaly detection in cyber-physical systems [35]. It consists of a treadmill that mimics an infinite highway, model cars that drive on the treadmill, and cameras that monitor the positions of those cars. See Figure 7.1 for an overview of the components we consider. The

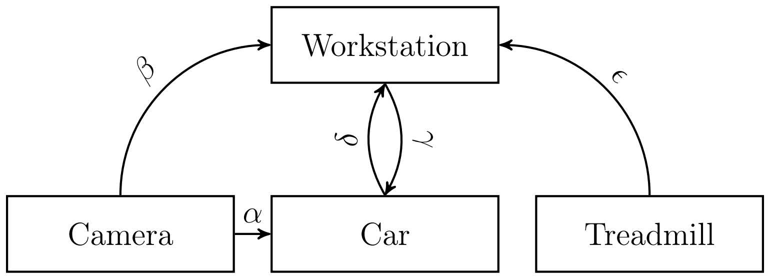


Figure 7.1: Abstracted ADAS Demonstrator Layout

software components of the system run on the Robot Operating System (ROS) [77]. Each component is encapsulated in a ROS *node* (distinct from a node in Palisade), and can use facilities provided by ROS as a publish-subscribe broker much like Redis. The ADAS platform is powerful, and we direct the reader to Shin's 2018 thesis [35] for further details on its construction.

For this case study, we focus on four subsystems within the ADAS demonstrator. The first and simplest subsystem we consider is the treadmill, which rotates in place under the cars and negates their movement, allowing them to remain in motion in their reference frame while staying stationary on the treadmill to an external observer. The treadmill has a *target velocity* (the target speed the treadmill is trying to maintain) and a *true velocity* (the actual speed as measured by a rotational encoder) that the treadmill is currently rotating.

The second subsystem we focus on is a car. While the ADAS demonstrator supports multiple cars in formation, for simplicity in this test case we consider only a single car. The car attempts to maintain its current position and velocity on the treadmill based on information from the camera and the speed of the treadmill. This scenario is called *free run*, and while the ADAS demonstrator supports more complicated scenarios, we consider only free run for simplicity. The car steers on a single axis and uses a kinematic bicycle physics model to drive [35]. It aims for a goal state and uses a Proportional-Integral-Derivative (PID) controller to maintain its position at a given goal position.

The third subsystem we focus on is the camera, which acts as a pseudo-Global Position System (GPS) within the confines of the treadmill. The camera is mounted above the treadmill and uses AprilTag [78] markers to locate the cars using image recognition techniques [35]. This location information serves as a surrogate GPS.

7.2 Experiment Setup

For simplicity, our experimental setup consists only of cases with a single car on the treadmill. This allows us to clearly show the detection and effects of anomalies on a small set of channels. We have augmented the workstation with additional instrumentation to collect CPU and memory usage. The CPU and memory usage and combine these with ROS messages in the

#	CAPEC	Name	Location	Expected Symptoms
1	148	Content Spoofing	α	Drift, Level Change
2	94	Main in the Middle Attack	β	Event Frequency Change
3	248	Command Injection	γ	Spike, S-Wave, Noise, Amplification, Level Change, Unexpected Event
4	548	Contaminate Resource	Car	Drifting, Noise, Amplification, Level Change
5	184	Software Integrity Attack	Treadmill	Event Frequency Change
6	125	Flooding	δ	Event Frequency Change
7	130	Excessive Allocation	Workstation	Drifting
8	594	Traffic Injection	ϵ	Spike, S-Wave, Drifting, Noise, Level Change
9	607	Obstruction	Camera	Loss, Periods of Silence
10	176	Configuration Manipulation	Car	Spike, S-Wave, Drifting, Noise, Clipping, Loss, Smoothing, Amplification, Level Change, Frequency Change
11	441	Malicious Logic Insertion	Car	Level Change

Table 7.1: Anomalies In ADAS Experiment

7.3 Anomaly Scenarios

Each of the scenarios described below (and in Table 7.3) is based on an attack from the CAPEC database [34]. These were selected based on existing published works demonstrating the viability of these attacks on cyber-physical systems. Using the restricted system model from Figure 7.1, we inject the described anomalies into the streams labelled therein.

7.3.1 CAPEC 148 - Content Spoofing

A content spoofing attack occurs when the adversary modifies the content from its original value without changing its source [34]. Fan et al. consider GPS spoofing to be a serious threat to cyber-physical systems [79].

For this case, we intercept the goal position messages sent to the car from the camera (channel α in Figure 7.1), as a surrogate for GPS. We alter the contents of those messages to direct the car to a different goal location. The car believes this to be the correct goal state.

7.3.2 CAPEC 94 - Man in the Middle

Man in the middle (MITM) attacks occur when an adversary inserts themselves within the communication channel between two system components [34], A and B. From there they can alter the contents of messages from A to B by pretending to be A from B's point of view, and alter messages from B to A by pretending to be B from A's point of view. Neither component knows they are not in direct communication with the other component. Conti et al. note that MITM attacks are among the most common attacks on computer systems today [80], and review over 200 papers on MITM attacks alone.

We implement a simple MITM attack between the camera and the workstation (channel β in Figure 7.1) where the adversary is doing something with the contents of the camera image. We delay the camera messages to arrive slightly slower to the workstation, simulating some additional processing or exfiltration that an actual MITM attacker might do with the image from the camera.

7.3.3 CAPEC 248 - Command Injection

An adversary conducts a command injection attack by inserting a command of their choosing into a data stream of otherwise legitimate commands [34]. This typically manifests in web applications in the form of SQL command injection [81], but Srivastava et al. discuss the use of command injection attacks to trip relays in a smart electrical grid [82].

We implement a command injection attack between the workstation and the car (channel γ in Figure 7.1). A command with a very large velocity is inserted into the stream of otherwise normal driving commands. An adversary would use such an attack to cause unintended acceleration leading to a crash.

7.3.4 CAPEC 548 - Contaminate Resource

A resource contamination attack occurs when an adversary inserts data into a data stream which that data stream should not have the appropriate permissions to handle [34]. The attacker inserts the privileged information into a communications channel that usually handles information at a lower privilege level. Giani et al. discuss various methods for exfiltrating sensitive data, including hiding information by contaminating unused fields in packet headers or network routing information [83].

We manifest a contaminate resource attack as the attacker compromising the car and passing out the battery voltage in the z-axis channel of the car's position on the track. While this information is usually available on other channels from the car, we pretend for this purpose that the battery information is privileged. The nodes that make decisions based on the car's velocity now have access to the battery state of the car, where the previously did nothing with the z-axis.

7.3.5 CAPEC 184 - Software Integrity Attack

Software integrity attacks occur when an attacker compromises the integrity of a software system component [34]. This is usually a side-effect, or an in-progress step within a more involved attack. Examples may be as simple as replacing an executable file or interpretable script file, or as involved as faking an entire network service. Sharma et al. observed cases where attackers of an open networked supercomputing environment replaced network service executable files with versions that stole authentication credentials from users attempting to log in [84].

We implement a software integrity attack by replacing the treadmill node with a new node that performs the same tasks as the treadmill node, but slightly slower (by adding an unnecessary wait). This simulates a compromised version of the treadmill node where the attacker is using the node as a remote access terminal, causing the actual functionality of the node to be delayed.

7.3.6 CAPEC 125 - Flooding

An adversary conducts a flooding attack by sending a large volume of messages to a target service intending to overwhelm it and manifest a denial of service [34]. Hahn et al. consider ethernet flooding attacks aimed at causing a denial of service as a demonstration of their cyber-security testbed [85]. Philips et al. note that flooding could be used to cause a denial of cooperation in a smart grid [86], and as a stepping stone to impersonating a node after forcing it offline by a denial of service flooding attack.

We implement a flooding attack by flooding the position channel between the car and the workstation (channel δ in Figure 7.1). This is the channel on which the car transmits

its estimated position to the workstation. By flooding this channel an attacker would aim to deny the workstation an accurate location on the car, potentially causing a crash.

7.3.7 CAPEC 130 - Excessive Allocation

An excessive allocation attack occurs when an attacker causes a system resource to allocate large amounts of unneeded resources to starve other legitimate processes on the same system [34]. Leiwo and Zheng note that an excessive allocation can cause denial of service, and provide an example of a process that duplicates itself, taking the majority of memory and CPU time [87].

We simulate and excessive allocation attack by starting a process on the workstation that progressively reserves one gigabyte of memory every second. This process is not a part of ROS, but it simulates an attacker getting some backdoor access to the workstation and attempting to cause a denial of service by reserving all the system memory.

7.3.8 CAPEC 594 - Traffic Injection

Traffic injection occurs when an attacker inserts traffic into a target network with the aim of degrading the connection and potentially modifying the contents of messages [34]. McLaughlin et al. describe the possibility of using a traffic injection attack to steal electricity from and advanced electrical metering architecture [88]. They note that traffic injection could be used to replace demand information from an electrical meter if cryptography was not correctly applied.

To cause a traffic injection attack, we insert messages about the velocity of the treadmill into the message channel between the treadmill and the workstation (channel ϵ in Figure 7.1). These messages cause a random variation in the reported speed of the treadmill, with the goal of confusing the car control logic into causing a crash.

7.3.9 CAPEC 607 - Obstruction

Obstruction occurs when an attacker blocks communication between system components [34]. This may manifest as simply as a network disconnection or physically blocking a sensor, or as complex as selectively dropping network packets. While we consider obstruction as a form of cyber-attack, it is just as possible that the obstruction may occur by accident, such as an object falling over in front of a camera. Xu et al. discuss attack and defense strategies for jamming attacks that obstruct wireless sensor networks [89]. They note that detecting jamming attacks is difficult because it can be hard to distinguish between normal and malicious sources of poor connectivity.

We simulate and obstruction attack by inserting a blocked area into the camera feed. Specifically, we modify the camera node to suppress and readings about car location within a chosen circular area. This mimics a blackout in the camera that could be caused by an attacker physically holding a blocking object in front of that area of the camera.

7.3.10 CAPEC 176 - Configuration Manipulation

CAPEC defines a configuration (or environment) manipulation attack as a scenario where an attacker alters files or settings for a target application that affect the operation of that application [34]. This is typically easier than altering the application itself, as the configuration parameters are intended to be changed by legitimate users of the application. By adjusting these settings the attacker masquerades as a user that made a mistake, rather than as a malicious actor. Lee et al. write in an analysis that in 2015 an attacker manipulated the configuration of a networked power supply such that it would fail during a wider attack on a power grid [90]. This impacted a target company's data centers, as their primary and backup power was disrupted.

We simulate a configuration manipulation attack by editing the configuration files the holds the coefficients for the PID controller that drives the car's control system. By altering these values we change how the car responds to commands, causing over-steering, excessive acceleration and deceleration, and crashes.

7.3.11 CAPEC 441 - Malicious Logic Insertion

A malicious logic insertion attack occurs when an attacker adds malicious code into an otherwise normally operating system component [34]. Typically this manifests as malware being installed on the target system, either by a malicious actor or as part of a self-propagating worm. Sridhar et al [91] and Mo et al. [92] consider malware a serious security threat to cyber-physical systems for smart electrical grids.

We implement a malicious logic insertion attack by modifying the logic in the car controller node. We change the channel to add an offset to all location related commands sent to the car, with the aim of causing the car to crash off the treadmill. While our approach is quite blunt, a more subtle malware might wait for the car to reach a high speed before offsetting the location to induce a more severe crash.

Number	CAPEC Name	Channel(s)
1	Content Spoofing	`/car/2/goal` `/car/2/pose`
2	Main in the Middle Attack	`/tracker/april_poses`
3	Command Injection	`/car/2/command`
4	Contaminate Resource	`/car/2/nav/goal`
5	Software Integrity Attack	`/treadmill/command_velocity`
6	Flooding	`/car/2/pose`
7	Excessive Allocation	`/available_memory` `/cpu_percent`
8	Traffic Injection	`/treadmill/velocity`
9	Obstruction	`/car/2/tag_pose_demuxed`
10	Configuration Manipulation	`/car/2/command` `/car/2/pose`
11	Malicious Logic Insertion	`/car/2/nav/goal`

Table 7.2: Anomaly Channels for ADAS Experiment

7.4 Channels

Each of the anomaly scenario presented above (Section 7.3) has one or more channels on which the anomaly is expected to be observed. That set of channels is not an exhaustive list of all the channels recorded from the ADAS system, however, for simplicity we consider only those channels. We call this set of possibly anomalous channels the *anomalous channel set*. This gives us a set of channels for all the anomaly scenarios where a subset should test positive for any given anomaly, but not every channel will have an anomaly for every scenario.

While we know which channels we expect the anomalies to manifest on by virtue of the experiment design, we run all the anomaly detectors on all available channels regardless. In a real-world Palisade installation we would not know which channels are anomalous; running all detectors on all channels is the most realistic simulation of the system in practice. We consider 33 channels in our anomalous channel set for this experiment, each of which is a component of the ten high-level channels covered in Table 7.4.

Chapter 8

Results and Discussion

8.1 Data

ROS provides a simple mechanism for recording messages within the ADAS system, called a ROSBAG (or just bag). For each of the anomaly scenarios detailed in Section 7.3 we induce the anomaly, then record a bag file. The moment the anomaly takes effect is not recorded. For some cases, we have multiple bag files available, for a total of 31 anomalous bag files spanning 32 minutes of data. We also take five recordings with no known anomalies present, totalling seven minutes of data. We reserve two of these recordings to train the detectors. While this is a small fraction of the total data available, the focus of this case study is on the power of the anomaly aggregators, not on the accuracy of the individual anomaly detectors. The remaining normal recordings (not used to train the detectors) are separated out into 18 samples. The 31 anomalous bag files are split into halves, for a total of 62 anomalous samples. We reserve 9 normal samples and 31 anomalous samples for aggregator training, and the remaining 9 normal samples and 31 anomalous samples for aggregator testing.

Here we run into a dichotomy: supervised anomaly detection, or unsupervised anomaly detection. This is a common problem in the training of anomaly detection systems [2]. Both options have their advantages, with unsupervised detection more applicable where the nature of anomalies is unknown, and supervised detection easier to train. Chandola et al. mention a middle ground: semi-supervised detection [2] where labels and training instances exist only for non-anomalous data.

In our case study, we adopt a semi-supervised model for the anomaly detectors and a fully supervised model for the aggregators. Anomaly detectors are trained (for those that need training) only on non-anomalous cases, but aggregation algorithms are trained with both anomalous and non-anomalous data.

8.2 Algorithm Selection

This primary objective of this experiment is to compare and contrast anomaly aggregation algorithms (as reviewed in Section 5). Therefore we select a simple, but representative set of anomaly detection algorithms. The detection algorithms themselves are not the focus of this experiment. Many of the algorithms below require a threshold to be selected where some metric exceeding the threshold will cause the detector to report an anomaly. In this thesis we do not discuss threshold selection methods (see future work in Section 9.2), so all the thresholds for anomaly detectors are selected by human observation on only the anomaly detector training data. We select the following detection algorithms for the experiment:

1. **Hash Detect** - We use the hash detector as described in Section 4.1. This runs on the executable script files and associated configuration files that make up the nodes of the ADAS system. It reports an anomaly with $P(anomaly) = 1$ if there are any changes to any of these files. We expect changes to manifest only in the Software Integrity Attack scenario and the Configuration Manipulation scenario.

2. **Spike Detect** - We use the spike detector as outlined in Section 4.2. We select a threshold of 3.0 standard deviations away from the mean to be considered a spike, and a buffer size of 40 samples (selected by human observation).

3. **Level-Change Detect** - The level change detector looks for changes in the mean of the data stream, looking for instances of the level change anomaly symptom (see Section 3.1.8). Recall Equation 3.6 reproduced here: given a time series y, an acceptable minimum level change threshold ℓ, a minimum number of samples the mean change must persist n, a level change has occurred over a window of w samples $y_{[t,t+w-1]}$ iff

$$|\bar{y}_{[t+w,t+w+n]} - \bar{y}_{[t-n-1,t-1]}| > \ell$$

   For this detector we select $w = n = 20$ samples (again selected by human observation). We select ℓ to be the largest level change observed in the training data.

4. **Range Check** - The range check detector is a simple bound checking watermark detector. We observe all the samples from the data stream in the training data

and keep track of the minimum and maximum values. Then we add a 20% margin (selected by human observation) to suppress false positives. During testing, if any value exceeds this threshold, we report an anomaly at that time with probability $P(anomaly) = 1$.

5. **Frequency Detect** - The frequency detector aims to detect Event Frequency Change anomaly symptoms (see Section 3.2.1). During training, we record the mean f_{mean} and standard deviation f_{std} of the inter-arrival times of messages on the target data stream. During testing, the frequency detector reports an anomaly if the inter-arrival time i of two messages satisfies $i < f_{mean} - f_{std}$ or $i > f_{mean} + 3f_{std}$. We arrived at these thresholds by noting that messages being delayed due to network activity is frequent, but the reverse is not nearly as common (messages that pile up during an outage still arrive at close to the expected frequency in ROS).

6. **Slope Detect** - The slope detector tries to drifting anomaly symptoms (see Section 3.1.2. It divides the input windows (which we have selected to be 50 samples long) and determines their average slope by linear least-squares regression. We then conduct a χ^2 test against the null hypothesis that the slope of the input window is zero. If the p-value of this test is less than 0.01, we report an anomaly with $P(anomaly) = 1$.

7. **Autoencoder** - We use the autoencoder as described in Section 4.3. We select an input window size of 50 samples, overlapping at 50% (selected by human observation). For each of these buffers X we run the autoencoder to produce $X' = d(e(X))$ and take the Euclidean distance Δ between the X' and X. If that exceeds the largest error observed in training, we report an anomaly with probability

$$P(anomaly) = 1 - \frac{1}{5\Delta + 1} \tag{8.1}$$

This gives us a probability that climbs rapidly to 1 as Δ increases. We chose the coefficient 5 arbitrarily. A more rigorous selection of probability function may yield better results from the autoencoder.

When selecting aggregation algorithms for this experiment we aim to select a representative sample of algorithms varying from simple voters to more complex machine learning based detectors. All the algorithms chosen for the experiment are discussed in Chapter 5. We use a simple majority vote aggregator, a weighted majority vote aggregator, a logistic regression aggregator, a probability sum aggregator, and a stack generalization based aggregator.

8.3 Methodology

The experiment proceeds in two phases: the detector phase, and the aggregator phase, both of which comprise of a training step and a testing step. In the detector phase, we begin by training the detectors in an unsupervised fashion on the training set reserved for the detectors (see Section 8.1). In this phase we also select the threshold above which each detector will report an anomaly. We choose this to be the zero-false-positive threshold (as discussed in Section 5). Then we proceed to the aggregator phase. In the aggregator phase we begin by running all the detectors on the aggregator training set, and producing the relevant $\mathcal{V}$ and $\mathcal{V}^q$ matrices based on the thresholds from the detector training phase. Since we have 7 detectors and 33 channels, we know the first two dimensions of $\mathcal{V}$ and $\mathcal{V}^q$ will be 7×33. We let q (the quantization parameter) be two seconds. For simplicity, we select t_{now} as the last time an anomaly is detected in a given sample, and quantize back from that point accordingly (for a justification of this approach see Section 5). We then train the aggregators, using the methods from Section 5, on all the $\mathcal{V}$ and $\mathcal{V}^q$ matrices.

Once training is complete, we run all the detectors on the aggregator testing set and use them to construct another set of $\mathcal{V}$ and $\mathcal{V}^q$ matrices. We also report the accuracy and F1-score of the detectors for each of the samples in the test set, and again for each quanta select during the construction of the associated $\mathcal{V}^q$ matrices for that sample. The results of this test can be seen in Table 8.4. The aggregators are then evaluated on $\mathcal{V}$ and $\mathcal{V}^q$ matrices for a range of threshold T values sufficient to generate the Receiver Operating Characteristic (ROC) curves seen in Figure 8.1.

8.4 Discussion

As can be seen in Table 8.4, the spike, slope, frequency, and autoencoder detectors all performed identically on a per-sample basis. This was because they all reported at least one anomaly in every possible testing sample. The other detectors did not perform much better. In general, such a result would necessitate an adjustment of the thresholds for each detector to be less sensitive. However, this behavior is ideal for the evaluation of our aggregation techniques, as it allows the strengths of the various detection algorithms to show. If the detectors actually reported zero false positives on the aggregator testing set, we would not have any results to share from the aggregators, as a pass-through aggregator (that reports Anomalous if any detector reports Anomalous) would be sufficient. Instead, the detector accuracies on a per-quantum basis give a much larger diversity between the strengths and weaknesses of each detector and provide a motivation for a time-based analysis.

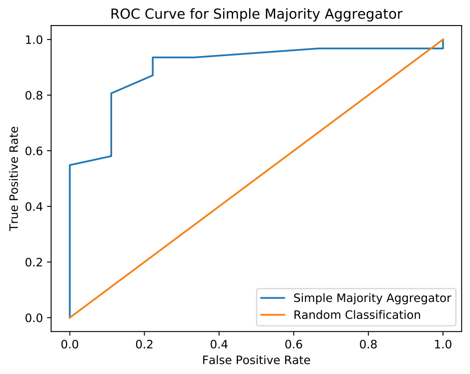
Detector	**Acc./Case**	**F1/Case**	**Acc./Quanta**	**F1/Quanta**
Hash Detect	0.350	0.278	0.255	0.280
Spike Detect	0.775	0.873	0.842	0.914
Level-Change Detect	0.800	0.886	0.672	0.784
Range Check	0.675	0.806	0.712	0.822
Slope Check	0.775	0.873	0.855	0.922
Frequency Check	0.775	0.873	0.881	0.934
Autoencoder	0.775	0.873	0.886	0.940

Table 8.1: Experimental Detector Results

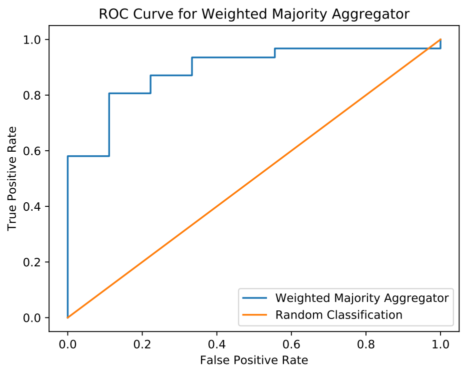
In Figure 8.1 we observe that both the voting aggregators and the probability sum aggregator provide a detection rate around 60% at a zero-false-positive rate. This is acceptable but low. The logistic regression aggregator does slightly better, 70%, but proceeds immediately up to 100% before reaching a 20% false positive rate. The stacked generalization aggregator gives 80% at a zero-false positive rate and rapidly climbs to 100% as we permit a handful of false positives.

The primary observation we make on the performance of aggregators is that voting aggregators are insufficient alone. The first three detectors (the voters and logistic regression) do not consider any history. This is a significant weakness of these approaches, and it shows in their mediocre accuracy. The probability sum aggregator is an ad-hoc approach and likely needs further refinement to produce significant results. The only method that provided a reasonable detection rate at zero false-positives was the stacked generalization aggregator. Unfortunately, it is difficult to ascertain if this was a consequence of the power of neural networks or a boon from the ability to consider the history of reported anomalies.

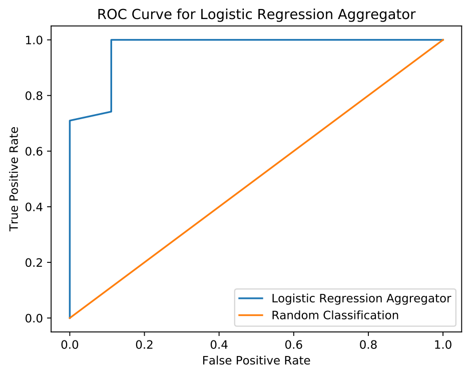
The main take-away from these results is that considering history is valuable, but there are many other factors that influence the strength of aggregation algorithms.



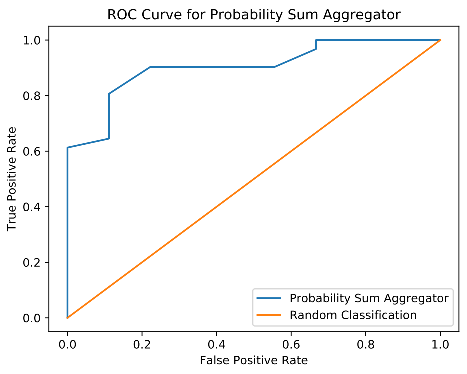
(a) Simple Majority



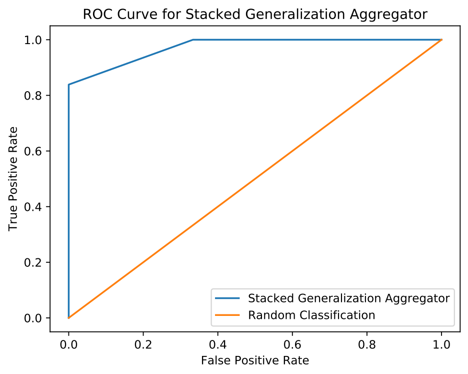
(b) Weighted Majority



(c) Logistic Regression



(d) Probability Sum



(e) Stacked Generalization

Figure 8.1: ROC Curves for Aggregators

Chapter 9

Future Work and Conclusion

9.1 Lessons Learned

During the construction of Palisade (see Chapter 6) we had to select a framework to use as a publish-subscribe middleware. For more information on the frameworks and methodology we used for making this decision see [71]. We considered four different publish-subscribe brokers: Redis, Kafka, NATS, and RabbitMQ. Redis and RabbitMQ had significantly better throughput and latency than either of Kafka or NATS. We choose Redis dues to the simplicity of the interface, but RabbitMQ is essentially an equivalent choice. This experiment produced our first lesson: selection of middleware matters.

We select sensors capable of recording data at up to 200 kHz in the case of the Palisade current sensor. A very early iteration of Palisade could only collect data from sensors at up to 1 kHz, and this lead to unusable data with errors so large it was impossible to get a coherent classification from any detectors, let alone produce a reliable final anomaly verdict. As we focused on increased sensor sampling rate as we developed Palisade into its current incarnation. The lesson here: anomaly detection performance is only as good as the input data, sensor quality and sample rate matter.

When collecting training data for anomaly detection systems, it is always easier to collect normal data than it is to collect anomalous data. Normal data can be collected by merely instrumenting the system during regular operation. Anomalous data collection requires a safe environment in which to induce the anomaly, an anomaly of a type that is known in advance (which are difficult to find, zero-day attacks are of considerable concern), and an anomaly that can be induced without causing permanent or costly damage to the target system. Inserting fake anomalies into already recorded normal data is often insufficient, as the side-effects of anomalies than manifest on side channels are often difficult to predict. Inducing symptoms (as outlined in Chapter 3) and trying to detect them is often the only possible solution. Fortunately in our case study in Chapter 7, we were able to induce many anomalies due to having full access to the system source code. This would be difficult to do for a black-box target system. The main lesson here is that finding anomalous data is difficult, so using detectors (and aggregators) that can be trained unsupervised (or only on normal data) is important.

Anomaly detection and aggregation algorithms often have low explainability. This means if Palisade is not operating as intended, it is difficult to determine what is at fault. Debugging is a difficult task in any distributed system. In a distributed system in which the results of an operation may vary because of a bug in the system, or because of an actual anomaly are even more difficult to understand. Thus good visualization tools are essential. The lesson here is that the ability to observe the underlying values in data streams, as well as the outputs of detectors and aggregators in real-time, is essential to deploying any distributed framework.

In the case study in Chapter 7 we considered 7 detectors over 33 channels. In a full Palisade installation, we may be considering dozens of detectors over hundreds of channels. The optimization problems that were simple in our case study become much more difficult in a real-world scenario. This further complicates explainability; as the dimension of the input increases, the individual contributions of data streams and detectors becomes more difficult to distinguish. The practical lesson here: select algorithms that have as much explainability as possible to increase scalability.

9.2 Future Work

In this work, we did not consider anomaly detectors that consume multiple input streams but render a single anomaly verdict. Such detectors may be statistical detectors that check some assumed physical relationship between system metrics. For example, a detector that checks that the water pressure in a pipe is correctly related to the speed of a pump feeding that pipe. These detectors pose an interesting challenge as they do not fit into anomaly matrices $\mathcal{V}$ and $\mathcal{V}^q$. Such detectors would warrant a new approach to the construction of anomaly aggregators.

There is very little literature on approaches that consider the classification history as a prior to aid in making an ensemble classification. Future work may want to investigate methods for ensemble learning from classifiers that produce multiple outputs that vary over time. Our quantization approach from Section 5 may not be the best method for considering history. Perhaps some Bayesian or Dempster-Shafer based approach that constructs a model of the component aggregators, and updates its beliefs in the presence of new samples, could be considered. Other methods might try convolution over the time domain in a deep learning based approach to recognize temporal patterns.

It may be possible to dynamically adjust the thresholds of the anomaly detectors that make up members of the aggregator ensemble. Perhaps increasing the thresholds of member detectors that report anomalies to often where the system as a whole disagrees, and decreasing the thresholds of the others may yield better system-wide results. Such an approach would have to be mindful of the effect of this adjustment on the learning models from previously trained aggregators.

In this thesis, we only considered stationary classifiers. Non-stationary classifiers complicate the aggregation problem by removing the ability for aggregators to assign member classifiers static weights. Some form of dynamic weight allocation scheme might be beneficial here. This may add significantly to the computational complexity of machine learning based aggregators, as training a neural network is much more computationally expensive than evaluating one. In additional, non-stationary member classifiers may update themselves at varying times. Some form of update coordination between different member classifiers would be needed to determine when to update the dynamic weights of the aggregator.

While anomaly detection in a practical cyber-physical system cannot tolerate false positives, it may be acceptable to raise the acceptable false positive rate for aggregators if there was an additional check before the anomaly reached the system operator. Another level of aggregator (a meta-aggregator) could perhaps enable the reduction of the false positive threshold of the (now member) aggregators, as the meta-aggregator could then have a zero-false-positive threshold set. Alternatively, a quantifiable explainability measure for the result of the aggregator might be able to inform the final anomaly verdict further.

9.3 Conclusion

In this thesis, we discussed a view of anomaly detection based on a set of formalized symptoms. We reviewed the literature on anomaly detection and ensemble learning as well as intrusion detection for cyber-physical systems. We discussed various anomaly detection algorithms and ensemble learning aggregation algorithms along with the construction of a data collection and anomaly detection framework called Palisade. We evaluated the above anomaly detection and aggregation algorithms on a dataset from a cyber-physical demonstrator system using attacks based on real-world scenarios and discussed the results.

There are two primary results of this work. The first is the assertion that considering history is valuable when designing anomaly detection systems. It is specifically useful for aggregating the results of multiple distributed detectors throughout a cyber-physical system. The second result is that the construction of a distributed anomaly detection system for cyber-physical systems is both practical and viable.

References

[1] Robert Mitchell and Ing-Ray Chen. A survey of intrusion detection techniques for cyber-physical systems. *ACM Computing Surveys (CSUR)*, 46(4):55, 2014.

[2] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)*, 41(3):15, 2009.

[3] Shikha Agrawal and Jitendra Agrawal. Survey on anomaly detection using data mining techniques. *Procedia Computer Science*, 60:708–713, 2015.

[4] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):15, 2012.

[5] Chenfeng Vincent Zhou, Christopher Leckie, and Shanika Karunasekera. Decentralized multi-dimensional alert correlation for collaborative intrusion detection. *Journal of Network and Computer Applications*, 32(5):1106–1123, 2009.

[6] Redis documentation. https://redis.io/documentation. Accessed: 2018-10-09.

[7] Bartosz Krawczyk, Leandro L Minku, João Gama, Jerzy Stefanowski, and Michał Woźniak. Ensemble learning for data stream analysis: A survey. *Information Fusion*, 37:132–156, 2017.

[8] Gerhard Widmer and Miroslav Kubat. Learning in the presence of concept drift and hidden contexts. *Machine Learning*, 23(1):69–101, 1996.

[9] Indrė Žliobaitė, Mykola Pechenizkiy, and Joao Gama. An overview of concept drift applications. In *Big Data Analysis: New Algorithms for a New Society*, pages 91–114. Springer, 2016.

[10] Xuan Dau Hoang, Jiankun Hu, and Peter Bertok. A program-based anomaly intrusion detection scheme using multiple detection engines and fuzzy inference. *Journal of Network and Computer Applications*, 32(6):1219–1228, 2009.

[11] Robi Polikar. Ensemble based systems in decision making. *IEEE Circuits and Systems Magazine*, 6(3):21–45, 2006.

[12] Robert E Schapire. The strength of weak learnability. *Machine Learning*, 5(2):197–227, 1990.

[13] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

[14] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[15] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[16] Lior Rokach. Ensemble-based classifiers. *Artificial Intelligence Review*, 33(1-2):1–39, 2010.

[17] Leo Breiman. Pasting small votes for classification in large databases and on-line. *Machine Learning*, 36(1-2):85–103, 1999.

[18] Nitesh V Chawla, Lawrence O Hall, Kevin W Bowyer, Thomas E Moore, and W Philip Kegelmeyer. Distributed pasting of small votes. In *International Workshop on Multiple Classifier Systems*, pages 52–61. Springer, 2002.

[19] Ralf Haeusler, Rahul Nair, and Daniel Kondermann. Ensemble learning for confidence measures in stereo vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 305–312, 2013.

[20] David H Wolpert. Stacked generalization. *Neural Networks*, 5(2):241–259, 1992.

[21] Srinivas Mukkamala, Andrew H Sung, and Ajith Abraham. Intrusion detection using an ensemble of intelligent paradigms. *Journal of Network and Computer Applications*, 28(2):167–182, 2005.

[22] Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. *Neural Computation*, 3(1):79–87, 1991.

[23] Tin Kam Ho, Jonathan J. Hull, and Sargur N. Srihari. Decision combination in multiple classifier systems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(1):66–75, 1994.

[24] Wray Buntine. *A theory of learning classification rules.* PhD thesis, University of Technology Sydney Australia, 1992.

[25] W Nick Street and YongSeog Kim. A streaming ensemble algorithm (sea) for large-scale classification. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 377–382. ACM, 2001.

[26] Martin Scholz and Ralf Klinkenberg. Boosting classifiers for drifting concepts. *Intelligent Data Analysis*, 11(1):3–28, 2007.

[27] Xueheng Qiu, Le Zhang, Ye Ren, Ponnuthurai N Suganthan, and Gehan Amaratunga. Ensemble deep learning for regression and time series forecasting. In *Computational Intelligence in Ensemble Learning (CIEL), 2014 IEEE Symposium on*, pages 1–6. IEEE, 2014.

[28] Yan Xu, Zhao Yang Dong, Jun Hua Zhao, Pei Zhang, and Kit Po Wong. A reliable intelligent system for real-time dynamic security assessment of power systems. *IEEE Transactions on Power Systems*, 27(3):1253–1263, 2012.

[29] Li Deng and John C Platt. Ensemble deep learning for speech recognition. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[30] Zhong Yin, Mengyuan Zhao, Yongxiong Wang, Jingdong Yang, and Jianhua Zhang. Recognition of emotions using multimodal physiological signals and an ensemble deep learning model. *Computer Methods and Programs in Biomedicine*, 140:93–110, 2017.

[31] Harris Drucker, Christopher JC Burges, Linda Kaufman, Alex J Smola, and Vladimir Vapnik. Support vector regression machines. In *Advances in Neural Information Processing Systems*, pages 155–161, 1997.

[32] Yang Zhao, Jianping Li, and Lean Yu. A deep learning ensemble approach for crude oil price forecasting. *Energy Economics*, 66:9–16, 2017.

[33] Huwaida Tagelsir Elshoush and Izzeldin Mohamed Osman. Alert correlation in collaborative intelligent intrusion detection systems—a survey. *Applied Soft Computing*, 11(7):4349–4365, 2011.

[34] MITRE Corporation. Common attack pattern enumeration and classification, 2018. https://capec.mitre.org/data/definitions/1000.html, Accessed 2018-10-29.

[35] Donghyun Shin. A platform for generating anomalous traces under cooperative driving scenarios. Master's thesis, University of Waterloo, 2018.

[36] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. In *Blackhat USA*. IOActive, 2015.

[37] Kelly Rollick, Allan Roczko, and Leslie Mitchell. Combustible gas detector sensor drift: Catalytic vs. infrared. *InTech Magazine*, Aug 2010.

[38] M. Seiter, H. J. Mathony, and P. Knoll. Parking assist. In *Handbook of Intelligent Vehicles*, pages 829–864. Springer, 2012.

[39] Jonathan Petit, Bas Stottelaar, and Michael Feiri. Remote attacks on automated vehicles sensors : Experiments on camera and lidar. In *Black Hat Europe*, pages 1–13, 2015.

[40] Subhojeet Mukherjee, Hossein Shirazi, Indrakshi Ray, Jeremy Daily, and Rose Gamble. Practical DoS attacks on embedded networks in commercial vehicles. In Indrajit Ray, Manoj Singh Gaur, Mauro Conti, Dheeraj Sanghi, and V. Kamakoti, editors, *Information Systems Security*, pages 23–42, Cham, 2016. Springer.

[41] Y. Mo and B. Sinopoli. Secure control against replay attacks. In *2009 47th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 911–918, Sept 2009.

[42] Alexander Bolshev, Jason Larsen, Marina Krotofil, and Reid Wightman. A rising tide: Design exploits in industrial control systems. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, Austin, TX, 2016. USENIX Association.

[43] Carlos Moreno, Sebastian Fischmeister, and M. Anwar Hasan. Non-intrusive program tracing and debugging of deployed embedded systems through side-channel analysis. In *Proc. of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*, pages 77–88, New York, USA, 2013. ACM.

[44] T. T. Y. Lin and D. P. Siewiorek. Error log analysis: statistical modeling and heuristic trend analysis. *IEEE Transactions on Reliability*, 39(4):419–432, Oct 1990.

[45] Steven M. Bellovin. Packets found on an internet. *SIGCOMM Comput. Commun. Rev.*, 23(3):26–31, July 1993.

[46] Abida Haque, Alexandra DeLucia, and Elisabeth Baseman. Markov chain modeling for anomaly detection in high performance computing system logs. In *Proceedings of the Fourth International Workshop on HPC User Support Tools*, HUST'17, pages 3:1–3:8, New York, USA, 2017. ACM.

[47] Farokh Marvasti, Mostafa Analoui, and Mohsen Gamshadzahi. Recovery of signals from nonuniform samples using iterative methods. *IEEE Transactions on Signal Processing*, 39(4):872–878, 1991.

[48] K. Sauer and J. Allebach. Iterative reconstruction of bandlimited images from nonuniformly spaced samples. *IEEE Transactions on Circuits and Systems*, 34(12):1497–1506, Dec 1987.

[49] H.G. Feichtinger. Discretization of convolutions and the generalized sampling principle. In *Progress in Approximation Theory*, pages 333–345, Boston; Toronto, 1991. Academic Press.

[50] Ke Wang and Salvatore J. Stolfo. Anomalous payload-based network intrusion detection. In Erland Jonsson, Alfonso Valdes, and Magnus Almgren, editors, *Recent Advances in Intrusion Detection*, pages 203–222, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[51] Gene H Kim and Eugene H Spafford. Experiences with tripwire: Using integrity checkers for intrusion detection. *Purdue Department of Computer Science Technical Reports*, 1994.

[52] Wojciech Samek, Thomas Wiegand, and Klaus-Robert Müller. Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models. *ArXiv Preprint arXiv:1708.08296*, 2017.

[53] Bill Chiu, Eamonn Keogh, and Stefano Lonardi. Probabilistic discovery of time series motifs. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 493–498. ACM, 2003.

[54] Xiaolei Li, Jiawei Han, Sangkyum Kim, and Hector Gonzalez. Roam: Rule-and motif-based anomaly detection in massive moving object data sets. In *Proceedings of the 2007 SIAM International Conference on Data Mining*, pages 273–284. SIAM, 2007.

[55] Grégoire Montavon, Wojciech Samek, and Klaus-Robert Müller. Methods for interpreting and understanding deep neural networks. *Digital Signal Processing*, 2017.

[56] National Institute of Standards and Technology. Secure hash standard - fips pub 180-2. *NIST Publications*, 2002.

[57] Robert Love. Kernel korner: Intro to inotify. *Linux Journal*, 2005(139):8, 2005.

[58] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, 2013.

[59] Jinghui Chen, Saket Sathe, Charu Aggarwal, and Deepak Turaga. Outlier detection with autoencoder ensembles. In *Proceedings of the 2017 SIAM International Conference on Data Mining*, pages 90–98. SIAM, 2017.

[60] Haowen Xu, Wenxiao Chen, Nengwen Zhao, Zeyan Li, Jiahao Bu, Zhihan Li, Ying Liu, Youjian Zhao, Dan Pei, Yang Feng, et al. Unsupervised anomaly detection via variational auto-encoder for seasonal kpis in web applications. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pages 187–196. International World Wide Web Conferences Steering Committee, 2018.

[61] Christoph Baur, Benedikt Wiestler, Shadi Albarqouni, and Nassir Navab. Deep autoencoding models for unsupervised anomaly segmentation in brain mr images. *ArXiv Preprint arXiv:1804.04488*, 2018.

[62] Sue Sendelbach and Marjorie Funk. Alarm fatigue: A patient safety concern. *AACN Advanced Critical Care*, 24(4):378–386, 2013.

[63] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. *Journal of Machine Learning Research*, 9(Aug):1871–1874, 2008.

[64] Srinivas Mukkamala, Guadalupe Janoski, and Andrew Sung. Intrusion detection using neural networks and support vector machines. In *Neural Networks, 2002. IJCNN'02. Proceedings of the 2002 International Joint Conference on*, volume 2, pages 1702–1707. IEEE, 2002.

[65] James Cannady. Artificial neural networks for misuse detection. In *National Information Systems Security Conference*, volume 26. Baltimore, 1998.

[66] Jake Ryan, Meng-Jang Lin, and Risto Miikkulainen. Intrusion detection with neural networks. In *Advances in Neural Information Processing Systems*, pages 943–949, 1998.

[67] Herve Debar, Monique Becker, and Didier Siboni. A neural network component for an intrusion detection system. In *IEEE Symposium on Security and Privacy*, pages 240–250, 1992.

[68] Jocelyn Sietsma and Robert JF Dow. Creating artificial neural networks that generalize. *Neural Networks*, 4(1):67–79, 1991.

[69] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533, 1986.

[70] Ajay Singh. A study of the power trace sampling frequency requirements in non-intrusive program tracing through power consumption monitoring. Master's thesis, University of Waterloo, 2018.

[71] Murray Dunne, Giovani Gracioli, and Sebastian Fischmeister. A comparison of data streaming frameworks for anomaly detection in embedded systems. In *International Workshop on Security and Privacy for the Internet-of-Things*, pages 30–33, 2018.

[72] Atmel Corporation. At32uc3a3/a4 series - complete datasheet, 2012. Rev. H-10/12.

[73] Carlos Moreno and Sebastian Fischmeister. Method and apparatus for non-intrusive program traching with bandwidth reduction for embedded computing systems, 2017. U.S. Patent Application.

[74] Atmel Corporation. Atmel 8-bit avr microcontroller with 2/4/8k bytes in-system programmable flash, 2013. Rev.: 2586Q–AVR–08/2013.

[75] Raspberry pi 3 model b - raspberry pi. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/. Accessed: 2018-10-19.

[76] Download raspbian for raspberry pi. https://www.raspberrypi.org/downloads/raspbian/. Accessed: 2018-10-19.

[77] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, Kobe, Japan, May 2009.

[78] Edwin Olson. Apriltag: A robust and flexible visual fiducial system. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 3400–3407. IEEE, 2011.

[79] Xiaoyuan Fan, Liang Du, and Dongliang Duan. Synchrophasor data correction under gps spoofing attack: A state estimation-based approach. *IEEE Transactions on Smart Grid*, 9(5):4538–4546, 2018.

[80] Mauro Conti, Nicola Dragoni, and Viktor Lesyk. A survey of man in the middle attacks. *IEEE Communications Surveys & Tutorials*, 18(3):2027–2051, 2016.

[81] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *ACM SIGPLAN Notices*, volume 41, pages 372–382. ACM, 2006.

[82] Anurag Srivastava, Thomas Morris, Timothy Ernster, Ceeman Vellaithurai, Shengyi Pan, and Uttam Adhikari. Modeling cyber-physical vulnerability of the smart grid with incomplete information. *IEEE Transactions on Smart Grid*, 4(1):235–244, 2013.

[83] Annarita Giani, Vincent H Berk, and George V Cybenko. Data exfiltration and covert channels. In *Sensors, and Command, Control, Communications, and Intelligence (C3I) Technologies for Homeland Security and Homeland Defense V*, volume 6201, page 620103. International Society for Optics and Photonics, 2006.

[84] Aashish Sharma, Zbigniew Kalbarczyk, R Iyer, and James Barlow. Analysis of credential stealing attacks in an open networked environment. In *Network and System Security (NSS), 2010 4th International Conference on*, pages 144–151. IEEE, 2010.

[85] Adam Hahn, Aditya Ashok, Siddharth Sridhar, and Manimaran Govindarasu. Cyber-physical security testbeds: Architecture, application, and evaluation for smart grid. *IEEE Transactions on Smart Grid*, 4(2):847–855, 2013.

[86] Laurence R Phillips, Bankim Tejani, Jonathan Margulies, Jason L Hills, Bryan T Richardson, Micheal J Baca, and Laura Weiland. Analysis of operations and cyber security policies for a system of cooperating flexible alternating current transmission system (facts) devices. *United States Department of Energy Publications*, 2005.

[87] Jussipekka Leiwo and Yuliang Zheng. A method to implement a denial of service protection base. In *Australasian Conference on Information Security and Privacy*, pages 90–101. Springer, 1997.

[88] Stephen McLaughlin, Dmitry Podkuiko, and Patrick McDaniel. Energy theft in the advanced metering infrastructure. In *International Workshop on Critical Information Infrastructures Security*, pages 176–187. Springer, 2009.

[89] Wenyuan Xu, Ke Ma, Wade Trappe, and Yanyong Zhang. Jamming sensor networks: attack and defense strategies. *IEEE Network*, 20(3):41–47, 2006.

[90] Robert Lee, Michael Assante, and Tim Conway. Analysis of the cyber attack on the ukrainian power grid. *Electricity Information Sharing and Analysis Center (E-ISAC)*, 2016.

[91] Siddharth Sridhar, Adam Hahn, Manimaran Govindarasu, et al. Cyber-physical system security for the electric power grid. *Proceedings of the IEEE*, 100(1):210–224, 2012.

[92] Yilin Mo, Tiffany Hyun-Jin Kim, Kenneth Brancik, Dona Dickinson, Heejo Lee, Adrian Perrig, and Bruno Sinopoli. Cyber–physical security of a smart grid infrastructure. *Proceedings of the IEEE*, 100(1):195–209, 2012.