Department of Computing, Imperial College London

MEng Individual Project

DeADA
Self-adaptive anomaly detection dataflow architecture

June 2013

Author

Andrei Bara

Supervisors

Prof. Wayne Luk

Submitted in part fulfillment of the requirements for the degree of Master of Engineering in Computing of Imperial College London

Contents

List of Figures

List of Tables

Listings

Abstract

Machine learning based network intrusion detection is increasingly becoming more difficult in the era of big-data, as not only does the amount of data increase, but the concept of data changes as new applications and systems are being integrated in a network. We propose an unsupervised self-adaptive anomaly detection algorithm based on One-Class Support Vector Machines which would be able to address the drift in data in an automated fashion, making it suitable for self-sustained network intrusion detection systems. We then integrate our algorithm in a highly parallel dataflow architecture designed as an end-to-end system for anomaly detection and analysis. Finally, we map the architecture to FPGA to assess its performance in comparison with an implementation on a 4 core CPU, obtaining a 4x improvement, with further theoretical improvements of up to 16x. We evaluate our proposed algorithm on synthetic benchmarks obtaining impressive results when compared against a one-off classifier model, and performance comparable to that of a supervised method.

Acknowledgements

I would like to thank:

- Professor Wayne Luk for providing guidance and for his continued interest in the project.
- Xinyu Niu for his help in understanding the Maxeler toolkit and his time spent helping me.
- Paul Grigoras for answering my questions related to Maxeler and for his introductory training day.
- Rory Allford for his initial contributions to the 3rd year project which formed the basis of the Ripple Framework and for writing the Shore-Probe packet capturing tool.
- Maxeler Technologies and Diego Ortega for the coaching provided at the Maxeler office.
- My friends and family for their continuous support and for not calling me on Skype everyday.

Chapter 1

Introduction

1.1 Motivation

The recent **Heartbleed** attack highlights a limitation of currently existing network intrusion detection (NID) techniques, which are unable to cope with unknown patterns, being very good in detecting attacks based on previously seen patterns, but can rarely detect novel threats. In addition, computing power is getting cheaper every year thus resulting in an increase in the volume of data transferred between systems. This impacts the ability of machine learning (ML) based NIDs to *accurately* distinguish between threats and ordinary traffic/information, as data are being exposed to *concept drift*. Furthermore, data are generated at high rates, being almost impossible for traditional machine learning based solutions to cope with real-time analysis, quite often resulting in offline analysis of data or network events. Companies such as BAE Systems Detica use large computer clusters to analyze data in a timely fashion using various algorithms like Naive Bayes or Neural Networks to recognize known attack patterns.

1.2 Challenges

1. **Handling diversity of attacks** is an active research area in the field of network security. Current **security information and event management** systems typically use either machine learning algorithms or rule based detection (e.g. Snort). However, the traditional problem with these approaches is that system administrators need to have prior knowledge of what defines an attack, hence reducing the space of *detectable* attacks to that of *known* attacks. **Heartbleed** is such a case, where the error was due to the lack of a bounds check in the OpenSSL code. Such exploits are hard to prevent as they most likely target a specific application higher in the OSI stack, and are determined by a combination of attributes. Scenarios that would cover every possible exploit are very hard to produce, but are needed in training new models which can *discriminate* between *normal* and *abnormal* data.

2. **Managing concept drift to prevent classifier degradation** is an

increasing problem in systems where new applications are introduced and is one of the generic problems of *anomaly detection*[4]. Nowadays systems usually produce data which no longer fits the model initially trained by a machine learning algorithm, resulting in decreased accuracy of the NID. Maintaining accuracy implies constantly generating new models, which in big-data environments implies manual analysis and labeling of data. Furthermore, data in NID systems is generally highly dimensional [4], which is not just a limiting factor when using machine learning algorithms but it also creates a diversity of ways in which concept drift can arise.

3. **Real-time analysis** of data is the main advantage rule-based systems have over machine learning algorithms. The relatively high computational complexity of the various MLs, restricts such techniques to offline use, with large data clusters analyzing packets captured over long period of time. Although the offline technique is effective in preventing *future* similar attacks, in many of the cases it might be too late, as attacks have already passed.

1.3 Contributions

The idea behind the project is to investigate and propose new ways in which these challenges could be addressed. For clarity purposes, this project will focus on *packet analysis*, however the techniques described in the report could be applied to *network event analysis*[8] as well. As such we define the following contributions:

1. **Unsupervised self-adaptive ensemble (USAE) algorithm** is based on One-Class Support Vector (OCSVM) machines, a variation of the classical Support Vector Machines (SVMs), but targeted for anomaly detection. We have chosen anomaly detection as the candidate for preventing new and unknown attacks. The idea behind is that only normal data are used in training the classifiers, whilst anything that differs from the normal pattern will be labeled as an anomaly. OCSVMs have been used successfully in the literature in detecting anomalies (also known as novelty detection), however there are few investigation, to the best of our knowledge, involving OCSVM and the concept-drift problem. We therefore propose an approach which not only tackles data drift by adapting to it, but also does it in an **unsupervised** manner, meaning there is no need for human interaction in order to adjust the models. The algorithm is presented in chapter 3.

2. **DeADA dataflow architecture** addresses the long computation time required by the Machine Learning programs, with a focus on OCSVM. Being based on SVMs, OCSVMs allow for greater parallelization of the computation and as such, a scalable architecture can be created. The solution integrates the USAE algorithm to improve the accuracy of the classifiers under concept drift and reduce the analysis time of data, thus making the prospect of real-time analysis more feasible when dealing with machine learning algorithms. Our proposed architecture can be found in chapter 4.

3. **The Ripple Framework** is a JAVA based library for rapid prototyping of dataflow architectures. We created the library based on Kahn Process Network principles, a type of flow-based computing part of the more general dataflow computing paradigms. The framework has proven useful in assessing the feasibility of DeADA as well as in measuring its performance on the CPU. The framework is described in chapter 5.

Chapters 3 and 4 form the basis for a research paper to be submitted for the 2014 International Conference on Field Programmable Technology, in Shanghai.

Chapter 2

Background

2.1 Anomaly detection

Anomaly detection is an active research area within machine learning which can be applied to a diversity of domains varying from finance to military. The aim of anomaly detection is to address the problem of discriminating between *normal* and *abnormal* behavior/data. These anomalies are often referred to as *outliers* or *exceptions* [4] and are quite often associated with binary classification problems (Figure 2.1). A subclass of anomaly detection is *novelty detection*, which addresses the issue of detecting previously unobserved patterns in the data, by *recognizing* positive instances of a concept rather than *differentiating* between them [12]. As Chandola et al. [4] adds, a key distinction to standard anomaly detection is that novel patterns are usually incorporated into the normal model once detected. Similar to novelty detection is *outlier* detection, however, the main difference resides in the fact that for novelty detection all training data are regarded as *normal*, whereas outlier detection is generally used when the data set representing the population is not clean [30].

Of importance to anomaly detection as stated in [4] are the data types. Chandola et al. identifies 3 main types of data:

- **sequence data**. Sequential data has ordered instances, generally as a function time, however different orderings may exist (e.g. lexicographic, etc.)
- **spatial data**. Spatial data, has instances which are related to its *neighboring* data (e.g. in sensor network, the sensors closer together will produce *spatial data*)
- **graph data**. Instances are represented as nodes in a graph, inter-connected by edges.

Building on these three types of data, Chandola groups anomalies into the following categories [4]:

Point Anomalies are instances of the data which are anomalous with respect to the rest of the data. As it can be seen from Figure 2.1 o_1 and o_2 are point anomalies as they lie outside the N_1 and N_2 sets.

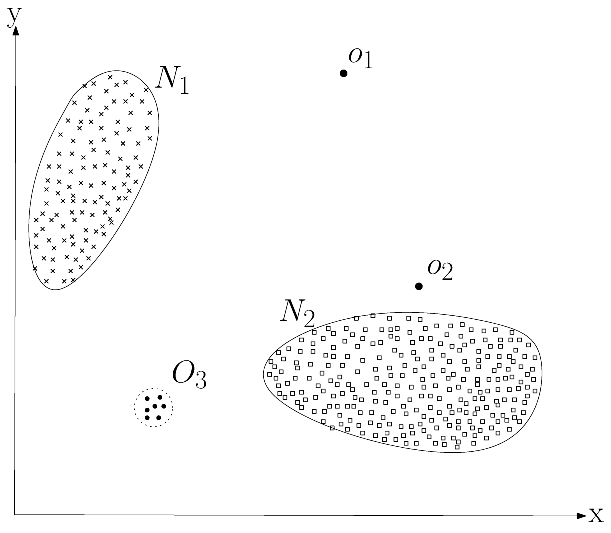


Figure 2.1: N_1 and N_2 are sets containing *normal data*, whereas O_1 and O_2 are outliers. *Source*: Chandola [4]

Contextual Anomalies are data instances which are anomalous only in certain *contexts* and normal otherwise. Data instances are defined by two types of attributes:

1. Contextual attributes, used to describe the neighborhood of that instance (e.g. geographical coordinates).
2. Behavioral attributes, used to describe the *non-contextual* characteristics of an instance (e.g. amount of rain in a certain geographical location).

Collective Anomalies are partitions of data which are anomalous with respect to the whole data set. As a consequence, anomalies can only be identified by analyzing them as a group. This is sometimes the case with anomalies in NIDs where multiple packets are re-assembled and analyzed together.

2.1.1 Support Vector Machines (SVM)

Support Vector Machines are a relatively new addition to the field of Machine Learning compared with other ML algorithms. The concept was introduced by Vapnik et al. [3] in 1992 and was based on the idea of identifying the *optimal* separation hyperplane for linearly separable data.

The Support Vectors (SV) are data points that lie closest to the decision boundary. Support vectors are computationally expensive to identify as it boils down to solving an optimization problem: increasing the *margin* around the hyperplane.

Suppose that we have the following training data $\{(x_1, y_1), (x_2, y_2)...(x_n, y_n)\}$, where x_i is the input and y_i is the class (represented as $\{-1, 1\}$). We can clearly see this is a binary classification problem.

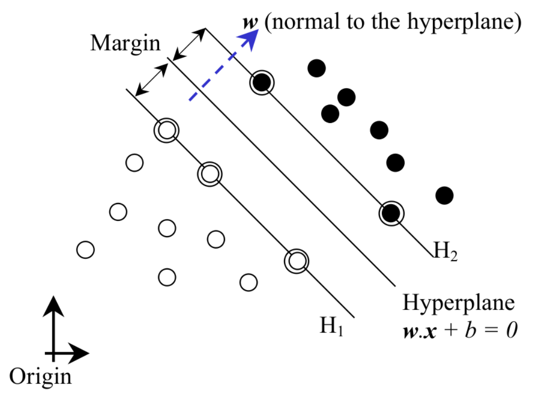


Figure 2.2: The hyperplane is the middle line, whereas H_1 and H_2 are the decision boundaries. *Source*: IDAPI course

We assume the 2 classes can be separated by a hyperplane H:

$$\text{Hyperplane}: w * x + b = 0 \tag{2.1}$$

We can now define the planes H_1 and H_2 as:

$$\begin{array}{l} \text{H1: } w * x_i + b \geq 1 \quad if\ y_i = +1 \\ \text{H1: } w * x_i + b \leq 1 \quad if\ y_i = -1 \end{array} \tag{2.2}$$

The distances from the origin to the two planes are $\frac{|1-b|}{|w|}$ and $\frac{|-1-b|}{|w|}$. As such the distance between H_1 and H_2 becomes $\frac{2}{|w|}$ and the distance between the hyperplane and the two planes forming the *margin* becomes $\frac{1}{|w|}$. Since we want to maximize the margin, we need to minimize $|w|$. Thus, this become an optimization problem for $|w|$:

$$\text{Minimize} f(w) : \frac{1}{2}||w||^2 \text{ s.t. } g(w,b) : y_i(w * x_i + b) - 1 \geq 0 \tag{2.3}$$

From this, we can express the Lagrangian, where n is the number of training instances:

$$L(w, b, \lambda) = \frac{1}{2}||w||^2 - \sum_i^n \lambda_i * [y_i(w * x_i + b) - 1] \tag{2.4}$$

We can now differentiate the Lagrangian to find the constraints:

$$\begin{array}{l} \frac{\partial L}{\partial w} = w - \sum_i^n \lambda_i * y_i * x_i \text{ and} \\ \frac{\partial L}{\partial b} = -\sum_i^n \lambda_i * y_i \end{array} \tag{2.5}$$

Setting the derivatives to 0 we have: $w = \sum_i^n \lambda_i y_i x_i$ and $\sum_i^n \lambda_i y_i = 0$. From this we can derive the *dual* formulation of the problem as:

$$
\begin{aligned}
&L(w^*, b^*, \lambda) = \textstyle\sum_i^n \lambda_i - b \sum_i^n \lambda_i * y_i - \sum_i^n \sum_j^n \lambda_i \lambda_j y_i y_j x_i^T x_j \implies \\
&L(w^*, b^*, \lambda) = \textstyle\sum_i^n \lambda_i - \sum_i^n \sum_j^n \lambda_i \lambda_j y_i y_j x_i^T x_j
\end{aligned}
\tag{2.6}
$$

We now have to find the $\lambda = \max_\lambda L$. Once we find the values for λ_i many of them will be 0. Those which are non-zero, will form the *support vectors*.

To prevent the SVM from overfitting on noisy data, a penalty constant C and a slack variable ξ can be introduced. Thus, we would have to add $C \sum_i^n \xi_i$ to the function to be minimized and modify the constraints so that $\xi_i \geq 0$ and $g(w, b) \geq 1 - \xi_i$

However, the inner product of the vectors is problematic when dealing with non-linear inputs. The solution to this is known as the *Kernel trick* and involves a kernel function $K(x_i, x_j) = \phi(x_i)\phi(x_j)$ (where $\phi(x) : \mathbb{R}^n \mapsto \mathbb{R}^m$), which maps the data points to higher dimensions where they become linearly separable. There are several options for choosing this kernel, most common being the *Sigmoidal*, *Gaussian radial basis*, and *Polynomial*.

2.1.2 One-Class Support Vector Machines

One-Class SVMs (OCSVM) are a recent addition to the field of machine learning algorithms which build on top of the classical SVMs, and deal with *identifying* whether new data are of the same class as the training data, thus becoming an attractive candidate for anomaly detection techniques. There are two main models used for describing the OCSVMs: one developed by Schölkopf et al.[29], the other one by Tax and Duin[33].

2.1.3 OCSVM according to Schölkopf

The idea behind the algorithm is to create a function f which maps most of data in some *small* region $+1$ and the rest to -1. During the *offline* phase the OCSVM considers the *origin* point to be the only negative example in the data set and tries to find a separating hyperplane while maximizing the margin between the data points and the origin (Figure 2.3) [36] [29].

The optimization problem is described the following way:

$$
\begin{aligned}
&\underset{w,\xi,\rho}{min} \frac{1}{2}||w||^2 + \frac{1}{\nu n} \textstyle\sum_i^n -\rho \text{ s.t.} \\
&(w * \phi(x_i)) \geq \rho - \xi_i \qquad \text{and} \\
&\xi_i \geq 0, \forall i \leq n
\end{aligned}
\tag{2.7}
$$

The decision function f thus becomes:

$$
f(x) = \text{sgn}\left(\sum_i^n \lambda_i K(x_i, x) - \rho\right), \text{ x = target data} \tag{2.8}
$$

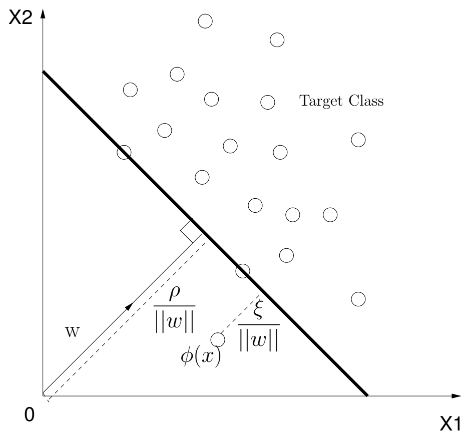


Figure 2.3: Hyperplane separation for the target class

Similar to the case of SVM, the optimization problem for finding λ (the support vectors) can be resolved using a QP solver for the following function:

$$\min_{\lambda} \frac{1}{2}\sum_{i}^{n} \lambda_i \lambda_j K(x_i, x_j) \text{ s.t. } 0 \leq \lambda_i \leq \frac{1}{\nu n} \text{and} \sum_{i}^{n} \lambda_i = 1 \tag{2.9}$$

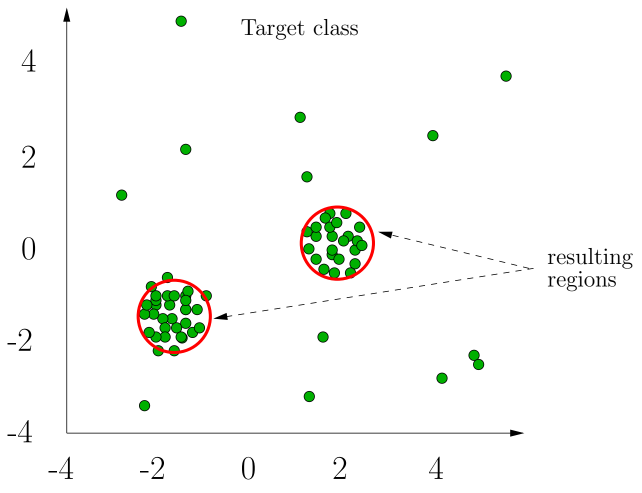


Figure 2.4: Learned frontier for One-class SVM

In Figure 2.4 we can see the learned frontiers of a set of two dimensional data points. These two regions are the result of mapping a higher dimension hyperplane generated using an RBF kernel back to a two dimensional representation. Although all of the points are part of the same class (i.e. *normal*) only the ones inside the frontier will be classified as normal, the rest being abnormal (these would be classified as training errors and are dependent on the selection of the ν parameter).

Importance of the ν parameter

m is the total number of training examples.

$$\frac{|Outliers|}{m} \leq \nu \leq \frac{|supportvectors|}{m} \tag{2.10}$$

According to Schölkopf et al. [28],[29] ν represents:

1. an upper bound on the total number of training examples which can be considered as outliers.
2. a lower bound on the number of resulting support vectors

2.1.4 OCSVM according to Tax and Duin

The formulation of Tax and Duin [33] differs from that of Schölkopf at el. in the sense that it takes a *spherical* rather than *planar* approach. This type of SVM is described by the authors as *Support Vector Data Description*(SVDD). The aim of the algorithms is to minimize the hypersphere so that only a few outliers are incorporated into the model [36]. The two points on the hyperplane will be the support vectors.

The hypersphere with the optimal volume R^2 has a radius $R > 0$ from the center a to any of the support vectors on the boundary. The center a is a linear combination of the support vectors. Similar to the previous formulation a slack variable ξ and a penalty constant C are being introduced (we can consider the $\frac{1}{\nu n}$ from Schölkopf as one possible solution for C). Using the Lagrangian and

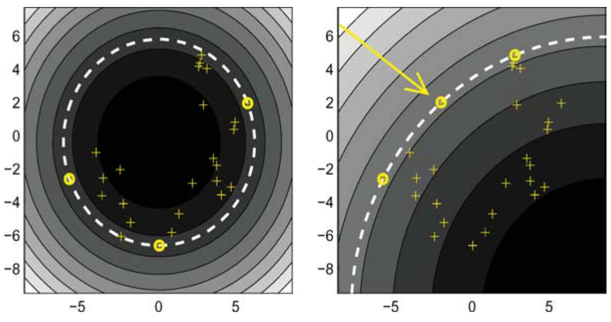


Figure 2.5: An example of data description without outliers(*left*) and with one outlier(*right*). *Source*: Tax and Duin [33]

the dual formulation, an optimization problem can be written as described in [33]. Thus, the decision function (where **z** is the input to be tested) can be written as:

$$||z - x||^2 = z^T z - 2\sum_{i}^{n} \lambda_i K(z, x_i) \sum_{i,j}^{n} \lambda_i \lambda_i \lambda_j K(x_i, x_j) \leq R^2 \tag{2.11}$$

2.1.5 Other techniques

A lot of investigations have been done in the field of *anomaly detection* and machine learning. However, to the best of our knowledge, few are trying to tackle the issues of the large amount of data in network system and attempt to perform anomaly detection in a timely fashion. Although many might not apply to our use case (Network Intrusion Detection) they are useful in discovering what could work and what could not.

One interesting area to look into is that of adapting existing machine learning algorithms which were not designed for anomaly detection but on which attempts have been made to achieve that. Japkowicz [12] presents a *novelty detection* algorithm based on *Neural Networks*, using an *autoencoder*. Classification/identification is possible because data instances which are normal can be reconstructed accurately, whereas the anomalies will not. However, neural networks have the downside of scaling poorly with the number of attributes. In order to deal with a broader spectrum of data (with an increased number of attributes), more *perceptrons* are required, and they are computationally expensive. Of course, techniques such as *attribute selection* [20] or *Principal Component Analysis* exist, but they abstract away some of the information that comes with those attributes which in areas like network security where minor changes can occur, might turn out to be undesirable.

2.2 Accelerating anomaly detection

Another interesting solution to anomaly detection has been proposed by Das et al. [5] based on Principal Component Analysis on top of FPGA. This is closely related to the goals of this project and provides a very useful insight into how one could turn PCA to anomaly detection. Traditionally PCA is considered to be a *preprocessing* or *postprocessing* step in machine learning, including SVMs.

The role of PCA is to find the underlying *correlations* in the data in order to reduce the dimension of the data (i.e. the number of attributes/features). Besides providing an FPGA cascaded implementation to *classification* (which could be turned into a data flow program), the authors make use of the *Mahalanbois distance* as a measure for the anomaly. This distance is a very good candidate for replacing the kernel in the OCSVM implementation, as it is consists of a series of matrix vector multiplication, which fit very well with *data flow computing*. However, a key point that comes out of the paper is that their focus is mainly on the *online* phase and not so much in the *offline* phase in terms of FPGA acceleration. The problem of accelerating the training phase is in fact a problem of accelerating the QP solving step for calculating the optimal values (i.e. the principal components in this case, and the support vectors in the SVM case).

There are quite a few solutions for the training problem, some creating an implementation for QP solvers [31] [13], while others using a geometrical approach to finding the support vectors [23]. The geometrical approach proposed by Markos et al. seems to be a viable options for the *offline* phase of the OCSVM, by providing a cascading FPGA architecture, which could be migrated to data flow programs (like MaxJ). However, the solution of having an FPGA based QP solver could prove to be more efficient when it comes to using the Mahalanbois distance as a kernel.

In fact, as Wei et al. [39] show in their paper using the Mahalanbois not only represents a good choice for the kernel, but it will also make the algorithm more robust as it will incorporate knowledge about the *correlations* in the data. Furthermore, they present a version which uses singular value decomposition to speed up the matrix multiplications. However, this solution still requires a QP solver and decomposing the problem in this manner might add extra computational steps that would need to be implemented in the FPGA, and is not clear (at the moment) whether it will result in a *speedup* or a *slowdown* of the offline training phase.

2.3 Handling concept drift

Tsymbal et al. [35] present three approaches to handling concept drift:

1. instance selection
2. instance weighting
3. ensemble learning

Instance selection is usually performed based on a windowed model moving over newly arrived instances in order to select those which are more relevant to the current concept. Then these instances are used to train classifiers for predictions on new data. Common techniques that fall under this category include case-based reasoning and case-editing strategies.

Instance weighting takes advantage of the properties of some Machine Learning algorithms which can deal with weighted inputs. SVM is one such algorithm and Krawczyk [16] propose an incremental learning and forgetting technique built on top of SVDD using *weighted One-Class SVM* [2]. Weights of the training data are changed as a function of time. One of the drawbacks of this technique is that all the training examples have to be stored in order to improve the accuracy of the model, and naturally as more data arrives the space requirements increase. Secondly, the training time also increases which might not be appropriate for applications regarding network intrusion detection. Thirdly, it has been shown by Klinkenberg[14] that *example weighting* may not be an appropriate model for handling concept drift.

Ensemble learning is the third option. It works by maintaining a set of models which are used in predicting the class of the incoming instance. Usually this technique involves a voting system and incremental learning with a heuristic for dynamically updating the ensemble members. The heuristic usually measures the *quality* of the base models with respect to the new data.

Windowed approaches can also be categorized as *ensemble learning* [25] with the ensemble being formed of different models trained at times $t, t+1...t+(N-1)$, where N is the *width* of the window. Klinkenberg et al. [15] present a solution for adaptive size windows, based on the $\xi\alpha$ (leave-one-out) estimator. The justification for dynamically adjusting the size of the ensemble is that concept might drift at different rates, and as such *small size* windows are better at learning fast changing concepts whereas larger windows are better at handling slow drifts (in terms of accuracy of the classifiers). However, with an adaptive window, estimating the time required to train and adjust the ensemble becomes more difficult and a variable training time might not be desirable depending on the application domain.

Parveen et al. [25] present an approach which uses a time shifted windowing method with One-Class SVM to detect concept drift for Insider Threat Detection. They propose a supervised solution, where the decision of which members of the ensemble to update is based upon the predictive accuracy of each individual model with respect to whole ensemble. For predicting the class of a new data point they use an accuracy weighted majority voting algorithm. The approach has proved to be quite efficient in detecting anomalies, but because they use the MIT Lincoln Data sets, we believe it is hard to quantify and describe the nature of the concept drift (i.e. it might just be the case that some of the data are "better" for training and thus increasing the accuracy of the updating model). However, in our analysis in section 6.3 we use the *supervised* algorithm described in [25] as reference for our *unsupervised* solution, but with a synthetic benchmark.

Tsymbal et al. [35] describe concept drift as being *local* or *global*. Local drift is determined with respect to the larger data set, and occurs between two consecutive time points. One important aspect of their paper is the relation between *rotating hyperplane*(Figure 2.6) [11] problem and the generation of *synthetic benchmarks*. The rotating hyperplane is a commonly used technique for simulating *gradual* concept drift of binary data (i.e. data with two classes). The formula for generating the hyperplane in d dimensions(i.e. features) is described by $\sum_{i=1}^{d} w_i x_i = w_0$. The hyperplane is then *rotated* by changing the weights $w_i, \forall i < K$ (K is the *magnitude* of change in terms of number of drifting features), with a certain $rate = s_i * \frac{T}{N}$ (change every T examples out of N total). The direction of the *drift* can be altered by switching the sign of s_i. The authors then proceed to tackling concept drift by using a Naive Bayes algorithm. We shall use an approach similar to the rotating hyperplane when generating our benchmarks for the One-class SVM, with the details of the differences described in section 3.1.

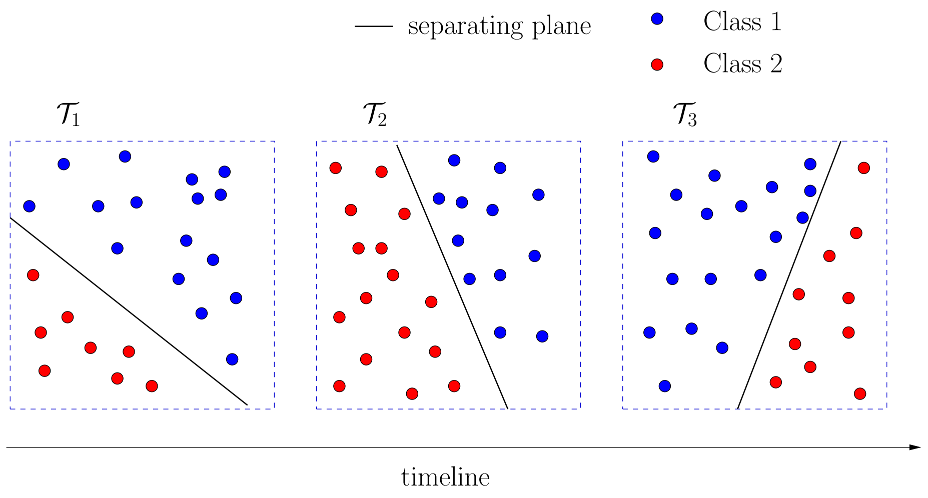


Figure 2.6: Rotating hyperplane for a binary problem

Type	Measures Drift	For anomaly detection
MIT Lincoln Dataset[25]	✗	✓
ECUE Dataset[16]	✗	✗
Rotating Hyperplane[35]	✓	✗
Our benchmark	✓	✓

Table 2.1: Analysis of different benchmarks proposed in the literature when handling concept drift. We are interested in whether the data sets have measurable concept drift and are suitable for drift in anomaly detection.

The methods described above are based on *supervised* learning. In general, supervised learning yields better results than the *unsupervised* version. However, in systems like computer networks with high volume of data, engaging humans to constantly monitor and sort the data (i.e. split it in normal/abnormal, evaluate the accuracy of the models, etc.) is a very tedious process, often leading to a good, but impractical solution. Lecomte et al. [17] propose an *unsupervised* approach for detecting anomalies in continuous streams of audio surveillance data. Their solution introduces a threshold λ, which translates the hyperplane of the model. This allows them to control the rate of false negatives and false positives by applying the function described Equation 2.12. We use a similar approach in our *unsupervised* algorithm as described in section 3.2.

$$\begin{array}{l} if\ f(x) \geq \lambda\ then\ x\ is\ normal \\ if\ f(x) < \lambda\ then\ x\ is\ abnormal \end{array} \tag{2.12}$$

Parveen et al. have also conducted studies into unsupervised learning for Insider Threat detection using a technique called *Graph Mining*[24]. However, they later show in [25] that a supervised window method outperforms the unsupervised solution.

A formal definition of concept drift as well as a possible way to measure it can be found in [38]. Wang et al. define the concept drift problem using three basic properties *intensions*, *extensions* and *labeling* which form the meaning of a concept. In order to be able to talk about *drift* with respect to a concept, the authors state that every concept must have some *rigid* properties (i.e. which don't change over time). The intension of a concept is formed by $int_r(C) \cup int_n r(C)$. Two concepts are considered *equal* (i.e. $C_1 = C_2$) if and only if their rigid intensions are equal $(int_r(C_1) = int_r(C_2))$.

The notion of *rigidity* is relevant to our work when creating our synthetic benchmarks, as we want to ensure the generated data does not undergo *concept shift*. Wang et al. describe *concept shift* as the change in a concept which makes it look more like a different concept. On the other hand, non-shifting concepts are subjected to *instability* as their *meaning* changes.

Table 2.1 gives a brief comparison of the different benchmarks presented in the literature. The MIT Lincoln Dataset (a.k.a. DARPA data set) is used to test the accuracy of a supervised ensemble of One-Class SVMs to detect anomalies under concept drift, however the authors do not give any measures of the drift, nor does the data set itself, to the best of our knowledge, have any information about the drift. The ECUE data set is used in [16], however the set contains already extracted features of spam and non-spam e-mail and has no measure of drift. In addition, it is our understanding that the method used to extract the features[7] creates a *hidden structure* in the data which might not be seen by the One-Class SVM [32][1] during the learning stage. The rotating hyperplane benchmark is a good benchmark for analyzing concept drift and has been previously tested with standard SVMs. However, due to the formulation of the problem in [35] it is not suitable for novelty detection (see section 3.3).

2.4 Dataflow Programming

Traditional sequential programming paradigms work in line with the von Neumann principles where data are considered *at rest*. Quite often this leads to inefficient accesses to memory, thus requiring the CPU cores to allocate a significant number of resources to caching data, performing branch predictions and other operations needed to run an arbitrary program, thus reducing the area of the chip available for computations. Multi-core processors can increase the computational power by having programmers parallelize the applications, however this can lead to complicated code as state information needs to be shared across the multiple cores or the parallel processing machines if in a distributed system.

On the other hand in dataflow paradigms, data are *moving* through the system. A dataflow program is constructed as a graph, where nodes perform the computations and edges representing inputs and outputs. The graph can be regarded as a large computational pipeline, where at each stage data are streamed in and out and values can be produces at every cycle. Instructions are embedded in the structure of the graph rather than being stored in memory. As a consequence a design running at 100Mhz could outperform it's CPU counterpart (as an algorithm) while consuming less energy[26].

2.5 FPGA Acceleration

Dataflow programming is a general paradigm which can be applied on different architectures CPUs, GPUs, ASICs[2] or FPGAs(Field Programmable Gate Arrays). We have chosen FPGAs as our architecture of choice due to their properties [5]:

- FPGAs are formed of re-configurable interconnected logic blocks, which can be wired to perform complex functions. A configuration is generally specified using a hardware description language (HDL).
- As opposed to ASICs (which are generally non-reconfigurable) FPGAs provide increased flexibility and a higher cost efficiency with low production volumes. On the other hand, the size of the FPGA chip constrains the design that can be uploaded onto it.
- FPGAs have a higher power efficiency when compared to multi-core CPUs and GPUs

[1]The authors state that One-Class SVMs do not consider dependence structure in the data

[2]Application Specific Integrated Circuits

- because there is no operating system and no synchronization requirement, scheduling is static and thus performance models are easier to create.

On the other hand FPGAs have the disadvantage of a longer development time as HDL specifications are cumbersome, and quite often hard to debug. Higher-level tools like MaxCompiler can simplify the design and development process, but compiling an FPGA design can take a very long time even when done on multi-core machines.

2.5.1 Resources

The FPGA resources are classified into the following categories:

1. **Look-up Tables (LUT)**: combinatorial elements used to implement logical functions.
2. **Flip-Flops (FFs)**: units intended for storage which can be used as registers
3. **Digital Signal Processors (DSP)**: special arithmetic units
4. **block RAM (BRAM)**: on-chip storage elements with quick access (also known as Fast Memory).

2.6 Maxeler Tools

Maxeler Technolgies provides custom hardware acceleration solutions, based on the concepts of data-flow programming. Designs are compiled using MaxCompiler, a proprietary compiler which maps dataflow designs to a hardware platform built around FPGAs.

There are several versions of the Maxeler hardware platform, however the current project will be using MAX3424A (MAX3) and Maia(MAX4) cards, with Xilinx Virtex 6 chips and Xilinx Virtex 7 respectively.

```java
class MovingAverageSimpleKernel extends Kernel {

	MovingAverageSimpleKernel(KernelParameters parameters) {
		super(parameters);

		DFEVar x = io.input("x", dfeFloat(8, 24));

		DFEVar prev = stream.offset(x, -1);
		DFEVar next = stream.offset(x, 1);
		DFEVar sum = prev + x + next;
		DFEVar result = sum / 3;

		io.output("y", result, dfeFloat(8, 24));
	}
}
```

Figure 2.7: A moving average function implemented using MaxJ

The MaxCompiler [19] provides a hybrid solution to dataflow programming by separating the code into two parts:

1. **CPU code**: which handles set-up and starting procedures of the dataflow engine (DFE) through a C API library, interaction with other applications, and manages the output from the DFE. The code (also known as *host code*) interacts with the Engine Code via PCIe.

2. **Engine code**: which forms the dataflow engine. A typical Maxeler application is formed of one or more Kernels[3] which are the *graph* nodes where computations are performed. The other important component is the *Manager* which specifies how information flows through the graph by linking various Maxeler components (e.g. State Machines, Kernels, Memory Pipelines, Inputs, Outputs, etc.) and thus forming a *dataflow design*. MaxCompiler provides a Java API for constructing the designs through the MaxJ meta-programming language.

```
max_file_t *maxfile = MovingAverage_init();
max_engine_t *engine = max_load(maxfile,"*");
max_actions_t *actions = max_actions_init(maxfile,"default");
max_queue_input(actions,"input",inputBuffer,inputSize);
max_queue_output(actions,"output",outputBuffer,bufferSize);
max_run(engine,actions);
```

Figure 2.8: Intialization code for the moving average function

A simple Maxeler kernel implementation of the moving average function is shown in Figure 2.7. Once the kernel has been built and the manager has linked all the components, the final part is to initialize the design from the host code. This can be done either through static Simple Live CPU Interface (SLiC), or through a dynamic one (Advanced Dynamic SliC). Figure 2.8 shows a snippet of how to initialize and run the MovingAverageSimple example via dynamic SLiC.

2.7 LibSVM

In order to train our models using One Class SVM we have chosen to use the LibSVM library. We preferred it to other libraries as it is lightweight and easy to integrate with our Ripple framework (see chapter 5). The OCSVM implementation provided by LibSVM is based on Schölkopf's hyperplane version. The optimization part of the training is done using Sequential Minimal Optimization (SMO) described in [27] a fast QP solving algorithm specifically designed for use with Support Vector Machines.

2.8 Summary

In this chapter we:

[3]The term Kernel here is different from that of the Kernel described in the context of the SVM

- introduced the concept of anomaly detection in the context of network security.
- presented the basic principles behind Support Vector Machines, and the One-Class Support Vector Machines which are crucial in understanding how anomaly detection and our proposed solutions are implemented. Two types of OCSVM approaches were presented, with the chosen one being that described by Schölkopf.
- looked at different machine learning algorithms applied in anomaly detection and analyzed their feasibility in the context of network security. The next step involved looking at various ways machine learning algorithms can be accelerated.
- presented the meaning of *concept drift* and provided a description of it, as well as existing work related to it. We also looked at different pieces of literature presenting ways of handling concept drift, and try to understand how it applies to anomaly/novelty detection.
- looked at the dataflow computing paradigms. We introduced the Maxeler tools to be used in section 6.4 as well as details about FPGAs, our target platform for the DeADA architecture presented in chapter 4.

Chapter 3

Self-adaptive ensemble

This chapter addresses the issues posed by concept drift in the context of anomaly detection performed with One-Class Support Vectors, and proposes a new algorithm for handling it. We hereby highlight the main contributions of this chapter:

- **Concept drift in anomaly detection**. We start off by stating the problem of drift with respect to the separating hyperplane of the OCSVM and explain the difference to the *rotating hyperplane problem* used in the literature. Hence, we introduce a definition of drift in terms of the *range shift* and *range expansion* of data (see section 3.1).

- **Unsupervised self-adaptive ensemble**. We introduce *USAE* our general use *online and fully automated* anomaly detection algorithm which handles concept drift. *USAE* exposes a parameter λ for fine tuning the ensemble, with respect to the strength of the drift. This forms the main contribution of this chapter and can be seen in section 3.2.

- **Synthetic benchmarks**. We propose a new method for generating synthetic benchmarks aimed for the specific use-case of anomaly detection, which unlike existing benchmarks [35],[7],[25] allows us to estimate the performance of the basic OCSVM and USAE with various **measurable** drift *rates* and *magnitudes*(see section 3.3).

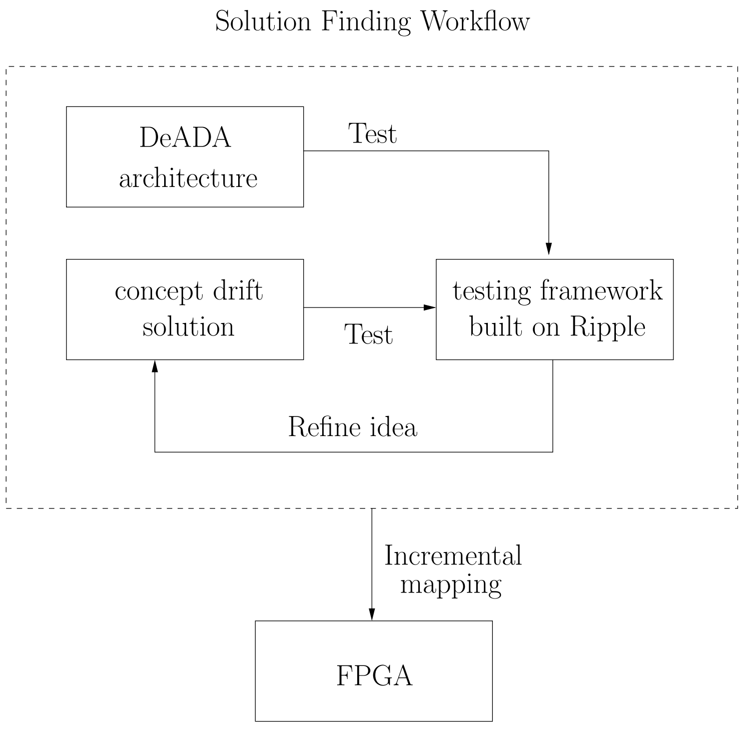


Figure 3.1: Workflow for arriving at a solution through iterative investigations and testing.

Figure 3.1 contains an overview of the work flow used in order to arrive to a solution to the problem of performing anomaly detection in computer networks with a high volume of data. We begin by defining a potential concept drift solution, then *test* and *refine* the ideas using the KPN Ripple Framework (see chapter 5). At the same time we construct and test DeADA which is described in chapter 4.

3.1 Concept Drift in Novelty Detection

Real-world systems have the tendency to change in time and so does the data they produce. The changes in data lead to modifications of the statistical properties of the target data-point we are trying to classify. Take, for example, the case of a company providing web-services. In order to access these web-services users need to specify different ports. The company is trying to prevent potential attacks its services by performing anomaly detection based on the port value and some other attributes.

Now suppose the company wants to add a new web-service, and as such they will use a different port number (e.g. $maximum\ of\ the\ allocated\ ports + 5$). The anomaly detection algorithm will now classify a *legitimate* request on this port as an anomaly, since the value is outside the *normal* range, and thus the data points ends outside the region described by the decision function $f(x)$.

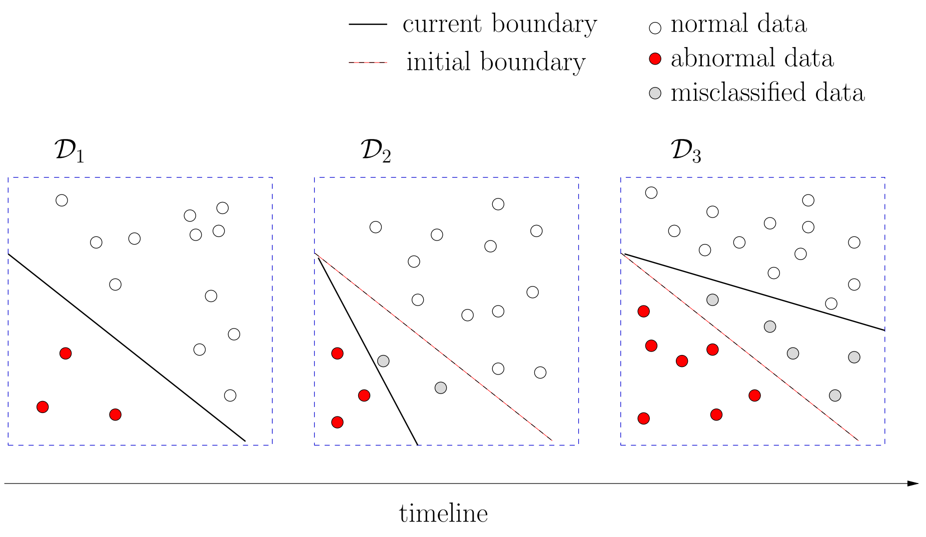


Figure 3.2: Different types of concept drift relative to the initial data $\mathcal{D}_1$

In Figure 3.2 we have the $\mathcal{D}_1$ data set on which we trained an initial classifier with the separating hyperplane shown as a solid line, whereas $\mathcal{D}_2$ and $\mathcal{D}_3$ describe two cases of the concept drift:

1. The decision boundary of the second data set moves below that of the first data set. By using the classifier trained on $\mathcal{D}_1$ two of the *normal* instances will be misclassified as abnormal, thus leading to an increase in the rate of *false positives*, despite being normal.

2. The decision boundary of the third data set moves above the initial boundary. In this scenario five of the *anomalies* would wrongly be considered normal, thus leading to an increase in the number of *false negatives*.

The change of the hyperplane in Figure 3.2 is very similar to that of the rotating hyperplane in Figure 2.6. However, the latter is applied to standard SVMs whereas the former is for One-Class SVM. The first difference is that anomalies seen in Figure 3.2 can only be mapped during the *online* phase, as the *offline* phase of the OCSVM contains only *normal* instances[1]. As seen in Figure 2.3 the maximum margin is calculated with respect to the origin (with no anomalies present below the hyperplane), whereas in Figure 2.2 the hyperplane is created in such way that it separates the two classes. Secondly, the *rotating hyperplane* is defined for a binary classifier (e.g. with labels -1 and 1), which means that both classes drift in such way that some of the instances previously labeled as -1 become 1 and vice versa, with the SVM being able to incorporate this knowledge into its learning phase. However, the OCSVM has no prior knowledge about the *anomaly* class (which we shall represent as -1).

Experimenting with one of the synthetic benchmarks (hyperplane9.arff) generated by Tsymbal et al. in [35], has revealed no deterioration of the accu-

[1]This is an important note. Despite the presence of the ν parameter which sets an upper bound on the training error, the *training error* refers to the number of *normal* data that can be left out and treated as anomalies, and does not directly deal with noise.

racy of the One-Class SVMs model even when applied to the strongest concept drift(K=8,T=1)[2], as can be seen in Figure 3.3.

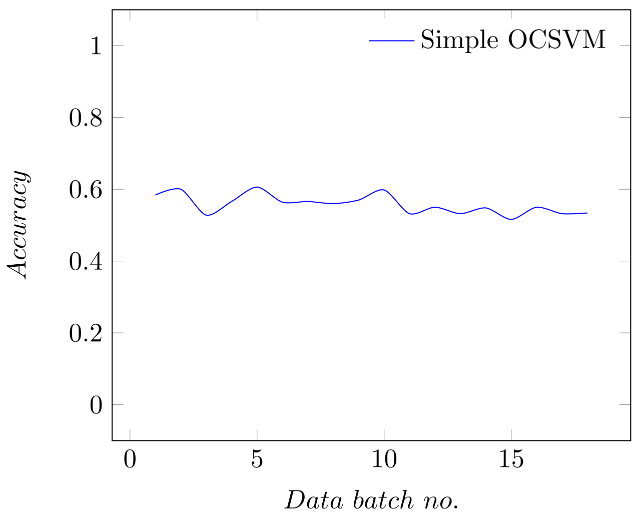


Figure 3.3: One-Class SVM applied to Tsymbal's et al. synthetic benchmark, T=1 (which is equivalent to T=10 in our synthetic benchmarks), K=8

We believe this to be due to the proportion of *normal*(1) and *abnormal*(−1) data being kept constant throughout the whole data set, and the overlap of the attribute ranges of the two classes during the rotation. Thus, we will need a different approach when generating concept drift for *novelty detection* as explained in section 3.3. Furthermore if we map the value of T to its corresponding value in our synthetic benchmark (described in section 3.1) we would get a drift rate of 0.0001 which is very small and is consistent with the results produced by the OCSVM when dealing with *slow drift* (see section 6.3, for T=50 and K=1). Therefore, we believe this exemplifies well the difference between concept drift in *anomaly detection* versus other machine learning techniques.

[2]The authors of the paper use the value of T as a proportion of 1000 elements in spite of the mathematical formulation presented in section 2.3

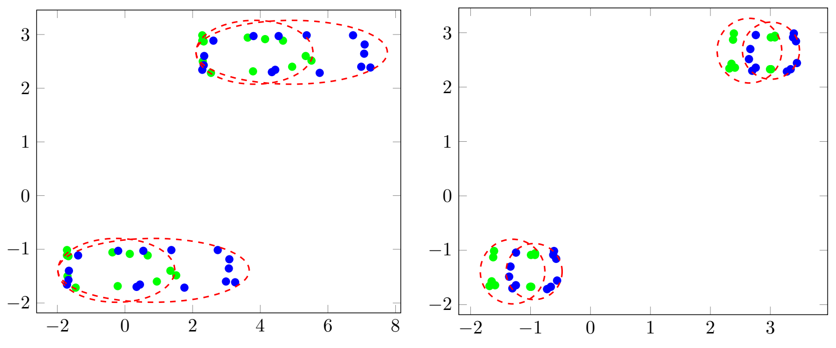


Figure 3.4: Hyperplane mapped into 2D space. The dashed line represents the boundary of the decision function

Figure 3.4 shows the regions delimited by the support vectors when applied to concept drift for OCSVMs with a *Radial Basis* kernel, by mapping the higher-dimension hyperplane back to a 2D representation. The data represented in the figure has two attributes corresponding to the horizontal axis and vertical axis. The green points represent the support vectors of the model trained on the $\mathcal{D}_1$ batch, whereas the blue dots represent the support vectors of a model trained with *drifted* data $\mathcal{D}_2$. Figure 3.4a deals with the case where one of the attributes drifts by *increasing* the *range* of values it can take. We can therefore notice how the boundary (represented by a dashed red line) expands with the drift. This correspond to the aforementioned Case 1 of the concept drift, where the initial classifier would lead to an increase in the number of *false positives*. Figure 3.4b on the other hand shows the case where the *range* of values *shifts*. This scenario incorporates both Case 1 and Case 2 from above, thus leading to both an increase in false positives as well as false negatives, when using the OCSVM model trained with $\mathcal{D}_1$ on $\mathcal{D}_2$ data. It is important to mention that the *range shift* used to create the concept drift is different from that of *concept shift* described in section 2.3 by Wang et al. In fact, shifting the range of the attributes has the desired result of creating *instability* in the concept drift, without changing the meaning of the concept.

3.2 Unsupervised self-adaptive ensemble(USAE)

Our main goal is to create a *self-adaptive* and *unsupervised* solution to handling concept drift in anomaly detection when using One-Class Support Vector Machines. The high-level view of our idea is presented in Figure 3.5. We decided to group data in batches $D_1, D_2, .., D_k$ not for training purposes, but rather to reflect snapshots of the concept drift. Hence, instances are analyzed *one at a time* making our solution truly **online**.

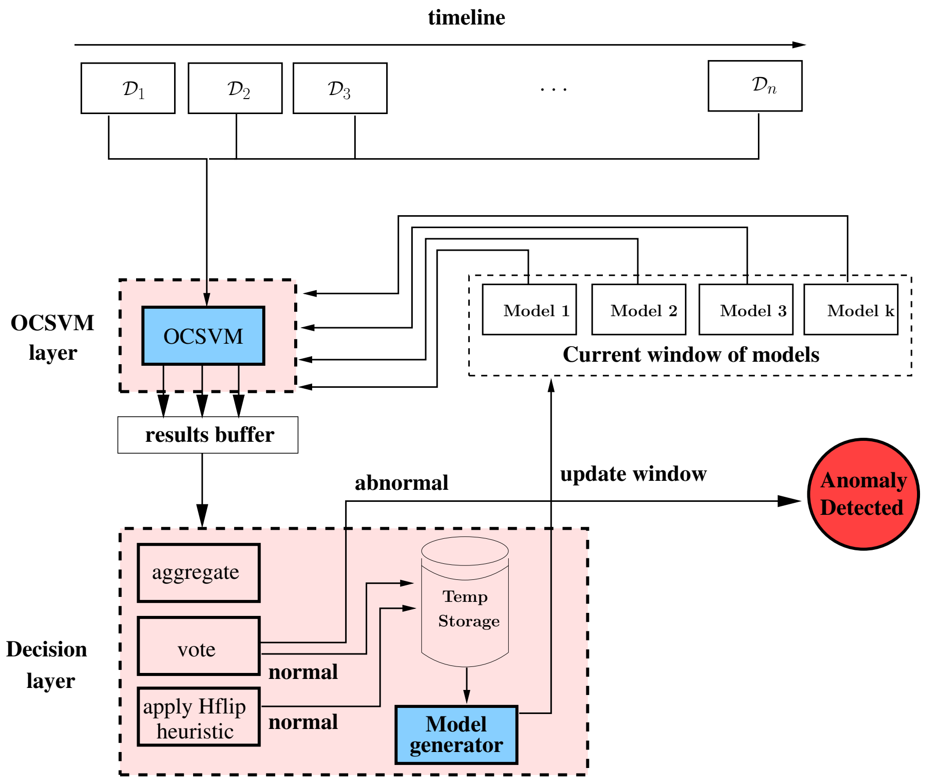


Figure 3.5: The *Unsupervised self-adaptive ensemble*(USAE) solution. $D_1, D_2, .., D_k$ represent data batches which are analyzed by USAE and used for *online* training. It is important to mention, that data are **NOT** consumed in batches, but one instance at time.

The *window* contains up to k models, where k represents the latest time instance. A new model is trained and added to the window once the *temporary store* has reached the training *batch size*(BS). The models are updated chronologically, that is, once the number of models exceeds the size of the window, the *oldest* $(Model\ 1)$ is being removed and the *newest* is added at the end of the list, thus becoming the new $Model\ k$. As mentioned in [25],[35],[14] using a window is a common approach in addressing concept drift: older classifiers ensure that *local concept drift* does not corrupt the *ensemble* of models, whereas newer models help reduce the number of false positives by incorporating the most recent information about data. Unlike in [25] our approach does not weight the models according to their accuracy.

In practical applications like network intrusion detection, the size of the window can be determined through experimentation and might be subjected to operational constraints (i.e. amount of memory available or amount of time taken to *compare* a new instance against all the *support vectors* of all k models). As opposed to Klinkenberg's et al. [15] *dynamic size window*, we prefer a *fixed size* approach in the context of NID with high-volume data, as this allows us to better estimate the performance of our DeADA architecture.

Algorithm 1 predict class of new instance

```
function PREDICT($z$)
    $predictions \leftarrow \emptyset$
    for $model \in models$ do
        if $Decision(z, model) > 0$ then
            $predictions = predictions \cup \{1\}$
        else
            $predictions = predictions \cup \{-1\}$
        end if
    end for
    $vote = majorityVote(predictions)$
    return $vote$
end function

function DECISION($z, model$)
    $x = model.supportVectors$
    $res = \sum_{i=0}^{N} \lambda_i K(x_i, z) - \rho$
    return $res$
end function
```

Algorithm 1 shows how a new instance is being classified upon arrival. The instance is compared against each model using the $Decision(z, model)$ function, and the result is stored in a list which is then used for *majority voting*[3]. If the instance is classified as *normal*, then it's being saved in the *temporary store* for use during the training phase of a new model. Note that the result of $Decision(z, model)$ is compared to 0, as opposed to having one of the standard values $\{-1, 1\}$. In fact, $Decision(z, model)$ is a version of the decision function described in Equation 2.8 with the sgn part removed. We will be re-using the absolute distance values when applying our *local distance* metric of the H_{flip} heuristic function described in subsection 3.2.1, prior to saving the instance to the local store. The *kernel* $K(x, z)$ of choice for our adaptive One-Class SVM is the *Radial Basis Kernel.*

[3]we use skeptical voting, that is an example is classified as normal if and only if $50\% + 1$ of the models have decided so

3.2.1 Local distance (LD)

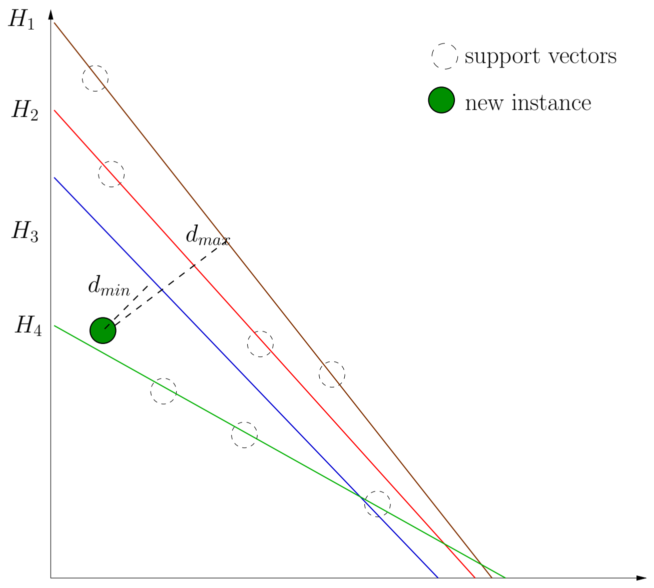


Figure 3.6: Local distance metric

The main issue with concept drift is that when *normal* data starts changing its properties and the quality of the already trained *classifiers* degrades. This leads to an increased rate of *false positives* which in the case of applications like intrusion detection can severely impact the performance of the overall system, be it a fully automated one (e.g. where data packets are being prevented from reaching the destination) or semi-automated (e.g. where a human analyst monitors potential alerts). We therefore propose a solution which tries to reduce the impact of concept drift on the *automation* of a network intrusion detection system, but which can be extended to more general uses cases of anomaly detection.

One simple approach to tackling concept drift in an unsupervised fashion would be to have a *single* or a *window* of continuously re-trained models. However, the problem arises when deciding which of the newly received instances are to be stored temporarily in preparation for training a new model. This is because we want to be able to distinguish between *true positives* and *false positives* (i.e between data which is truly anomalous and data which is considered anomalous because the concept drifted).

To address this issue we introduce a *heuristic function* based on a new variable λ which allows us to control the rate of true positives and false positives, with respect to *magnitude* and *rate* of the concept drift. This new heuristic function H_{flip} is applied after a *prediction* occurs if the majority vote decided the instance is *anomalous*, and prior to the data being fed into the *temporary store*. H_{flip} has the last word in this scenario, thus deciding whether to consider an abnormal instance normal or leave it as the majority has decided.

$$H_{flip}(x) = \begin{cases} if\ \dfrac{d_{min}}{d_{max}} < \lambda\ then\ x.label = 1 \\ \\ else\ x.label = x.label \end{cases} \tag{3.1}$$

Figure 3.6 illustrates the relation of the $\frac{d_{min}}{d_{max}}$ ratio with respect to hyperplanes $H_1, ..., H_4$ of a windowed model of size four, and we will define this relation as a *local distance metric*[4]. Distances d_{min} and d_{max} are in fact the outcome of the $Decision(z, model)$ function from Algorithm 1. The *new instance* has been classified as *abnormal* by three of the OCSVM models, and *normal* by one model. We can see from the figure that $d_{min} < d_{max}$ meaning the new instance is relatively close to H_3 however it's quite far from H_1 which in turn indicates a potentially high concept drift rate. Thus, the value λ specifies an upper bound on our *local distance* and can be correlated with the drift rate. If an *abnormal* instance is very close to the hyperplane of a model it is most likely *normal*(depending on its accuracy) especially in the case where the majority vote is not *unanimous*, and thus the λ cap would not play any major role. However, the cap is important when we have an *unanimous* vote, as it stops *false negatives* from polluting the training data for future models. As such λ can also be seen as an upper bound on the number of anomalies allowed as *noise*.

3.2.2 Training

Once the class of an incoming instance has been predicted and mediated by H_{flip} it will be stored temporarily until there is enough data (i.e. the batch size has the desired number of elements) for a new model to be trained. The training phase could be performed *offline* and in an independent manner, therefore there is no immediate need for accelerating this phase. Once the new model has been generated the window list is updated by inserting it at the end of the list and then removing the first one(if there are more than k).

3.3 Generating the synthetic benchmarks

As mentioned at the beginning of section 2.3 the rotating hyperplane approach shown in [35][11] does not impact the accuracy of the classifiers and does not lead to their degradation, since it is intended for simulating concept drift in binary classification problems (SVM or otherwise) and not for *novelty detection.* However, we construct an equivalent synthetic benchmark using a similar approach.

Changing the orientation of the One-Class SVM hyperplane can be done by altering the range of values for individual attributes of a data instance(see Figure 3.4). We therefore decide to use the method of *shifting* the ranges as this will generate data, some of which will be classified as either *false positive* or *false negatives*, rather than just false positive as for the range *expansion* case.

[4]The term *local* comes from the metric only being valid for the current instance under the current windows, both subsets of a greater set of values

It is important to understand the difference between having data which are wrongly classified as abnormal and actual anomalies. In the context of many applications including network security, anomalies can never become normal. Should that be the case, it is most likely because it was a *false positive* and the OCSVM model did not have sufficient training examples to incorporate that information in its generalization. As such we reserve the range $[0, 20]$ to represent anomalous values of the attributes. This way we avoid generating abnormal data which at some future time might be considered normal and wrongly represent the concept drift, thus "helping" the anomaly detection algorithm to perform better.

For every attribute $a_i, i < d$ where d is the number of attributes of the data, we define the amount of shift as $shift_i$. We define a parameter K to handle the *magnitude* of the drift. The magnitude of the drift is given by the number of attributes which are subjected to range shift. Thus, $\forall i, 1 \leq i \leq K, K \leq d$ we update $shift_i$ of a_i with the quantity described in Equation 3.2, where $direction \in \{-1, 1\}$, T represents the number of instances after which the concept drifts, N is the total amount of data to be generated and $\frac{T}{N}$ is the *drift rate*. While creating a *normal* example, *direction* has a 0.1 probability of becoming -1, but is reset after every example.

$$shift_i = shift_i + direction * \frac{T}{N} \tag{3.2}$$

When creating a *normal* data point we first choose a discrete random value between $[0, 50]$ added to $shift_i + 20$, to avoid overlap with the *anomalous* range. Initially $shift_i = 0$ and thus all the attributes have the same range. After ever generated example $shift_i$ gets updated for $\forall i, i < K$. *Abnormal* data are created with a 0.3 probability by randomly setting one of the attributes a_i to a discrete random value in the $[0, 20]$ interval. The rest of attributes are set in the same way as for a *normal* data point. In both cases we divide the random value by 70 to avoid the scenario in which numbers could grow past the representational range of the CPU. We deliberately avoid normalizing the data in the $[0, 1]$ interval to better simulate a real scenario in which, because of the potential drift, true ranges are not known, as it would require future possible values to be known at the current time. However, because SVMs are not scale invariant all the attributes of the *normal* data can only choose random values from the $[0, 50]$ interval, so each attribute has the same *weight* in the decision. We create several data sets of 10000 examples with varying values for the K and T parameters, to be used in section 6.3.

3.4 Summary

We began this chapter by describing the concept drift problem in anomaly detection, what it means with respect to the separation hyperplane of OCSVM, and assessed its impact on classification. We then explained the difference between the *rotating hyperplane* problem [35] and concept drift.

In the subsequent sections we presented our own solution to addressing concept drift, by creating a novel algorithm based on an *unsupervised self-adaptive ensemble(USAE).* We then described the *local distance* metric defined in terms of λ and the distances between *a new point* and the *hyperplanes* of the models in the ensemble.

Finally, we gave a solution for creating a synthetic benchmark that would allow us to measure and test USAE. The validity of our benchmark can be seen in section 6.3 where a *Simple* classifier degrades in time (as we would expect), as opposed to the rotating hyperplane problem, shown not to be a good benchmark for concept drift in anomaly detection.

Chapter 4

DeADA - a dataflow engine architecture

In this chapter we present DeADA a dataflow architecture designed as **end-to-end system** capable of performing **live network analysis** using the USAE algorithm for detecting anomalies under concept drift. The main points discussed are as follows:

- We first demonstrate that a synchronous set-up is ineffective in a live environment and highlight the need of a scalable solution. We propose two different techniques for addressing the issue and define a **Drop Rate** metric, for measuring the amount of data that needs to be skipped so the system would not get overloaded. The approach is also described in the evaluation section (see section 4.1 and section 6.2).
- We then describe the **DeADA** architecture, and focus on the scalability of the design by looking at different levels of parallelism, with a focus on the *OCSVM layer* as that is the bottleneck. Being a dataflow architecture **DeADA** can be implemented on different architecture, such as CPU, FPGA, GPU, etc. However, for the current project we will focus on CPU and FPGA (see section 4.2).
- We finish the chapter by describing an implementation targeted for FPGA using Maxeler's MaxJ and MaxCompiler tools (see section 4.3). The implementation is evaluated and compared against a CPU targeted implementation in section 6.4.

4.1 Bottlenecks

Since we are trying to achieve real-time anomaly detection in computer networks, the first step is to identify the bottlenecks when dealing with a *serial*, but pipelined system. Thus, we create a simple setup where one *Receiver* thread receives a series of packets and *enqueues* them for processing by the *OCSVM* thread. The queue between those two threads is capped at 6000 elements as to simulate memory constraints. The total number of Support Vectors used in the

OCSVM computation is 20000 with 20 attributes each, equivalent to having an ensemble of 4 models with ≈ 5000 support vectors/model.

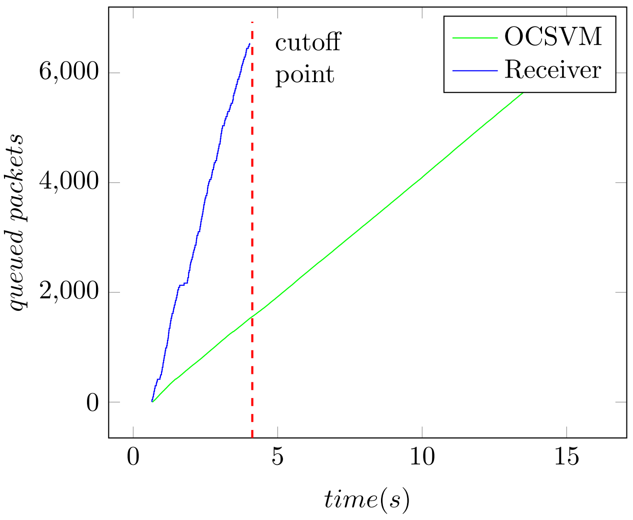


Figure 4.1: Packet processing times for a receiver and for the classifier

For producing the results we used a data set with 483945 packets(totaling 72MB) captured over a period of one day of normal browsing. However, the data set contains a large number of TCP segments resulting in 80855 re-assembled packets. Features extracted from the packets are then received over a period of $\approx 31.44s$, being the equivalent of an $\approx 18Mbps$ throughput (see Equation 4.1) The *arrival* rate is ≈ 2571.95 $packets/s$, whereas the *classification* rate is ≈ 461.36 $packets/s$. Figure 4.1 illustrates how after a few seconds the *receiving* queue gets filled (no more packets will be received), while the *OCSVM* node is still computing.

$$Throughput = \frac{TotalFileSize * 8}{Time}(Mbps) \tag{4.1}$$

4.1.1 Mitigating the lack of speed

At this point, there are several options one could undertake in order to avoid the scenario where the application crashes because it runs out of memory, and thus leading to no classification at all.

First option would be to skip the analysis of four out of five packets. This might be appropriate for situations when a *pessimistic* approach is employed to increase our chances of detecting an attack even when not all the information is available (i.e. the attack might happen over multiple packets, but we only analyze a fifth of them).

Second option would be to skip packets until the queue has been emptied. The performance of this solution depends on the memory capacity and on the intended application. We believe this might be appropriate for networks were attacks/anomalies occur in a repeated (either regular, or irregular) manner. However, the main issue is that attacks might not be spotted in time (e.g. a malicious application sends some data every two days; it might be that when the intrusion is detected, the attack has already finished).

We prefer the first option as it provides continuous analysis of data, which is to be desired in most scenarios. Therefore in Equation 4.2 we define a way of measuring the performance of the system in terms of the *drop-rate* of packet analysis. The drop-rate directly affects the accuracy of the *USAE* method as it misses packets, and thus the lower the drop-rate is, the better the performance of *USAE* will be when implemented in network systems with high volumes of data.

$$Drop\ Rate = \frac{arrival\ rate}{processing\ rate} \tag{4.2}$$

To simplify the relation between the data size, the number of support vectors and the number of vector attributes we define *DataSize* as in Equation 4.3. This will enable us to measure the *DropRate* with respect to the amount of data stored in the models.

$$Data\ Size = |attributes| * \sum_{i=1}^{M}\sum_{j=1}^{S} 1 \tag{4.3}$$

4.1.2 Support Vectors and throughput

The previous example had 20000 support vectors used for classification. We therefore briefly analyze the impact of *data size* on the number of support vectors in a model. Table 4.1 shows the correlation between the two main One-Class SVM parameters ν and γ and the total number of support vectors, by using our **Heartbleed experiment** as a reference.

ν	γ	support vectors
0.008	0.0003	28
0.0001	0.0003	10
0.01	0.0003	26
0.1	0.16	261
0.001	0.16	78
0.0001	0.16	79
0.5	0.16	1143
0.8	0.16	1811
0.8	30	1807

Table 4.1: Impact of ν and γ parameters on the number of support vectors

We can see that ν has a major impact. This is due to Equation 2.10 and the interpretation that ν is a *lower bound* on the number of support vectors and an *upper bound* on the number of outliers. For example, when training a classifier for detecting the **Heartbleed** bug, we used $\nu = 0.008$, $\gamma = 0.0003$ resulting in 28 support vectors. This is to be expected as $\lceil 0.008 * 2232 \rceil = 18$ (2232 is the total number of training examples), which means there will be *at least*

18 them. However, the asymptotic behavior of the number of vectors depends on the combination of the two parameters as described in [28].Techniques for capping their numbers have been proposed in the literature [6], but they impact the performance of the classifiers.

Thus, we can give an estimate of how many support vectors to expect based on the above throughput(18Mbps) and arrival rate(≈ 2571.95 *packets/s*), for a 100Mbps network switch, if the *OCSVM* wasn't a bottleneck. The new estimated arrival rate is therefore 14283 *packets/s*. Suppose that 90% of the data are normal and can be used as training examples for a new classifier. If we were to train a new model every 10 minutes[1], the amount of data points would be $14283 * 60 * 10 = 8569800$. For $\nu = 0.008$ we will have at least 68559 support vectors.

With an *adaptive ensemble* of size 4 there will be 274236 vectors, a lot more than in the bottleneck example presented earlier. Furthermore, SVMs are well known for being able to handle data with a large number of attributes without impacting the accuracy of the models [22], thus adding another factor to the computation time.

An ensemble of size 4 with a new model being trained every 10 minutes is most likely an optimistic version as it would cover very little historical information about the *normality* of a system, a typical application having either larger sized windows (days or weeks) or several ensemble models. On top of that, modern computer networks operate at speeds of 1/10Gbps with a new 100Gbps protocol created in 2011 thus generating more data to be stored and analyzed.

With this in mind, we believe it is safe to assume that a real-world system will have quite a large amount of information to process from, even when dealing with medium (100Mbps) speed networks (i.e. with models incorporating more historical information).

4.2 The Architecture

As previously seen performance is an issue when attempting *live* anomaly detection on network traffic even with throughput as low as 18Mbps, thus justifying the need for a scalable and parallelizable solution. In order to achieve this, one can either use distributed computing (but then network latency becomes an issue) or high-performance computing techniques based on hardware acceleration (either GPU or FPGA). Thus, we have created DeADA, a dataflow architecture (Figure 4.2), designed as an end-to-end system, containing all the steps from data capturing and pre-processing, enabling us to exploit the benefits of anomaly detection in network intrusion detection.

[1]This might be a bit too often for real-world systems thus leading to an *underestimate*

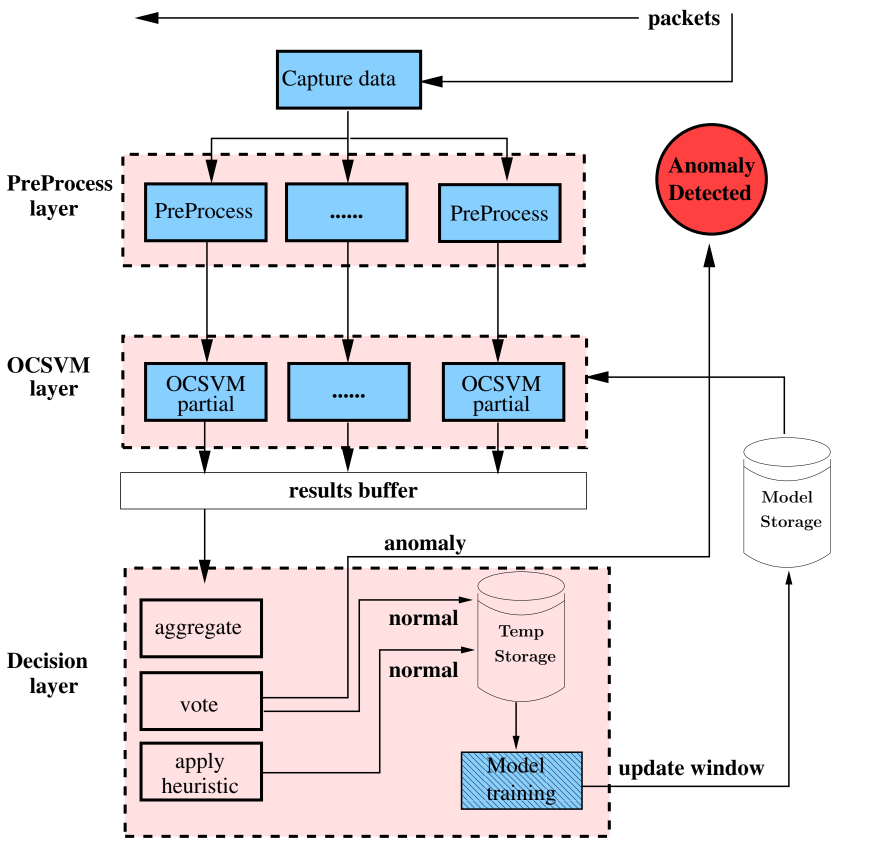


Figure 4.2: DeADA architecture. Each blue box represents a computation node, at the various level. The *OCSVM layer* and *Decision layer* are the implementation of *USAE* from Figure 3.5. Note the *Model Training* node is stripped as this phase can be done via the architecture, or as an offline and separate process.

The aim of DeADA is to sit as a standalone node in the network, possibly behind *edge routers* or *switches* and analyze incoming data, decide whether it is anomalous or not, and based on this perform certain actions (e.g. notify administrators, shut down the network, etc.). Our dataflow engine architecture was designed to take advantage of the re-configurable and heterogeneous properties of FPGAs. We will demonstrate this by mapping the `OCSVM prediction` layer to an FPGA implementation using Maxeler tools (see section 6.4). We also implement the DeADA architecture on top of our RIPPLE framework to allow us to test the concept-drift solution (see section 3.1) as well as overall *functionality* and *feasibility*, before migrating different components of DeADA to FPGA.

From Figure 4.2 we can see our design allows for parallel computations at the `PreProcess` and `OCSVM` layers. Since we based our design on *dataflow programming* concepts, the overall architecture forms a graph where *nodes* perform different tasks and in which computations are pipelined. Packets are captured from the network using tools like Wireshark or TcpDump, the required *packet fields*(e.g. headers, content, flags, etc.) extracted and sent to the `PreProcess` layer.

4.2.1 Parallelism at the pre-processing layer

The purpose of this layer is to transform the received features into a numerical representation which can then be fed into the One-Class classifiers.

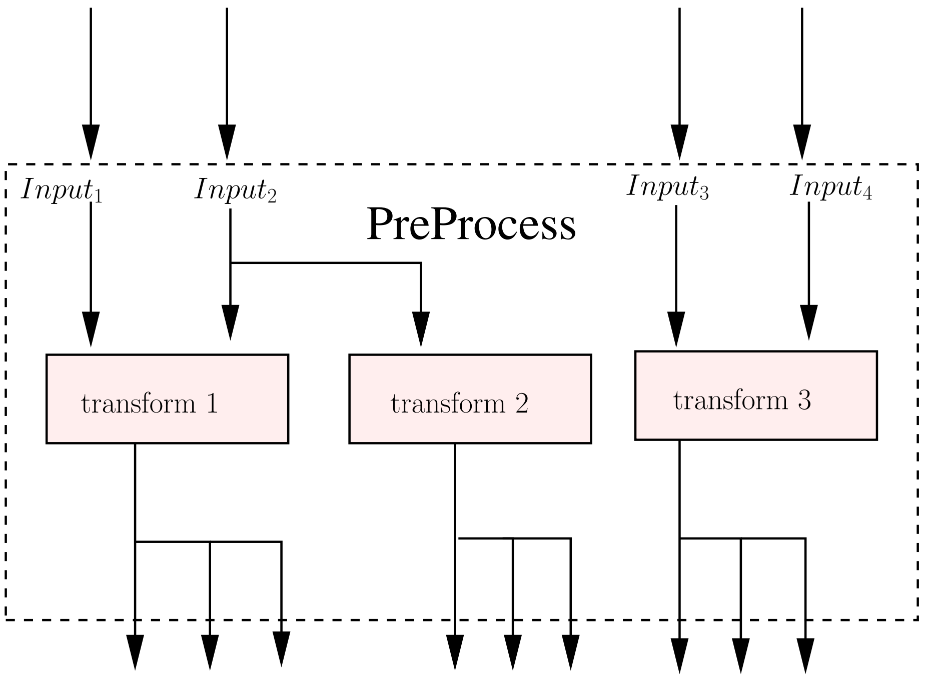


Figure 4.3: Multiple outputs

We decided to separate the `PreProcess` layer from that of the OCSVM for the following reasons:

- We want to minimize the time spent on data conversion and feature extraction at the OCSVM layer as this is the bottleneck of the NID (as seen in Figure 4.1).The functions applied at this step could be complex and might involve interactions with external applications (e.g. getting the geographic coordinates of an IP address). Thus, we prefer to apply the transformation once, and then replicate it across the *outputs* of a node as seen in Figure 4.3.
- Each of the pre-processing nodes might apply different transformation functions for the same data, but we might decide that we want to speed up the pre-processing by splitting them across different `PreProcess` nodes (Figure 4.4). Despite of the outputs being generated at different rates before being fed into the OCSM layer, dataflow computations are executed in lock step, and as long as the *order* of the generated data us maintained, the *sink* node will only perform a computation once data are available on all inputs.

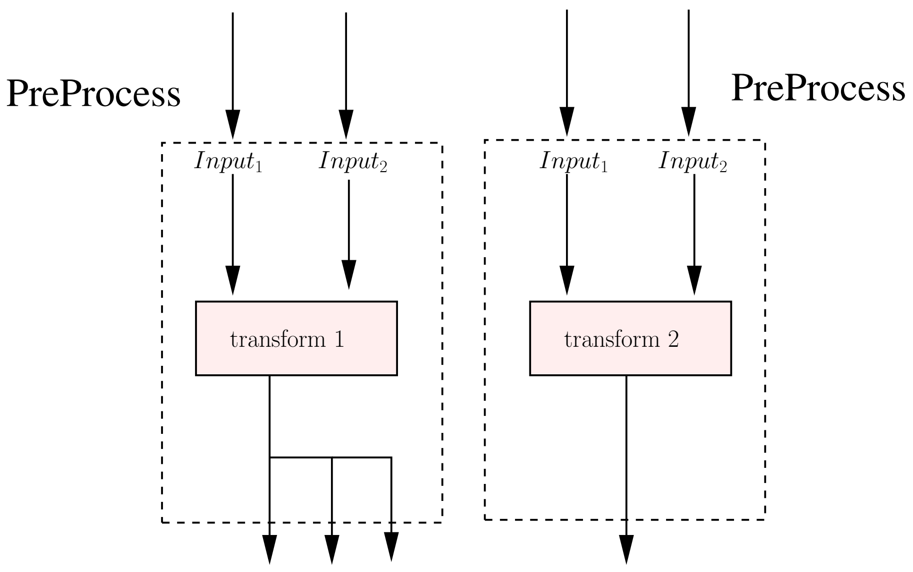


Figure 4.4: Multiple pre-process nodes with different functions

In practice a `PreProcess` node will most likely be a combination of the two scenarios presented above, however in our experiments and the Heartbleed demo we use the first case.

4.2.2 Parallelism at OCSVM layer

In our architecture the *anomaly detection* phase is performed in the OCSVM layer. For the CPU based version we use LibSVM for training the models, however when *classifying* new instances we use our *ensemble based* solution to concept drift described in section 3.1. From Figure 4.1 we have seen that the bottleneck for anomaly detection is at the *OCSVM* prediction phase. This is due to the potentially large number of support vectors resulted from training classifiers on network data.

It is therefore imperative that we *parallelize* this step to make our approach feasible for practical applications.

Data are fed into an `OCSVMpartial` node from the `PreProcess` layer in numerical format. Figure 4.5 shows how to parallelize the computation so that every node handles the support vectors from a particular model of the ensemble. The results are then put in the *results buffer* and aggregated so the decision function from Equation 4.4 can be applied for each of the models. The resulting predictions are then used for majority voting and in applying the H_{flip} heuristic.

$$decision(z) = \sum_{i=0}^{N} \lambda_i K(x_i, z) - \rho \tag{4.4}$$

In Figure 4.5, 1 *to* M_1 represents the range of Support Vectors(SV) which will be processed by the first `OCSVMpartial` node, where M_1 is the total number of SVs stored in Model 1. Support vectors from Model 2 are split into two parts: the first part 1 *to* k is fed into the second `OCSVMpartial` node whereas the second part k *to* M_2 goes into the last node.

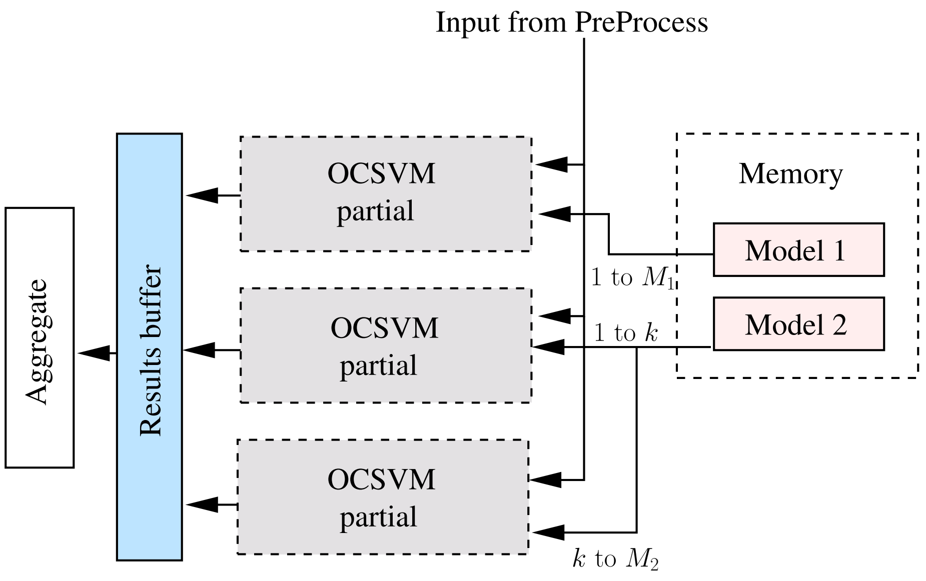


Figure 4.5: Multiple OCSVM nodes with their own models

Once the *temporary storage* has been filled or the required number of training examples reached, the classifier ensemble will be updated with a newly trained model. In our experiments we use LibSVM's default `svm_train` function and `svm_model` to store the classifier.

4.2.3 FPGA mapping of OCSVM

Before mapping the OCSVM layer to FPGA it is important to understand the structure of the data and the structure of the inputs. Support Vector Machines have the advantage of being able to cope with a large number of attributes (technically no upper bound), when compared to other machine learning techniques like Neural Networks, or Radial Basis functions[22].

However, some of these attributes may or may not exist for some instances, thus many implementations store the support vectors in sparse matrix format as $[index, value]$ pairs, along with their corresponding λ coefficients. Similarly the incoming instance is formed of $[index, value]$ pairs, where *index* is the *ordinal* of the attribute (i.e. even if some of them might be missing, the number of *possible* attributes is always fixed). Also, for both the instance and the SVs the pairs are stored in increasing order of their indices. In our implementation we have chosen the Gaussian Radial Basis for the SVM *kernel trick* as shown in Equation 4.5.

$$K(x, x') = exp\left(-\frac{||x - x'||_2^2}{2\sigma}\right) \tag{4.5}$$

The *kernel* computation is a time consuming step in a `OCSVMpartial` node, as it has to be applied for every support vector assigned to it. Storing the data in sparse matrix format does help up speeding up the process, however, it is not enough as seen from Figure 4.1.

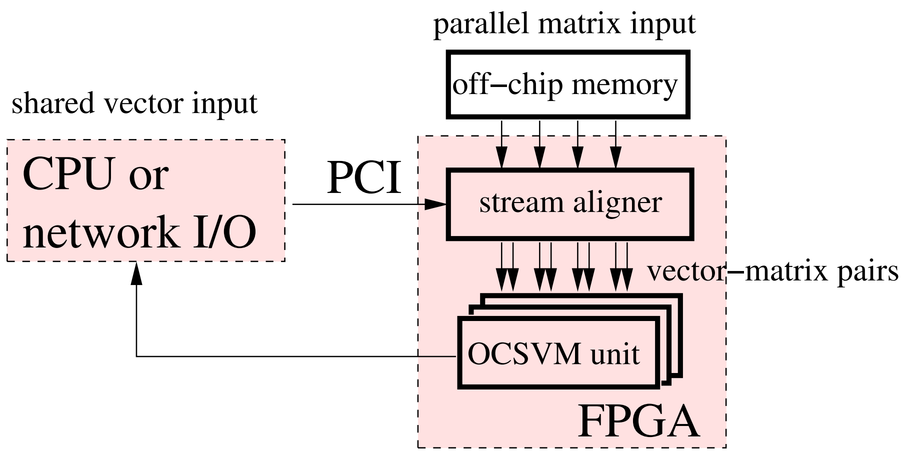


Figure 4.6: OCSVM node architecture

Figure 4.6 represents the equivalent FPGA implementation of Figure 4.5. The *shared vector input* is the *instance vector* for of $[index, value]$ pairs, and is received from the `PreProcess` layer by the CPU side of the accelarated implementation. Data are then fed into the *stream aligner* via PCI, while support vectors are loaded from the off-chip memory (with large storage capabilities) via the memory bus. The purpose of the stream aligner is to ensure the indices of the SV and of the instance match when calculating $||x - x'||$ from Equation 4.5, since we are using the sparse matrix format. The results are then put on an *output buffer* and sent back to the CPU. At the CPU side results are aggregated and decisions made using Equation 4.4. The sum from Equation 4.4 is not done of the FPGA to reduce the logic required to keep track of which support vectors, from which model, are being used during the computation, but instead increase the on-chip space to accommodate more OCSVM units.

4.3 Maxeler implementation

In order to evaluate the performance of our proposed architecture we implement the design from the previous section using Maxeler tools.

Figure 4.7 illustrates how the equivalent OCSVM design from Figure 4.6 looks when implemented on the Maxeler platform. We will call the grouping of a *stream aligner* and a *DeADA kernel* a **DeADA unit**. The dataflow diagram has a **replication factor** of 4(i.e. there are four DeADA units). Information is passed *in* and *out* of the Maxeler kernels via PCIe or memory bus. Figure 4.8 contains the code from the manager showing how various components are linked together for a variable replication factor.

The **ROM kernel** block has the purpose of replicating the $[index, value]$ pairs of the *Incoming Instance* and store them in the **on-chip** memory (ROM). The kernel will continuously produce the stored pairs, resetting every time a $K(x_i, z)$ computation has finished (x_i represents the support vector, z is the analyzed instance).

The **RAM kernel** block reads *bursts* of data from the **off-chip** memory (RAM). In Maxeler terms the RAM is known as Large Memory (LMem) and

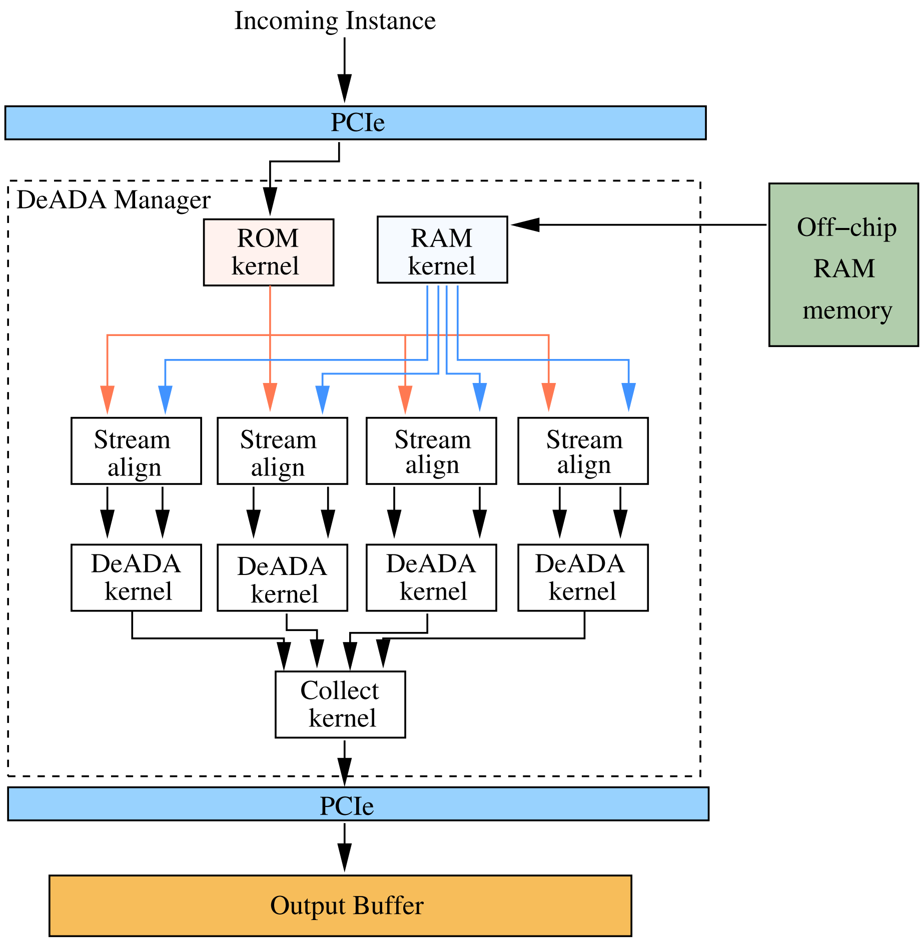


Figure 4.7: Maxeler implementation for the FPGA architecture

has storage capacities that can reach hundreds of GBs. The chunk is then split and distributed to the DeADA units.

The **Stream align** block is a *manager state machine* which *synchronized* the data received from the ROM and RAM kernels, based on the value of their indices.

The **DeADA kernel** block then performs the computation described in Equation 4.5, after which the results are sent to the *output buffer* via PCIe.

Collect kernel aggregates all the outputs from the DeADA kernels in order to serialize them via a single output stream. This is necessary due to limitations of the Maxeler platform with respect to the number of input/output streams.

In general, input data (both PCI or memory) has to be padded because transfers occur in bursts. On a MAX3 card, data read from the memory is read in chunks of 384 bytes, while PCIe output and input streams need to be multiples of 16 bytes, and aligned to a 16 byte boundary.

```
for(int i=0;i<replication;i++){
        sms.get(i).getInput("instIndex") <==
            kmROM.getOutput("instIndex"+i);
        sms.get(i).getInput("instData") <==
            kmROM.getOutput("instData"+i);
        sms.get(i).getInput("svmIndex") <==
            kmRAM.getOutput("svmIndex"+i);
        sms.get(i).getInput("svmData") <== kmRAM.getOutput("svmData"+i);
        kernelBlocks.get(i).getInput("inv") <==
            sms.get(i).getOutput("instOut");
        kernelBlocks.get(i).getInput("svv") <==
            sms.get(i).getOutput("svmOut");
        addStreamToCPU("output"+i) <==
            kerneBlocks.get(i).getOutput("res");
}
```

Figure 4.8: Sample code for connecting the blocks. *sms* is a list of manager state machines and *kernelBlocks* is the list of DeADA kernels

4.3.1 Fixed point arithmetic

The CPU version of the architecture uses *single* or *double* precision for the computations. However, when mapped to FPGA floating point operations have a latency greater than one cycle. This means the FPGA would have to run for more cycles than necessary, leading to suboptimal performance.

In order to address the issue we have implemented the computations using *fixed point arithmetic* with the results then being casted to the required precision. Operations involving fixed points can be done in one cycle (however when building for hardware the pipelining factor has to be adjusted by calling `optimization.pusPipeliningFactor(0)`).

Empirical results based on our synthetic benchmark have shown that for single precision, a fixed point with 8 bits for the integer part and 24 for the fractional part was sufficient. However, these numbers are application dependent and should not be taken for granted. It is important to understand the type of data we are dealing with (i.e. range of values or precision of the fractional part). For example, data normalized in the $[0, 1]$ interval will need only one bit for the integer part and the rest for the fractional one.

4.4 Summary

In this chapter we first show the bottlenecks of a very simple system performing anomaly detection on live network traffic whilst using a total of 20000 One-Class Support Vectors. As expected, the anomaly detection rate is significantly slower than the *arrival* rate, hence we propose two different approaches that could be used in overcoming the issue, and settle on using the first one.

The next section describes the relation between the amount of data and the number of support vectors resulted from training models on it, with respect to the ν and γ parameters.

We then move on and propose DeADA, a dataflow architecture with different levels of parallelism for improving the anomaly detection rate. We also incorporate the USAE algorithm into the architecture. The following steps describe a more generic approach to parallelization the OCSVM layer on FPGA, finalizing with the description of a Maxeler implementation which we will use for analyzing performance of DeADA when targeted for FPGA.

Chapter 5

The Ripple Framework

We introduce **Ripple**, a Java framework for easy modeling of dataflow programs and incremental mapping to FPGAs. Although Ripple is not one of the goals of this project, the initial framework we built in order to test DeADA proved so useful that we decided to improve the base code and expose it as a library which could be used in a diversity of applications.

The chapter contains the following key points:

- **A description of Kahn Process Networks** and their relation to dataflow programming and dataflow applications.
- **Ripple Framework** and its core components which are used to form Kahn Process Networks, for fast **prototyping** and **simulation** of dataflow designs. The core components are the lightweight `Box` threads which form the node is the KPN graph, and `BoxPins` which when linked form the edges.
- **Communication channels** as implemented in Ripple. Communication is done via *message passing* using communication queues.
- **Pre processing** as one of the strengths of Ripple. We believe this to be a very powerful feature as it allows data to be casted on the fly, at pin level, before being used in the computation and various functions to be applied automatically. Ripple supports dynamic creation of new data types and pre-processing functions.
- **Shore** is an application created using Ripple and used as a library for DeADA's packet capturing component.

5.1 Prototyping dataflow graphs

FPGA and dataflow designs are known to have a long development cycle caused by different factors as highlighted in section 2.5. Therefore we needed a solution which would allow us to test and simulate our *dataflow designs* before implementing them on a specific architecture. Dataflow programming languages such as [9], Pythonect, LabView or even VHDL exist, however, what we wanted was a programmable *plug-and-play* solution which would not have the overhead of learning a new language, and creating designs which target specific architectures.

5.1.1 Kahn Process Networks

As it turns out, Kahn Process Networks (KPNs) are what we were looking for. KPNs are groups of *deterministic sequential* processes with unbound FIFO channels forming a distributed computing model. Data are read and written in an atomic fashion, the behavior of the networked processes being agnostic to *communication* and *computation* delays. Reads from a channel are *blocking* (i.e. a process will stall if there is no input on the channels) whereas writes are *non blocking* (i.e. it always succeeds)[1]. Dataflow networks have been show to be a special case of KPNs[18] and are quite often used to implement streaming applications, thus making it a good candidate for creating applications using the dataflow programming paradigm. Furthermore Kahn Process Networks have been used to model FPGA implementations [21] and have also been proposed as alternatives to the traditional MapReduce design pattern [37].

5.1.2 Implementation requirements

As such, we identified the following requirements for Ripple:

- the building blocks of the framework should allow us to construct a **graph** with nodes and edges.
- each **node** should act as an independent actor.
- the **communication channels**(the edges) should have unbound FIFO queues and use message passing.
- **lock-step execution**. A node will stall if there are no data on the inputs.

5.2 Implementation

Ripple is built on top of KPN principles and Flow-based programming, where processes act as *black boxes* which can be interconnected in various ways to form complex designs and will be available as open source under an MIT license[2]. The existing source code for Ripple is based an *monolithic* application created during the 3rd year software engineering project together with Rory Allford (rda10@imperial.ac.uk) and has been re-factored to form a standalone library.

The framework has two mains building blocks: *Processing Nodes* and *Communication Channels* as described in the following sections.

5.2.1 Processing nodes

A `Box` represents a KPN processing node and is the fundamental building block of Ripple. Figure 5.1 illustrates the high-level architecture of the `Box` which is a lightweight thread operating in three steps:

[1]In practice, this is limited by the amount of memory allocated to the queues

[2]Some more code refactoring as well as integration with Akka actors is required before it will be made publicly available

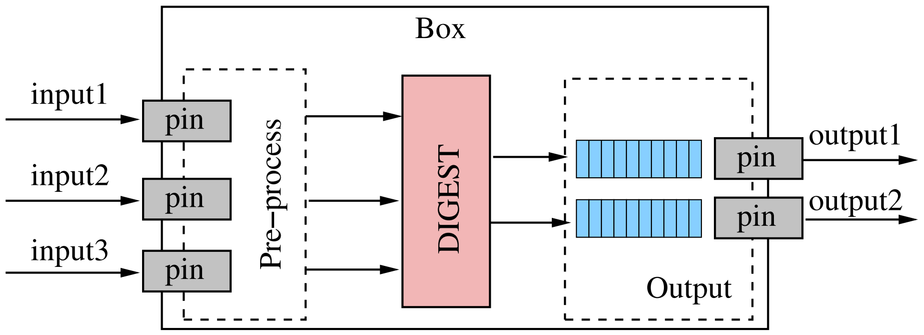


Figure 5.1: A Ripple *Box* node

1. Receive data on input pins: incoming data are received on the input pins. However, the pins themselves do not contain the actual data, only information about the type of data to be received so that it can be *pre-processed*, evaluated and converted to a *data type* to be used in the next step. Data are stored on the *communication channel*. Listing 5.1 shows how inputs are initialized during the creation of a `Box`

2. Digest the inputs: values are extracted from the incoming channels through lazy evaluation. This step is where computations should be placed, by implementing the `execStep` method of the abstract `Box`. Listing 5.2 shows how the input is *digested* and casted to an Integer (from an array of bytes not shown in here) during the execution of the `execStep` function. Notice that in order to lazily evaluate an input we need to call `getValue(sequenceNumber)`. The sequence number represents *the ordinal value* of the message on that channel.

3. Produce outputs: finally once the computation has been completed, `execStep` must return an array of objects (as many as the output pins) which are either *null* or *non-null*. The results of the computations will then be passed to the next nodes in the dataflow graph. Listing 5.2, however, does not produce any output (i.e. the `Box` node might have been a terminal node in the graph).

Listing 5.1: Setting up the pre-processor and input pins/values in a Box

```
    PPVariableTable varTable = new PPVariableTable();
    for(BoxPin pin : listInputPins()){
        varTable.create(pin.getPinName(),pin.getPinType());
    }
    /* Expressions */
    PPType[] expressionTypes = utils.
    parseTypeList(config.getConfigMulti("expressionTypes"));
    PPBase[] inputValues =
        utils.parseTuples(config.getConfig("expression"),
    expressionTypes,varTable);
```

Listing 5.2: Performing a computation during execStep, with lazy evaluation of the input values. Notice that there are no objects returned by this call, meaning the Box has no output pins

```
    @Override
    protected Object[] execStep(Object[] objects, Object o) throws
        BoxError, InterruptedException {
    for(int i=0;i<inputValues.length;i++){
            if(inputValues[i].getValue(sequenceNumber)!=null){
                System.out.println((Integer)inputValues[i].getValue(sequenceNumber));
            }
        }
        return new Object[]{};
}
```

Operating modes

The `Box` implementation provides a *hook* for dynamically changing the *state* of a node. The abstraction has two operating modes which are specific to any implementation of the `Box`: `BoxMode.Connected` and `BoxMode.Disconnected`. When the processing node is started, it will automatically check whether the input nodes (if any) have been connected and switch to `BoxMode.Connected`. When the mode is `BoxMode.Connected` a `SubMode` can be used to diversify the behavior of the processing node. Unlike `BoxMode` the generic `SubMode` is specified during the implementation (it could be an enum too, however that is required if the class is comparable).

The *hook* can be created by implementing the `startupOrSwitch` method. Four parameters are provided by the method which can be used in changing the behavior of the `Box`:

- `BoxMode newMode`: which can be either *disconnected* or *connected*. If the mode hasn't changed than it will be the same. This would only change if the internal state of the `Box`, most likely in critical scenarios.
- `BoxMode prevMode`: the previous mode, before the mode was changed.
- `<SubMode> nextMode`: is the next mode as selected via `Box.setMode(SetMode mode)`
- `<SubMode> prevMode`: similarly the parameter has the value of the mode before the change.

5.2.2 Communication Channels

Key in understanding the way a dataflow graph can be constructed using Ripple are the *communication channels* and how they work. Figure 5.2 illustrates the principles behind the Ripple *message passing* channel. On the left, we have the *output pin types* together with the *payload queues* forming a structure we call `PinSource`. On the right end we have the *input pin types* which we define as the `PinSink`. Messages are enqueued at the source level using an *unbounded* `LinkedBlockingQueue`. This means the channel is always able to receive messages, however, should the source queues be empty, the receiving end will block.

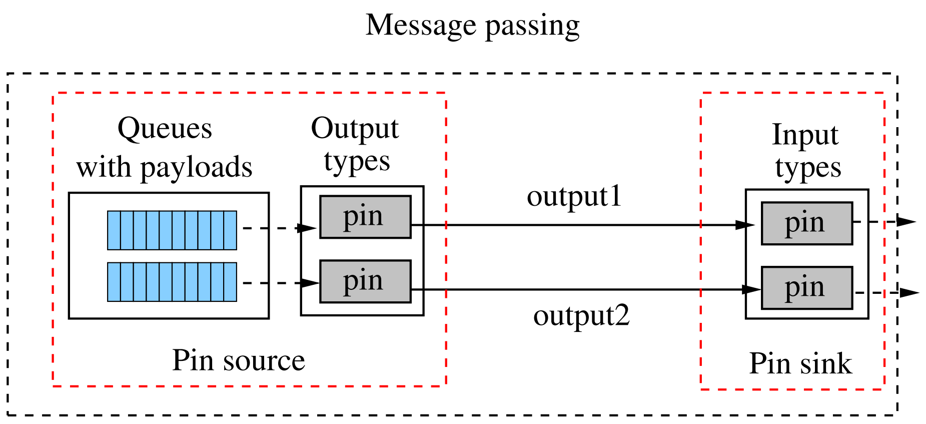


Figure 5.2: Two Ripple message passing channels, as seen when connecting two *Box* nodes, with the first node having two outputs, whilst the second node having to inputs. The *source* of the channel is on the left, whereas the *sink* is at the right end

Pins

There are 2 kinds of pins, both implementing the `BoxPin` abstract class : *input pins* and *output pins.*

1. **Output pins** are the *source* of new data, and are connected to the input pins of another computational node. The `BoxPin` contains information about the `PPType` (more details in the next section) of a pin. The combination of a `BoxPin` and the blocking queues forms the `PinSource` of the communication channel. A `PinSource` implements the `PPBase` class which allows for data to be casted automatically when `PinSink` connects to it. Figure 5.3 illustrates the relation between these different components which are further explained in subsection 5.2.4.

2. **Input pins** act as the sink of the `Box`. Similar to the output pins, `PinSink` also contains a `BoxPin` field alongside with type information, but rather than implementing `PPBase` it contains a reference to a `PinSource`. Thus, by linking these two components we obtain a queue based message passing channel between the nodes.

The output from a pin can only be connected to a *single* input pin, and similarly, an input pin can be connected to only one output pin, the channel being *uni-directional.*

5.2.3 Lockstep Execution

In our framework the KPN processing nodes execute in *lock step.* We implement this behavior in two stages:

1. Input synchronization: the `execStep()` function will only be called if all the input channels/streams have data, otherwise the executing thread will *block* until the empty streams are non-empty. Therefore, if all data are

to be processed than every input must have received an *equal* amount of information. This approach is similar to that used by the Maxeler kernels.

2. Output synchronization: all new outputs are produced during the same cycle.

Our implementation allows for *non-monotonic* sequences of data: if for some reason an inputs do not have the same *sequence* than the inputs will be skipped so that all of them reach the highest existing sequence number. In addition, the `Box` nodes can be configured to skip *null* data, by skipping the execution step and filling all the outputs with *null* values. Should that be the case, the remaining non-null input values will be lost and will not be used in any computation.

Event ordering

Inside a channel values are in *total order*, however at node level, each channel will most likely receive the data at different rates. Therefore, the amount of time a node is idling is dependent on the receiving rate.

5.2.4 Input pre-processing

We believe the *pre-processing* components which are part of the Ripple framework are a differentiating factor when compared to similar tools. Figure 5.3 illustrates the high level interaction between them during the KPN *initialization* and *runtime* phases. Pre-processing is used in two ways:

1. Casting: for mapping different types to each other, so when data are transmitted on the channel there is no need for explicit transformations at the receiving end.

2. Function composition/application: various functions can be specified in the Spring configuration files, which take as parameters either pins (identified by name) or constant expressions.

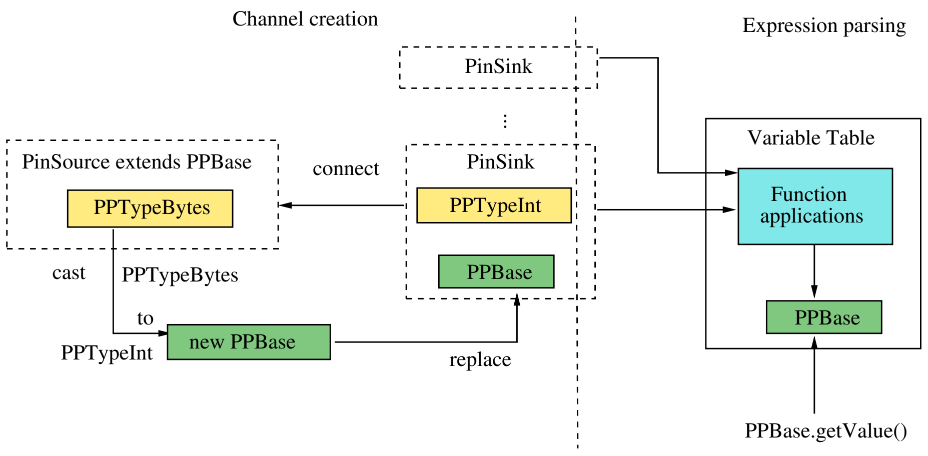


Figure 5.3: Relation between the different pre-processing components and how they play together to achieve lazy evaluation

- `PPBase` is basic abstract block used for pre-processing. From Listing 5.1 we can see the `inputValues` used during `execStep` are initialized as `PPBase`. Besides containing the `Payload` as retrived from the channel queues, it also stores a corresponding `PPType`. The type is used when connecting `PinSink` to `PinSource` for automatic casting during *channel creation*.

- `PPType` is the abstract class representing a pre-processed type, such as `PPTypeBytes`, `PPTypeInt`, etc. Each implementation of a `PPType` must override the `cast(PPType newType, PPBase value)` method. `cast()` is called every time type casting is required, and returns a new `PPBase` value with the new type. Besides the previous scenario, casting is also used when the *Variable Table* is created and expressions are parsed.

- `PPAppBase` are an extension of `PPBase` and are used for the *function application* stage inside the *Variable Table*. The function parameters are all `PPBase` and can be either *identifiers*, *constants* or the result of another function, since the result of applying a function will be `PPBase` itself.

5.3 Spring integration

Spring is a very popular dependency injection library, found in many enterprise level applications. As we believe that Ripple has the potential of being used as a tool for modeling distributed/dataflow computing, we have decided to use Spring for DI and configuration. Listing 5.3 shows how the pin configuration can be specified using `BoxConfig` and Spring beans for injection in a `Box` bean.

Listing 5.3: Configuration example for a *Box* using *BoxConfig*. The function *iprange* is applied on the pin named *ip* which is of type *PPTypeBytes* and results in a *PPTypeInt*

```xml
    <bean id="modelConfig" class="com.septacore.ripple.node.BoxConfig">
        <property name="properties">
            <map>
                <entry key="inputPinNames" value="ip;"/>
                <entry key="inputPinTypes" value="PPTypeBytes;"/>
                <entry key="expression" value="iprange(ip);"/>
                <entry key="expressionTypes" value="PPTypeInt;"/>
            </map>
        </property>
    </bean>
```

Besides specifying configuration, Spring also allows us to extend the range of types included in Ripple, as well as the set of function applications. Listing Listing 5.4 is an example of a newly created `PPAppFunction`. The name of the function (which is used in the Spring configuration) is case insensitive and is declared using Spring's `@Component(value=name)` annotation.

Listing 5.4: Example of a newly created function IPRANGE to be used as an expression/function application

```java
@Component(value = "IPRANGE")
public class IpRange implements PPAppFunction {

   private class IpRangeApp extends PPAppBase{

       private boolean checkNotNull(Object[] objects){
           for(Object o: objects){
               if(o==null) return false;
           }
           return true;
       }

       private IpRangeApp(){
           super(new PPTypeInt(), new PPType[]{new PPTypeBytes()});
       }

       @Override
       public Object applyPreprocessor(Object[] objects) {
           if(!checkNotNull(objects)) return null;
           /*
           Get the first byte and return it as an int.
            */
           ByteBuffer buff = ByteBuffer.allocate(4);
           int buffL=4;
           buff.put(--buffL,((byte[])objects[0])[0]);
           buff.order(ByteOrder.BIG_ENDIAN);
           return buff.getInt();
       }
   }

   @Override
   public PPAppBase create() {
       return new IpRangeApp();
   }
}
```

The annotation is required for Spring to be able to pick-up the new class and register in the `PPAppRegistry` (which is used in the internal calls of the *Variable Table* when parsing the `expressions` from the configuration file).

Listing 5.5 is an example of how a new type can be added to Ripple's type database by extending `PPType` and overriding the `cast()` method.

Listing 5.5: Example of a newly created type PPTypeGeneric. Note that the name of the class is used to identify the new type inside the switch statement

```java
@Component
@Scope(BeanDefinition.SCOPE_PROTOTYPE)
public class PPTypeGeneric extends PPType{
    @Override
    public PPBase cast(PPType newType, PPBase value) throws
         PPError.PPSemanticError {
        switch (newType.getClass().getSimpleName()){
            case "PPTypeGeneric":
                return value;
            default:
                throw new PPError.PPTypeError(this, newType);
        }
    }

    @Override
    public Object defaultValue() {
        return new Object();
    }
}
```

The final step, is to create an application by inter-connecting different `Box`es. A `Box` implementation should be instantiated using `ApplicationContext.getBean`. The pins should be connected using `Box.setInputPin(PinSource source)`. The `PinSource` can be obtained by calling `Box.getOutputPin(BoxPin pin)`. Once the nodes and links have been created, we add them to `RippleManager`, a wrapper around Spring's `ThreadPoolTaskExecutor`. The dataflow application can then be switched off via `RippleManager.terminateNetwork()`.

5.4 Shore

Shore is a small library we created using Ripple. We will use this library when creating our **Heartbleed** experimental set-up, in conjunction with **Shore-Probe**. Shore-Probe is a C++ program which uses the Wireshark library to capture packets, extract the packet fields we are interested in and then forward them to Shore via an Ethernet connection. Shore will then receive the information a `Box` node, named `DispatchReceiver`, which will form the entry point(data source) of our dataflow graph.

The `DispatchReceiver` can be configured in the same way as a standard `Box` node. We use the *names of the output pins* to specify which of the packet fields we are interested in. Listing 5.6 is an example configuration with which we extract two SSL packet fields.

Listing 5.6: Specifying the output pin names for retrieving wireshark packet fields

```
  <entry key="outputPinNames"
      value="ssl.ssl.record.ssl.record.content_type;
             ssl.ssl.record.ssl.record.length;
             ssl.ssl.record.ssl.handshake.ssl.handshake.extensions_length;"/>
```

The `DispatchReceiver` has three operating modes:

- **Online** mode, will tell Shore-Probe to analyze live packets on the adapter specified when starting the probe.
- **Replay** mode, will replay packets recorded in a `libpcap` file. The file is specified in the `capturefile` configuration field.
- **Recording** mode will instruct the probe to capture the packets and save them to the `capturefile` when the system is shut-down.

In addition to the three operating modes, `DispatchReceiver` can also use Wireshark display filters, to filter the packets from which fields are being extracted, by setting `filter` in the `BoxConfig` bean.

The source code for Shore-Probe can be found `https://bitbucket.org/bandrei/shore-probe`with the corresponding wiki pages at `https://bitbucket.org/bandrei/shore-probe/wiki/Home`. Shore Java library can be found at `https://bitbucket.org/bandrei/shore`, and similarly the Ripple library at `https://bitbucket.org/bandrei/ripple`.

5.5 Summary

This chapter introduced the Ripple Framework a Kahn Process Network based library, which resulted from our need to quickly prototype dataflow designs to test various theories before moving on and implementing them on FGPAs, or using tools such as those in the Maxeler suite.

We presented the various components of the Ripple Framework, with a detailed description of the `Box` nodes implemented as single standalone threads and of the communication channels between the nodes. As part of Ripple we include extensible pre-processing capabilities for input pins. Pre-processing the data reduces the amount of boilerplate code associated with casting various types of data coming through the pins while executing a computation in lock-step.

We showcased the working of the framework and the benefits of having Spring integrated with it by presenting code snippets from real applications. We finally describe Shore, an application we have built using Ripple for the purposes of capturing packets from the network.

Overall we believe that Ripple:

- can be used as an option for **fast prototyping** of dataflow designs, based on Kahn Process Network Principles.
- can take advantage of multi-core processors through its implementation of the *processing nodes* as threads.
- is easy to use, similar to a *plug-and-play* system where input and output pins are connected between nodes to form a dataflow graph.
- supports extensible functionality, such as new configurable data types, or addition of new pre-processing function via Spring Dependency Injection.

Chapter 6

Evaluation

This chapter contains the evaluation of our contributions from the previous chapters. As such, notable results are:

- **correctness validation** of the Synthetic Benchmarks from section 3.3 by using the *Simple* method with a pure One-Class SVM model, whose accuracy degrades, as expected, when exposed to concept drift. As such our Synthetic Benchmark solution can be used as an alternative to the one described by Tsymbal et al. [35]. See section 6.3.

- **excellent performance** of our *USAE* algorithm at the general level (not just for network intrusion detection) while testing on the Synthetic Benchmarks. Under concept drift, and with the right selection of λ, *USAE* is able to constantly maintain a **high level of accuracy** of $\approx$ 80% (compared to the 37% of the simple OCSVM), and at its best is between $\approx$ 5 and $\approx$ 7 percentage points below the optimal *Supervised* approach [25]. However, we believe this to be acceptable given that *USAE* is a **fully automated** online learning solution, whereas the supervised approach involves manual labeling of data. See section 3.1 for details of the algorithm and section 6.3 for results.

- **improved anomaly detection** in systems with high volume of data. When targeting FPGA, our DeADA architecture has a drop rate of 130.66, which is $\approx$ 4x lower than the best CPU implementation. As a consequence DeADA is able to maintain an average classification accuracy of 78% even when packets are dropped and data are under concept-drift, compared to the 20% low when not using DeADA. See section 6.4 and section 6.5.

6.1 The setup

For conducting our experiments we used a machine with a 4 Core Intel i73630QM processor and 8GB or RAM, the Maxeler MAX342A card with a Virtex 6 chips, and a MAIA card with Virtex 7. The machine is used for assessing the performance of the *host* code in the DeADA architecture, testing the *self-adaptive ensemble (USAE)* method and showcasing the **Heartbleed** bug.

The throughput rates in chapter 4 are obtained when running an installation of Ubuntu 12.04 Server inside a VMWare virtual environment. Capturing is done with Wireshark and our Shore-Probe.

6.1.1 Ripple Implementation

One of the exciting, but unexpected results of this project is the Ripple Framework. Because we want to compare the performance of the DeADA architecture on FPGA versus CPU, Ripple has proved very useful in implementing DeADA on the CPU. The threaded nature of the `Box` nodes allows us to take advantage of the multi-core architecture of the i7 processors.

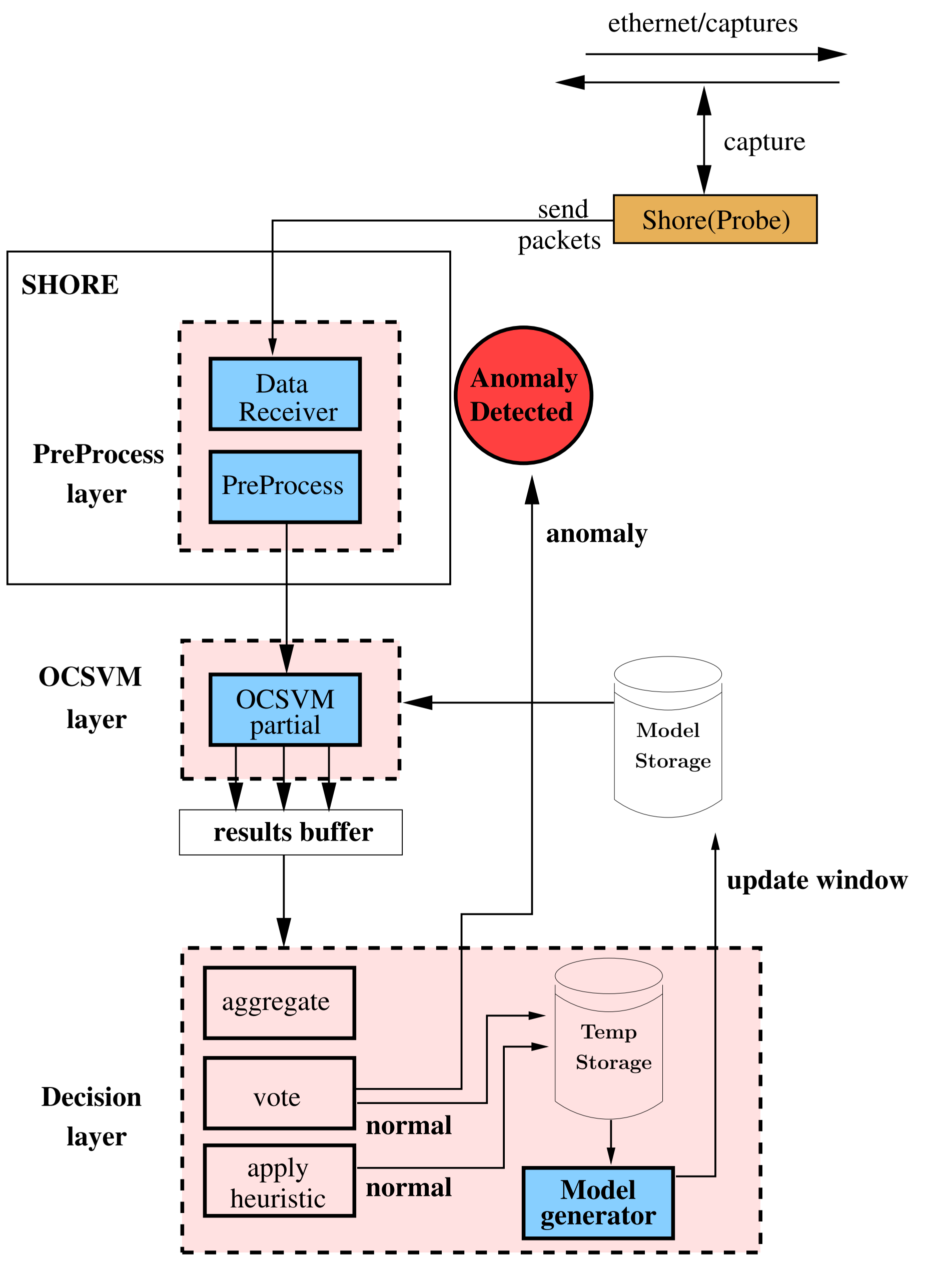


Figure 6.1: Ripple implementation of DeADA for CPU. Note that the OCSVM layer has been reduced in size so the diagram fits on the page, but it's still parallel as indicated by the multiple outputs and the *results buffer*.

Figure 6.1 represents our implementation of DeADA using Ripple and Shore. We can see that it is consistent with Figure 4.2 from chapter 4, with the only addition of the Shore component (see section 5.4) as the entry point in the dataflow graph.

We use the Ripple implementation for the following purposes:

- *Test the USAE* algorithm. Note that Shore is removed in this case as seen in Figure 6.3.
- *Simulation* (see section 6.5), as not all of DeADA components have been mapped to FPGA.

6.2 Heartbleed case-study

The strength of the anomaly detection lies in its ability to detect *unseen* patterns and *distinguish* them from the *known* ones. In the context of network intrusion detection, this means anomaly detection is able to see unknown attacks (i.e. attacks which have not been reported, and about which little is known). However, the techniques has its disadvantages as well. Because it is very good at recognizing if something is different, it fails at identifying the type of *different*[4]. Fortunately, the task of *labeling* the attacks can be delegated to different tools; if they fail, it means that either we had a *false positive* or more likely we have caught a *novel type of attack* and thus prevented a potential disaster such as **Heartbleed**.

We therefore create a *hypothetical* scenario as if we were trying to prevent attacks prior to the reporting of Heartbleed, in order to demonstrate the potential of *anomaly detection* and highlight some of the challenges which arise from using it for network intrusion detection.

6.2.1 Description of the bug

Heartbleed is a new security breach reported by Google's security team on April 1 2014. It affects versions of OpenSSL 1.0.1 starting from March 14, 2012[1] when a new feature of the SSL protocol was enabled by *default*. The new feature was an SSL request meant to check the status of the connection and keep it alive (thus the *Hearbeat* name). The Heartbeat can be identified in the SSL packet by looking for value 24 in the `Content Type` field.

The bug exploits a *bounds check* in the code, by naively trusting the values sent by the client. The relevant parts of a Heartbeat message are as follows:

- The message length: size of the message in bytes.
- The message content: formed of a *message type*, a *payload length* and the actual *payload* which is expected as a response from the server if the request is successful.

Suppose we have the following scenario: `Message length`=3, `Type`=1 and `Payload length`=16384. We therefore have the type as the first byte, and the payload size as the remaining two bytes, however there is no payload. Since the server sees the length of the payload is 16384, but does not check whether that is the case, it will allocate 16384 bytes of memory to fill the response. Because the payload is 0 bytes long, the response will contain 16384 bytes of the server's memory, thus revealing sensitive information such as cookies, usernames, passwords, etc.

[1]git commit hash 4817504

6.2.2 Methodology

We proceed to using a **Simple** One-Class Support Vector machine classifier as a solution to preventing Heartbleed. We create a scenario in which we are trying to secure our OpenSSL server against potential threats, but **without having any prior knowledge** about the structure of the attacks or the bugs they might exploit. We are interested in analyzing the requests coming from the clients on the allocated SSL port (443).

Since we lack an actual server which would receive data from different clients, we record the packets as seen from the client's perspective while accessing multiple SSL servers, and filter it out based on the port number. This is equivalent to the sever recording the incoming client packets.

As we only care about the security of the SSL protocol, we shall capture and analyze only SSL related packets. There are four types of SSL requests: *Cypher Exchange*, *Heartbeat*, *Application Data*, *Handshake*, and *Alert*. We do not want to analyze Application Data requests as they contain the actual information transmitted using the SSL channel. We chose to inspect the following protocol fields (specified in Wireshark format) as we believe an attack would be an *unexpected* combination of these:

- `ssl.record.content_type`
- `ssl.record.length`
- `ssl.record.version`
- `ssl.handshake.type`
- `ssl.handshake.length`
- `ssl.handshake.version`
- `ssl.handshake.extensions_length`

We train an initial classifier on 2232 examples with $\nu = 0.008$ and $\gamma = 0.0003$ resulting in a total of 28 support vectors.

We then setup an OpenSSL server with version 1.0.1f (which contains the Heartbleed bug) in Ubuntu Server 12.04. We then use a python script which tests the server for the vulnerability by performing an attack on it, thus generating network data as for a real situation.

Important notes:

- The packets have **no concept drift**
- The architecture has **no replication** and hence no parallelism.

6.2.3 Results

We obtain the following results:

- The *Simple* classifier is able to detect 100% of the anomalies with an overall *Accuracy* of 98% for the classified data. As investigations pointed out, the existence of the *Heartbeat* itself is an anomaly, with the recorded data set containing no such messages, since most SSL protocol implementations don't even use it.

- As explained in chapter 4 the arrival rate is $\approx 2271 packets/s$ corresponding to an $18Mbps$ network speed. However, due to the difference in the *arrival rate* and *processing rate* we notice that after $\approx$ 4 seconds the Receiver **cannot** accommodate any more data, as memory allocated to the 6000 elements buffer has been filled. Despite the high classification *Accuracy*, we were able to analyze only a fraction of the whole data, more specifically $\approx$ 8200 packets out of the 483945 recorded packets, before the system shutdown/restart (i.e. until the *cutoff* point in Figure 6.2). This is clearly unacceptable in a real-world scenario, as a system which does not work or analyze the data automatically implies $Accuracy = 0\%$.
- One of the ways to allow the system to continuously analyze data is to drop packets (see chapter 4), but the drop-rate impacts the accuracy of the classification. section 6.5 takes this experiment further, and evaluates the impact of dropping packets when using this simple setup, versus the $DeADA$ setup, highlighting the need to have lower drop-rates(which can be obtained through DeADA).

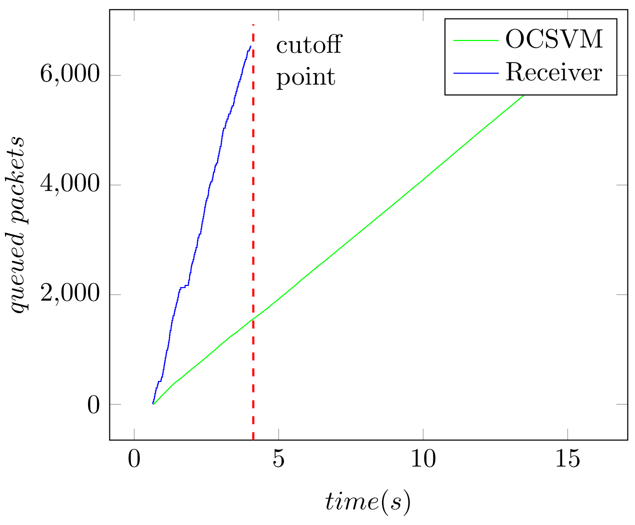


Figure 6.2: Packet processing time for a Receiving node and for the OCSVM node

Based on aforementioned results we can define the following challenges, which will be addressed in the next sections, with an overall evaluation in section 6.5:

- solving the issue resulting from the difference of the *arrival rate* and *processing* rate, and creating a system which would be able to continuously analyze data (i.e. DeADA).
- test the accuracy of the One-Class SVM model in detecting Heartbleed attacks under concept drift, which has not been covered under this scenario.

6.3 USAE evaluation

One of the main contributions of the project is our solution to addressing concept drift in the context of *novelty* detection. In order to validate our theory, we first set-up a dataflow architecture based on the DeADA architecture in Figure 4.2, but with a replication factor of 1 at both the *Pre-processing layer* and *OCSVM layer*.

We decided that replication is not necessary in testing our theory as the *USAE* algorithm described in section 3.2 has no such requirement. Furthermore the heuristic functions are applied after results are aggregated, a step which cannot be parallelized. `ModelGenerator` will perform *aggregation*, *voting*, application of the H_{flip} heuristic and will act as the temporary store for the training examples.

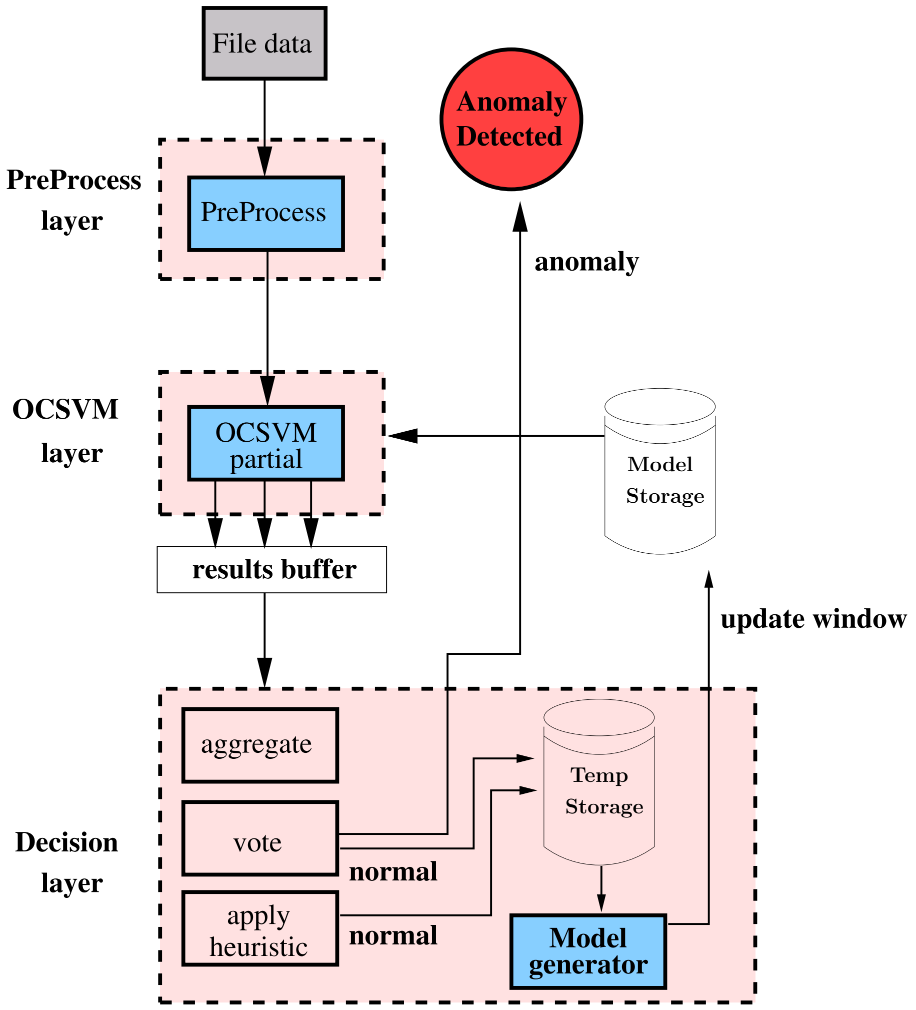


Figure 6.3: Set-up architecture used for testing the self-adaptive ensemble theory

Figure 6.3 illustrates what our set-up looks like. Note the `Capture` node is removed as we will be experimenting with our synthetic benchmarks from section 3.3, which are stored in files.

6.3.1 Methodology

Techniques used for comparison

We are interested in evaluating the performance of our approach when compared to the following techniques:

- a **Simple** classifier trained only once, and used for analyzing the rest of the data set. We train an initial classifier on the first 100 *positive* examples out of a total of 10000 available in the data sets. This is the simple, unaltered version of One-Class Support Vectors, as described by Schölkopf et al.
- a **Supervised** window ensemble as described by Parveen et al. [25], where models are removed based on their accuracy on previous data. Since in their approach data are labeled, a supervised algorithm should produce better results than an unsupervised one, as the former has knowledge not available to the latter. We shall consider this as the **optimal** solution when using ensembles of models. We will set the size of the ensemble to 4 (same as for $USAE$).
- a **Forward** feeding classifier, with periodic model update. This is an unsupervised approach which is neither ensemble based, nor does it use our self-adaptive algorithm, but where the model used for anomaly detection is always trained on data labeled by *previous models*, hence the term *forward feeding.*
- **USAE**, our own contribution as described in section 3.1

There are a total of six data sets generated using our synthetic benchmark algorithm with various values for the T and K parameters. Each example in the data set has a total of 10 attributes.

Measuring performance

For each of the four methods analyzed in this section we measure their accuracy based on Equation 6.1, as well as on a *modified* Brier score as described in Equation 6.2. Note when using $BS_{modified}$ a lower value is *better*, just as it would be with the standard Brier Score.

We do not use the *classical* formulation of the Brier Score since it is usually intended for measuring the quality of classifiers which give predictions as a *probability*, whereas a One-Class SVM returns a discrete value from $\{-1, 1\}$.

$$Accuracy = \frac{TrueNeg + TruePos}{TrueNeg + TruePos + FalseNeg + FalsePos} \tag{6.1}$$

$$BS_{modified} = \frac{1}{n} \sum_{i=1}^{|data|} f(prediction, label), f(x) = \begin{cases} 1 \; if \; prediction \neq label \\ \\ 0 \; otherwise \end{cases} \tag{6.2}$$

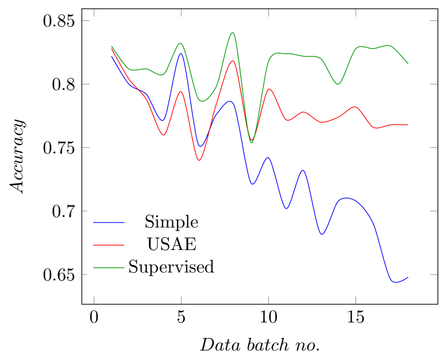
Benchmarks

section 3.3 describes a method for generating Synthetic Benchmarks specifically for use in *anomaly detection*. As such we generated several benchmarks with varying degrees of concept drift by setting the T and K parameters.

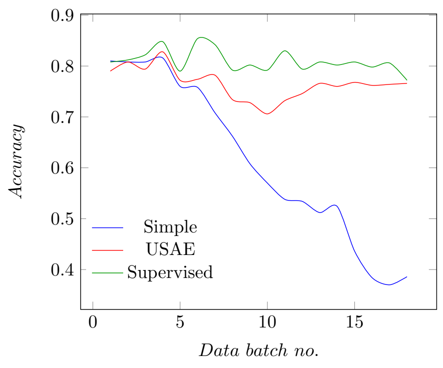
For each of the synthetic benchmarks we reserve the first 400 positive examples for training the base classifiers and the ensembles (100 examples for each model in the ensemble). We then measure the performance of the algorithms on the remaining examples (both negative and positive). Once 100 *positive* examples have been collected by the `ModelGenerator` queue, we generate a new model and update the ensemble. Each benchmark has a total of 10000 mixed examples. We measure $BS_{modified}$ and *Accuracy* in batches of size 500, resulting in ≈ 18 data points.

6.3.2 Results

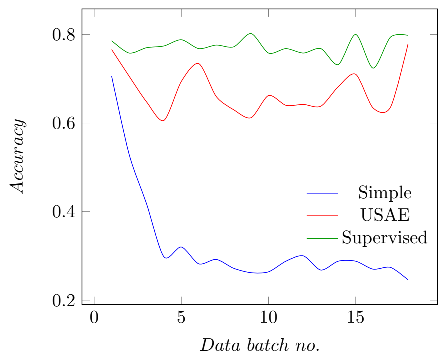
Figure 6.4 compares the accuracy results of the *USAE* method against that of the *Simple* base classifier and the *Supervised* OCSVM ensemble, for varying parameters of T and K. Figure 6.5 shows the $BS_{modfied}$ scores for the same methods. In Figure 6.4a and Figure 6.5a we can see the concept drift is not very strong.



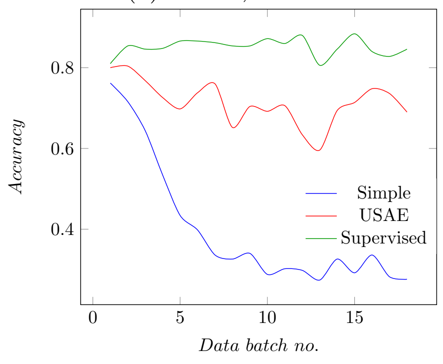
(a) T=50, K=1



(b) T=100, K=1



(c) T=500, K=1

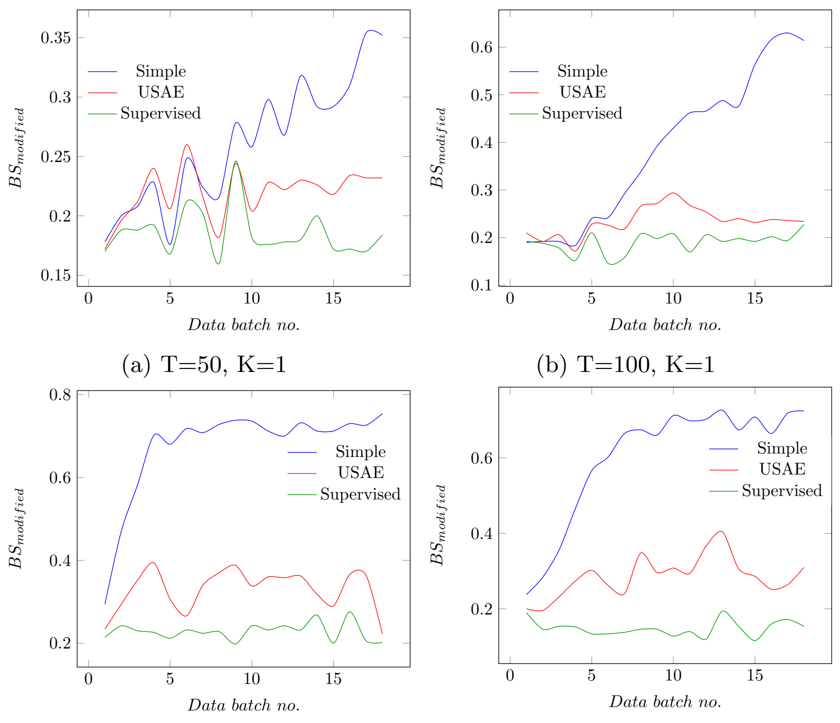


(d) T=100, K=7

Figure 6.4: Accuracy measurements for the **Simple** base classifier, the **Supervised** ensemble, and our **USAE** method.

The highs and lows in the graph are a consequence of the drift changing direction with a probability of 0.1 for some random attribute $a_i, 1 \leq i \leq 10$ (see section 3.3). We can see that in some cases the *USAE* method has an ≈ 10 *to* 15 percentage points difference in the accuracy when compared to *Supervised*, which is also confirmed by $BS_{modified}$. This is due to a poor selection of the λ parameter in *USAE*, as we set it to 0.3 which is the optimal value for the $T = 100, K = 1$ scenario.

Indeed, we can see in Figure 6.4b and Figure 6.5b that *USAE* is performing a lot better, having an accuracy almost on par with the *Supervised* method. Figure 6.4c shows the case where the *rate* of the drift is higher (i.e. $T = 500$), whereas Figure 6.4d shows the case where the *magnitude* of the drift is increased (i.e. the K attributes affected by drift).



(a) T=50, K=1

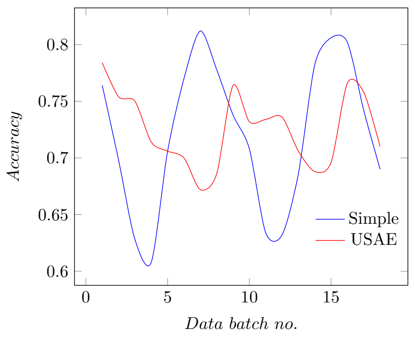
(b) T=100, K=1

(c) T=500, K=1

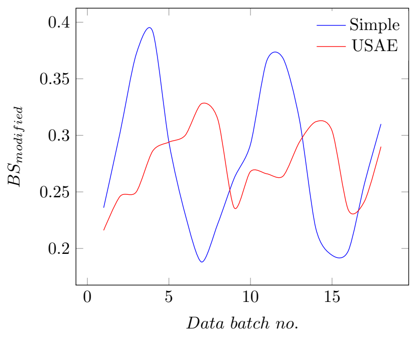
(d) T=100, K=7

Figure 6.5: Brier scores for different values of K and T

In all of the cases presented here, the *USAE* method performs significantly better than the *Simple* classifier, and is almost as good as the *Supervised* method, when the λ parameter is selected accordingly, being able to maintain an *almost* constant precision.



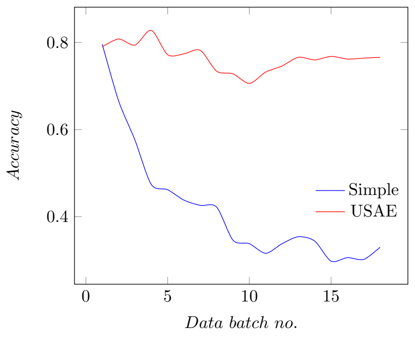
(a) Accuracy



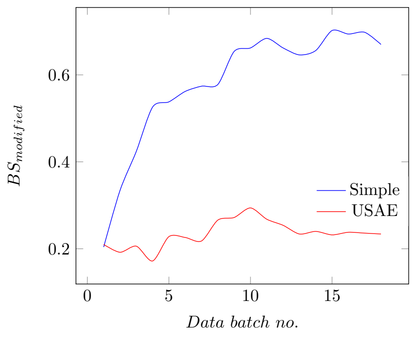
(b) $BS_{modified}$

Figure 6.6: Highly oscillating concept-drift

Figure 6.6a and Figure 6.6b consider the case with a high drift rate, and a frequent change of direction (i.e. every 2000 examples, the direction of drift is reversed for all K attributes). We can see *USAE* is in *anti-phase* with the *Simple* method. As expected, the *Simple* classifier starts performing better as data re-enters its original range and drops as it moves away from it. However, because *USAE* is an ensemble based method it incorporates historical information about the data, and thus adapts more slowly to change. In fact, for this particular example, the accuracy of *USAE* increases just as that of *Simple* starts to decrease, thus the smaller amplitude of the *Accuracy* and $BS_{modified}$ measures. This means *USAE* might not be such a good option for systems where the drift has a high *magnitude* and the direction of the drift changes very often.



(a) Accuracy



(b) $BS_{modified}$

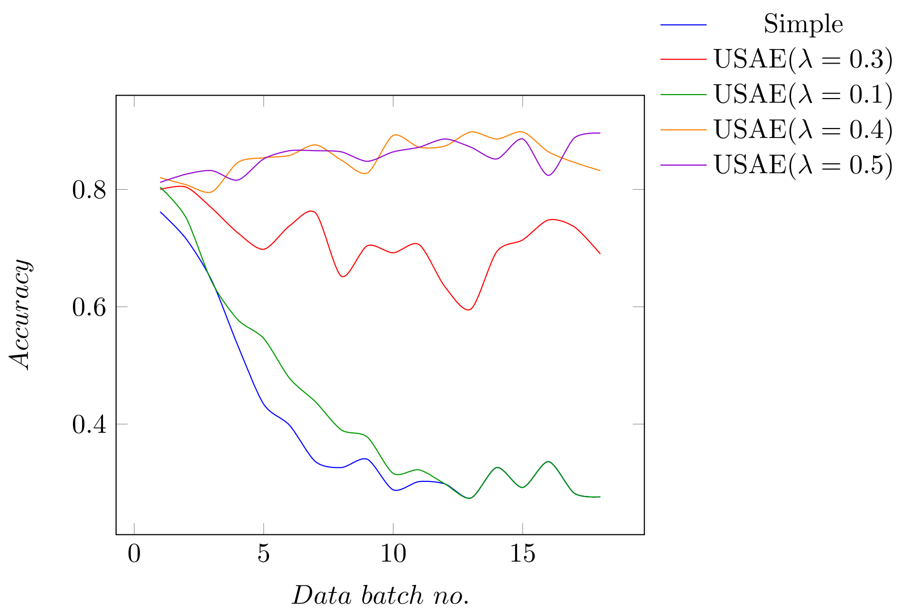
Figure 6.7: Continuously self-adaptive **Forward** classifier and **USAE**. The simple classifier is updated with a new model, once enough positive data (as labeled by the current classifier) has been gathered. We observe high accuracy for our **USAE** method, as well as very low (0.2) $BS_{modified}$ score.

Figure 6.7 compares the *Forward* feeding approach described earlier, to our proposed *USAE* method. We can see that a simple, but continuously updated classifier does not perform very well, its performance being similar to that of a *Simple* classifier used for the whole life-time of the system.

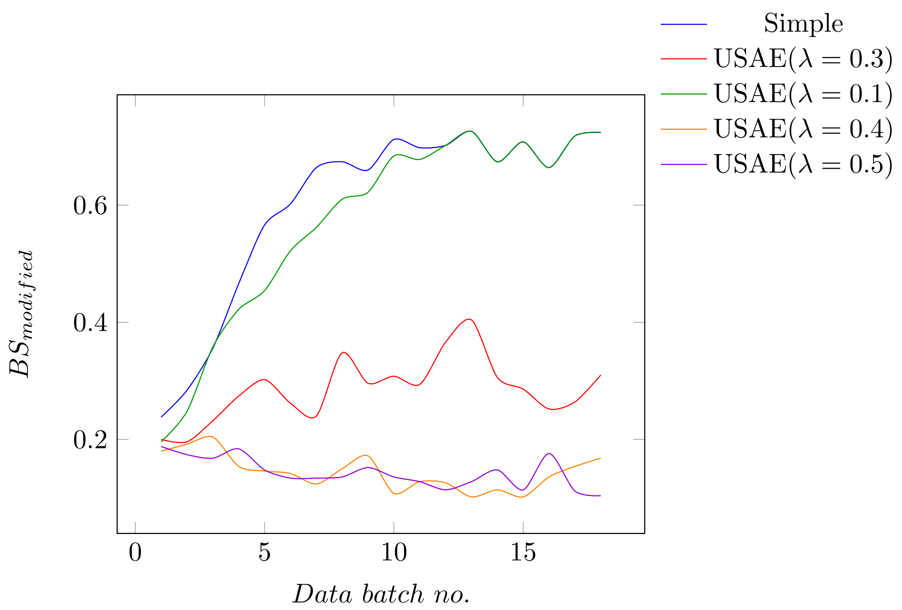
Impact of λ

We have previously seen that a general value for λ does not work well in all scenarios and is correlated with the characteristics of the concept drift. Figure 6.8a and Figure 6.8b analyze the impact of choosing the right λ. We can see that $\lambda = 0.5$ and $\lambda = 0.4$ perform best when handling concept drift with a magnitude of $K = 7$.

This is caused by data distancing itself from the separating hyperplane of the models. The further away a point is from any of the hyperplanes, the smaller the relative difference is (i.e. $d_{max} - d_{min}$ compared to d_{min} and d_{max}) and thus, the ratio $\frac{d_{min}}{d_{max}}$ gets closer to 1. Choosing the right value for λ can be done either through experimentation, analyzing historical data or by creating mathematical models for the concept drift.



(a) *Accuracy*



(b) $BS_{modified}$ score

Figure 6.8: Impact of varying λ, $T = 100$, $K = 7$ on applying **USAE** and compared to **Simple**. $\lambda = 0.5$ and $\lambda = 0.4$ perform the best in terms of both *Accuracy* and $BS_{modified}$.

		Simple	USAE($\lambda = 0.3$)	Supervised
T=50	Low	0.64	0.76	0.75
K=1	High	0.82	0.82	0.83
T=100	Low	0.37	0.70	0.77
K=1	High	0.81	0.80	0.85
T=500	Low	0.24	0.63	0.73
K=1	High	0.70	0.77	0.80
T=100	Low	0.27	0.59	0.88
K=7	High	0.76	0.8	0.81

Table 6.1: Table containing the lowest and highest accuracies of the three main methods. As we can see **USAE** has a very narrow oscillation for $T = 100, K = 1$ since $\lambda = 0.3$ was selected to handle this type of drift. The results are consistent with the figures, meaning **USAE** performs far better than **Simple** and is approximately as good as the optimal **Supervised** method.

To sum up the section:

- we have shown that our **Unsupervised Self-Adaptive Ensemble** online algorithm is far superior to that of a **Simple** One-Class SVM model/classifier or a self-adaptive **Forward** feeding version of it.
- **USAE** is *highly resilient* when facing concept drift, being able to continuously maintain the accuracy as high as **80%** (see Figure 6.4b) when a good value for λ is selected.
- despite being an unsupervised online algorithm **USAE** is almost as good as the **Supervised** method (percentage point difference in accuracy can get as low as 1), whilst being **fully automated**.
- the λ parameter allows for fine-tuning of the the **USAE** algorithm, and is related to the strength of the concept drift.

6.4 Maxeler implementation

We have chosen to map the OCSVM layer to FPGA by using Maxeler tools. The designs were compiled using Maxcompiler 2013.2.2 for MAX3 and Maia cards with a Xilinx Virtex 6 chip and Virtex 7 chip respectively. The performance metrics of MAX3 were gathered by running the design on `maxstation2.doc.ic.ac.uk` under normal load. All the designs were built for a running frequency of 100Mhz, unless stated otherwise.

We define *replication* as being the number of **DeADA units** we can fit on a chip (see section 4.3 for details).

6.4.1 Double vs Single Precision

One of the advantages of FPGAs is the ability to control the word size resulting in a more efficient usage of the chip's resources with respect to the data. One such simple, but common, optimization is the use of *single precision* instead of

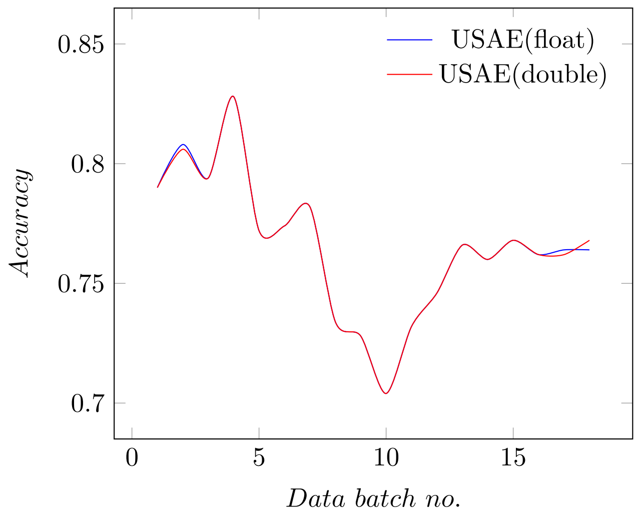


Figure 6.9: Difference in accuracy of Double Precision and Singe Precision implementations on CPU side

double precision as long as it does not impact the accuracy of the information. Figure 6.9 shows almost no difference when using floating point values in our *USAE* approach on the synthetic benchmark. We therefore decided to use a 32bit design for our architecture with *fixed point arithmetic*(as described in chapter 4) for the internal calculations and *floating point* values as inputs and outputs.

6.4.2 MAX3 resource usage

We compile the design from section 4.3 and obtain the resources usages described in Table 6.2, Table 6.3 and Table 6.4, for three main replication factors: $replication \in \{1, 48, 96\}$. Additional resources usage tables can be found in Appendix A.

LUTs	FFs	BRAMs	DSPs	component
11.03%	8.30%	8.88%	0.20%	total
0.82%	0.58%	0.94%	0.20%	kernels
10.05%	7.69%	7.94%	0.00%	manager
0.13%	0.04%	0.00%	0.00%	stray resources

Table 6.2: Resource usage for a build with replication=1

LUTs	FFs	BRAMs	DSPs	component
43.25%	30.89%	30.31%	9.52%	total
21.97%	16.79%	0.94%	9.52%	kernels
20.36%	13.97%	29.37%	0.00%	manager
0.88%	0.12%	0.00%	0.00%	stray resources

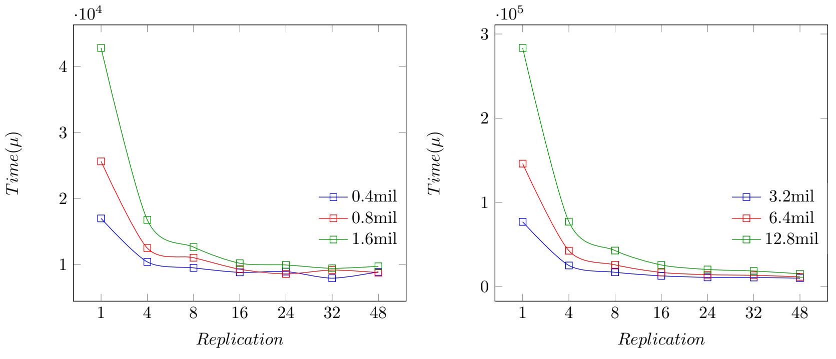
Table 6.3: Resource usage for a build with replication=48 on MAX3

LUTs	FFs	BRAMs	DSPs	component
42.86%	28.66%	49.01%	9.78%	total
24.80%	17.33%	1.40%	9.78%	kernels
17.86%	11.14%	47.18%	0.00%	manager
35.29%	24.82%	40.49%	9.78%	stray resources

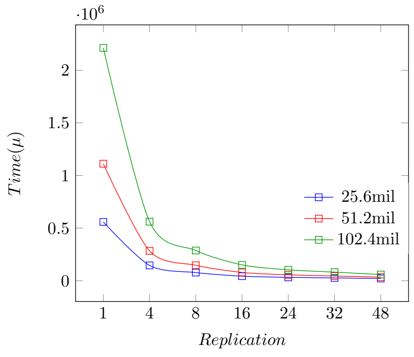
Table 6.4: Resource usage for a build with replication=96 on MAIA

6.4.3 Results

In order to create a unified measure of how much information is to be processed by the FPGAs we define the *DataSize* as in Equation 4.3. Multiple data sets were generated using the synthetic benchmark algorithm and include a variable number of support vectors: $2^i * 10000, 1 \leq i \leq 9$, with 20 attributes each.



(a) Small data size (b) Average data size



(c) Large data size

Figure 6.10: Time taken to compute data of different size on an architecture with varying replication levels.

Figure 6.10 shows the performance of our design as a function of the data size and the replication factor. As expected $replication = 48$ is able to maintain the lower bound imposed by the *initialization time*, whereas for *large data* it is asymptotically close.For $replication = 48$ we measured the average initialization time (i.e. by providing data for only one computation cycle) to be $\approx 8325\mu s$, while for $replication \in \{1, 24\} \approx 8125\mu s$

We measure the performance of both MAX3 and CPU implementations with respect to the *Drop Rate* described in chapter 4 and Equation 4.2.

Unfortunately in the current setup there is no connection between the *data capturing* nodes, and the FPGA accelerated implementation, meaning there is no continuous stream of newly arrived packets on which to perform analysis and obtain the processing rate. Data presented in Figure 6.10a is measured by analyzing a single *instance* against varying data sizes, resulting in the *processing time per instance*. However, we can use the *arrival rate* and the inverse of the *processing rate* from the CPU side, to obtain the drop rate. Equation 4.2 and Equation 6.3 are therefore equivalent.

This substitution allows us to obtain the drop rates for the FPGA implementation using an arrival rate of ≈ 2571 $packets/s$ as discussed in chapter 4.It is also important to note there is a *one-time* initialization cost of the FPGA which varies for different setups (i.e. different replication factors) and is present in each of measurements, Hence, the initialization time will be subtracted when calculating the *DropRate*.

$$\begin{aligned} Drop\ Rate &= (analysis\ time) * (arrival\ rate) \\ analysis\ time &= \frac{1}{processing\ rate} \end{aligned} \tag{6.3}$$

Drop Rates						
Data Size(M)	CPU		MAX3			
	1 core	4 cores	r=1	r=24	r=48	r=72
0.4	4.16	1.46	22.68	1.96	1.35	0.8
0.8	8.24	3.14	44.90	1.06	1.07	0.77
1.6	16.30	6.15	89.10	4.54	3.15	1.02
3.2	33.17	11.52	176.93	7.34	4.38	3.81
6.4	65.05	22.96	354.71	15.57	8.65	6.22
12.8	129.13	45.56	707.99	31.48	17.78	13.45
25.6	313.15	106.99	1415.89	62.69	33.83	38.31
51.2	560.13	219.18	2838.63	123.59	67.16	58.40
102.4	1367.55	469.16	5667.95	245.91	130.66	118.61

Table 6.5: Drop rates on CPU and MAX3

We can see that for small data (0.4 and 0.8 million) the drop rates for FPGA are quite low and are comparable to that of a 4 core CPU. We refrain from assessing whether it is better or worse, as measurements with this little data are very sensitive to the accuracy of our *initialization* time estimates. On the other hand, we can see the MAX3 implementation is able to maintain the *DropRate* at a lower bound of ≈ 1 as data increases from 0.4 million to 0.8 million, whereas the *DropRate* doubles on a 4 core CPU.

6.4.4 Further analysis

Despite the fact that our design consumes only $\approx 40\%$ of the FPGA chip for $replication = 48$, increasing the number of *DeADA units* would not result an increase in performance (as we can see from $replication = 72$). The memory bus width of the MAX3 card is a limiting factor as it is only 3072 bits wide, which means in the optimal use case we have at most 48 units capable of consuming data at the same time, as resulted from Equation 6.4.

$$Replication_{max} = \frac{Memory\ Bus\ Width}{64} \tag{6.4}$$

However, we can give performance estimates for implementations on the Maia card which has double the memory bus width(6144 bits) of MAX3. Based on Table 6.5 we can estimate the *DropRate* for $replication = 96$ to be 69.5, $\approx$8x better than the best CPU implementation.

Further work can be done in exploiting the remaining on-chip resources (possibly by incorporating the *PreProcess* layer on the same chip) and connecting the accelerated OCSVM layer to the data receiving/pre-processing layer in order to measure performance on continuous streams of incoming packets.

Finally we summarize the results:

- we have successfully obtained a 4x decrease in the *DropRate* for the FPGA implementation when compared to the best(4 core) CPU implementation, with a further possible improvement of up to 8x.
- we successfully fit our FPGA design of the OCSVM layer on chip, and increased the parallelism until stopped by the DRAM memory bus width.
- we highlighted the trade-off between lower *DropRates* and *DataSize*

6.5 DeADA and Heartbleed

We have previously seen in section 6.2 a case study of Heartbleed and the potential of *anomaly detection* in defending against such threats. Despite the 100% accuracy of the OCSVM model, the intrusion detection system stopped analyzing any new packets as the memory got filled, thus making it unfeasible for practical applications. Furthermore, the data used to test and train the machine learning model had no concept drift in it.

Hence, we addressed two main issues in the remaining sections of the evaluation:

- we evaluated the *USAE* algorithm on a Synthetic benchmark to demonstrate its utility for the more general use case where data (not just network packets) are subjected to concept drift.
- we evaluated the impact of parallelising the One-Class SVM computations from the **OCSVM layer** of DeADA on the *DropRate*.

6.5.1 Methodology

In this section, we revisit the **Heartbleed** bug, however under a **more aggressive** scenario, using DeADA in a set-up where packets are analyzed continuously, without the system crashing. In addition, we transform the 483945 packets such that data are exposed to drift (based on the range of values in the data set), in a similar way to the synthetic benchmark generation mechanism. Figure 6.1 shows the setup for this experiment.

Because the FPGA acceleration section is not fully integrated with the rest of the DeADA architecture, we simulate the behavior using our Ripple based implementation, and remove the memory limitation, but keep the drop rate.

6.5.2 Results

Since packets are now dropped at different rates, the *DropRate* becomes significant in analyzing the accuracy of the system as a whole. From Figure 6.11 we can see how the *DropRate* impacts the accuracy of the overall system:

- for the **Simple** classifier with a $DropRate = 1367.55$ accuracy decreases greatly as data drifts. If data had no concept drift the accuracy would have been constant, but we would still be able to analyze only a limited number of packets.
- for **DeADA** we have two versions. The one with the higher $DropRate$ corresponds to $replication = 1$, whereas the lower $DropRate$ is for $replication = 48$. Hence, for $DropRate = 1367.55$ even with $USAE$ the accuracy of the system is very bad (as low as 20%). The potential of DeADA as **end-to-end** architecture is thus emphasized when $DropRate = 130.66$, with the system maintaining an accuracy as high as 80%.

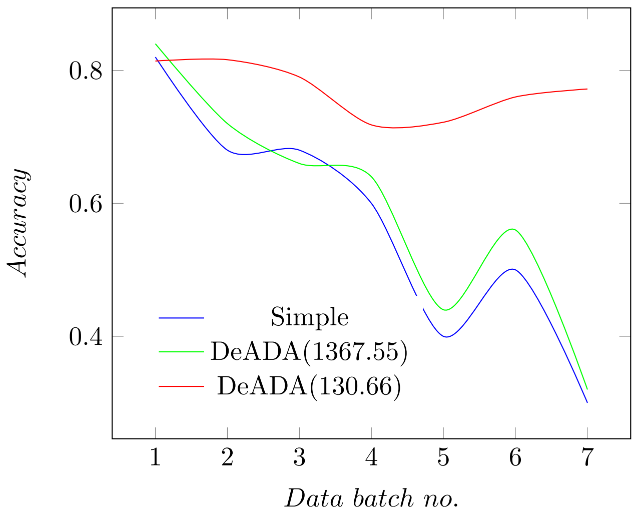


Figure 6.11: Detecting Heartbleed in environments with concept-drift. The figure shows the impact of the *Drop Rate* when aiming to detect Heartbleed. The *Simple* and *DeADA*(1367.55) have the same $DropRate = 1367.55$ (see). We can see the lower the $DropRate$ is, the higher the accuracy of the system becomes. The drift is set by $T = 100, K = 1$.

Overall, not only does $DropRate$ impact the system's ability to analyze all the data (and thus miss potential attacks), but it also affects the accuracy of the classification when data are subjected to drift. From our evaluation, we believe there is a lot of potential for a system like DeADA to become a reality and one day perform **live intrusion detection** with very low drop rates, even in Gigabit network setups.

6.6 Beyond the memory limit

As seen in section 6.4 our design is consuming ≈40% of the chip resources, however it is limited by the bus width of the memory. Given Equation 6.4 we can hypothesize on the impact of DeADA should these constraints be removed, or reduced:

- We estimated the potential of the implementation on the Maia card to be 8x better than on Max3, due to its wider memory bus width. However, even for Maia the resource usage allows for twice as more replication, with an estimated ≈16x improvement in the *DropRate*.
- Thus, we can get an estimated *DropRate* of ≈30, which drastically improves the detection capabilities of DeADA in terms of the number of analyzed packets. Furthermore, from section 6.5 we have seen the positive impact of an improved *DropRate* on the accuracy of the whole system.

By overcoming these constraints, we believe DeADA could become the network intrusion detection system of choice, given the inherit ability of anomaly detection to handle unknown attacks, the self-adaptive and unsupervised nature of our USAE algorithm, and the low *DropRates* resulted from higher parallelism. Future work could see DeADA optimized for handling speeds of 100Mbps up to 1Gbps and beyond.

6.7 Summary

The chapter evaluates the potential of our *USAE* algorithm as well as the performance of the DeADA architecture. We investigate the improvement of the *drop rate* when targeting DeADA for FPGAs over CPU. The designs are synthesized using MaxCompiler 2013.2.2.

We start by showcasing the potential of *anomaly detection* and *USAE* in identifying previously unseen attacks. **Heartbleed** is a recent bug discovered in OpenSSL and as such we believe it a very good candidate for our experiment. We train a classifier using OCSV on a range of TCP fields extracted using Wireshark. The fields are chosen with no particular preference, but have been chosen such that they contain useful information about the incoming data. The classifier was successful in identifying all the simulated attacks.

The next section of the chapter evaluates our proposed unsupervised self-adaptive ensemble method on a range of synthetically generated benchmarks. We analyze the impact of the λ parameter from the H_{flip} heuristic on the overall quality of *USAE*. We obtain very good results, comparable to the reference solution (i.e. a supervised ensemble as described in [25]).

Finally, we evaluate the performance of DeADA when mapped to a MAX3 card. We obtain an improvement of ≈4x in the *Drop Rates* when compared to 4 cores CPU, with further possible improvements when using a Maia card.

Chapter 7

Conclusion

In this project we have addressed the issues posed by using anomaly detection on live data subjected to concept drift in the context of network security. We have shown the potential of our proposed *unsupervised* machine learning solution compared to supervised anomaly detection. We then constructed a scalable dataflow architecture around it, with improved performance over CPU when analyzing live traffic.

7.1 Summary of achievements

We propose the *Unsupervised Self-Adaptive Ensemble(USAE)* as an alternative to handling concept drift in anomaly detection, a branch of machine learning. By combining different techniques we created a solution which is comparable to existing supervised techniques, when using One-Class Support Vector Machines. Thus, we automated the process of data classification and example gathering for the purposes of training new classifiers capable of handling concept-drift, and as a consequence enable the use of anomaly detection in handling high-volume data without any external interventions, addressing Challenges 1 and 2 from section 1.2.

To address Challenge 3, consisting of the high processing times required by machine learning algorithms for classifying network traffic as *good* or *bad*, we created DeADA, a scalable architecture which incorporates the power of OCSVM in anomaly detection with our USAE algorithm, exploiting the benefits of dataflow computing to speed-up computations and achieve *live analysis.* Our solution resulted in increased computing speeds when implemented on FPGA, meaning fewer packets were skipped in the process compared to the CPU implementation, improving the overall accuracy of the system in detecting novel threats (see section 6.5).

Besides addressing the main challenges of the project we have also created Ripple, a Java library for fast prototyping of *dataflow desings.* Ripple is based on Kahn Process Network principles, allowing us to construct dataflow architectures and test their functionality prior to transitioning them to FPGAs. The library was very useful for testing and simulating DeADA, and we believe it could be of help to the community, hence it will be available as open source under an MIT license.

To evaluate our contributions we started off with a case study of Heartbleed, a recent bug in the OpenSSL implementation of the SSL protocol, highlighting the benefits of using anomaly detection as an option for network intrusion detection, when defending against novel/unknown threats. We evaluated the *USAE* concept drift solution on a synthetically generated benchmark, obtaining very good results. Building on the information collected from the case study we evaluated the performance of our proposed architecture with respect to packet *Drop Rate* when targeted for FPGA resulting in 4x improvement compared its CPU counterpart, with possible 16x improvements when memory constraints are overcome. Finally we tested DeADA on a data set containing Heartbleed attacks, while normal data was drifting. The results highlighted the importance of a reduced *Drop Rate* in increasing the accuracy and reliability of a network intrusion detection system based on anomaly detection.

7.2 Future work

We believe that we have managed to achieve good results and fulfill on the objectives set in section 1.2, with our proposed solutions showing a lot of potential. However there are several future improvements and experiments that could improve the quality of our work.

- *Gathering of network data* with measurable concept drift is critical to putting a verdict on the relevance of our USAE method. In fact we believe such a data set would also benefit the community by providing an adequate benchmark for assessing various solutions to the concept drift problem, particularly in the case of anomaly detection.
- *Assessing the performance of the* $replication = 96$ implementation of DeADA, is one of the areas which we have not managed to fully cover.
- *Mapping the whole DeADA.* Future work should focus on mapping all of the layers of DeADA to FPGA, in order to achieve higher data throughput, remove the bottlenecks and create an easy to manage system which could then be installed in a computer network. Furthermore, linking the pre-processing straight to the FPGA will result in better performance measurements.
- *Akka with Ripple.* Since Ripple shows a lot of potential for quickly prototyping dataflow designs, we believe it also has the potential of becoming an alternative to dataflow based distributed computing. As such, changing the existing thread implementations with ones built on top of Akka actors, not only will improve the quality of the library, but will also open it to new use cases such as constructing a dataflow application where the nodes in the graph are spread across different machines.
- *Intrusion detection response mechanism.* Current work focuses on simply detecting whether an anomaly occurred in the network, however it does not cover the *counter-measures* aspect of network intrusion detection. We believe there is a lot of potential in this area, particularly when using anomaly detection and trying to label the attacks with more information than just *normal* of *abnormal*. Such an example would be the use of

a hybrid setup where anomaly detection is used to capture all attacks (including unknown ones), whereas a subsequent layer of machine learning algorithm such as Naive Bayes or Neural Networks would then distinguish between *known* attack scenarios.

- *Network event analysis.* Our investigation have been based around anomaly detection on network packets, however we believe that our solutions can easily be extended to analyzing network events. Network events can provide more information about attacks than simple packets analysis as generally they incorporate more complex behaviors of the network.

- Since *USAE* is a generic algorithm for performing anomaly detection under concept drift, we believe it could be extended to a variety of application, such as aerospace engineering[10],[34], environmental monitoring[1], and audio surveillance[17].

Bibliography

[1] J.C. Bezdek, S. Rajasegarar, M. Moshtaghi, C. Leckie, M. Palaniswami, and T.C. Havens. Anomaly detection in environmental monitoring networks [application notes]. *Computational Intelligence Magazine, IEEE*, 6(2):52–58, May 2011.

[2] Manuele Bicego and Mario A. T. Figueiredo. Soft clustering using weighted one-class support vector machines. *Pattern Recogn.*, 42(1):27–32, January 2009.

[3] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, pages 144–152, New York, NY, USA, 1992. ACM.

[4] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009.

[5] Abhishek Das, Sanchit Misra, Sumeet Joshi, Joseph Zambreno, Gokhan Memik, and Alok Choudhary. An efficient fpga implementation of principle component analysis based network intrusion detection system. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pages 1160–1165, New York, NY, USA, 2008. ACM.

[6] Ofer Dekel and Yoram Singer. Support vector machines on a budget. In *In NIPS*, 2006.

[7] Sarah Jane Delany, Pádraig Cunningham, Alexey Tsymbal, and Lorcan Coyle. A case-based technique for tracking concept drift in spam filtering. *Know.-Based Syst.*, 18(4-5):187–195, August 2005.

[8] Big Data Working Group. Big data analytics for security intelligence. Technical report, CLOUD SECURITY ALLIANCE, 2013.

[9] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[10] Paul Hayton, Bernhard Schölkopf, Lionel Tarassenko, and Paul Anuzis. Support vector novelty detection applied to jet engine vibration spectra. In *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 13*, pages 946–952. MIT Press, 2000.

[11] Geoff Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '01, pages 97–106, New York, NY, USA, 2001. ACM.

[12] Nathalie Japkowicz, Catherine Myers, and Mark A. Gluck. A novelty detection approach to classification. In *IJCAI*, pages 518–523, 1995.

[13] Juan Luis Jerez, George Anthony Constantinides, and Eric C. Kerrigan. An fpga implementation of a sparse quadratic programming solver for constrained predictive control. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 209–218, New York, NY, USA, 2011. ACM.

[14] Ralf Klinkenberg. Learning drifting concepts: Example selection vs. example weighting. *Intell. Data Anal.*, 8(3):281–300, August 2004.

[15] Ralf Klinkenberg and Thorsten Joachims. Detecting concept drift with support vector machines. In *In Proceedings of the Seventeenth International Conference on Machine Learning (ICML*, pages 487–494. Morgan Kaufmann, 2000.

[16] Bartosz Krawczyk and Michał Woźniak. Incremental learning and forgetting in one-class classifiers for data streams. In Robert Burduk, Konrad Jackowski, Marek Kurzynski, Michał Wozniak, and Andrzej Zolnierek, editors, *Proceedings of the 8th International Conference on Computer Recognition Systems CORES 2013*, volume 226 of *Advances in Intelligent Systems and Computing*, pages 319–328. Springer International Publishing, 2013.

[17] S. Lecomte, R. Lengelle, C. Richard, F. Capman, and B. Ravera. Abnormal events detection using unsupervised one-class svm - application to audio surveillance and evaluation -. In *Advanced Video and Signal-Based Surveillance (AVSS), 2011 8th IEEE International Conference on*, pages 124–129, Aug 2011.

[18] Edward A. Lee and Thomas M. Parks. Readings in hardware/software codesign. chapter Dataflow Process Networks, pages 59–85. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[19] Maxeler. Maxcompiler white paper, 2011. `http://www.maxeler.com/media/documents/MaxelerWhitePaperMaxCompiler.pdf`, Accessed June 2014.

[20] Linda Milne. Feature selection using neural networks with contribution measures, 1995.

[21] Hristo Nikolov, Todor Stefanov, and Ed Deprettere. Modeling and fpga implementation of applications using parameterized process networks with non-static parameters. In *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '05, pages 255–263, Washington, DC, USA, 2005. IEEE Computer Society.

[22] Oracle. Support vector machines, 2014. `http://docs.oracle.com/cd/B28359_01/datamine.111/b28129/algo_svm.htm#CHDDJFDJ`, Accessed June 2014.

[23] M. Papadonikolakis and C. Bouganis. A scalable fpga architecture for non-linear svm training. In *ICECE Technology, 2008. FPT 2008. International Conference on*, pages 337–340, Dec 2008.

[24] P. Parveen, J. Evans, Bhavani Thuraisingham, K.W. Hamlen, and L. Khan. Insider threat detection using stream mining and graph mining. In *Privacy, security, risk and trust (passat), 2011 ieee third international conference on and 2011 ieee third international conference on social computing (socialcom)*, pages 1102–1110, Oct 2011.

[25] Pallabi Parveen, Zackary R. Weger, Bhavani Thuraisingham, Kevin Hamlen, and Latifur Khan. Supervised learning for insider threat detection using stream mining. In *Proceedings of the 2011 IEEE 23rd International Conference on Tools with Artificial Intelligence*, ICTAI '11, pages 1032–1039, Washington, DC, USA, 2011. IEEE Computer Society.

[26] Oliver Pell and Oskar Mencer. Surviving the end of frequency scaling with reconfigurable dataflow computing. *SIGARCH Comput. Archit. News*, 39(4):60–65, December 2011.

[27] John C. Platt. Advances in kernel methods. chapter Fast Training of Support Vector Machines Using Sequential Minimal Optimization, pages 185–208. MIT Press, Cambridge, MA, USA, 1999.

[28] Bernhard Schölkopf. Learning with kernels, 2006. `http://dip.sun.ac.za/~hanno/tw796/lesings/mlss06au_scholkopf_lk.pdf`, Accessed June 2014.

[29] Bernhard Schölkopf, Robert C Williamson, Alex J Smola, John Shawe-Taylor, and John C Platt. Support vector method for novelty detection. *NIPS*, 12:582–588, 1999.

[30] Sci-kit.org. Novelty and outlier detection, June 2010. `http://scikit-learn.org/stable/modules/outlier_detection.html`, accessed June 2014.

[31] Y. Shimai, J. Tani, H. Noguchi, H. Kawaguchi, and M. Yoshimoto. Fpga implementation of mixed integer quadratic programming solver for mobile robot control. In *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pages 447–450, Dec 2009.

[32] Yale Song, Zhen Wen, Ching-Yung Lin, and Randall Davis. One-class conditional random fields for sequential anomaly detection. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI'13, pages 1685–1691. AAAI Press, 2013.

[33] David MJ Tax and Robert PW Duin. Support vector data description. *Machine learning*, 54(1):45–66, 2004.

[34] D. Tolani, M. Yasar, Shin Chin, and A. Ray. Anomaly detection for health management of aircraft gas turbine engines. In *American Control Conference, 2005. Proceedings of the 2005*, pages 459–464 vol. 1, June 2005.

[35] Alexey Tsymbal, Mykola Pechenizkiy, Pádraig Cunningham, and Seppo Puuronen. Dynamic integration of classifiers for handling concept drift. *Inf. Fusion*, 9(1):56–68, January 2008.

[36] Roemer Vlasveld. Introduction to ocsvm, 2013. `http://rvlasveld.github.io/blog/2013/07/12/introduction-to-one-class-support-vector-machines/`, Accessed April 2014.

[37] Z. Vrba, P. Halvorsen, C. Griwodz, and P. Beskow. Kahn process networks are a flexible alternative to mapreduce. In *High Performance Computing and Communications, 2009. HPCC '09. 11th IEEE International Conference on*, pages 154–162, June 2009.

[38] Shenghui Wang, Stefan Schlobach, and Michel Klein. What is concept drift and how to measure it? In Philipp Cimiano and H.Sofia Pinto, editors, *Knowledge Engineering and Management by the Masses*, volume 6317 of *Lecture Notes in Computer Science*, pages 241–256. Springer Berlin Heidelberg, 2010.

[39] Xunkai Wei, Yinghong Li, Dong Liu, and Liguang Zhan. Mahalanbois support vector machines made fast and robust.

Appendices

Appendix A

MAX3 resource usage

This section contains the resource usage results obtained for various replication factors when built for MAX3, with a frequency of 100Mhz.

LUTs	FFs	BRAMs	DSPs	component
13.63%	10.10%	10.71%	0.79%	total
2.16%	1.55%	0.94%	0.79%	kernels
11.22%	8.50%	9.77%	0.00%	manager
0.21%	0.04%	0.00%	0.00%	stray resources

Table A.1: Resource usage for a build with replication=4

LUTs	FFs	BRAMs	DSPs	component
16.43%	12.42%	13.31%	1.59%	total
3.94%	2.86%	0.94%	1.59%	kernels
12.24%	9.51%	12.17%	0.00%	manager
0.23%	0.05%	0.00%	0.00%	stray resources

Table A.2: Resource usage for a build with replication=8 on MAX3

LUTs	FFs	BRAMs	DSPs	component
21.64%	15.89%	19.83%	3.17%	total
7.74%	5.83%	0.94%	3.17%	kernels
13.44%	10.00%	18.89%	0.00%	manager
0.42%	0.06%	0.00%	0.00%	stray resources

Table A.3: Resource usage for a build with replication=16 on MAX3

LUTs	FFs	BRAMs	DSPs	component
26.09%	19.18%	17.39%	4.76%	total
11.16%	8.57%	0.94%	4.76%	kernels
14.55%	10.54%	16.45%	0.00%	manager
0.36%	0.08%	0.00%	0.00%	stray resources

Table A.4: Resource usage for a build with replication=24 on MAX3

LUTs	FFs	BRAMs	DSPs	component
32.82%	24.24%	31.06%	6.35%	total
14.73%	11.31%	0.94%	6.35%	kernels
17.49%	12.84%	30.12%	0.00%	manager
0.57%	0.09%	0.00%	0.00%	stray resources

Table A.5: Resource usage for a build with replication=32 on MAX3

Appendix B

Maxeler code

The code for the state machine which implements the stream aligned can be found at `https://bitbucket.org/niushin/network_security/`, together with the rest of the code under the various DeADA folders.

```
import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.KernelMath;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.CounterChain;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEFix.SignMode;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEType;

class DeADAKernel extends Kernel {

    protected DeADAKernel(KernelParameters parameters) {
        super(parameters);


        DFEType fixType = dfeFixOffset(32,-8, SignMode.TWOSCOMPLEMENT);
        DFEVar carry = fixType.newInstance(this);

        /* incoming scalar vector */
        DFEVar supportValue = io.input("svv", dfeFloat(8,24)).cast(fixType);
        /* incoming instance */
        DFEVar instValue = io.input("inv", dfeFloat(8,24)).cast(fixType);
        DFEVar gamma = io.scalarInput("gamma",dfeFloat(8,24)).cast(fixType);
        DFEVar isPoison = supportValue === -1;

        DFEVar count = control.count.makeCounter(control.count.makeParams(32).
        withReset(stream.offset(isPoison,-1))).getCount();

        DFEVar isCountZero = count === 0;
        DFEVar preRes = isCountZero ? 0.0 : carry;
        DFEVar mult = isPoison ? 0.0 : (instValue - supportValue) * (instValue - supportValue);
        optimization.pushPipeliningFactor(0);
        DFEVar result = mult + preRes;
        optimization.popPipeliningFactor();
        DFEVar exp = result * gamma * -1;
        DFEVar res2 = KernelMath.exp(exp);
        //debug.simPrintf(isPoison,"Count:%d Result: %f %f Poison %d \n",count,result.cast(dfeFloat(8,24)),supportValue.cast(dfeFloat(8,24)),isPoison);
        carry.connect(stream.offset(result,-1));
        io.output("res", res2.cast(dfeFloat(8,24)), dfeFloat(8,24), isPoison);
    }

}
```

Figure B.1: DeADAKernel code.

Appendix C

Experimental data

Processing Rate (packets/s)		
Data Size(M)	CPU	
	1 core	4 cores
0.4	617.27	1752.33
0.8	312.94	818.31
1.6	157.72	417.99
3.2	77.49	223.10
6.4	39.52	111.95
12.8	19.91	56.42
25.6	8.21	24.03
51.2	4.59	11.73
102.4	1.88	5.48

Table C.1: Processing rates on CPU

Processing time per instance (μs)							
Data Size(M)	MAX3						
	r=1	r=4	r=8	r=16	r=24	r=32	r=48
0.4	16947	10356	9454	8775	8891	7893	8853
0.8	25590	12451	10994	9252	8539	9124	8734
1.6	42782	16724	12605	10159	9893	9387	9693
3.2	76943	25114	17035	12806	10982	10824	10029
6.4	146094	42680	25890	16854	14183	13441	11690
12.8	283501	77150	42829	25700	20373	18564	15243
25.6	558843	145942	77702	43472	32510	27089	21487
51.2	1112222	284221	147106	79193	56199	45316	34449
102.4	2212698	561655	287593	150993	103776	81961	59148

Table C.2: Analysis speed in μs on a Max3 card

Data batch	T,K	Tn	Tp	Fn	Fp	BS_m	T,K	Tn	Tp	Fn	Fp	BS_m	T,K	Tn	Tp	Fn	Fp	BS_m
1		291	114	28	67	0.19		297	114	44	45	0.178		258	95	40	107	0.294
2		297	107	33	63	0.192		300	100	44	56	0.2		146	119	27	208	0.47
3		295	109	34	62	0.192		286	110	42	62	0.208		68	141	13	278	0.582
4		302	106	22	70	0.184		297	89	41	73	0.228		8	141	9	342	0.702
5		261	119	32	88	0.24		302	110	37	51	0.176		0	160	2	338	0.68
6		263	116	27	94	0.242		270	106	46	78	0.248		0	141	7	352	0.718
7		223	131	23	123	0.292		286	102	45	67	0.224		0	146	8	346	0.708
8		204	127	31	138	0.338		279	113	46	62	0.216		0	136	6	358	0.728
9	T=100	195	109	23	173	0.392	T=50	246	115	56	83	0.278	T=500	0	131	5	364	0.738
10	K=1	174	111	25	190	0.43	K=1	253	118	37	92	0.258	K=1	0	132	9	359	0.736
11		162	107	18	213	0.462		247	104	45	104	0.298		0	144	4	352	0.712
12		145	122	14	219	0.466		240	126	36	98	0.268		0	150	9	341	0.7
13		112	144	8	236	0.488		228	113	52	107	0.318		0	134	6	360	0.732
14		108	154	7	231	0.476		244	110	38	108	0.292		0	144	9	347	0.712
15		92	126	8	274	0.564		219	135	23	123	0.292		0	144	9	347	0.712
17		56	136	9	299	0.616		217	128	29	126	0.31		0	135	4	361	0.73
17		47	138	8	307	0.63		202	121	33	144	0.354		0	137	11	352	0.726
18		39	154	5	302	0.614		210	114	22	154	0.352		0	123	5	372	0.754

Table C.3: Experimental results for the *Simple* classifier, containing the number of true positives(Tp), true negatives(tn), false positives(fp) and false negatives(fn).

Data batch	T,K	Tn	Tp	Fn	Fp	BS_m	T,K	Tn	Tp	Fn	Fp	BS_m
1		275	123	19	83	0.204		256	126	40	78	0.236
2		204	129	11	156	0.334		232	117	30	121	0.302
3		152	136	7	205	0.424		179	135	26	160	0.372
4		111	126	2	261	0.526		183	121	42	154	0.392
5		84	147	4	265	0.538		249	104	46	101	0.294
6		77	142	1	280	0.562		272	113	47	68	0.23
7		59	154	0	287	0.574		299	107	48	46	0.188
8		53	158	0	289	0.578		287	102	51	60	0.222
9	T=100	42	131	1	326	0.654	T=100	266	103	46	85	0.262
10	K=1	33	136	0	331	0.662	K=7	226	128	45	101	0.292
11	forward feed	34	124	1	341	0.684	oscillating	186	131	36	147	0.366
12		33	136	0	331	0.662		199	117	31	153	0.368
13		25	152	0	323	0.646		237	106	50	107	0.314
14		12	160	1	327	0.656		295	96	48	61	0.218
15		15	134	0	351	0.702		297	106	48	49	0.194
17		8	145	0	347	0.694		307	94	43	56	0.198
17		5	146	0	349	0.698		262	109	54	75	0.258
18		6	159	0	335	0.67		237	108	40	115	0.31

Table C.4: Experimental results for the *Simple* classifier, containing the number of true positives(Tp), true negatives(tn), false positives(fp) and false negatives(fn) with the *forward feeding* case and the *oscillating* case

Data batch	T,K	Tn	Tp	Fn	Fp	BS_m	T,K	Tn	Tp	Fn	Fp	BS_m	T,K	Tn	Tp	Fn	Fp	BS_m
1		299	115	43	43	0.172		282	113	29	76	0.21		278	105	30	87	0.234
2		301	101	43	55	0.196		295	109	31	65	0.192		240	113	33	114	0.294
3		287	107	45	61	0.212		295	102	41	62	0.206		194	130	24	152	0.352
4		290	90	40	80	0.24		310	104	24	62	0.172		177	126	24	173	0.394
5		286	111	36	67	0.206		267	119	32	82	0.228		211	136	26	127	0.306
6		254	116	36	94	0.26		268	119	24	89	0.226		253	114	34	99	0.266
7		281	111	36	72	0.216		265	126	28	81	0.218		200	130	24	146	0.34
8		279	130	29	62	0.182		247	120	38	95	0.266		189	126	16	169	0.37
9	T=50	263	115	56	66	0.244	T=100	269	95	37	99	0.272		186	120	16	178	0.388
10	K=1	277	121	34	68	0.204	K=1	254	99	37	110	0.294		215	116	25	144	0.338
11		283	103	46	68	0.228		262	104	21	113	0.268		200	120	28	152	0.36
12		275	114	48	63	0.222		260	113	23	104	0.254		187	134	25	154	0.358
13		265	120	45	70	0.23		263	120	32	85	0.234		211	108	32	149	0.362
14		272	115	33	80	0.226		249	131	30	90	0.24		216	125	28	131	0.318
15		269	122	36	73	0.218		270	114	20	96	0.232		225	130	23	122	0.29
17		252	131	26	91	0.234		269	112	33	86	0.238		191	126	13	170	0.366
17		273	111	43	73	0.232		257	125	21	97	0.236		188	130	18	164	0.364
18		278	106	30	86	0.232		254	129	30	87	0.234		284	105	23	88	0.222

Table C.5: Experimental results for the *USAE* algorithm for a setting of $\lambda = 0.3$

Data batch	T,K	Tn	Tp	Fn	Fp	BS_m	T,K	Tn	Tp	Fn	Fp	BS_m	T,K	Tn	Tp	Fn	Fp	BS_m
1		285	117	43	55	0.196		304	106	54	36	0.18		305	101	59	35	0.188
2		247	129	15	109	0.248		301	103	41	55	0.192		321	92	52	35	0.174
3		183	138	9	170	0.358		288	110	37	65	0.204		326	90	57	27	0.168
4		149	140	7	204	0.422		297	126	21	56	0.154		310	98	49	43	0.184
5		131	142	3	224	0.454		307	120	25	48	0.146		334	92	53	21	0.148
6		90	149	3	258	0.522		303	126	26	45	0.142		334	99	53	14	0.134
7		80	139	1	280	0.562		324	114	26	36	0.124		340	93	47	20	0.134
8		44	151	1	304	0.61		296	129	23	52	0.15		327	105	47	21	0.136
9	T=100	26	163	0	311	0.622	T=100	286	128	35	51	0.172	T=100	321	103	60	16	0.152
10	K=1,	17	141	0	342	0.684	K=1	329	117	24	30	0.108	K=1	347	85	56	12	0.136
11	$\lambda = 0.1$	10	151	0	339	0.678	$\lambda = 0.4$	308	128	23	41	0.128	$\lambda = 0.5$	342	94	57	7	0.128
12		0	149	0	351	0.702		324	113	36	27	0.126		344	99	50	7	0.114
13		0	137	0	363	0.726		326	123	14	37	0.102		345	91	46	18	0.128
14		0	163	0	337	0.674		311	132	31	26	0.114		335	91	72	2	0.148
15		0	146	0	354	0.708		335	114	32	19	0.102		353	90	56	1	0.114
17		0	168	0	332	0.664		312	120	48	20	0.136		331	81	87	1	0.176
17		0	141	0	359	0.718		322	101	40	37	0.154		354	90	51	5	0.112
18		0	138	0	362	0.724		306	110	28	56	0.168		356	92	46	6	0.104

Table C.6: Experimental results for the *USAE* algorithm for varying λ

Data batch	T,K	Tn	Tp	Fn	Fp	BS_m	T,K	Tn	Tp	Fn	Fp	BS_m
1		293	107	53	47	0.2		277	115	51	57	0.216
2		288	114	30	68	0.196		288	89	58	65	0.246
3		258	126	21	95	0.232		268	107	54	71	0.25
4		230	133	14	123	0.274		256	101	62	81	0.286
5		209	140	5	146	0.302		260	93	57	90	0.294
6		223	146	6	125	0.262		244	106	54	96	0.3
7	T=100	250	130	10	110	0.24		227	109	46	118	0.328
8	K=7	184	142	10	164	0.348	T=100	234	109	44	113	0.314
9		193	159	4	144	0.296	K=7	277	105	44	74	0.236
10	strong	212	134	7	147	0.308	oscillating	256	110	63	71	0.268
11	drift	203	150	1	146	0.294	drift	265	102	65	68	0.266
12		171	146	3	180	0.366		282	86	62	70	0.264
13		169	129	8	194	0.404		256	97	59	88	0.294
14		189	158	5	148	0.306		255	89	55	101	0.312
15		215	142	4	139	0.286		240	108	46	106	0.304
17		216	158	10	116	0.252		284	99	38	79	0.234
17		236	132	9	123	0.264		271	108	55	66	0.242
18		211	134	4	151	0.31		253	102	46	99	0.29

Table C.7: Experimental data for *USAE* for strong drift (K=7,T=100) and for oscillating drift (i.e. the drift changes direction every 2000 elements)

Data batch	T,K	Tn	Tp	Fn	Fp	BS_m	T,K	Tn	Tp	Fn	Fp	BS_m	T,K	Tn	Tp	Fn	Fp	BS_m
1		295	120	38	47	0.17		286	118	24	72	0.192		282	111	24	83	0.214
2		301	105	39	55	0.188		295	111	29	65	0.188		268	111	35	86	0.242
3		301	105	47	47	0.188		305	106	37	52	0.178		269	116	38	77	0.23
4		313	91	39	57	0.192		313	111	17	59	0.152		275	112	38	75	0.226
5		304	112	35	49	0.168		274	121	30	75	0.21		274	120	42	64	0.212
6		278	116	36	70	0.212		317	110	33	40	0.146		275	109	39	77	0.232
7		295	104	43	58	0.202		299	122	32	47	0.158		270	118	36	76	0.224
8		296	124	35	45	0.16		275	121	37	67	0.208		273	113	29	85	0.228
9	T=100	259	118	53	70	0.246	T=100	314	87	45	54	0.198		291	110	26	73	0.198
10	K=50	292	117	38	53	0.182	K=1	307	89	47	57	0.208		265	114	27	94	0.242
11		304	108	41	47	0.176		321	94	31	54	0.17		269	115	33	83	0.232
12		287	124	38	51	0.178		291	106	30	73	0.206		258	121	38	83	0.242
13		287	123	42	48	0.18		292	112	40	56	0.192		279	105	35	81	0.232
14		282	118	30	70	0.2		277	124	37	62	0.198		252	114	39	95	0.268
15		289	125	33	53	0.172		298	106	28	68	0.192		280	120	33	67	0.2
17		292	122	35	51	0.172		301	98	47	54	0.202		249	113	26	112	0.276
17		306	109	45	40	0.17		299	104	42	55	0.194		277	120	28	75	0.206
18		307	101	35	57	0.184		276	110	49	65	0.228		291	108	20	81	0.202

Table C.8: Experimental data for the *Supervised* approach, for varying parameters of T and K.