Data Driven Device Failure Prediction

THESIS

Paul L. Jordan, 1st Lt, USAF

AFIT/GCS/ENG/17-M

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/17-M

DATA DRIVEN DEVICE FAILURE PREDICTION

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Science

Paul L. Jordan, B.S.C.S.

1st Lt, USAF

April 9, 2016

AFIT/GCS/ENG/17-M

DATA DRIVEN DEVICE FAILURE PREDICTION

THESIS

Paul L. Jordan, B.S.C.S.
1st Lt, USAF

Committee Membership:

Dr. M. J. Mendenhall
Chair

Dr. G. L. Peterson
Member

Maj A. C. Lin, PhD
Member

Maj A. J. Sellers, PhD
Member

AFIT/GCS/ENG/17-M

Abstract

This is the start of my abstract that I will write later.

Acknowledgements

I would like to thank...

Paul L. Jordan

Table of Contents

List of Figures

List of Tables

DATA DRIVEN DEVICE FAILURE PREDICTION

I. Introduction

We as humans have always shared a curiosity about the future. Being able to predict events in the future offers tremendous application in todays technologically advanced world. While actually being able to accurately predict the future has unfortunately not been proven possible, there has been an enormous amount of time and energy spent over the past several decades attempting to make educated predictions about the failure of machine in order to avoid failures. In this research we explore the application of a new framework developed to automatically re-train machine learning based failure predictors. Failures in software based computer systems have still not been eliminated due to the fact that software is still developed by humans and is therefore exposed to human error. There a number of ways to reduce the number of errors produced by a piece of software, but the software development life-cycle is shrinking and less time and effort are being devoted to reducing errors before deployment. This leaves real-time error prevention or handling.

In recent years, it seems many of the cloud based computing companies have attempted to solve this problem by making all of their services massively redundant. As hardware becomes more affordable, this is an effective approach in many ways but ultimately is still not cost efficient. In some cases, funds may not be available to acheive this sort of redundancy. Consequently, this research focuses on a small piece of the general field of reliable computing: online failure prediction (OFP). OFP is the act of attempting to predict when failures are likely so that they can be avoided. A great deal of work has been done in this field which we outline in chapter II, but much of it has gone unimplemented due to the complex and manual task of training a prediction model. If the underlying system changes at all (which in todays world is a common occurrence due to the aforementioned shrinking software development life-cycle) the efficacy of a prediction model can be drastically reduced if not rendered completely useless until it is retrained. This research explores an implementation of a new framework for automatically retraining a predictor after such an underlying system change. More specifically, we present our results after implementing this framework using a Microsoft Windows Server domain controller. We then apply successive software updates until the model we have selected becomes useless and allow the framework to re-train our predictor.

1.1 Problem Statement

Predicting and alerting on impending network service failures currently uses thresholds and rules on discrete items in enterprise system logs. For example, if the central processing unit (CPU) and memory usage on a device exceeds 90%, then an alert may be issued. This approach works, but only for certain types of failures and in order to minimize the false positives, it only makes recommendations minutes before a failure, or when the system is in an already degraded performance mode. To maintain network resilience, the operational organizations responsible for communications support desperately need some means of gaining lead-time before a service failure occurs.

Preceding a service failure event, multiple indicators spread disparate sources, perhaps over a long period of time, may appear in system logs. The log entries of interest are also quite rare compared with normal operations. Because of these constraints, identifying failure indicators can be nearly impossible for humans to perform. Further, in most cases, restoring service is more important than identifying the indicators that may or may not have existed.

Failure prediction can be approached in many ways. Arguably the simplest approach is to use everyday statistical analysis to, for example, determine the mean time between failures of specific components. The analysis of all components making up a system can be aggregated to make predictions about that system using a set of statistics-based or business-relevant rules. Unfortunately, the complexity of modern architectures has outpaced such off-line statistical-based analysis, which has driven the advancement of OFP. OFP differs from other means of failure prediction in that it focuses on classifying the current running state of a machine as either failure prone or not, or in such a way that it describes the confidence in how failure prone a system is at present [19].

Fortunately, over the past several decades many machine learning-based approaches to identifying indications of pending failure in log messages or *reported errors* have been presented. These data-driven approaches are categorized in a 2010 survey paper by Salfner et al. [19] on OFP. They categorize these data-driven approaches along with several others in a taxonomy which we extend in this research. They also categorize OFP under a much broader area of study called proactive fault management (PFM).

Unfortunately, in recent years much of the work in OFP has gone unused due to the dramatic decrease in cost and complexity involved in building hardware-based redundant systems. Furthermore, in most cases OFP implements machine learning algorithms that require manual re-training after underlying system changes. More troubling is that these system changes are becoming more frequent as the software development life cycle moves toward a more continuous integration model. To help solve these challenges, the framework presented in [12] uses simulated faults to automatically re-train a prediction algorithm to make implementing OFP approaches easier. We propose to expand the work in [12] to capture developments since its writing and generalize it so it works for a broader class of devices.

1.2 Impact of Research

Every day, many of the Air Force's critical missions depend on our computer infrastructure. An essential piece of this infrastructure is the authentication mechanisms that protect our sensitive information. Unfortunately, the software at the core of this infrastructure is written and maintained by humans and thus susceptible human error. This research will enable the Air Force and many others that use the Microsoft Enterprise Infrastructure to accurately predict pending service outages thereby providing lead-time in order to avoid those outages. The result is cost savings in personnel, equipment, but isn't limited to cost savings. It is difficult to quantify the risk of mission failure due to network service outage.

1.3 Assumptions and Limitations

1.4 Alternate approach (from paper)

II. Overview of Online Failure Prediction

This chapter reviews current research regarding online failure prediction and its many approaches to build a foundation for this research. Further, a taxonomy of approaches was developed here [19], this chapter updates that taxonomy and classifies approaches since its publication using it.

2.1 Background

In 2010, Salfner et al. published a survey paper that provides a comprehensive summary of the state of the art on the topic of OFP [19]. In addition to the review of the literature up to the point of publication, they provide a summary of definitions (see Section 2.1) and measures of performance (see Section 2.1) commonly used in the community for couching the OFP discussion.

Definitions.

Proactive Fault Management:.

Figure 1 shows the components in the PFM process, and in that process the final three stages deine how much lead time is required to avoid a failure when predicted during OFP. Lead time, although defined later, is a critical element in any approach to OFP.

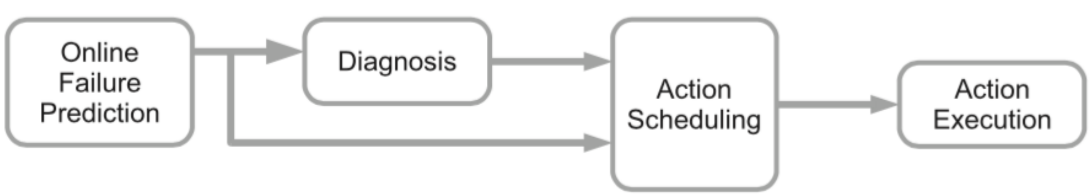

Figure 1. The four stages of proactive fault management and how they integrate [19].

Although there is a lot of utility in studying PFM in its entirety, the crux to its success is at the beginning of this four stage process. This research focusses mainly on that first step however, we include here a brief overview of the remaining steps as they define how much *lead time* is required to avoid a predicted failure in order for the first step (OFP) to be of any use. *Lead time* is formally defined later in this section, but it should be noted now that it is a critical element of a failure prediction approach.

OFP is defined as the first step in PFM seen in Figure 1. The last three stages of PFM are diagnosis, action scheduling, and action execution. Once failure has been predicted, a fault tolerant system must determine what will cause the failure. This stage is called the *diagnosis* stage or "root-cause analysis" stage. During the *diagnosis* stage, the analysis must be conducted so that a system knows which remediation actions are possible. After it is determined what will cause a failure, a fault tolerant system must schedule a remediation action that is either performed by an operator or done automatically. This stage is known as the *action scheduling* stage and normally takes as input the cost of performing an action, confidence in prediction, effectiveness/complexity of remedy action and makes a decision about what action to perform based on that input. In some cases a remedy action can be so simple that even if the confidence in the prediction is low, the action can still be performed with little impact on the overall system and its users. A thorough analysis of the trade-off between cost of avoidance and confidence in prediction and the associated benefits is described in [3]. Finally, in order to avoid failure, a system must execute the scheduled remediation action or let an operator know which actions can be taken in a stage called *action execution*.

Faults, Errors, Symptoms, and Failures:.

As [19] points out, many attempts have been made to define faults, errors, symptoms, and failures. In this research we use the definitions from [1] as interpreted and extended in [19] for the following terms: failure; error (detected versus undetected); fault; and symptom.

Failure is an event that occurs when the delivered service deviates from correct service. In other words, things can go wrong internally; as long as the output of a system is what is expected, failure has not occurred. An *error* is the part of the total state of the system that may lead to its subsequent service failure. *Errors* are characterized as the point when things go wrong [19]. Fault tolerant systems can handle errors without necessarily evolving into failure. There are two kinds of errors. First, a *detected error* is an error that is reported to a logging service. In other words, if it can be seen in a log then it is a detected error. Second, *undetected errors* are errors that have not been identified by an error detector. Undetected errors are things like memory leaks. The error certainly exists, but as long as there is usable memory, it is not likely to be reported to a logging service. Once the system runs out of usable memory, undetected errors will likely show up in logs and become a detected errors. A *fault* is the hypothesized root cause of an error. Faults can remain dormant for some time before manifesting themselves and causing an incorrect system state. In the memory leak example, the missing *free* statement in the source code would be the fault. A *symptom* is an out-of-norm behavior of a system's parameters caused by errors, whether detected or undetected. In the memory leak example, a possible symptom of the error might be delayed response times due to sluggish performance of the overall system.

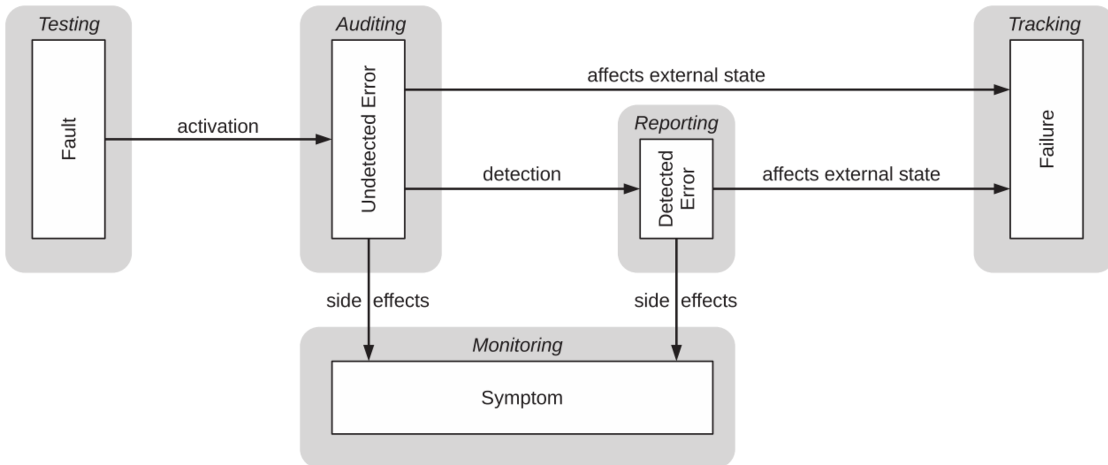

Figure 2. How faults and errors evolve into failure with the associated methods for detection represented by enclosing gray boxes [19].

Figure 2 illustrates the differences between faults, errors, symptoms and failures. It further shows how faults can evolve into failures and how specifically at each of the steps, they can be detected. We define these specifically because in Section 2.2 we outline a taxonomy of OFP approaches introduced in [19] which divides the OFP approaches into four major categories based on the four methods for detecting errors in running systems: auditing, reporting, monitoring, and tracking.

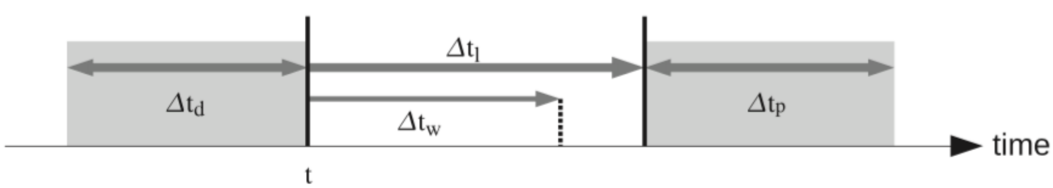

Figure 3. The timeline for OFP [19].

Figure 3 demonstrates the timeline associated with OFP. It does so based on a set of parameters drawn from the literature from the work in [19]. The parameters used by the community to define a predictor are as follows:

- Present Time: t
- Lead Time: Δt_l, is the total time at which a predictor makes an assessment about the current state.
- Data Window: Δt_d, represents the time from which data is used for a predictor uses to make its assessment.
- Minimal Warning Time: Δt_w, is the amount of time required to avoid a failure if one is predicted.
- Prediction Period: Δt_p, is the time for which a prediction is valid. As $\Delta t_p \rightarrow \infty$, the accuracy of the predictor approaches 100% because every system will eventually fail. As this happens, the usefulness of a predictor is diminished.

As the above parameters are adjusted, predictors can become more or less useful. For example, it is clear that as we look further into the future potentially increasing *lead time*, our confidence in our prediction is likely to be reduced. On the other hand, if *lead time* is too small, we may not be able to effectively implement a remediation action. In general, OFP approaches seek to find a balance between the parameters, within an acceptable bound depending on application, to achieve the best possible performance.

Measures of Performance.

In order to accurately compare OFP approaches, standard measures of performance are used. A widely accepted set of measures of performance used by the community is presented in [19]. Before we begin the outline of the measures of performance used to evaluate and compare failure prediction approaches, it is important to note that OFP is done based on statistical analysis of known data. In other words, in order to calculate the following outlined measures of performance, predictors must be evaluated against labeled data sets. Typically, the labeled data set is divided into three parts:

1. Training Set: A data set that allows a prediction model to establish and optimize its parameters

2. Validation Set: The parameters selected in the training phase are then validated against a separate data set

3. Test Set: The predictor is finally run against a final previously unevaluated data set to assess generalizability

During the test phase, true positives (negatives) versus false positives (negatives) are determined in order to compute the measures of performance in this section. The following terms and associated abbreviations are used: *True Positive* (TP) is when failure has been predicted and then actually occurs; *False Positive* (FP) is when failure has been predicted and then does not occur; *True Negative* (TN) is when a state has been accurately classified as non-failure prone; *False Negative* is when a state has been classified as non-failure prone and a failure occurs.

Precision and Recall:.

Precision and recall are the most popular measures of performance used when presenting and comparing OFP approaches. The two are related and often times improving precision results in reduced recall. For example, if a predictor is more accurate with a prediction then that predictor may be less likely to classify a state as failure-prone in an effort to minimize false-positives.

Precision is the number of correctly identified failures over number of all predicted failures. In other words, it reports, out of the predictions of a failure-prone state that were made, how many were correct. In general, the higher the precision the better the predictor. Precision is expressed as:

$$Precision = \frac{TP}{TP + FP} \in [0, 1]$$

Recall is the ratio of correctly predicted failures to the number of true failures. In other words, it reports, out of the actual failures that occurred, how many the predictor classified as failure-prone. In conjunction with a higher precision, higher recall is indicative of a better predictor. Recall is expressed as:

$$Recall = \frac{TP}{TP + FN} \in [0, 1]$$

F-Measure, as defined by [18], is the harmonic mean of precision and recall and represents a trade-off between the two. A higher F-Measure reflects a higher quality predictor. F-Measure is expressed as:

$$F\text{-}Measure = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} \in [0, 1]$$

False Positive Rate and Specificity:.

Precision and recall do not account for true negatives (correctly predicted non-failure-prone situations) which can bias an assessment of a predictor. The following measures of performance take true negatives into account to help evaluators more accurately assess and compare predictors.

False Positive Rate (FPR) is the number of incorrectly predicted failures over the total number of predicted non-failure-prone states. A smaller FPR reflects a higher quality predictor. The False Positive Rate is expressed as:

$$FPR = \frac{FP}{FP + TN} \in [0, 1]$$

Specificity the number of times a predictor correctly classified a state as non-failure-prone over all non-failure-prone predictions made. In general, specificity alone is not very useful since failure is rare. Specificity is expressed as:

$$Specificity = \frac{TN}{FP + TN} = 1 - FalsePositiveRate$$

Negative Predictive Value (NPV) and Accuracy:.

In some cases, we wish to show that a prediction approach can correctly classify non-failure-prone situations. The following measures of performance usually can not stand alone due to the nature of failures being rare events. In other words, a highly "accurate" predictor could classify a state 100% of the time as non-failure-prone and still fail to predict every single true failure.

Negative Predictive Value (NPV) is the number of times a predictor correctly classifies a state as non-failure-prone to the total number all non-failure-prone states during which a prediction was made. Higher quality predictors have high NPVs. The NPV is expressed as:

$$NPV = \frac{TN}{TN + FN}$$

Accuracy is the ratio of all correct predictions to the number of predictions made. Accuracy is expressed as:

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN}$$

Precision/Recall Curve:.

Much like with other predictors, many OFP approaches implement variable thresholds to sacrifice precision for recall or vice versa. That trade-off is typically visualized using a precision/recall curve as seen in Figure 4.

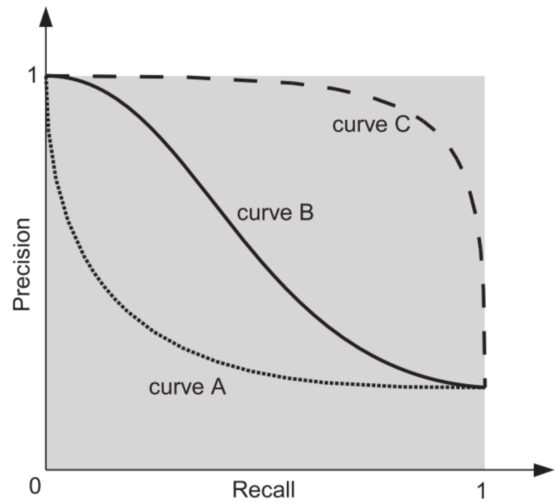

Figure 4. Sample precision/recall curves [19]. Curve *A* represents a poorly performing predictor, curve *B* represents an average predictor, and curve *C* represents an exceptional predictor.

Another popular visualization is the receiver operating characteristic (ROC) curve. By plotting true positive rate over false positive rate one is able to see the predictors ability to accurately classify a failure. A sample ROC curve is shown in Figure 5.

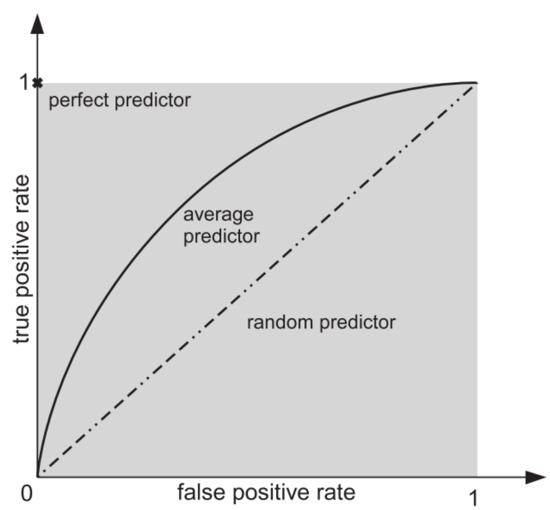

Figure 5. ROC plots of perfect, average, and random predictors [19].

The ROC curve relationship can be further illustrated by calculating the area under the curve (AUC). Predictors are commonly compared using the AUC which is calculated as follows:

$$AUC = \int_0^1 tpr(fpr)\, d\,fpr \in [0, 1],$$

where tpr = true positive rate (recall), and fpr = false positive rate. A pure random predictor will result in an AUC of 0.5 and a perfect predictor a value of 1. The AUC can be thought of as the probability that a predictor will be able to accurately distinguish between a failure-prone state and a non-failure-prone state, over the entire operating range of the predictor.

2.2 Approaches to Online Failure Prediction (OFP)

OFP Taxonomy.

Taxonomies offer natural ways of organizing information and due to the significant body of work surrounding OFP, one was published by Salfner et al. in 2010 [19] classifying many of the OFP approaches in the literature into four major categories. As stated in Section 2.1, these four major categories are defined by the four techniques used to detect faults in real-time: auditing, monitoring, reporting, and tracking. In this research, we update the *reporting* category by classifying approaches to OFP published since its inception. Since this research focusses on real-time *data-driven* device failure prediction approaches, our focus is on the *reporting* category of Salfner's taxonomy. The *reporting* category organizes failure prediction techniques that attempt to classify a state as failure prone based on reported errors. Salfner et al. further organize the reporting category into five sub-categories: rule-based systems; co-occurrence; pattern recognition; statistical tests; classifiers and they are defined as follows:

Rule-Based Systems attempt to classify a system as being failure-prone or not based a set of conditions met by reported errors. Since modern systems are far too complex to build a set of conditions manually, these approaches seek to find automated ways of identifying these conditions in training data. *Co-occurrence* predictors generate failure predictions based on the reported errors that occur either spatially or temporally close together. *Pattern Recognition* predictors attempt to classify patterns of reported errors as failure prone. This research focusses on pattern recognition OFP approaches, which can be visualized in Figure 6. *Statistical Tests* attempt to classify a system as failure-prone based on statistical analysis of historical data. For example, if a system is generating a much larger volume of error reports than it typically does, it may be a sign of pending failure. *Classifiers* assign labels to given sets of error reports in training data and then make failure predictions based on observed labels in real-time data.

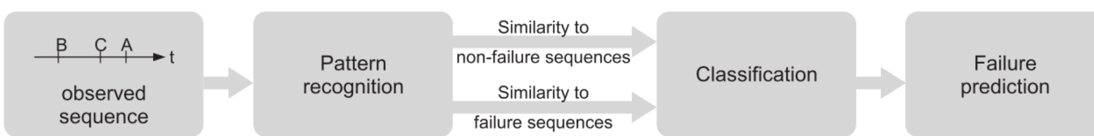

Figure 6. How pattern recognition is accomplished in reported errors [19].

Data-Driven Online Failure Prediction.

The survey published by Salfner et al. covered approaches in every sub-category of the *reporting* category. Since the publication of the survey, we found approaches in two of the subcategories, *pattern recognition* and *classifiers.* We therefore only cover the approaches in those sub-categories of the reporting category here. We found some of the approaches published since Salfner's survey to be difficult to classify because they employ aspects of the other sub-categories in the *reporting* category. More specifically, many of the modern techniques seem to be a blend between the two sub-categories *pattern recognition* and *classifiers*. We believe these categories have been blended because these approaches seem to follow general human intuition when looking for software failures. In other words, we have found that cyber operators tend to look for patterns in reported errors and then classify a situation based on those patterns. We therefore categorize these approaches as *hybrid* approaches.

Pattern Recognition:.

In 2006, Salfner et al. proposed an approach to predicting failures by learning patterns of similar events using a semi-Markov chain model [21]. The model learned patterns of error reports that led to failure by mapping the reported errors to the states in the Markov chain and predicted the probability of the transition to a failure-prone state. They tested the model using performance failures of a telecommunication system and reported a precision of 0.8, recall of 0.923, and an F-measure of 0.8571, which drastically outperformed the models to which it was compared.

Given the results, the semi-Markov Chain model is compelling however, it depends on the sequence of reported errors to remain constant in order to be effective. Today, most software is multi-threaded or distributed so there is no guarantee that the sequence of reported errors will remain constant. Further, the authors reported that this approach did not scale well as the complexity of the reported errors grew.

In 2007, Salfner et al. extended their previous work in [21] using semi-Markov models [20]. They generalized the Hidden Semi-Markov process for a continuous-time model and called it the “Generalized Hidden Semi-Markov Model (GHSMM)” By making this generalization, the model was able to effectively predict the sequence of similar events (or in this case, errors) in the continuous time domain. The authors then tested the model and training algorithm using telecommunication performance failure data and compared it to three other approaches. While this GHSMM model did not perform as well as their previous work, it did outperform the models to which it was compared and more importantly did not depend on the sequence of reported errors. In other words, this new GHSMM model predicted failure for permutations of a known failure-prone sequence making it more suited for a distributed or parallel system.

The GHSMM approach has been well received by the community, although appears to be limited in use to a single system. Unfortunately, this approach as well as its predecessor, does not scale well and does not adapt to changes to the underlying system without retraining.

Classifiers:.

In 2002, Domeniconi et al. [5] published a technique based on support vector machines (SVM) to classify the present state as either failure prone or not based on a window of error reports as an input vector. As Salfner points out in [19], this SVM approach would not be useful without some sort of transformation of the input vector since the exact same sequence of error messages, rotated by one message, would not be classified as similar. To solve this permutation challenge, the authors in [5] used singular value decomposition to isolate the sequence of error reports that led to a failure.

This SVM approach used training data from a production computer environment with 750 hosts over a period of 30 days. The types of failures the system was trying to detect was the inability to route to a web-page and an arbitrary node being down. Many approaches involving SVMs have been explored since and seem to be popular in the community [7, 8, 16, 5, 12].

Hybrid Approaches:.

Since 2012, *Fujitsu Labs* has published several papers on an approach for predicting failure in a cloud-computing environment [22, 25, 24]. Watanabe et al. report on findings after applying a Bayesian learning approach to detect patterns in similar log messages [24, 25]. Their approach abstracts the log messages by breaking them down into single words and categorizing them based on the number of identical words between multiple messages. This hybrid approach removes the details from the messages, like node identifier, and IP address while retaining meaning of the log message.

Watanabe et al.'s hybrid approach attempts to solve the problem of underlying system changes by learning new patterns of messages in real-time. As new messages come in, the model actively updates the probability of failure by Bayesian inference based on the number of messages of a certain type that have occurred within a certain time window. The authors claim that their approach solves three problems: 1) The model is not dependent upon a certain lexicon used to report errors to handle different messages from different vendors; 2) The model does not take into account the order of messages necessarily so in a cloud environment where messages may arrive in different orders, the model is still effective; and 3) The model actively retrains itself so manual re-training does not need to occur after system updates. The model was then tested in a cloud environment over a ninety day period. The authors reported a precision of 0.8 and a recall of 0.9, resulting in an F-measure of 0.847.

In 2012, Fronza et al. [7] introduced a pattern-recognition/classifier hybrid approach that used an SVM to detect patterns in log messages that would lead to failure. The authors used random indexing to solve the problem previously discussed of SVMs failing to classify two sequences as similar if they are offset by one error report. The authors report that their predictor was able to almost perfectly detect non-failure conditions but was poor at identifying failures. The authors then weighted the SVMs to account for this discrepancy by assigning a larger penalty for false negatives than false positives and had better results.

Industry Approaches to Online Failure Prediction.

We thought it necessary to add a brief discussion on how industry seems to have addressed the failure prediction dilemma in recent years. Because hardware has become so easy to acquire, industry seems to have avoided the problem by implementing massive redundancy in their systems. The work in [24, 12] have attributed this problem avoidance to the fact that until recently, implementing and maintaining a failure predictor was difficult. As we decrease the length of the software development life cycle, software updates are being published with increasing frequency leading to rapid changes in underlying systems. These changes can often render a predictor useless without re-training, which is often a manual and resource intensive process.

Adaptive Failure Prediction (AFP) Framework.

The Adaptive Failure Prediction (AFP) Framework by Irrera et al. in [12] seen in Figure 7 presents a new approach to maintaining the efficacy of failure predictors given underlying system changes. The authors conducted a case study implementing the framework using virtualization and fault injection on a web server.

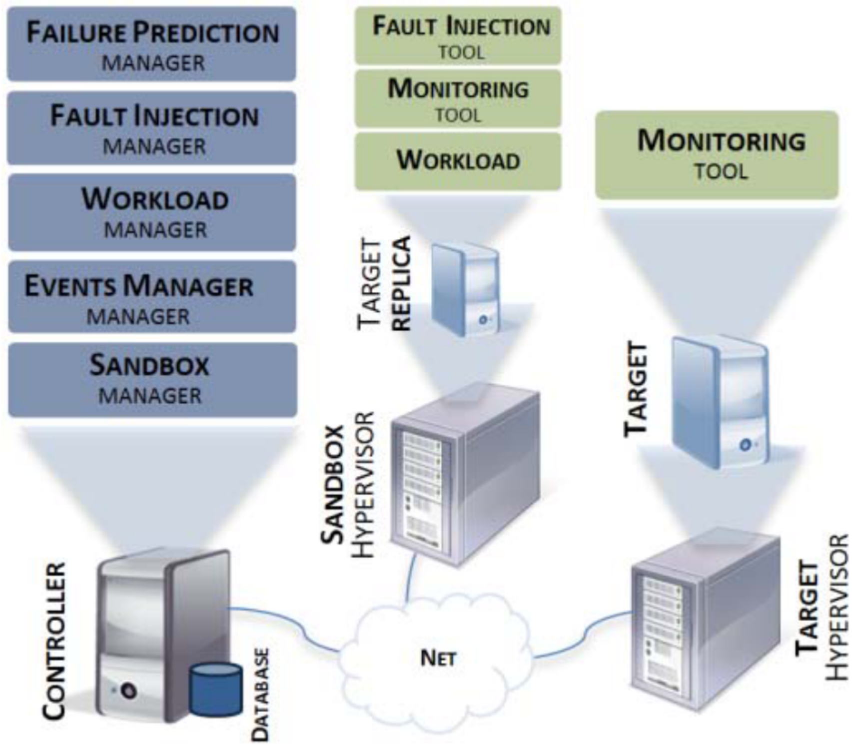

Figure 7. How the AFP framework is implemented [12].

The concept reported used past work by Irrera et al. in [9, 11] to generate failure data by injecting software faults using a tool based on G-SWFIT [6] in a virtual environment for comparing and automatically re-training predictors. In general, the use of simulated data is not well received by the community, however the authors in [10, 11] report evidence supporting the claim that simulated failure data is representative of real failure data. Further, the authors suggest that since systems are so frequently updated and failures are in general rare events, real failure data is often not available. Moreover, the literature shows that even if there is a certain type of failure in training data and a predictor can detect and predict that type of error accurately, it will still miss failures not present in the training data. By injecting the types of faults that one can expect, each failure type will be represented in the training data.

The authors then conducted a case-study using a web server and an SVM predictor, and report their findings demonstrate their framework is able to adapt to changes to an underlying system which would normally render a predictor unusable.

2.3 Summary

This chapter covered the definitions, measures of performance, and approaches that are relevant to this research as organized under the subsection of *reporting* within the OFP field of study. There has been a tremendous amount of research surrounding the topic of OFP and many prediction approaches have been presented. Unfortunately, these approaches do not appear on modern operational systems and failures are still relatively prevalent. Recent approaches as covered here have sought to make predictors more adaptive to the changes in underlying systems in an effort to make implementing existing failure predictors easier. In this work, we plan to extend the adaptive failure prediction framework and further generalize the approach.

III. Methodology

In this chapter we outline our extension to the Adaptive Failure Prediction (AFP) Framework as seen in [12] and our experiment to validate that extension and further generalize the framework. The chapter is split into two sections the first of which outlines each module of the AFP framework in detail. The second describes the process that the framework executes in order to automatically train a failure prediction algorithm to detect pending failures in a target system.

3.1 Architecture

AFP Framework Extension.

Our experiment replicates the experiment in [12] except in place of the web-server a Microsoft (MS) Windows Server running Active Directory (AD) Domain Services. We have applied multiple prediction techniques using this framework to further generalize and validate the framework. The original AFP architecture can be seen in Figure 8 with the parts that would need to be modified for our experiment highlighted and numbered 8,9, and 12. The rest of this section is organized to detail each element of Figure 8 by the associated number.

Components of the Experiment.

In [12], the authors outline multiple modules into which they have broken the AFP Framework for organizational purposes. Our work does not modify these modules, instead, we have taken a more granular approach and present a modified architecture and detail each element of that architecture.

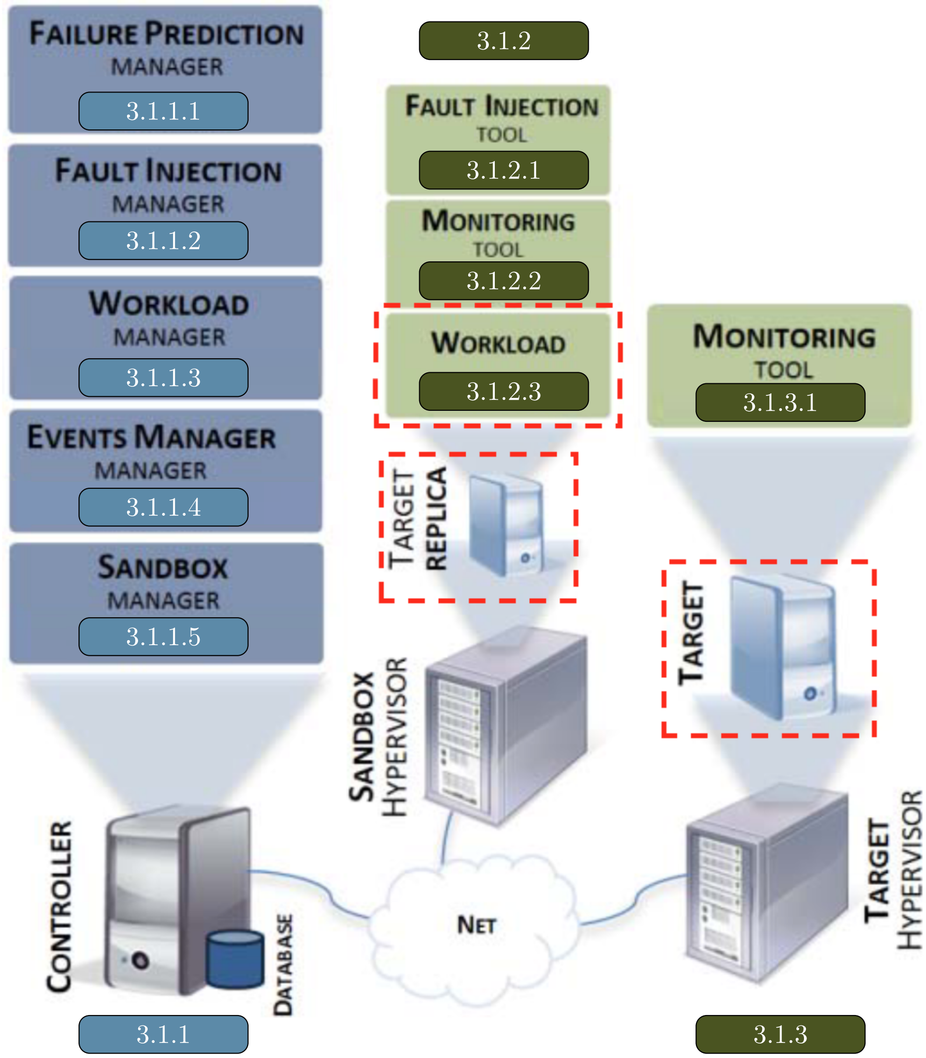

Figure 8. How the AFP framework is implemented [12] with modified components highlighted and numbered for reference.

The following sections will reference the virtual environment in which this architecture was constructed. For reference, this virtual environment was hosted on two VMWare ESXi 5.5 hypervisors each with two 2.6 GHz AMD Opteron 4180 (6 cores each) CPUs and 64 GB memory. The individual virtual machines are detailed in Tables 1, and 2.

Table 1. **Hypervisor 1 Configuration (Sandbox/Target)**

Qty.	Role	Operating System	CPU / Mem.
1	DC	Win. Server 2008 R2	2 / 2 GB
5	Client	Win. 7	1 / 512 MB

Table 2. **Hypervisor 2 Configuration (Controller)**

Qty.	Role	Operating System	CPU / Mem.
1	RDP	Win. Server 2008 R2	1 / 4 GB
1	Log	Ubuntu 14.04 LTS	1 / 1 GB

3.1.1 Controller Hypervisor.

The controller functions in our experiment are split between two systems on a single hypervisor seen on Table 2. One system is a Microsoft Windows Server responsible for workload management and fault injection management. The additional Windows server also hosts remote desktop services to allow our load generator to execute third party authentication with the domain controller. The other system is an Ubuntu 14.04 LTS server that performs the failure prediction management and event management. Each of these functions is detailed in the following sections.

3.1.1.1 Failure Prediction.

The purpose of this module is to actually predict failure using machine learning algorithms that are trained using the labelled training data generated by the rest of this framework. This module is constantly either training a new predictor because a software update occurred, or predicting failure based on log messages and other features produced by the production system.

In [12], the failure prediction management function is performed by an SVM predictor using libsvm. Additionally, the original experiment made use of a database that stored the features used for the failure prediction training algorithm. Our experiment does not modify this module drastically as it has already been shown ([19])

that we are not experiencing a shortage in the number of available quality predictors. We have however chosen a different tool-set to execute the training and predicting phases. Due to its widespread use in the statistical community, our prediction and training algorithms made use of the R programming language.

In our experiment, we train an SVM predictor as done in [12], as well as a... TODO maybe the method in [24]?

3.1.1.2 Fault Injection.

This module is responsible for managing the fault injector installed on the clone of the production target system. The purpose of the fault injector is to force a loaded software application to fail in a realistic way so that the indicators of that failure can be used to train the failure prediction algorithm.

Irrera et al. use a tool implementing the G-SWFIT technique for this module in [12]. G-SWFIT was developed at the University of Coimbra in Coimbra Portugal by Joao Duraes and Henrique Madeira [6]. The method is widely implemented for use in software fault injection both commercially and academically [17, 11, 4, 23]. Recently, studies have questioned the representativeness of the failures generated by G-SWFIT [13]. In each case, the workload generated was critical in creating representative faults. We address this in Section 3.1.1.3.

An additional concern has been that some faults that have been injected by use of the G-SWFIT technique may not elude modern software testing and as a result never actually occur in production software [17]. The recommended remedy is to conduct source code analysis to determine which pieces of code get executed most frequently and avoid fault injection in those areas since they are most likely to be covered by unit tests. Unfortunately, our target is not an open source project and as a result, some of our faults and resulting failures may never happen in a production environment. Fortunately, the tool that we have developed to do software fault injection automatically scans each library loaded by the target executable for fault injection points and then is capable of evenly distributing the faults it does inject.

For our experiment, the original plan was to use the same tool used in [12] for fault injection. Unfortunately, that tool and all prior implementations of the G-SWFIT were incapable of injecting faults into x86-64 binary executables. Further, even many of the commercial products that we evaluated were incapable of dealing with modern address space layout randomization (ASLR). As a result, we have developed a tool capable of injecting faults into all user and kernel mode applications on modern MS Windows operating systems. In this work we introduce an x86-64 implementation of the G-SWFIT technique that we call W-SWFIT for Windows Software Fault Injection Tool. We have published the source code as open source on Github[1] and hope that others may find it useful for many of the reasons cited in the original G-SWFIT paper [6].

The key contributions of W-SWFIT are ASLR adaption and the x86-64 translation we have performed. G-SWFIT works by scanning binary libraries already in memory for patterns (or operators) that match compiled errors in software development. The faults were based on the Orthogonal Defect Classification [2] and can be seen in 3. As pointed out in [19] and [6], failures are ultimately the result of software developer errors. Unfortunately, much of the work done in [6] was on encoding common development errors as IA32 assembly instructions so that working binary executable code could be mutated in memory to introduce these errors in running applications. Our target is strictly an x86-64 (also known as x64 or amd64) application and the patterns identified in [6] are incompatible. Consequently, we needed our tool to mutate x86-64 instructions in the same way. We have implemented each of the operators in [6] in

[1] `https://github.com/paullj1/w-swfit/`

the x86-64 language by translating the operators seen in 3 from IA32 to x86-64. A simple example of this translation can be seen on the entry/exit points of a function in Tables 4, and 5. The rest of the translations can be seen in the source code. In many cases, the translation was very simple, but in others, the IA32 patterns did not cleanly map to x86-64 byte code. When this happened, great care was taken to ensure the pattern was correctly mapped to x86-64.

Table 3. Table of Faults Injected [6]

Type	Description	ODC Classes
MIFS	Missing "If (cond) { statement(s) }"	Algorithm
MFC	Missing function call	Algorithm
MLAC	Missing "AND EXPR" in expression used as branch	Checking
MLPC	Missing small and localized part of the algorithm	Algorithm
WVAV	Wrong value assigned to a value	Assignment
MVI	Missing variable initialization	Assignment
MVAV	Missing variable assignment using a value	Assignment
WPFV	Wrong variable used in parameter of function call	Interface

Table 4. Funtion Entry/Exit Patterns (IA32) [6]

Module Entry Point		Module Exit Point	
Instruction Sequence	**Explanation**	**Instruction Sequence**	**Explanation**
push ebp	stack frame	move esp,ebp	stack frame
mov ebp, esp	setup	pop ebp	cleanup
sub esp, immed		ret	

Table 5. Funtion Entry/Exit Patterns (x86-64) [6]

Module Entry Point		Module Exit Point	
Instruction Sequence	**Explanation**	**Instruction Sequence**	**Explanation**
push rbp	stack frame	add rsp, immed	stack frame
sub rsp, immed		pop rbp	cleanup
mov rbp, rdx	setup	ret	

3.1.1.3 Workload.

The Workload module is responsible for orchestrating the load against the target system in the sandbox hypervisor. Without this module, it could take a very long time for an injected fault to manifest itself as a failure. Consider a missing *free* statement and the consequent memory leak. A production target server may have a considerable amount of memory and the leak could be very small. To accelerate the possibility of failure occurring, realistic load must be generated against the sandbox clone of the production target.

In the original AFP case study, a Windows XP based web-server was used for a target and therefore the load generation was done by a simple web request generator. As previously mentioned, realistic workload is critical in generating realistic failure and consequently training a useful predictor. As a result, we have placed much emphasis on this module. Since our target is not a web server, we could not use the same load generator used in [12]. Our initial search for a load generator yielded a tool developed by Microsoft that initiated remote desktop connections to aid in sizing a terminal services server[2]. By executing a remote desktop session, the authentication and DNS functions of the domain controller would also be loaded. Unfortunately, this tool is no longer maintained and would not execute on our target machine[3]. Further searches for tools that would sufficiently load our domain controller did not produce any results. Consequently, we began developing our own tool and introduce it here.

The Distributed PowerShell Load Generator (D-PLG) is a collection of Microsoft PowerShell scripts designed to generate realistic traffic that will sufficiently load a Microsoft domain controller. Other network traffic generators typically work by replaying traffic captured on a live network. Unfortunately, due to the cryptographic

[2] `http://www.microsoft.com/en-us/download/details.aspx?id=2218`

[3] `https://social.technet.microsoft.com/Forums/windowsserver/en-US/2f8fa5cf-3714-4eb3-a895-c30e2b26862d/debug-assertion-failed-sockcorecpp-line-623`

nature of authentication, simply replaying traffic will not load a service since the timestamps and challenge responses will no longer be valid. As a result, any replayed traffic will be dropped and ignored by a live domain controller. D-PLG solves this problem by making native authentication requests by use of built-in PowerShell cmdlets (command-lets). By doing this, we can guarantee that realistic authentication requests are sent to our domain controller and are actually processed. We have evaluated the functions our domain controller performs and have built D-PLG to be able to sufficiently load each of the services responsible for performing those functions.

For our purposes, we considered the DC as employed in the Air Force Network architecture. After careful analysis, we have determined that the major roles in that architecture being performed are authentication and domain name service (DNS). Additionally, by use of native cmdlets, D-PLG is capable of generating four kinds of traffic: web, mail, file sharing, and MS remote desktop protocol (RDP). We use MS's Powershell environment to generate the traffic in an effort to make the traffic as real as possible. After building the tool we constructed and tested it on a scale model of a production environment. The scaled simulation network was built using the recommendations of the Microsoft community for sizing a domain controller [14] and tested by running the tool on five client machines against our domain controller for five rounds of five minutes. The results of this test can be seen in Figures 9, 10, 11.

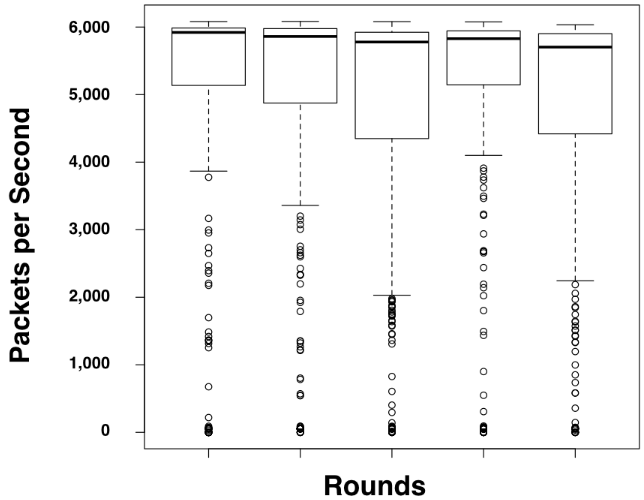

Figure 9. How many packets per second were sent or received by the domain controller across all five rounds of the first test. In each test, we captured approximately 1.8 million packets.

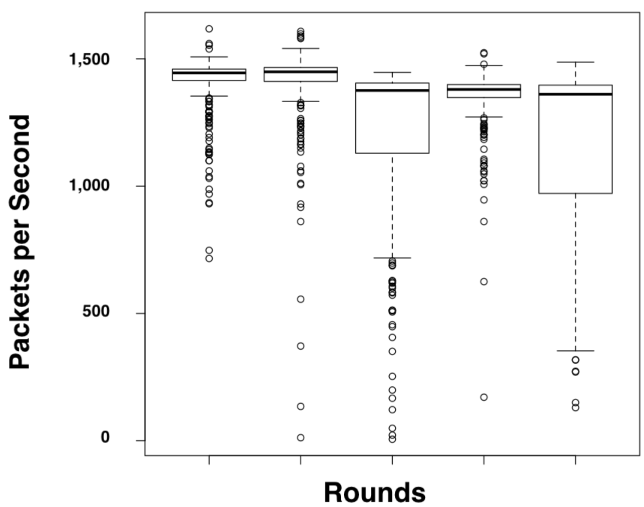

Figure 10. How many packets per second were sent or received by one of the clients across all five rounds of the first test.

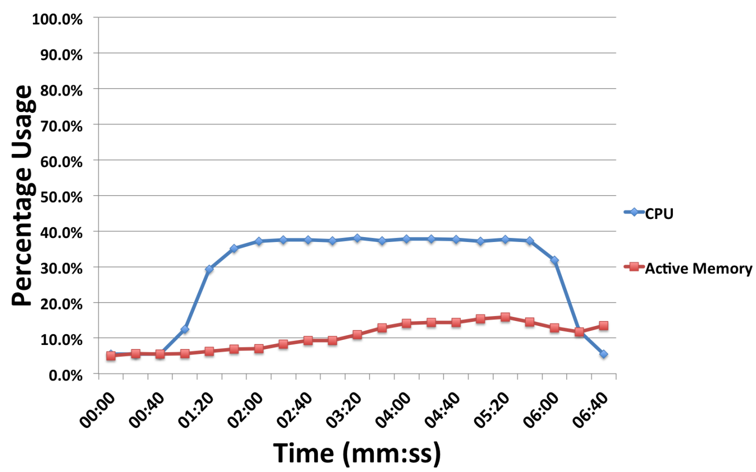

Figure 11. Domain controller CPU and memory utilization during the first test.

D-PLG makes use of client machines running a Windows operating system with PowerShell version 4.0 or newer. The controller asks each machine to generate a configurable list of requests at evenly spaced intervals for a configurable duration of time. While this may not be realistic network traffic, it will produce realistic load against a domain controller. Since D-PLG depends on the use of client machines, we recommend conducting any load generation during off-peak hours if spare client sized machines are not available. It should be noted however, that even with our poorly resourced client machines (seen in 1), we were able to generate fifteen thousand authentication sessions over a five minute period; approximately 10 authentication sessions per machine, per second. With modern workstations, the impact on these client machines will likely be negligible and could likely be in use during load generation.

Based on these results, and that a production domain controller should be at approximately 40% CPU utilization during peak utilization [14], we have concluded that D-PLG is capable of sufficiently loading our domain controller over a sustained period of time for the purposes of implementing the AFP framework and is used in this research. Further, we have concluded that D-PLG is capable of scaling to provide load against higher capacity domain controllers by using only a few client machines. There are many more uses for a load generator of this type and we were not able to find a tool that is capable of creating the same type of load so we have published these scripts on Github[4] for others to use.

3.1.1.4 Events Manager.

This module is responsible for receiving and managing log messages and other events that may be used to train the failure prediction algorithm. Irrera et al. use the *Logman* tool for event management in their original case study in [12]. Since we have modelled our environment after the Air Force enterprise environment, we have chosen the *Solar Winds* log forwarding tool to perform the functions in this module as it is already present on many of the Air Force domain controllers. The domain controllers on the Sandbox and Target hypervisors forward all events to the Ubuntu virtual machine with the *rsyslog* server daemon configured to receive all messages. These messages are then processed and added to a SQL database for training and prediction.

3.1.1.5 Sandbox Management.

The purpose of the sandbox management module is to orchestrate the virtual clone of the production system that is made when a new predictor is to be trained. As Irrera et al. point out, we cannot reasonably expect to be able to inject faults and cause failures in production systems, so a virtual clone must be created for that purpose.

Our sandbox is managed manually using Virtual Machine (VM) snapshots. After an initial stable state was configured, snapshots of every component of the architecture were taken so that they could be reset after iterations of the experiment. It is

[4]https://github.com/paullj1/AFP-DC/tree/master/D-PLG

important to note here that because VMWare has documented APIs, in future work, this function could be automated.

3.1.2 Sandbox Hypervisor.

The sandbox has been constructed on a single hypervisor implemented as seen on Table 1. The following sections outline each module within this module.

3.1.2.1 Fault Injection.

This module is responsible for causing the target application to fail so that labelled failure data can be generated in a short period of time. As described in Section 3.1.1.2, we have developed a tool that implements the G-SWFIT technique developed by Duraes et al. in [6] for fault injection. The execution is controlled by our Windows Server virtual machine on the Controller hypervisor through PowerShell remote execution to reduce our interaction and potential to introduce bias into the training data. The tool allows us to inject a comprehensive list of faults into the AD Services processes and binary libraries which are mostly contained within the 'lsass.exe' process. Since many of the critical functions performed by the AD Services processes are performed in one library called 'ntdsa.dll', it is the focus of our fault injection. We also focus attention in generating the training data

3.1.2.2 Monitoring.

The purpose of this module is to capture some evidence or indication of pending failure so that it may be used to train a prediction algorithm. Irrera et al. use the *Logman* tool in their original study but because we are modelling our architecture after production Air Force networks, we have chose the *Solar Winds* log forwarding tool. The tool is a lightweight application that simply forwards windows events to a syslog server.

3.1.2.3 Workload.

The workload module is likely the most critical module in the entire framework. Its purpose is to create realistic work for the target application to do before faults are injected. If this workload is not realistic, then the failures that occur after fault injection will not be representative of real failures and any data or indicators collected cannot be used to train an effective prediction algorithm.

Irrera et al. used a web traffic generator called TPC-W in their original study because their target was a web server. Because our service does not respond to web-requests and a tool had not previously been written for this application, we have designed D-PLG, a tool that generates approximately ten full-stack authentication sessions requests per second in order to sufficiently load the domain controller. In our experiment, this tool is installed on five client machines which make it capable of sufficiently loading the domain controller that we have built for this experiment.

3.1.3 Target Hypervisor.

The target hypervisor was constructed as a clone of the sandbox hypervisor seen on Table 1. The following section outlines the monitoring tool installed on the domain controller on this hypervisor. It should be noted here that while we did clone the client machines for convenience, they were not used in our experiments.

3.1.3.1 Monitoring.

This module is exactly the same as the sandbox monitoring module and for our experiment, we have used the *Solar Winds* syslog forwarding tool. To ensure that the messages that are sent are uniquely identified by the controller, the hostname of the target machine must be different from the hostname of the sandbox target machine.

3.2 Experiment

In this work, we seek to generalize the Adaptive Failure Prediction framework outlined in [12] by conducting another case study with a Microsoft Windows Server acting as an active directory service. We then take the case study a step further and show that the approach holds for other predictors (maybe). Specifically, we will report findings after implementing this framework with the predictor used by Irrera et al. in their case study and the predictor used by Watanabe et al. here [24].

We set up the experiment by simulating a production active directory server. In order to generate realistic load on this server we employed a tool by Microsoft for terminal services size planning [15]. The tool is capable of executing up to fifty remote desktop connections to a remote desktop server from a single client workstation. By initiating a remote desktop connection, the client workstation must authenticate using the active directory server which gives us realistic traffic that simulates the actual role domain controllers play in production environments.

In this section we outline the step-by-step procedure by which we evaluate the AFP framework and how effective it is when used on Windows Server deployments. We do this by dividing the steps taken in our experiment into three major phases as seen in Figure 12: Running State; Re-Training; Predictor Replacement.

Running State.

In this phase, the system has a working predictor providing input to either an automated decision system or a human-machine team. The system in this phase is making failure predictions about the current state based on the last run of the AFP.

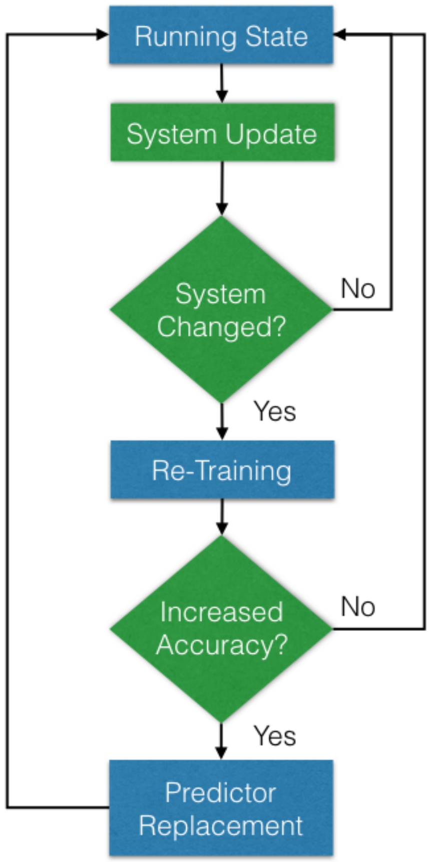

Figure 12. The flow of the major steps involved in the AFP framework execution [12].

IV. Experimental Results and Analysis

V. Conclusion and Future Work

In conclusion...

Bibliography

1. A. Avižienis, J. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, 2004.

2. Norm Bridge and Corinne Miller. Orthogonal Defect Classification Using Defect Data to Improve Software Development. *Software Quality*, 3(1):1–8, 1998.

3. G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot-a technique for cheap recovery. In *OSDI*, volume 4, pages 31–44, 2004.

4. D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa. Experimental Analysis of Binary-Level Software Fault Injection in Complex Software. *2012 Ninth European Dependable Computing Conference*, pages 162–172, 2012.

5. C. Domeniconi, C. Perng, R. Vilalta, and S. Ma. A classification approach for prediction of target events in temporal sequences. *Lecture notes in computer science*, pages 125–137, 2002.

6. J. Duraes and H. Madeira. Emulation of software faults: A field data study and a practical approach. *Software Engineering, IEEE Transactions on*, 32(11):849–867, Nov 2006.

7. I. Fronza, A. Sillitti, G. Succi, M. Terho, and J. Vlasenko. Failure prediction based on log files using Random Indexing and Support Vector Machines. *Journal of Systems and Software*, 86(1):2–11, 2013.

8. E. Fulp, G. Fink, and J. Haack. Predicting Computer System Failures Using Support Vector Machines. *Proceedings of the First USENIX conference on Analysis of system logs*, pages 5–5, 2008.

9. I. Irrera, J. Duraes, H. Madeira, and M. Vieira. Assessing the impact of virtualization on the generation of failure prediction data. *Proceedings - 6th Latin-American Symposium on Dependable Computing, LADC 2013*, pages 92–97, 2013.

10. I. Irrera, J. Duraes, M. Vieira, and H. Madeira. Towards Identifying the Best Variables for Failure Prediction Using Injection of Realistic Software Faults. *2010 IEEE 16th Pacific Rim International Symposium on Dependable Computing*, pages 3–10, 2010.

11. I. Irrera and M. Vieira. A Practical Approach for Generating Failure Data for Assessing and Comparing Failure Prediction Algorithms. *2014 IEEE 20th Pacific Rim International Symposium on Dependable Computing*, pages 86–95, 2014.

12. I. Irrera, M. Vieira, and J. Duraes. Adaptive Failure Prediction for Computer Systems: A Framework and a Case Study. *2015 IEEE 16th International Symposium on High Assurance Systems Engineering*, pages 142–149, 2015.

13. N. Kikuchi, T. Yoshimura, R. Sakuma, and K. Kono. Do injected faults cause real failures? A case study of linux. *Proceedings - IEEE 25th International Symposium on Software Reliability Engineering Workshops, ISSREW 2014*, pages 174–179, 2014.

14. S. Makbulolu and G. Geelen. Capacity planning for active directory domain services, 2012.

15. Microsoft. Download remote desktop load simulation tools from official microsoft download center.

16. J. Murray, G. Hughes, and K. Kreutz-Delgado. Machine Learning Methods for Predicting Failures in Hard Drives: A Multiple-Instance Application. *J. Mach. Learn. Res.*, 6:783–816, 2005.

17. R. Natella, D. Cotroneo, J. Duraes, and H. Madeira. Representativeness analysis of injected software faults in complex software. *Proceedings of the International Conference on Dependable Systems and Networks*, pages 437–446, 2010.

18. C.J. Rijsbergen. v.(1979). *Information retrieval*, 2, 1979.

19. F. Salfner, M. Lenk, and M. Malek. A survey of online failure prediction methods. *ACM Comput. Surv.*, 42(3):10:1–10:42, March 2010.

20. F. Salfner and M. Malek. Using Hidden Semi-Markov Models for Effective Online Failure Prediction. *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pages 161–174, 2007.

21. F. Salfner, M. Schieschke, and M. Malek. Predicting failures of computer systems: a case study for a telecommunication system. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8 pp.–, April 2006.

22. M. Sonoda, Y. Watanabe, and Y. Matsumoto. Prediction of failure occurrence time based on system log message pattern learning. *Network Operations and Management Symposium (NOMS), 2012 IEEE*, (4):578–581, 2012.

23. K. Umadevi and S. Rajakumari. A Review on Software Fault Injection Methods and Tools. pages 1582–1587, 2015.

24. Y. Watanabe. Online Failure Prediction in Cloud Datacenters. 50(1):66–71, 2014.

25. Y. Watanabe, H. Otsuka, M. Sonoda, S. Kikuchi, and Y. Matsumoto. Online failure prediction in cloud datacenters by real-time message pattern learning. *CloudCom 2012 - Proceedings: 2012 4th IEEE International Conference on Cloud Computing Technology and Science*, pages 504–511, 2012.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
10–02–2017	Master's Thesis	Sept 2015 — Mar 2017

4. TITLE AND SUBTITLE

DATA DRIVEN DEVICE FAILURE PREDICTION

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Paul L. Jordan

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCS/ENG/17-M01

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Information Assurance Education and Training Program
9800 Savage Road
Fort Meade, Maryland 20755-6744
410-854-6206
Email: gmellis@nsa.gov; aeshaff@nsa.gov

10. SPONSOR/MONITOR'S ACRONYM(S)

NIETP

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

TODO

15. SUBJECT TERMS

Thesis, Failure Prediction, Machine Learning

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. M. J. Mendenhall, AFIT/ENG
					19b. TELEPHONE NUMBER *(include area code)*
U	U	U	U	???	(937) 255-????, x????; mike.mendenhall@afit.edu