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

The views expressed in this document are those of the author and do not reflect the official policy or position of the United States Air Force, the United States Department of Defense or the United States Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

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

May 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. G. L. Peterson
Chair

Maj A. C. Lin, PhD
Member

Dr. M. J. Mendenhall
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

Page

Abstract iv

Acknowledgements v

List of Figures viii

List of Tables ix

I. Introduction 1

1.1 Problem Statement 2
1.2 Hypothesis 4
1.3 Research Goals 4
1.4 Impact of Research 5
1.5 Assumptions and Limitations 5
1.6 Results 5

II. Overview of Online Failure Prediction 7

2.1 Background 7
2.1.1 Definitions 7
2.2 Approaches to Online Failure Prediction (OFP) 11
2.2.1 OFP Taxonomy 11
2.2.2 Data-Driven Online Failure Prediction 12
2.2.3 Industry Approaches to Online Failure Prediction 15
2.2.4 Adaptive Failure Prediction (AFP) Framework 16
2.3 Summary 17

III. Methodology 19

3.1 Failure Data Generation 19
3.1.1 Preparation Phase 20
3.1.2 Execution Phase 20
3.1.3 Training Phase 22
3.2 Implementation of the AFP 26
3.2.1 AFP Framework Implementation 26
3.2.2 AFP Modules 26
3.2.3 Controller Hypervisor 27
3.2.4 Sandbox Hypervisor 36
3.2.5 Target Hypervisor 38
3.3 Extenstions to the AFP 39

Page

IV. Experimental Results and Analysis 40

4.1 Performance Measures 40
4.1.1 Precision and Recall: 41
4.1.2 False Positive Rate and Specificity: 42
4.1.3 Negative Predictive Value (NPV) and Accuracy: 42
4.1.4 Precision/Recall Curve: 43

V. Conclusion and Future Work 45

Appendix A. W-SWFIT 46

A.1 FaultInjection.cpp 46

Appendix B. ResourceLeak 51

B.1 resourceleak.cpp 51
Bibliography 53

List of Figures

Figure Page

1 Proactive Fault Management [19] 8

2 Failure Flow Diagram [19] 10

3 Online Failure Prediction [19] 10

4 Pattern recognition in reported errors [19] 12

5 AFP Framework Implementation [13] 16

6 AFP Execution Phase [13] 21

7 AFP Training Phase [13] 23

8 Annotated AFP Framework [13] 27

9 Domain Controller Packets per Second 34

10 Client Packets per Second 34

11 Test 1: Domain Controller Metrics 35

12 Sample Precision/Recall Curves [19] 43

13 Sample ROC Plots [19] 44

List of Tables

Table Page

1 Hypervisor 1 Configuration (Sandbox/Target) . . . 26

2 Hypervisor 2 Configuration (Controller) . . . 28

3 Table of Faults Injected [7] . . . 31

4 Funtion Entry/Exit Patterns (IA32) [7] . . . 31

5 Funtion Entry/Exit Patterns (x86-64) [7] . . . 31

DATA DRIVEN DEVICE FAILURE PREDICTION

I. Introduction

As dependency upon computers grows, so too do the associated risks. Computer systems are all around us. Some of these computer systems play insignificant roles in our lives while others are responsible for sustaining our lives. Unfortunately, the software that controls these systems is written by humans and consequently subject to human error. As a result, these systems are prone to failure which in many cases may not be a big deal, but in others, could have severe consequences. Every day, critical infrastructure and Air Force missions systems depend on the reliability of computer systems. As a result, being able to predict pending failure in computer systems can offer tremendous and potentially life-saving applications in today's technologically advanced world. While actually being able to accurately predict failure has unfortunately not been proven possible, there has been work over the past several decades attempting to make educated predictions about the failure of machines through the use of machine learning algorithms [19]. Unfortunately, much of this work has gone unused.

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 problems caused by machine failure 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 achieve 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. Chapter II outlines the recent work done in this field, much of which has gone unimplemented due to the complex and manual task of training a prediction model. If the underlying system changes at all, the efficacy of a prediction model can be drastically reduced if not rendered completely useless until it is retrained. Furthermore, training requires access to labelled training data. Since failure is such a rare event, access to this type of training data may not be possible.

Irrera et al. [13] presented a potential solution in 2015 to automate the process of dynamically generating failure data and using it to train a predictor after an underlying system change. They defined a framework for implementing this process and called it the Adaptive Failure Prediction (AFP) Framework. This research explores an implementation of that framework. More specifically, it presents results after implementing the AFP using a Microsoft Windows Server domain controller. Successive software updates are then applied until the model selected becomes useless, the framework is then allowed to re-train a new predictor.

1.1 Problem Statement

According to the operators at the 33d Network Warfare Squadron, 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 several ways. 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].

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 [13]. Furthermore, in most cases OFP implements machine learning algorithms that require manual re-training after underlying system changes. More troubling is that 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 [13] uses simulated faults to automatically re-train a prediction algorithm to make implementing OFP approaches easier. This work extends that framework to capture developments since its writing and generalize it so it works for a broader class of devices by translating and implementing the fault injection tool from the IA32 architecture to the x86-64 architecture and developing the fault load to produce even more representative failures.

1.2 Hypothesis

The implementation of the AFP framework with a more representative fault load for the Microsoft Windows enterprise infrastructure will lead to accurate failure prediction with better lead time than is available today without any prediction model. This research will test this hypothesis by implementing the AFP in a scaled virtual environment and evaluating its performance after successive software updates. Prior to this research, the implementation of the AFP was not possible on modern Microsoft Windows infrastructure because the fault injection tool had not been written for the x86-64 architecture. Additionally, the faults produced and consequently predicted were the result of first-order software failure. This research evaluates the performance of the AFP when second and third order failures are introduced.

1.3 Research Goals

The goal of this research is to inform decision makers about the potential benefits of implementing a machine learning based failure prediction model to predict failures in computer systems. This research should demonstrate the efficacy of the AFP framework and proposed extensions on the Microsoft Windows enterprise architecture. A long-term goal of this research is to drive the improvement of the AFP framework and increase its adoption and resulting cost savings. In the near-term, the translation of the IA32 G-SWFIT tool to the x86-64 architecture enables the same advantages of software fault injection for 32-bit systems on 64-bit systems [7].

1.4 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 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 and equipment. Further advantages are difficult to quantify such as a decreased risk of mission failure due to network service outage.

1.5 Assumptions and Limitations

This research assumes indicators of failure are present and available with enough lead-time to accurately make decisions and take mitigation action should failure be predicted based on these indicators. Furthermore, it has not been proven possible to accurately predict future events without a priori knowledge. This research presents a method of predicting failure, but this method is completely useless at predicting *act of God* events. Further, this method is capable of predicting system failure based on underlying software failures and will unlikely provide any useful information about malicious attacks against the target system.

1.6 Results

Because a prediction method is not presently deployed on any Air Force network, any level of dependable prediction will be better than what is currently available. This research will show that after an underlying system change, this method of predicting failure will be capable of automatically training a more effective prediction algorithm so that this technique can be implemented on an Air Force network with little to no impact on manpower. Consequently, it is expected that this research will inform decision makers and allow them to implement this technique in a production environment.

Specifically, the technique presented in this research could most effectively be implemented and used by the Cyber Security and Control System (CSCS) weapon system employed at the 561st and 83d Network Operation Squadrons (NOS) and their associated detachments to reduce the number of network service outages, increasing uptime, leading to improved mission effectiveness in both the support and operational domains. Further, this technique should be general enough to be employed outside of the Air Force to increase mission effectiveness across the Department of Defense (DOD). External to the DOD, this research further generalizes an approach that could be used to help increase availability of nearly any computer system.

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.

The rest of this chapter is organized as follows. In Section 2.1, a brief background on the topic of online failure prediction (OFP) is given including definitions, terminology, and measures of performance used by the community. In Section 2.2, the approaches relevant to this research are presented followed by a brief summary in Section **??**.

2.1 Background

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

2.1.1 Definitions.

2.1.1.1 Proactive Fault Management:.

Salfner, et al. [19] define proactive fault management (PFM) as the process by which faults are handled in a proactive way, analogous with *fault tolerance* and basically consisting of four steps: online failure prediction, diagnosis, action scheduling, and action execution as shown in Figure 1. The final three stages of PFM define how much lead time is required to avoid a failure when predicted during OFP. *Lead time* is defined as the time between when failure is predicted and when that failure will occur. Lead time is one of the most critical elements of a failure prediction approach.

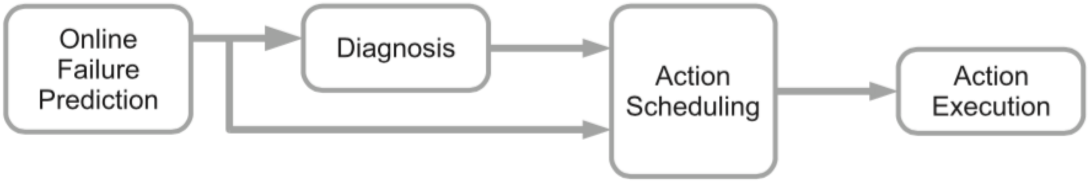

Figure 1. The stages of proactive fault management [19].

OFP is defined as the first step in PFM shown in Figure 1. OFP is the act of analyzing the running state of a system in order to predict a failure in that system. 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*.

2.1.1.2 Faults, Errors, Symptoms, and Failures:.

This research uses 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 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 appear 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.

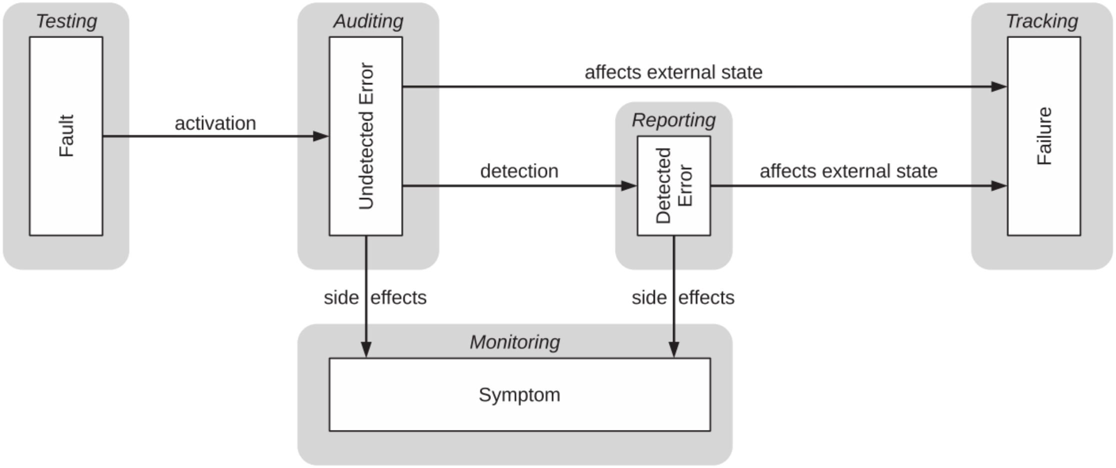

Figure 2. How faults and errors evolve into failure with the associated methods for detection represented by enclosing gray boxes [19].

Figure 2 illustrates how a software fault can evolve into a failure. Faults, errors, symptoms, and failures can be further categorized by how they are detected also shown in Figure 2. Salfner, et al. **??** introduces a taxonomy of OFP approaches and classifies failure prediction approaches by the stage at which a fault is detected as it evolves into a failure: auditing, reporting, monitoring, and tracking. Testing is left out because it does not help detect faults in an online sense.

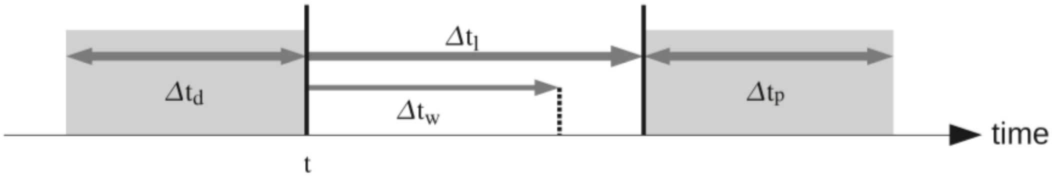

Figure 3. The timeline for OFP [19].

Figure 3 demonstrates the timeline associated with OFP. 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 $\Delta t_p \to \infty$, 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 a predictor looks further into the future potentially increasing *lead time*, confidence in its prediction is likely to be reduced. On the other hand, if *lead time* is too small, there will likely not be enough time to effectively take 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.

2.2 Approaches to Online Failure Prediction (OFP)

2.2.1 OFP Taxonomy.

The taxonomy by Salfner, et al. [19] classifies many of the OFP approaches in the literature into four major categories. These four major categories are defined by the four techniques used to detect faults in real-time: auditing, monitoring, reporting, and tracking.

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. [19] further organize the reporting category into five sub-categories: rule-based systems; co-occurrence; pattern recognition; statistical tests; and classifiers.

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 4. *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.

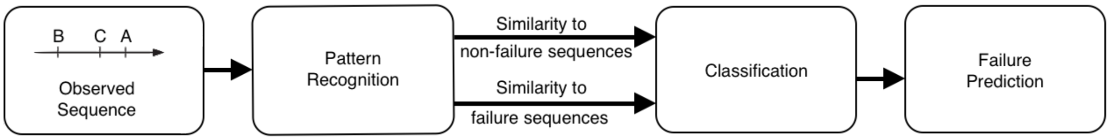

Figure 4. How pattern recognition is accomplished in reported errors [19].

2.2.2 Data-Driven Online Failure Prediction.

2.2.2.1 Pattern Recognition:.

Salfner, et al. [21] proposed an approach to predicting failures by learning patterns of similar events using a semi-Markov chain model. 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.

2.2.2.2 Classifiers:.

Domeniconi, et al. [6] 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 [6] 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 [8, 9, 16, 6, 13].

2.2.2.3 Hybrid Approaches:.

Fujitsu Labs has published several papers on an approach for predicting failure in a cloud-computing environment [22, 25, 24]. Watanabe, et al. [24, 25] report on findings after applying a Bayesian learning approach to detect patterns in similar log messages. 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 [24] 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.

Fronza, et al. [8] 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.

2.2.3 Industry Approaches to Online Failure Prediction.

Because hardware has become so easy to acquire, industry has sought to avoid the problem of software failure by implementing massive redundancy in their systems. The work in [24, 13] attributes the 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.

Redundancy is not without problems however. Implementing redundant systems to avoid the failure problem can be expensive and can add overhead and complexity making a system more difficult to manage.

2.2.4 Adaptive Failure Prediction (AFP) Framework.

The Adaptive Failure Prediction (AFP) Framework by Irrera, et al. in [13] seen in Figure 5 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.

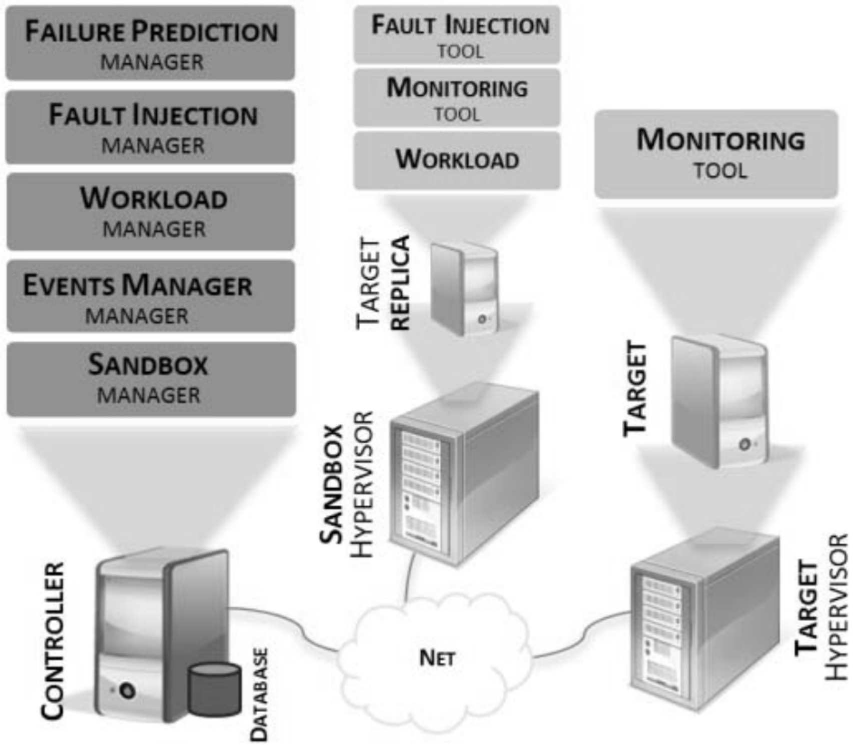

Figure 5. How the AFP framework is implemented [13].

The concept reported used past work by Irrera et al. [10, 12] to generate failure data by injecting software faults using a tool based on G-SWFIT [7] 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 [11, 12] 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 is 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. They reported good results and concluded that the AFP is an effective tool. Unfortunately, the AFP is not a universal solution and requires significant work to be implemented on a modern Microsoft Windows enterprise network. Furthermore, the fault load previously explored does not completely represent all possible failures.

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

The purpose of the AFP framework is to automate the generation of realistic labelled failure data for the purposes of automatically training a failure prediction algorithm. The framework breaks down into modules so that the it can be more easily adapted for different applications. This chapter presents three topics. The first describes the process that the extended framework executes in order to generate the labelled training data and automatically train a failure prediction algorithm. The second outlines each module of the extended AFP framework and their associated extensions in detail. The final section outlines extensions to the AFP not covered in the other two sections.

This chapter outlines the implementation and extensions to the Adaptive Failure Prediction (AFP) Framework as seen in [13] as well as an experiment to validate those extensions and further generalize the framework. The AFP was originally tested on a single system running an operating system that has been deprecated. Consequently, the results from the case study conducted using the AFP are limited in utility and require generalization to be useful to the general community.

3.1 Failure Data Generation

This work extends the Adaptive Failure Prediction framework [13] by conducting another case study with a Microsoft Windows Server acting as an active directory service with a new implementation of the G-SWFIT technique for the x86-64 architecture as well as a more representative fault load. The case study is then taken a step further by showing that the approach holds for other predictors (maybe). Specifically, findings are reported after implementing this framework with the predictor used by Irrera, et al. [13] in their case study and the predictor used by Watanabe, et al. [24].

This section outlines the step-by-step procedure by which the extended AFP is evaluated to show how effective it is when used on Windows Server deployments. This is done by dividing the steps taken in an experiment into the three major phases as defined in [13]: preparation phase, execution phase, and training phase.

3.1.1 Preparation Phase.

In this phase the AFP is prepared to run for the first time as described in [13]. The Cross Industry Standard Process for Data Mining (CRISP-DM) [4] should be applied to this situation when evaluating how to best apply the AFP for a particular target. For the purposes of this research, our focus is on the Microsoft Windows Directory Services and predicting failure in those services. To demonstrate the efficacy of the AFP, a predictor must be evaluated before and after a significant software update. As a result, the most critical preparation made in evaluating this framework is to hold back all software updates on the target system prior to the first run of the execution phase.

This phase is ultimately the implementation of the framework. Each module of the implementation for this work is detailed in Section 3.2 and is therefore not discussed further here.

3.1.2 Execution Phase.

A general outline of this process can be seen in Figure 6. This phase is divided into three major states: data collection and failure prediction, event checking, and training/update as described in this section.

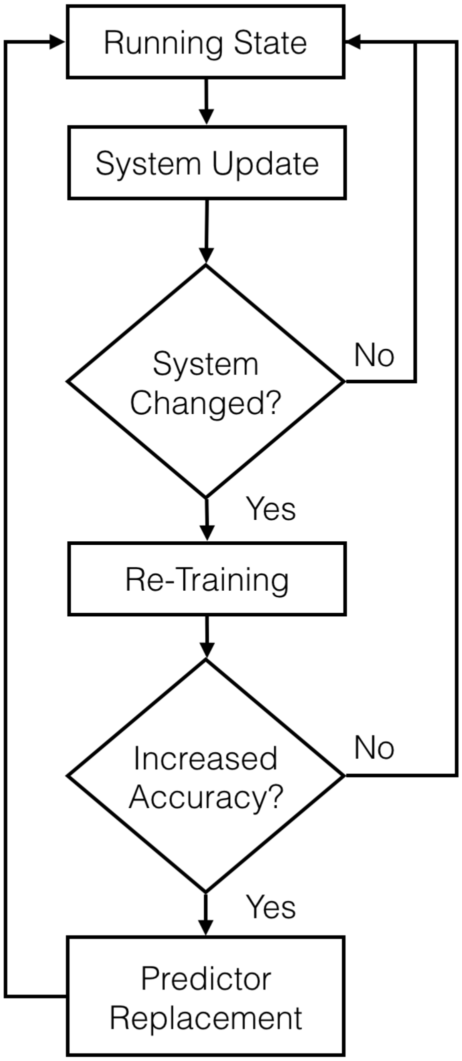

Figure 6. The flow of the major steps involved in the AFP framework execution phase [13].

3.1.2.1 Data Collection and Failure Prediction.

In this phase, the system has a working predictor providing input to some sort of decision system. It should be noted here that this decision system does not have to be automated. The system in this phase is making failure predictions about the current state based on the last run of the training phase. For the Air Force application, an operator will be the decision maker and the output of the AFP will be a message to that operator.

3.1.2.2 Event Checking.

Concurrent with the data collection and failure prediction sub-phase, the AFP continuously monitors events that may alter the underlying system. For this experiment, these events are software updates. The output of each episode of this phase is a binary decision to either begin the training phase, or not. In this experiment, the training phase is manually triggered upon completion of a major software update.

3.1.2.3 Failure Predictor (Re-)Training and Update.

The purpose of this sub-phase is to initiate the training phase and compare its results (a new predictor) with the currently employed predictor. Should the new predictor perform better, the old predictor is replaced by the new.

3.1.3 Training Phase.

The training phase is broken down into five major steps: target replication, data generation & collection, dataset building, predictor training, and analysis. The general flow can be seen in Figure 7. Each phase is outlined in the following sub-sections.

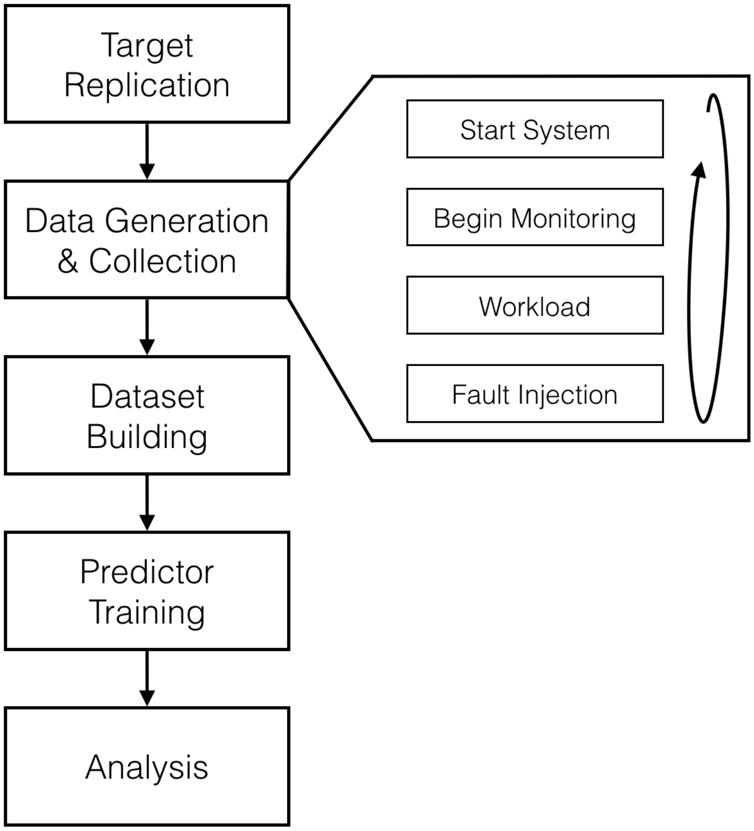

Figure 7. The flow of the major steps involved in the AFP framework training phase [13].

3.1.3.1 Target Replication.

During this phase a virtual clone of the target is made. After the clone is made, the fault injection and monitoring software must be installed. In this experiment, the monitoring tool is the same as the production system but care must be taken to ensure the host-name is changed so the log messages generated during this phase are not confused with messages from the production system.

3.1.3.2 Data Generation & Collection.

The purpose of this phase is to generate the data to train a new prediction algorithm. As a result, this sub-phase must be executed several times to generate statistically meaningful datasets. In this phase, the controller triggers the cloned target startup. Once startup is complete and the system enters an idle state, the monitoring tool begins collecting data from the target. After monitoring has begun, the workload is started. Once the workload has entered a steady state, the fault load is started. Finally, when failure occurs, monitoring stops, the workload stops, and the system is rebooted for the next run. To generate golden data, the first run omits the fault injection step.

The most critical part of this process is labelling the data when failure occurs. For the purposes of this experiment, D-PLG reports when authentication fails and transmits a syslog message to the controller. For this research, failure has been defined by two criteria, the first is when authentication with known good credentials fails. The second is classified by when authentication with known good credentials takes longer than thirty seconds (and it can then be assumed that the service has crashed).

3.1.3.3 Dataset Building.

In this phase, the raw syslog messages are formatted and encoded to train the predictor. The messages generated by the load generator are removed and in each case the neighboring message with the closest timestamp is labelled as when the failure occurred.

Irrera, et al. [13] loaded all event messages into a database for processing. In this work, the events are initially stored in a flat file on the Ubuntu machine by the syslog daemon. During this phase, a tool is run that divides each of the n event messages into n observations with the following three features: failure, timestamp, and message. The failure feature is a binary bit set if the event occurred within a configurable time window δt before the actual failure as defined by Salfner, et al. [19]. The messages are processed by assigning a unique integer for each unique word in the log messages. The integers are then concatenated resulting in a single feature (TODO: Better process?). The features for each observation are then combined in an $n \times 3$ matrix for training.

3.1.3.4 Predictor Training.

The purpose of this phase is to use the data generated by the forced failure of the virtual clone to train a machine learning algorithm to classify a system as failure prone or not.

During this phase, each of the k datasets produced by the k runs of the execution phase, each containing a single failure, are used to train a support vector machine. Each dataset is an $n \times 3$ matrix where n is the number of events recorded in the syslog for each run of the execution phase. These k datasets are used to conduct a k-fold cross validation training and evaluation process where the first $k - 1$ datasets are used to train a support vector machine. The remaining set is used to validate the trained model. The data is then rotated and repeated k times.

At each phase, the number of events that are correctly classified as failure prone and occurring within the specified time window δt are recorded as well as the number of events incorrectly classified as failure prone occurring within the time window. These figures are then used to calculate the measures of performance in the *Analysis* phase.

3.1.3.5 Analysis.

During this phase, the precision, recall, and area under the ROC curve are computed using the figures measured in the previous phase so that the new predictor can be compared against the old. In this experiment, the focus is on the precision and recall of the algorithm. Since failure is such a rare event, accuracy is not a very meaningful measure of performance and thus is not used.

3.2 Implementation of the AFP

3.2.1 AFP Framework Implementation.

This experiment replicates the experiment in [13] except in place of the web-server a Microsoft (MS) Windows Server running Active Directory (AD) Domain Services. In addition, several extensions to the original experiment are made and presented here. Multiple prediction techniques have been applied using this framework to further generalize and validate the framework. The original AFP architecture is shown in Figure 8 with the parts that are modified in this work highlighted.

3.2.2 AFP Modules.

In [13], the authors outline multiple modules into which they have broken the AFP Framework for organizational purposes. This research does not modify these modules, instead, it takes a more granular approach and presents a modified architecture and details each element of that architecture.

The following sections detail 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 described 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

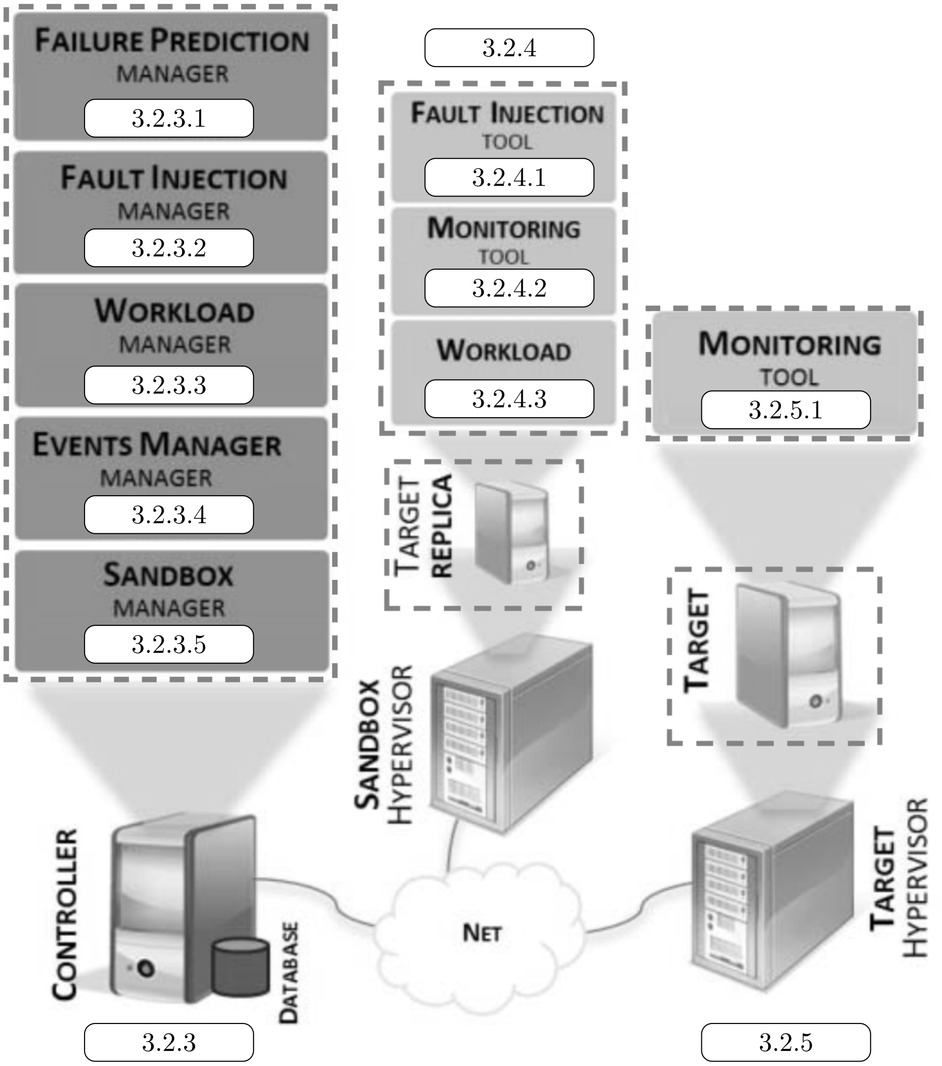

Figure 8. The AFP framework implementation [13] with modified components highlighted.

3.2.3 Controller Hypervisor.

The controller functions in this 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 the 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

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

event management. Each of these functions is detailed in the following sections.

3.2.3.1 Failure Prediction.

The failure prediction module predicts failure using machine learning algorithms 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.

THe AFP failure prediction function as outlined in [13], is performed by a support vector machine (SVM) predictor using *libsvm*. Additionally, the original experiment made use of a database that stored the features and observations used for the failure prediction training algorithm. This experiment does not modify the failure prediction module drastically as it has already been shown in previous work that the online failure prediction area of study is well explored [19]. This research makes use of a different tool-set to execute the training and predicting phases. Due to its widespread use in the statistical community, the prediction and training algorithms make use of the R programming language.

In this experiment, an SVM predictor is trained as done in [13], as well as a... TODO maybe the method in [24]?

3.2.3.2 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. [13] use a tool implementing the G-SWFIT technique for this module. G-SWFIT was developed at the University of Coimbra in Coimbra Portugal by Joao Duraes and Henrique Madeira [7]. The method is widely implemented for use in software fault injection both commercially and academically [17, 12, 5, 23]. Recently, studies have questioned the representativeness of the failures generated by G-SWFIT [14]. In each case, the workload generated was critical in creating representative faults. This concern has been addressed in this research and is discussed in Section 3.2.3.3.

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, the target is not an open source project and as a result, some of the faults and resulting failures may never happen in a production environment. Fortunately, the tool that has been developed for this research 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.

This work introduces an x86-64 implementation of the G-SWFIT technique called W-SWFIT for Windows Software Fault Injection Tool. The source code for W-SWFIT has been published as open source on Github[1] so that others may use it for any of the reasons cited in the original G-SWFIT paper [7]. For this research, the original plan was to use the same tool used in [13] for fault injection. Unfortunately,

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

that tool and all prior G-SWFIT implementations were incapable of injecting faults into x86-64 binary executables. Further, many of the commercial products that were evaluated for this research were incapable of dealing with modern address space layout randomization (ASLR). As a result, W-SWFIT was developed for this research and is capable of injecting faults into all user and kernel mode applications on modern MS Windows operating systems.

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 [7], failures are ultimately the result of software developer errors. Unfortunately, much of the work done in [7] 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. The target application in this research is strictly an x86-64 (also known as x64 or amd64) application and the patterns identified in [7] are incompatible. Consequently, a fault injection tool capable of mutating x86-64 instructions in the same way was required. W-SWFIT implements many of the operators in [7] in the x86-64 language by translating the operators seen in Table 3 from IA32 to x86-64. A simple example of this translation is shown on the entry/exit points of a function in Tables 4, and 5. The rest of the translations can be seen in source code for W-SWFIT. 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 [7].

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) [7].

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) [7].

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	

3.2.3.3 Workload Managment.

The Workload module creates realistic work for the target system in the sandbox hypervisor to do as a way of generating computational load. 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, much emphasis is been placed on this module. Since the target is not a web server, it was not possible to use the same load generator as was used in [13]. Initial searches for a load generator suitable for this research yeilded 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 the target machine[3]. Further searches for tools that would sufficiently load the domain controller did not produce any results. Consequently, a tool to produce realistic load for a domain controller was developed for this research and is introduced 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

[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`

Microsoft domain controller. Other network traffic generators typically work by replaying traffic captured on a live network. Unfortunately, due to the cryptographic 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, realistic authentication requests are sent to the domain controller and are actually processed. The functions performed by the domain controller have been evaluated and D-PLG is designed to sufficiently load each of the services responsible for performing those functions. functions.

In this experiment, the DC is configured as it is in the Air Force Network architecture. After careful analysis, it has been determined that the major roles in the Air Force 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). D-PLG uses the MS Powershell environment to generate the traffic in an effort to make the traffic as real as possible. After building the tool, an experiment was constructed and executed 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 [15] and tested by running the tool on five client machines against the domain controller for five rounds of five minutes. The results of this test can be seen in Figures 9, 10, 11.

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

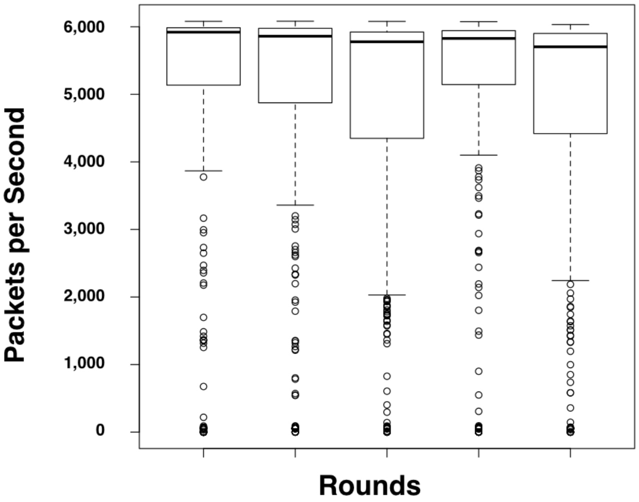

Figure 9. How many packets per second were sent or received by the domain controller across all five rounds of the first test. In each test, we captured approximately 1.8 million packets.

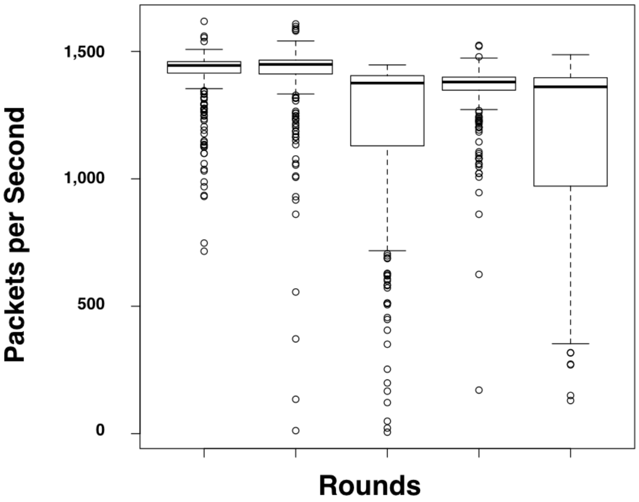

Figure 10. How many packets per second were sent or received by one of the clients across all five rounds of the first test.

against a domain controller. Since D-PLG depends on the use of client machines, it is recommended that any load generation be conducted during off-peak hours if spare client sized machines are not available. It should be noted however, that even

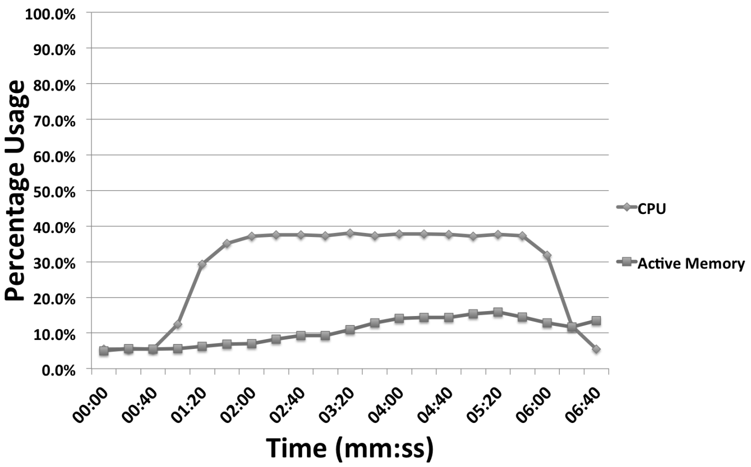

Figure 11. Domain controller CPU and memory utilization during the first test.

with poorly resourced client machines (seen in 1), D-PLG was 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 [15], we have concluded that D-PLG is capable of sufficiently loading the 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.

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

3.2.3.4 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. [13] use the *Logman* tool for event management in their original case study. Since the experimental environment was modelled after the Air Force enterprise environment, the *Solar Winds* log forwarding tool is used 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.

3.2.3.5 Sandbox Management.

The purpose of the sandbox management module is to supervise the virtual cloning of the production system that is made when a new predictor is to be trained. As Irrera, et al. [13, 10] point out, it is typically inappropriate to inject faults and cause failures in production systems, so a virtual clone must be created for that purpose.

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

3.2.4 Sandbox Hypervisor.

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

3.2.4.1 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 3.2.3.2, W-SWFIT has been developed to serve this purpose and implements the G-SWFIT technique developed by Duraes, et al. [7] for fault injection. The execution is controlled by the Windows Server virtual machine on the Controller hypervisor through PowerShell remote execution to reduce the 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 fault injection.

This function is extended by this research to include failure as a result of excessive load and failure as a result of a corrupt database. Section 3.3 covers these extensions in more depth.

3.2.4.2 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. [13] use the *Logman* tool in their original study but because the experimental infrastructure used in this research is modelled after production Air Force networks, the *Solar Winds* log forwarding tool is used because it is already present in the Air Force architecture. The tool is a lightweight application that simply forwards windows events to a syslog server.

3.2.4.3 Sandbox Workload.

The sandbox 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 the 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. [13] used a web traffic generator called TPC-W installed on a single machine in their original study because their target was a web server. Because the domain controller does not respond to web-requests and a tool had not previously been written for this application, a tool was developed for this research called D-PLG. D-PLG is a tool that generates approximately ten full-stack authentication sessions requests per second in order to sufficiently load the domain controller. D-PLG is a distributed tool and requires the use of client machines as a result. This module is represented by those client machines. In this experiment, the client portion of D-PLG is installed on five client machines managed by the central workload manager as discussed in Section 3.2.3.3.

3.2.5 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 the client machines were cloned as well for convenience, they were not used in this experiment.

3.2.5.1 Monitoring.

The monitoring module is exactly the same as the sandbox monitoring module and for this experiment, the *Solar Winds* syslog forwarding tool is used. 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.

3.3 Extenstions to the AFP

This section outlines a few additional extensions to the Adaptive Failure Prediction Framework specifically with respect to the fault load. One additional type of failure that may be used to train a predictor can be generated by under-allocating resources for the domain controller in the sandbox hypervisor. Under some circumstances, this may not be considered a realistic form of failure. However, one reason an organization may want to implement the AFP may be that monetary resources are not available to implement an adequately redundant domain controller and as a result, it may be possible that an adequately sized domain controller is not an option. Consequently, load based failure may be a realistic challenge faced by some organizations and knowing that such a failure may occur might be valuable.

Another type of failure that the extended architecture evaluates goes one layer deeper in analyzing the function performed by the target application. In addition to targeting the main library that performs authentication on a domain controller with fault injection, the extended framework will also target the library responsible for interacting with the user database. In this way, the extended AFP framework is capable of simulating a corrupted database that may occur as a result of a corrupted sector on the disk where the database is stored.

By adding these two additional types of failures to the data used to train a prediction algorithm, the resulting algorithm will be able to predict a wider range of realistic failures.

IV. Experimental Results and Analysis

This chapter reports results after conducting the experiments laid out in Chapter III. First, common measures of performance are reviewed followed by a short discussion of the findings. The chapter concludes with a detailed analysis of the outcomes of this research.

4.1 Performance Measures

This section reviews the performance measures used in this chapter to demonstrate the efficacy and quality of the models trained in this research. These measures are commonly used in the field of machine learning to compare and assess predictors and are taken from a survey of online failure prediction methods written by Salfner et al. [19].

This research utilizes a technique called cross-validation in which a set of labelled training data is broken into three parts as follows:

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 performance measures 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.

4.1.1 Precision and Recall:.

Precision and recall are the most popular performance measures used when for comparing OFP approaches. The two are related and often times improving precision results in reduced recall. Precision is the number of correctly identified failures over the 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:

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

4.1.2 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 performance measures 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:

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

4.1.3 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 performance measures 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:

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

4.1.4 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 shown in Figure 12.

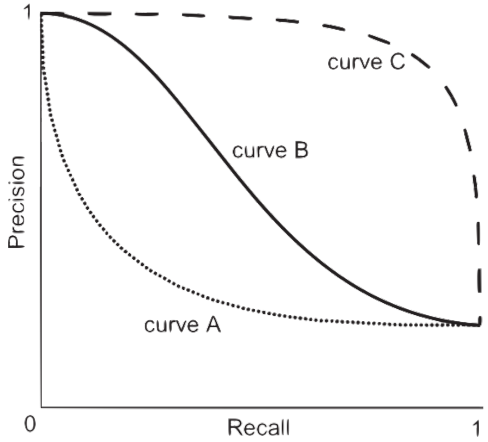

Figure 12. Sample precision/recall curves [19]. Curve *A* represents a poorly performing predictor, curve *B* represents an average predictor, and curve *C* represents an exceptional predictor.

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 13.

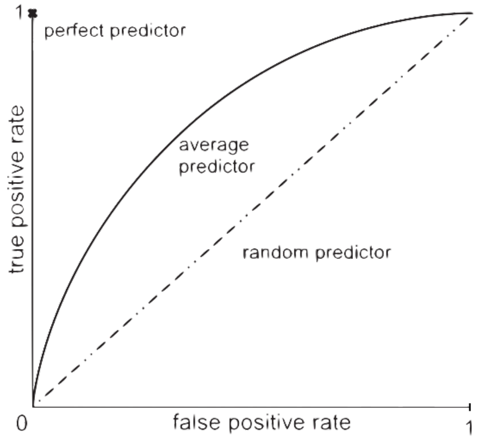

Figure 13. ROC plots of perfect, average, and random predictors [19].

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.

V. Conclusion and Future Work

Notes on future work:

- More automation using VMWare APIs
- Implement more of the operators from G-SWFIT and better automate injection/training phases
- Automate the event checking process... need a way to better determine when underlying system has changed
- Implement more predictors?
- Better define method for determining when to run AFP? Sliding time window? Machine learngin?
- Continuously running AFP? Let it run continuously in the background to capture new failures. The same way our tool reports failure, have a health checking daemon running in the background that will report to syslog when failure has occured so that it gets labelled

Appendix A. W-SWFIT

A.1 FaultInjection.cpp

```
// FaultInjection.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include "globals.h"

#include "Operators.h"
#include "Operator.h"
#include "Library.h"

using namespace std;

bool SendSyslog();

int _tmain(int argc, _TCHAR* argv[]) {

        // Declarations
        DWORD aProcesses[1024], cbNeeded, cProcesses;
        TCHAR szProcessName[MAX_PATH] = TEXT("<unknown>");
        unsigned int i;

        // Get All pids
        if (!EnumProcesses(aProcesses, sizeof(aProcesses), &cbNeeded)){
                cerr << "Failed␣to␣get␣all␣PIDs:␣" << GetLastError() << endl;
                return -1;
        }

        // Get screen width
        CONSOLE_SCREEN_BUFFER_INFO csbi;
        GetConsoleScreenBufferInfo(GetStdHandle(STD_OUTPUT_HANDLE), &csbi);
        int dwidth = csbi.srWindow.Right - csbi.srWindow.Left;

        cout << "Running␣Processes" << endl;
        printf("%-6s␣%-*s\n", "PID", dwidth-7, "Process");
        cout << string(3, ’-’) << "␣" << string(dwidth - 7, ’-’) << endl;
        cProcesses = cbNeeded / sizeof(DWORD);
```

```
for (i = 0; i < cProcesses; i++) {
    if (aProcesses[i] != 0) {
        HANDLE hProc = OpenProcess(PROCESS_QUERY_INFORMATION |
          PROCESS_VM_READ, FALSE, aProcesses[i]);
        if (hProc != NULL) {
            HMODULE hMod;
            DWORD cbNeededMod;
            if (EnumProcessModules(hProc, &hMod, sizeof(hMod), &
              cbNeededMod)) {
                GetModuleBaseName(hProc, hMod, szProcessName,
                   sizeof(szProcessName) / sizeof(TCHAR));
            }

            _tprintf(TEXT("%6u␣%-*s\n"), aProcesses[i], dwidth-7,
               szProcessName);
            CloseHandle(hProc);
        }
    }
}

// Which process?
string s_pid = "";
cout << endl << "Into␣which␣process␣would␣you␣like␣to␣inject␣faults?␣[PID]:␣"
  ;
getline(cin, s_pid);
int pid = stoi(s_pid);

HANDLE hTarget = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);
if (!hTarget) {
    cerr << "Failed␣to␣open␣process␣(check␣your␣privilege):␣" <<
      GetLastError() << endl;
    return -1;
}

// Enumerate modules within process
HMODULE hmods[1024];
cout << "DLLs␣currently␣loaded␣in␣target␣process:" << endl;
printf("%-4s␣%-*s\n", "ID", dwidth-5, "Module␣Name:");
cout << string(4, '-') << "␣" << string(dwidth - 5, '-') << endl;
if (EnumProcessModules(hTarget, hmods, sizeof(hmods), &cbNeeded)) {
        for (i = 0; i < (cbNeeded / sizeof(HMODULE)); i++) {
                TCHAR szModName[MAX_PATH];
                if (GetModuleFileNameEx(hTarget, hmods[i], szModName, sizeof(
                    szModName) / sizeof(TCHAR))) {
                        _tprintf(TEXT("%4d␣%-*s\n"), i, dwidth-5, szModName);
                } else {
                        cerr << "Failed␣to␣Print␣enumerated␣list␣of␣modules:␣
                            " << GetLastError() << endl;
                }
        }
} else {
        cerr << "Failed␣to␣enum␣the␣modules:␣" << GetLastError() << endl;
}

// Which Module?
string s_mod_id = "";
cout << "Into␣which␣module␣would␣you␣like␣to␣inject␣faults?␣[ID]:␣";
getline(cin, s_mod_id);
int mod_id = stoi(s_mod_id);

MODULEINFO lModInfo = { 0 };
cout << "Dll␣Information:" << endl;
if (GetModuleInformation(hTarget, hmods[mod_id], &lModInfo, sizeof(lModInfo))
    ){


        cout << "\t␣Base␣Addr:␣" << lModInfo.lpBaseOfDll << endl;
        cout << "\t␣Entry␣Point:␣" << lModInfo.EntryPoint << endl;
        cout << "\t␣Size␣of␣image:␣" << lModInfo.SizeOfImage << endl << endl;

} else {
        cerr << "Failed␣to␣get␣module␣information:␣" << GetLastError() <<
            endl;
        return -1;
}

// Get module name
TCHAR szModName[MAX_PATH] = TEXT("<unknown>");
GetModuleFileNameEx(hTarget, hmods[mod_id], szModName, sizeof(szModName) /
    sizeof(TCHAR));
```

```
        // Build library object
        Library *library = new Library(hTarget, (DWORD64)lModInfo.lpBaseOfDll,
                                                                          lModInfo.
                                                                              SizeOfImage
                                                                              , string
                                                                              ((char *)
                                                                              &
                                                                              szModName
                                                                              ));

        // Save library for future static analysis
        library->write_library_to_disk("C:\\memdump.dll");

        library->inject();

        // Send syslog message
        SendSyslog();

        return 0;
}

bool SendSyslog() {
        WSADATA wsaData;
        int iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
        if (iResult != NO_ERROR) {
                cerr << "Couldn't␣send␣syslog␣message" << endl;
                return false;
        }

        SOCKET ConnectSocket;
        ConnectSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
        if (ConnectSocket == INVALID_SOCKET) {
                cerr << "Couldn't␣send␣syslog␣message" << endl;
                WSACleanup();
                return false;
        }

        sockaddr_in clientService;
        clientService.sin_family = AF_INET;
        clientService.sin_addr.s_addr = inet_addr("192.168.224.7");
        clientService.sin_port = htons(514);

        iResult = connect(ConnectSocket, (SOCKADDR *)&clientService, sizeof(
            clientService));
        if (iResult == SOCKET_ERROR) {
                cerr << "Couldn't␣send␣syslog␣message" << endl;
                closesocket(ConnectSocket);
                WSACleanup();
                return false;
        }

        char *sendbuf = "FAULT_INJECTED_SUCCESSFULLY";
        iResult = send(ConnectSocket, sendbuf, (int)strlen(sendbuf), 0);
        if (iResult == SOCKET_ERROR) {
                cerr << "Couldn't␣send␣syslog␣message" << endl;
                closesocket(ConnectSocket);
                WSACleanup();
                return false;
        }
        cout << "Successfully␣sent␣syslog␣message" << endl;

        closesocket(ConnectSocket);
        WSACleanup();
        return true;
}
```

Appendix B. ResourceLeak

B.1 resourceleak.cpp

```
/****************************************************************************/
/*                                                                          */
/* resourceleak.cpp                                                         */
/* Project: W-SWFIT: Resource Leak                                          */
/* Authors: Paul Jordan                                                     */
/* Date Created:  8 May 2016                                                */
/*                                                                          */
/* Description: Small app designed to fill up memory, disk, or CPU at a     */
/* configurable rate  in order to force a system to fail.  This application */
/* simulates a poorly written third-party application which might cause     */
/* failure in an underlying system.                                         */
/*                                                                          */
/* Copyright (c) 2016                                                       */
/*                                                                          */
/****************************************************************************/

#include "globals.hpp"
#include "memory.hpp"
#include "cpu.hpp"
//#include "disk.hpp"

using namespace std;

int main(int argc, char *argv[]) {
  // Process Command Line Args
  if ( argc < 3 ) {
    cerr << "Need␣to␣specify␣which␣type␣of␣leak␣[m]emory,␣or␣[c]pu." << endl;
    cerr << "usage:␣"<< argv[0] <<"␣-[m|c]␣<rate>" << endl;
    return 1;
  }

  Resource *leak = NULL;
  if      ( string(argv[1]).compare("-m") == 0 )
    leak = new Memory();
  else if ( string(argv[1]).compare("-c") == 0 )
    leak = new CPU();
  else {
    cerr << "Unrecognized␣leak␣type.␣Specify␣[m]emory,␣or␣[c]pu." << endl;
    return 1;
  }

  int rate;
  string str_rate = string(argv[2]);
  if ( ! (istringstream(str_rate) >> rate) ) rate = 0;

  if (rate <= 0 || rate > 100) {
    cerr << "Unrecognized␣rate.␣Specify␣rate␣between␣1-100." << endl;
    return 1;
  }

  if (leak)
    leak->start(1);

  while(true) { this_thread::sleep_for(chrono::seconds(1)); }
  return 0;
}
```

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. P. Chapman, J. Clinton, R. Kerber, T. Khabaza, T. Reinartz, C. Shearer, and R. Wirth. CRISP-DM 1.0 Step-by-step data mining guide. Technical report, The CRISP-DM consortium, August 2000.

5. 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.

6. 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.

7. 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.

8. 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.

9. 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.

10. 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.

11. 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.

12. 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.

13. 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.

14. 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.

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

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)* 10–09–2016	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From — To)* Sept 2015 — Sep 2016

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** Dr. G. Peterson, AFIT/ENG
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(include area code)*
U	U	U	U	???	(937) 255-????, x????; gilbert.peterson@afit.edu