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.

1st Lt, USAF

July 24, 2016

AFIT/GCS/ENG/17-M

DATA DRIVEN DEVICE FAILURE PREDICTION

THESIS

Paul L. Jordan, B.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

As society becomes more dependent upon computer systems to perform increasingly critical tasks, ensuring those systems do not fail also becomes more important. Many organizations depend heavily on desktop computers for day to day operations. Unfortunately, the software that runs on these computers is still written by humans and as such, is still subject to human error and consequent failure. A natural solution is to use statistical machine learning to predict failure. However, since failure is still a relatively rare event, obtaining labelled training data to train these models is not trivial. This work explores new simulated fault loads with an automated framework to predict failure in the Microsoft enterprise authentication service in an effort to increase up-time in desktop computers and improve mission effectiveness. These new fault loads were successful in creating realistic failure conditions that could be accurately identified by statistical learning models.

Acknowledgments

Nothing worth doing, is possible alone. This work is no exception. Thanks to my advisors, course instructors, and committee members for working with me and guiding me through this exciting endevour. Thanks to my fellow classmates for commiserating with me through the unrelenting flood of coursework. Finally, but most importantly, thanks to my wife for always being there, supporting my often erratic work schedule, and making sure I never forgot to eat.

Paul L. Jordan

Table of Contents

Page

Page

List of Figures

List of Tables

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 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, and in some cases that failure could have catastrophic consequences. Every day, critical infrastructure and Air Force mission 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 [1]. Unfortunately, much of this work has gone unused.

Failure has been defined as the result of a software fault or error [1]. There are 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 [4]. This leaves real-time error prevention or handling. In recent years, it seems the recommended solution to this problem is to make massively redundant systems that can withstand failure [5]. 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 is not done in production environments due to the complex and manual task of training a prediction model. If the underlying system changes, the efficacy of a prediction model can be drastically reduced 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. [2] presented a framework in 2015 to automate the process of dynamically generating failure data and using it to train a predictor after an underlying system change. This framework is called the Adaptive Failure Prediction (AFP) framework and this research explores an implementation of it. More specifically, this research presents results after implementing a modernized AFP framework using a Microsoft (MS) Windows Server Domain Controller (DC) that is capable of generating more diverse and specific failure data for training. Successive software updates are then applied until the model selected becomes useless, the framework is then allowed to re-train a new more effective predictor. Finally, the iplementation is validated by running the same experiment on a web server.

1.1 Problem Statement

According to the operational community, 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 when the system has already entered a degraded

performance mode. To maintain network resilience, the operational organizations responsible for communications support desperately need some means of gaining accuracy and lead-time before a service failure occurs.

To increase that lead time and make more accurate predictions, this research explores predicting failure by analyzing data reported by a target system. Preceding a service failure event, multiple indicators from 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. For example, the simplest approach is to use everyday statistical analysis to 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. 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 [1].

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 [2]. 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,

Irrera, et al. [2] introduced a framework that 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 exploring and developing the fault load. Specifically, this work explores additional and more realistic types of faults, modernizes the fault injection tool by translating it from the IA32 architecture to the x86-64 architecture, and explores the use of reported errors or log messages instead of system health information.

1.2 Hypothesis

An implementation of the AFP framework with a more representative fault load for the MS Windows enterprise infrastructure using data in log messages will lead to accurate failure prediction with better lead time than is available today without any prediction model. This hypothesis is tested by implementing the AFP framework in a scaled virtual environment and evaluating its performance after successive software updates. To validate the approach, the same AFP framework implementation is evaluated against an Apache web server. Prior to this research, the faults produced and consequently predicted by the AFP framework were the result of first-order software faults. This research evaluates the performance of the AFP framework when second and third order faults are introduced. Additionally, the implementation of the AFP framework was not possible on modern MS Windows infrastructure because the fault injection tool used, had not been written for the x86-64 architecture, and was incapable of injecting faults in protected system processes. Finally, the initial case study of the AFP framework used system health information to train the prediction model. As is pointed out by Salfner, et al. [1], this sort of prediction may have difficulty distinguishing between normal operations and actual errors which may evolve into

failure. This work explores the use of reported errors to train the model instead to overcome this shortcoming. It is expected that this modification will allow for more accurate predictions.

1.3 Research Goals

A goal of this research is to develop a machine learning based failure prediction model to predict failures in enterprise network services. This research should demonstrate the efficacy of the AFP framework and proposed extensions when used on the MS 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 increased representativeness of the faults generated should lead to better predictions and increased availability in enterprise services. Finally, the translation of the IA32 General Software Fault Injection Technique (G-SWFIT) tool to the x86-64 architecture should enable the same advantages of software fault injection for 32-bit systems on 64-bit systems [3].

1.4 Impact of Research

Every day, many of the Air Force's critical missions depend on 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 MS 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. Finally, this method is capable of predicting system failure based on underlying software faults and not indicators about malicious attacks against the target system.

1.6 Results

This research shows that when used in conjunction with the modernized AFP framework the additional fault loads enable the generation of failure data that can be used to train predictors to alert on pending failure with better precision and recall than is currently available today. Furthermore, without this modernization, the use of the AFP framework would not have been possible on modern MS Windows operating systems. The results of this work demonstrate the efficacy of the approach by showing that these new fault loads and modernized framework work on both the MS DC as well as a MS web server.

This research also shows that after an underlying system change, this method of predicting failure is 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.

1.7 Summary

This work outlines a technique that is effective in predicting failure in modern MS Windows systems that can adapt to underlying system changes. The remainder of this document will outline exactly how this modernization and the additional fault loads were implemented and tested. The impact of this work is that it could readily be adapted and implemented in many enterprise system architectures with little manpower burden. Specifically, in the Air Force, it 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. Finally, this technique is 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 (OFP)

Traditionally, failure is predicted using statistical information about past failures offline before a system is implemented. Unfortunately, given the complexity of modern computer systems and the infinite number of ways in which they can be configured, this sort of offline analysis is not helpful. OFP is the act of evaluating a running system in real time to make a prediction about what the future state will be.

This chapter reviews current research regarding OFP and its many approaches to build a foundation for this research. Further, the taxonomy of approaches developed by Salfner, et al. [1], is updated by classifying approaches since its publication and creating a new sub-category. The rest of this chapter is organized as follows. In Section 2.1, a brief background on the topic of 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. This chapter then concludes with a brief summary.

2.1 Background

In 2010, Salfner, et al. [1] 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. The remainder of this section reviews those definitions to build a foundation for the rest of this work.

2.1.1 Definitions.

2.1.1.1 Proactive Fault Management (PFM).

Salfner, et al. [1] define PFM as the process by which faults are handled in a proactive way, analogous with *fault tolerance* and basically consisting of four steps: OFP, 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.

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 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, analysis must be conducted to identify possible remediation actions. 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 [6]. 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*.

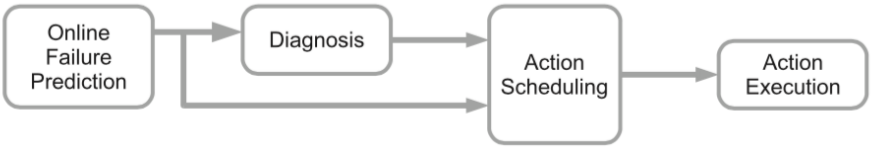


Figure 1. The stages of proactive fault management [1].

2.1.1.2 Faults, Errors, Symptoms, and Failures.

This research uses the definitions from [7] as interpreted and extended in [1] 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 [1]. 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 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. [1] 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 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 $\Delta t_p \rightarrow \infty$, 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 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

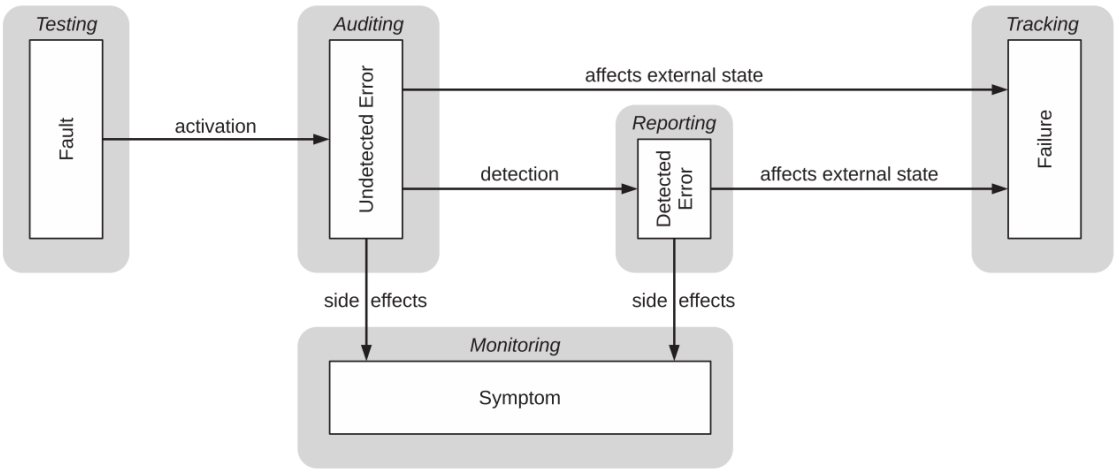


Figure 2. How faults and errors evolve into failure with the associated methods for detection represented by enclosing gray boxes [1].

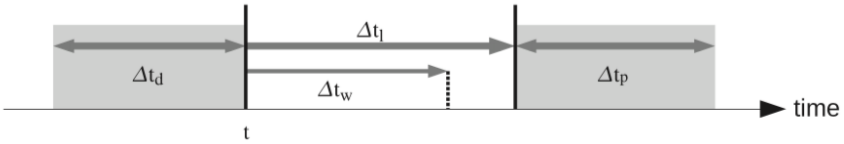


Figure 3. The timeline for OFP [1].

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 OFP

2.2.1 OFP Taxonomy.

The taxonomy by Salfner, et al. [1] 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 as illustrated in Figure 2. The taxonomy is shown in Figure 4.

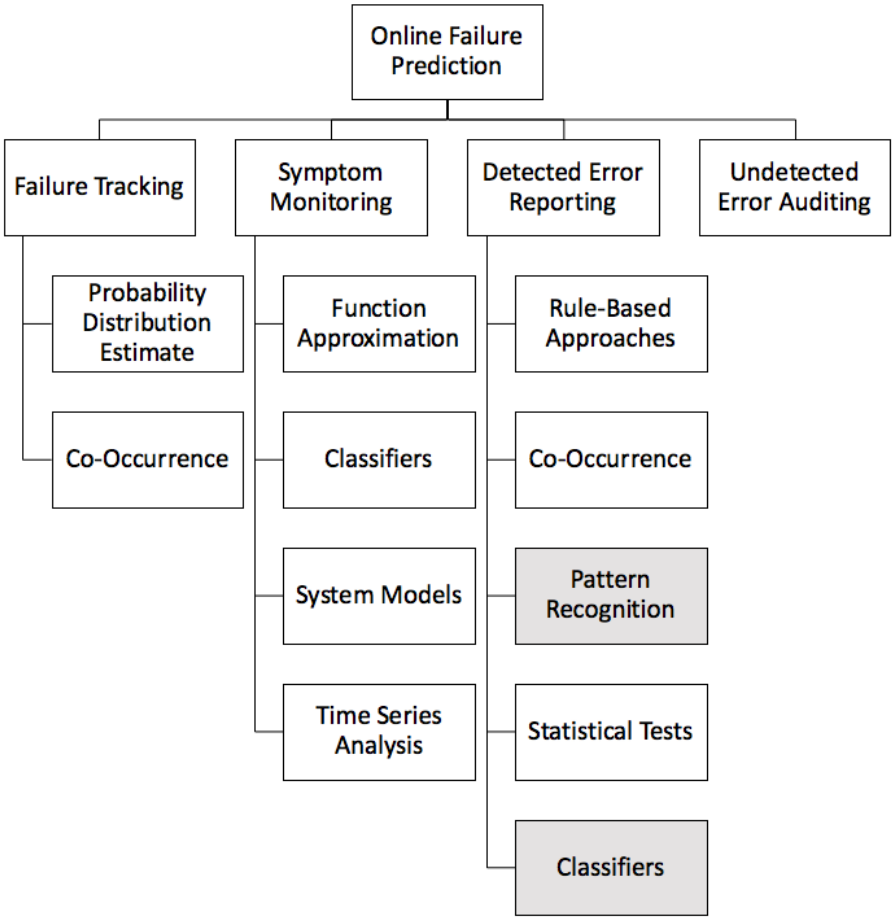


Figure 4. Taxonomy of approaches to online failure prediction [1]. The two categories into which this research falls are highlighted.

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. This is different from prediction methods that rely on observing the current state of a machine such as auditing and monitoring. As pointed out by Salfner, et al. [1], in general these methods have difficulty distinguishing a system under peak load and one that may be about to fail.

The reporting category of the taxonomy is further organized 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. *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.

This research focusses on pattern recognition OFP approaches, which are shown in Figure 5. Strategies employed in the other sub-categories are closely related and thus are also explored in this research.

2.2.2 Data-Driven OFP.

2.2.2.1 Pattern Recognition.

Salfner, et al. [8] 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

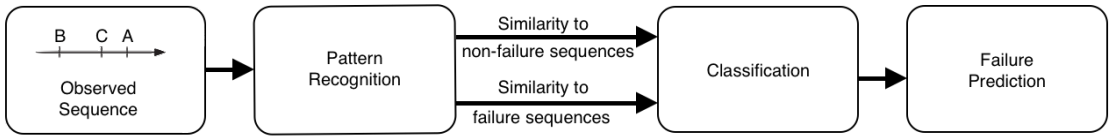


Figure 5. How pattern recognition is accomplished in reported errors [1].

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 [8] using semi-Markov models [9]. 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. [10] published a technique that used Support Vector Machine (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 [1], 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 [10] 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 [2, 10–13].

2.2.2.3 Hybrid Approaches.

Fujitsu Labs has published several papers on an approach for predicting failure in a cloud-computing environment [14–16]. Watanabe, et al. [15, 16] 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 Internet Protocol (IP) address while retaining meaning of the log message.

Watanabe et al.'s [16] 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. [11] 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 OFP.

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 [2, 16] 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 AFP framework by Irrera, et al. [2] shown in Figure 6, 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.

The framework built upon past work by Irrera, et al. [17, 18] to generate failure data by injecting software faults using a tool based on General Software Fault Injection Technique (G-SWFIT) [3] in a virtual environment for comparing and automatically re-training predictors. With the introduction of the framework, Irrera, et al. [2] report results of a case-study. After implementing the AFP framework using a web server and an SVM predictor, they report that their findings demonstrate their framework is able to adapt to changes to an underlying system which would normally render a predictor unusable.

In general, the use of simulated data is not well received by the community, however the authors in [18,19] 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

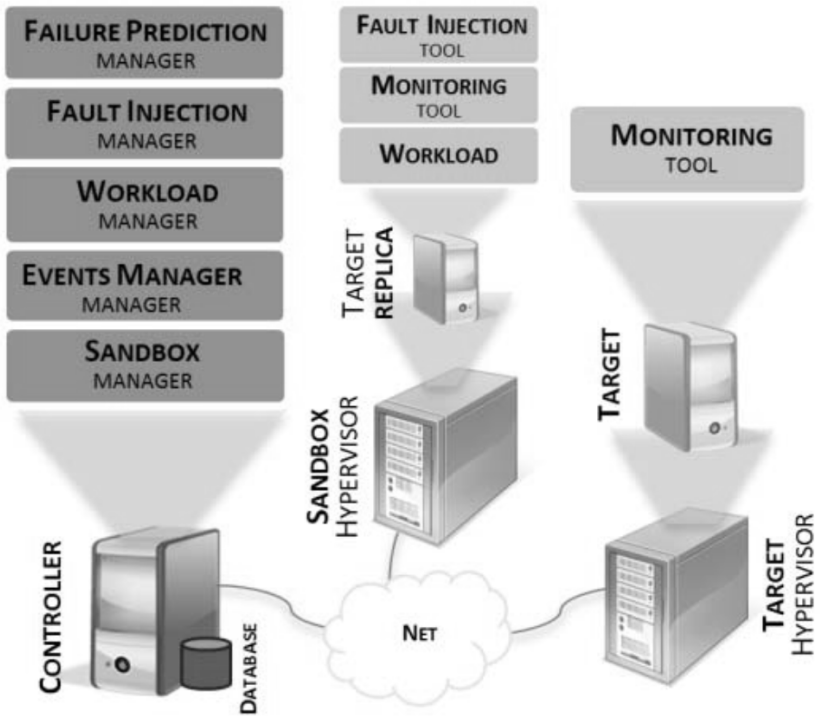


Figure 6. How the AFP framework is implemented [2].

injecting the types of faults that one can expect, each failure type is represented in the training data.

Irrera, et al. [2] reported good results and concluded that the AFP framework is an effective tool. Unfortunately, the AFP framework is not a universal solution and requires significant work to be implemented on a modern Microsoft (MS) Windows enterprise network. Furthermore, the fault load previously explored does not completely represent all possible failures [20].

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 AFP framework and further generalize the approach.

III. Methodology

The purpose of the Adaptive Failure Prediction (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 it can be more easily adapted for different applications. This chapter presents three topics. The first describes the process that the framework executes in order to generate the labelled training data and train a failure prediction algorithm. The second describes each module of the extended AFP framework. The final section details extensions to the AFP framework explored by this research.

This chapter outlines the implementation and extensions to the AFP framework [2] as well as an experiment that was conducted to validate those extensions and further generalize the framework. The AFP framework 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 framework are limited in utility and require generalization to be useful to the general community.

3.1 Failure Data Generation

This work extends the AFP framework [2] by presenting results after conducting another case study with an Microsoft (MS) Windows Server acting as an Active Directory (AD) service with a more representative fault load as well as a new implementation of the General Software Fault Injection Technique (G-SWFIT) technique for the x86-64 architecture.

The case study was done using three new types of faults: third-party memory leak, third-party Central Processing Unit (CPU) hog, and process memory corruption. For completeness, the standard G-SWFIT technique was also used. Another important

modification was made in the actual data collected. The original case study used status and machine state information polled every second. Salner et al. [1] points out that this technique does not properly distinguish between underlying errors and normal workload. In this case study, reported errors are used instead.

Finally, findings are reported after implementing this framework using two different statistical machine learning techniques on reported errors (log messages): boosted decision trees and the weighted Support Vector Machine (SVM). The weighted SVM was used because of it performs well on imbalanced data and it is popular in the Online Failure Prediction (OFP) community [1]. The boosted decision tree was used because it is non-parametric, it is capable of classification, and it is particularly suited for imbalanced data. In both cases, feature reduction was performed as is done by Fulp et al. [12], on a sliding time window as is done by Irrera, et al. [21] and Vaarandi [22].

This section outlines the step-by-step procedure by which the extended AFP framework was evaluated to show how effective it is when used on Windows Server deployments. This is done by dividing the steps taken in the experiment into the three major phases as defined in [2]: preparation phase, execution phase, and training phase.

3.1.1 Preparation Phase.

In this phase the AFP framework is prepared to run for the first time as described in [2]. The Cross Industry Standard Process for Data Mining (CRISP-DM) [23] should be applied to this situation when evaluating how to best apply the AFP for a particular target. For the purposes of this research, the focus was on the MS Windows Directory Services and predicting failure in those services. To demonstrate the efficacy of the AFP, a predictor was evaluated before and after a significant software update. As a result, the most critical preparation made in evaluating this

framework was to hold back all software updates on the target system prior to the first run of the execution phase. The performance of various prediction techniques was evaluated both before and after the Windows Update application was allowed to run. A complete list of the updates installed is shown in Appendix C.

This phase is essentially comprised of the manual act of implementing 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 phase is shown in Figure 7. This phase is divided into three major steps: data collection and failure prediction, event checking, and training/update as described in this section.

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 makes failure predictions about the current state based on the last run of the training phase. This function was not implemented in this research as it is application specific. The output of this process in this experiment was a warning message indicating a predicted failure.

3.1.2.2 Event Checking.

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

training phase was manually triggered upon completion of these updates.

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. In this experiment, this phase was accomplished manually.

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 is shown in Figure 8. Each phase is outlined in the following sub-sections.

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 is installed. In this experiment, the monitoring tool was the same as was already installed on the production system so the extra step of installing the monitoring software was unnecessary.

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 (or data with no failures present to aid training), 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, failure was defined by the log message ID 4625: An account failed to log on[1]. When this occurred in conjunction with known valid credentials on an enabled account, the preceding data window defined for the experiment was labelled as failure prone. Additionally, the workload generator used in this research reported when authentication failed and transmitted a syslog message to the controller.

3.1.3.3 Dataset Building.

In this phase, the raw syslog messages are formatted and encoded to train the prediction model. The purpose of this phase is to prepare the raw messages to be used as numeric inputs for the training phase. Irrera, et al. [2] loaded all data into a database for processing. In this work, the events were stored in a flat file on the Ubuntu machine by the syslog server daemon. An example of one of these messages is shown below:

```
May  8 14:31:52 dc.afnet.com MSWinEventLog 5 Security 3 Sun May 08 14:31:50 2016 4672 Microsoft-
     Windows-Security-Auditing N/A Audit Success dc.afnet.com 12548 Special privileges assigned to new
      logon. Subject: Security ID: S-1-5-21-2379403389-181978965-2953995107-500 Account Name:
     Administrator Account Domain: AFNET Logon ID: 0x9beb4e7a Privileges: SeSecurityPrivilege
     SeBackupPrivilege SeRestorePrivilege SeTakeOwnershipPrivilege SeDebugPrivilege
     SeSystemEnvironmentPrivilege SeLoadDriverPrivilege SeImpersonatePrivilege
     SeEnableDelegationPrivilege
```

The messages were formatted using the *Snare*[2] MSWinEventLog format which is generally divided into several categories. The first is the time-stamp and host name of

[1] https://support.microsoft.com/en-us/kb/977519

[2] http://wiki.rsyslog.com/index.php/Snare_and_rsyslog

the sender prepended by the syslog server daemon: *May 8 14:31:52 dc.afnet.com*. The remainder of the message contains tab delimited values where the keys (and consequent features) are shown in Table 1. Of these features, Criticality, EventLogSource, EventID, SourceName, and CategoryString were selected for further encoding.

The raw messages were then encoded. First, the events were filtered by EventID as is done by Fulp et al. [12] to reduce the noise generated by successful login attempts. Log messages with IDs shown in Table 2 were filtered from the input.

Next, to encode the time dimension and reduce the sequential message ordering dependency, a sliding time window was created by counting each unique entry for each feature within the data window (Δt_d) as is done by Vaarandi [22]. During this stage, the number of messages that were reported in the data window were also recorded and used as a feature.

Finally, each time window preceding the failure within Δt was labelled as failure prone as is done by Irrera, et al. [2]. This encoding enables the use of classification algorithms in the training phase. An example of the final encoding is shown in Table 3.

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.

In this experiment, the execution phase was run k times. During this phase, each of the k datasets produced by the k runs of the execution phase (each containing a single failure), were used to train a statistical classification model. Each dataset was an $n \times p$ matrix where n was the number of sliding time windows and p was the number of predictors present in the output of the dataset building phase. These k datasets were used to conduct a $k-1$-fold cross validation training and evaluation

Table 1. Typical authentication message sent as keys that correspond to the values as designated in the *Snare* protocol for MSWinEventLog used by the SolarWinds syslog agent.

Key	Value
HostName	dc.afnet.com
Criticality	5
EventLogSource	Security
Counter	3
SubmitTime	Sun May 08 14:31:50 2016
EventID	4672
SourceName	Microsoft-Windows-Security-Auditing
UserName	N/A
SIDType	Audit Success
EventLogType	dc.afnet.com
ComputerName	12548
CategoryString	Special privileges assigned to...
ExtendedDataString	Security ID: S-1-5-21-2379403...

Table 2. Microsoft log message IDs[3].

ID	Message
4624	An account was successfully logged on.
4634	An account was logged off.
4672	Special privileges assigned to new logon.
4769	A Kerberos service ticket was requested.
4770	A Kerberos service ticket was renewed.
4776	The computer attempted to validate the credentials for an account.

process where the first $k-2$ datasets were used to train the statistical model. The remaining set was used to validate the trained model. The data was then rotated and the process was repeated $k-1$ times. Parameters for the classification model were selected based on the output of this cross validation. Finally, statistics and performance was reported on the final model's performance on the held out data set.

[3] `https://support.microsoft.com/en-us/kb/977519`

Table 3. Sample message data window after translation.

Predictor	Value
FailureWindow	0
NumObservations	2
Criticality: 6	2
Criticality: 2	0
Criticality: 4	0
EventLogSource: Application	1
EventLogSource: System	1

3.1.3.5 Analysis.

During this phase, the precision, recall, f-measure, and area under the Receiver Operating Characteristic (ROC) curve are computed using the figures measured in the previous phase so that the new predictor can be compared against the old. If a new predictor outperforms the old, the old is replaced with the new. Upon completion of this phase, control flow returns to the *Event Checking* phase. In this phase, this analysis was done manually.

3.2 Implementation of the AFP

3.2.1 AFP Framework Implementation.

This experiment replicated the experiment in [2] with the following modifications. Most importantly, since the focus of this research is on reported errors, log messages were used to train the predictor as is done in many other recent approaches [9, 10, 12, 16]. Instead of only using fault injection to induce failure, three additional fault loads were explored. In addition to using the SVM model, boosted decision trees were evaluated. Finally, in addition to the Apache web-server, the primary target was the MS Windows Server running AD Domain Services. The purpose of Apache web server was to validate the approach and additional fault loads. The original AFP architecture is shown in Figure 9 with the parts that were modified in this work

highlighted.

3.2.2 AFP Modules.

Irrera, et al. [2] 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 Gigahertz (GHz) AMD Opteron 4180 (6 cores each) CPUs and 64 Gigabyte (GB) memory. The specifications of the individual Virtual Machine (VM)s are shown in Tables 4, and 5.

3.2.3 Controller Hypervisor.

The controller responsibilities in this experiment were split between two systems on a single hypervisor shown in Table 5. One system was the MS Windows Server responsible for workload management and fault injection management. The other system was an Ubuntu 14.04 server that performed the failure prediction management and 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 possibly other features produced by the production system.

Table 4. Hypervisor 1 configuration (sandbox/target).

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

Table 5. 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

In the original case study, this module was implemented using an SVM prediction model using the *libsvm* software library. In this experiment, the statistical models were trained on input built as described in Section 3.1.3.3 using the popular statistical learning software suite *R*.

3.2.3.2 Fault Injection.

This module is responsible for managing the fault load used to create realistic failure data. Irrera, et al. [2] use a single tool implementing the G-SWFIT for this module and pointed out that this module is the most critical piece of the AFP implementation. G-SWFIT was developed by Duraes, et al. [3] to emulate software failures for the purposes of software testing. The method is widely implemented for use in software fault injection both commercially and academically [18, 24–26].

Recently, studies have questioned the representativeness of the failures generated by G-SWFIT [20, 24]. 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 regarding fault injection has been that some injected faults may not elude modern software testing and as a result never actually occur in production software [25]. 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. Unfortunately, the target of this research 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 fault injection tool that has been developed for this research 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.

Because of the concerns with fault injection, the experiment conducted in this research tested three additional types of fault load to more exhaustively represent realistic faults that may be encountered by a target process. This experiment trained a predictor using failures generated by third-party applications purposefully written to slowly consume all available resources on the target systems. Specifically, the third-party application contains a memory leak that slowly allocates all free system memory until the target application crashes. Next, failures were recorded as the result of a third-party application consuming all CPU time. Source code for this application is included in Appendix B. Finally, failure was recorded after corrupting heap space in memory (versus program memory as done by the G-SWFIT). This type of fault could be caused by privileged third party applications such as hardware drivers inadvertently writing to the target processes allocated memory. Finally, for completeness, this experiment uses a tool developed for this work that implements the G-SWFIT technique.

This work introduces an x86-64 implementation of G-SWFIT called Windows Software Fault Injection Tool (W-SWFIT). The source code for W-SWFIT has been published as open source on Github[4] so that others may use it for any of the reasons cited in the original G-SWFIT paper [3]. For completeness, the source is also included

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

in Appendix A.

For this research, the original plan was to use the same fault injection tool used in the original case study by Irrera, et al. [2]. Unfortunately, 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 translations that have been performed. Further, as pointed out by Irrera, et al. [21], prior implementations of the G-SWFIT were not capable of injecting faults into protected (kernel mode) processes. Since the focus of this research is on a protected system process, this capability was critical, and as a result, W-SWFIT was implemented in a way that made protected process injection possible.

G-SWFIT works by scanning binary libraries already in memory for known patterns (or operators). These operators are then mutated to match compiled errors that could have been made during development. The errors targeted by G-SWFIT were discovered by analyzing open source project bug reports and code repositories. The errors were then classified based on the Orthogonal Defect Classification (ODC) [27] and are shown in Table 6. The point of this mutation is that failure is ultimately the result of developer error [1, 2], and that fault injection accurately simulates those errors [3]. Unfortunately, G-SWFIT has only previously been implemented for Java applications [28, 29], and the IA32 instruction set [3, 25]. Furthermore, the target applications in this research are strictly x86-64 (also known as x64 or amd64) applications, and the patterns identified previously are incompatible. Consequently, to

implement the AFP framework completely, a fault injection tool capable of mutating x86-64 instructions in the same way was required. W-SWFIT implements two of the operators from the original G-SWFIT shown in Table 6: OMFC and OMLPA. The translation of these operators was not trivial given the complexity of the x86-64 architecture. However, a simple example of this translation is shown using the entry/exit points of a function in Tables 7, and 8. The rest of the translations were done using the *Capstone* [5] library and can be seen in source code for W-SWFIT.

In summary, for the purposes of this research, fault injection was performed four ways: software fault injection with W-SWFIT, under-resourced memory, under-resourced CPU, and heap space corruption. Apart from W-SWFIT, these new fault loads are covered in more detail in Section 3.3.

3.2.3.3 Workload Management.

The workload management module controls the generation of computational load by directing the sandbox workload module to create realistic work for the virtually cloned target to accomplish. Without this module, it could take too long for an injected fault to evolve into a failure. Consider a missing *free* statement and the consequent memory leak. A production target server may have a large amount of available memory and the leak could be relatively 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 the target and the load generation management was collocated with the actual load generator - a simple web request generator [2]. In this experiment, the management and actual load generator roles have been divided and a new tool has been developed: Distributed PowerShell Load Generator (D-PLG). The rest of this section outlines

[5] `http://www.capstone-engine.org`

Table 6. Fault operators used for fault injection [3].

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
MLPA	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 7. Function entry/exit patterns in IA32 bytecode [3].

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

D-PLG and how it fulfills the workload and workload management functions of the AFP framework.

Realistic workload is critical in generating realistic failure and consequently training a useful predictor. Initial searches for a load generator suitable for this research yielded a tool developed by MS that initiated Remote Desktop Protocol (RDP) connections to aid in sizing a terminal services server[6]. By executing an RDP session, the authentication and Domain Name System (DNS) functions of the Domain Controller (DC) would also be loaded. Unfortunately, this tool is no longer maintained and would not execute on the target machine[7]. Further searches for tools that would sufficiently load the DC did not produce any results which led to the development of D-PLG.

D-PLG is a collection of remotely executed MS PowerShell scripts managed by

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

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

Table 8. Function entry/exit patterns in x86-64 bytecode [3].

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

a central script designed to generate realistic traffic that will sufficiently load MS enterprise services including a web server and DC. Other network traffic generators typically work by replaying traffic captured on a live network [30]. This would likely work against an unsecured web server, but unfortunately, due to the cryptographic nature of authentication on a DC, simply replaying traffic will not load such 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 DC. D-PLG solves this problem by making native authentication requests by use of built-in PowerShell cmdlets (pronounced command-lets). By doing this, realistic authentication requests are sent to a DC and are actually processed. Finally, the DNS role can be stressed by sending the authentication requests using domain names without allowing local caching.

By use of native cmdlets, D-PLG is capable of generating four kinds of traffic designed to stress the following services: authentication, web, mail, file sharing, and MS 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 MS community for sizing a DC [31] and tested by running the tool on five client machines against the DC for five rounds of five minutes. The results of this test are shown in Figures 10, 11, 12.

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 does produce realistic load against a DC. 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 with poorly resourced client machines (shown in Table 4), 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 is negligible and they can be in use during load generation.

Based on these results, and that a production DC should be at approximately 40% CPU utilization during peak usage [31], D-PLG is capable of sufficiently loading the DC over a sustained period of time for the purposes of implementing the AFP framework and was used in this research. Further, D-PLG is capable of scaling to provide load against higher capacity DCs by using only a few client machines. D-PLG is available on Github[8] for others to use.

In this experiment, D-PLG was used as the central workload manager. Furthermore, the client portion of D-PLG was used installed on five client machines and used as the sandbox workload generator as discussed in Section 3.2.4.3.

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. [2] use the MS *Logman* tool from the remote controller for event management in their original case study. *Logman* was configured to poll 170 system variables on the target machine once per second.

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

Since the focus of this research is on *reported errors*, and the experimental environment in this work was modelled after modern enterprise environments where this sort of polling could produce too much data, this experiment implemented an *rsyslog* server daemon and the target was configured to forward logs to it. Moreover, because syslog is a standard protocol, it is already in use in many enterprise networks today. The messages forwarded to the events manager were then processed and added to a Structured Query Language (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. [2, 17] 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. Furthermore, the virtualization of the target process has little affect on generated data [17].

For this experiment, the sandbox was managed manually using 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 Application Programming Interface (API)s, in future work, this function could be automated.

3.2.4 Sandbox Hypervisor.

The sandbox hypervisor hosts the virtual clone of the production environment where faults are injected and from which failure data is collected. Cloning the production environment ensures that the production system is not be affected and service are maintained during the training phase. For the purposes of this experiment, the

sandbox was constructed on a single hypervisor implemented as shown in Table 4. 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. [3] for fault injection. The execution is controlled by the Windows Server VM on the 'Controller' hypervisor through PowerShell remote execution to reduce the interaction and potential to introduce bias into the training data. The tool allowed 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'[9], it was the focus of fault injection.

This function was extended by this research to include failure as a result of third-party memory and CPU leaks, and memory corruption. 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 at the target host level so that it may be used to train a statistical prediction model. Since Irrera, et al. [2] use the *Logman* remotely, no additional software was needed on the host. In this experiment, syslog was used and while it is a recognized standard, syslog messages are not produced natively in Windows. Fortunately, several forwarding agents are available to translate and forward native Windows log messages

[9] `https://technet.microsoft.com/en-us/library/cc780455(v=ws.10).aspx`

to a syslog server. For this experiment, the *Solar Winds* syslog forwarding tool was used because of its popularity in the security community and existing presence on many enterprise networks. The tool is a lightweight application that simply forwards Windows events to a syslog server.

3.2.4.3 Sandbox Workload.

The purpose of this module 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 [2,20,24].

Irrera, et al. [2] used a web traffic generator called TPC-W installed on a single machine in their original study because their target was a web server. This would be the ideal tool for the validation test on the Apache web server in this experiment but unfortunately, this tool has been deprecated and no substitute has been written [10]. As a result, D-PLG was used as the work load generator for both the DC and web requests.

D-PLG is a distributed tool and requires the use of client machines. This module is represented by those client machines. In this experiment, the client portion of D-PLG was 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 shown in Table 4. The following section outlines the monitoring tool installed on both the DC and web server on this hypervisor.

[10] `http://www.tpc.org/tpcw/`

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 was used. The only modification worth noting here is that to ensure that the messages sent were uniquely identified by the controller, the hostname of the target machine must be different from the hostname of the sandbox target machine.

3.3 Extensions to the AFP

This section outlines the extensions to the AFP framework explored by this research. Given that fault injection isn't always considered representative [20], the next three sub-sections outline three addition fault loads explored. Next, an outline of the changes in how data was collected from the target is presented. Finally, this chapter concludes with a brief summary of these extensions.

3.3.1 Under-Resourced CPU.

A CPU may become under-resourced in a few ways. The organization implementing the target service may not accurately anticipate the amount of load the service may experience. Alternatively, a third-party application installed on the same physical machine may inadvertently consume all CPU time. The result in both of these situations is the target process gets starved of CPU time.

This condition was simulated in two ways to accurately capture both scenarios outlined above. First, by downsizing the number of virtual CPUs available to the target VM. Second, by introducing a third-party application that ran at 100% CPU. The source code for this application is shown in Appendix B.

3.3.2 Under-Resourced Memory.

Available memory can be limited in a few ways. As with the under-resourced CPU, the implementing organization may under estimate the amount of memory that will be needed by a server to handle the required demand. Additionally, a third-party application could contain a memory leak. In both cases, the target application may not have enough memory to accomplish the work it has been assigned.

To test this fault load, this experiment created both conditions outlined above. First, as was done for the CPU, the amount of memory available to the target VM was reduced. Second, a third-party application with an intentional memory leak was run on the target system. The source code for this application is also shown in Appendix B.

3.3.3 Heap Space Corruption.

Finally, heap-space corruption can happen in a production environment in a few ways. First, in the Windows operating system, device drivers share critical kernel mode libraries and have elevated permissions [32]. If a hardware device driver developer inadvertently writes to an area of memory not allocated for his software, say by forgetting to dereference a pointer, Windows may not warn him. Consequently, he may corrupt the memory of another process.

In this experiment, the focus of this fault load was on the user database. First, users that had been cached by the DC process were corrupted. Next, to simulate a disk failure, the same user was corrupted on disk. To do this, the W-SWFIT code was modified to be able to search and write anywhere in a processes memory. This code is shown in Appendix A.

3.3.4 Reported Errors.

Finally, this research focusses on reported errors instead of system information using the *Logman* tool in the original study [2]. As pointed out by Salfner, et al. [1], a predictor only given system information is not typically able to determine the difference between a system that is going to fail and one that is perhaps under higher than average load. It may be able to pick up on *undetected errors*, but there is little to distinguish those from every day use. Consider the DC and a memory leak situation. According to Russinovich, et al. [32], the MS DC will use as much memory as is available to cache user credentials. This consumption of all available memory may appear very similar to a memory leak if system information is all that is being recorded.

3.3.5 Summary.

In summary, by adding these additional faults and considering reported errors when generating failure data used to train a prediction algorithm, the resulting algorithm will be able to predict a wider range of realistic failures.

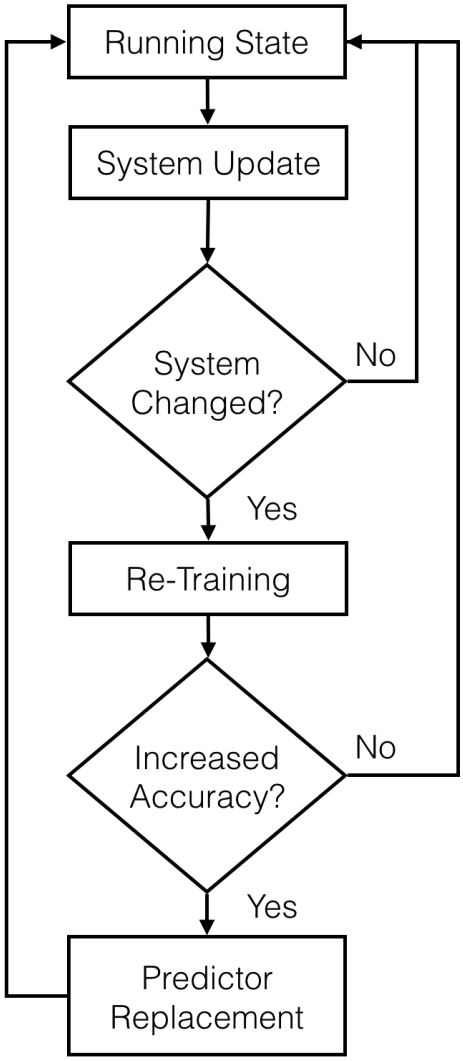


Figure 7. The flow of the major steps involved in the AFP framework execution phase [2].

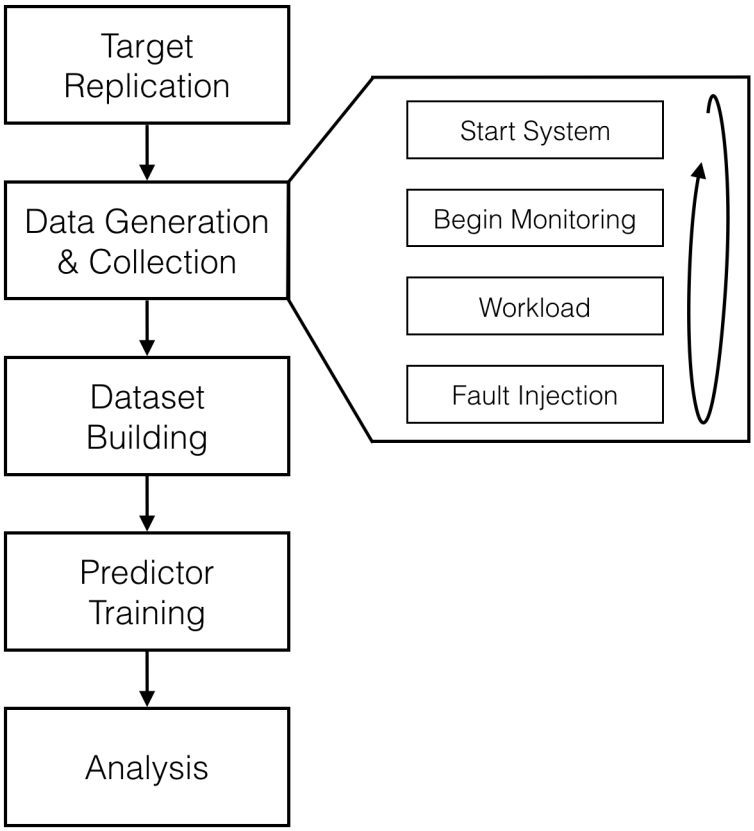


Figure 8. The flow of the major steps involved in the AFP framework training phase [2].

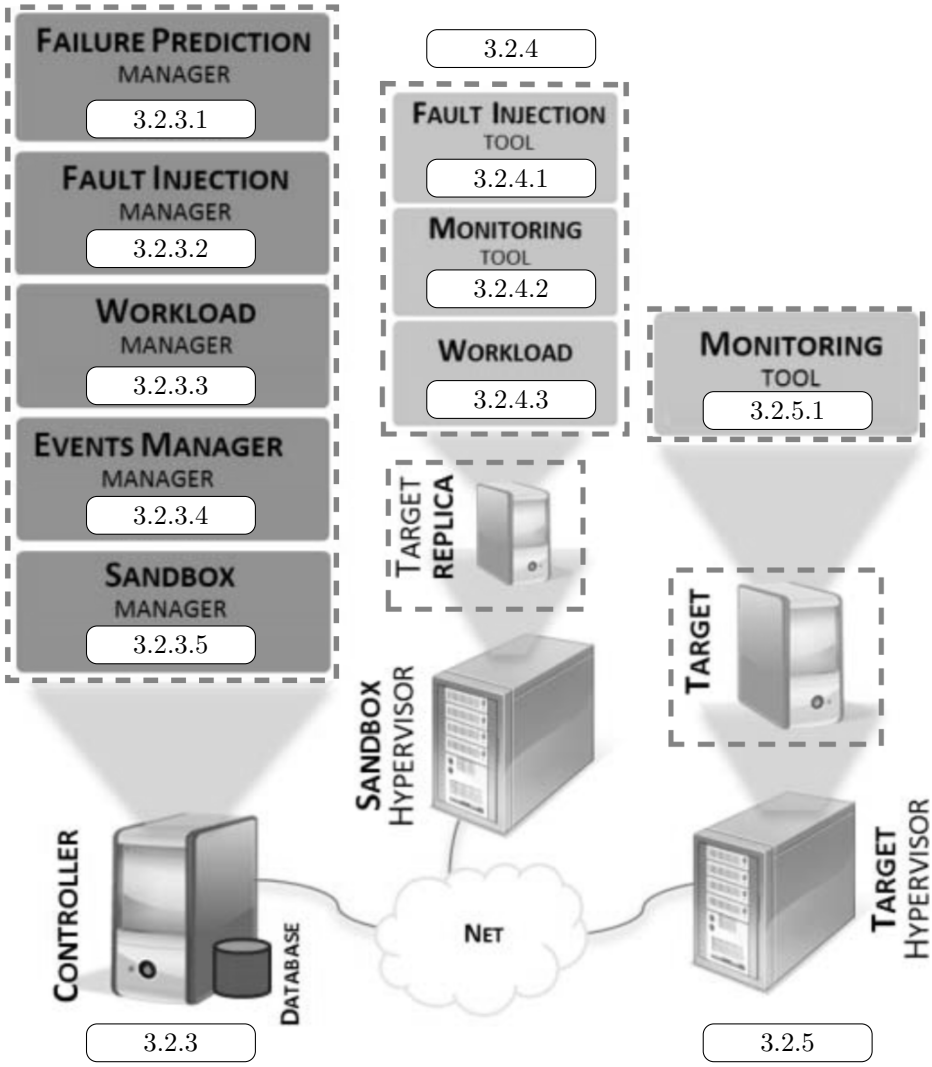


Figure 9. The AFP framework implementation [2] with modified components highlighted.

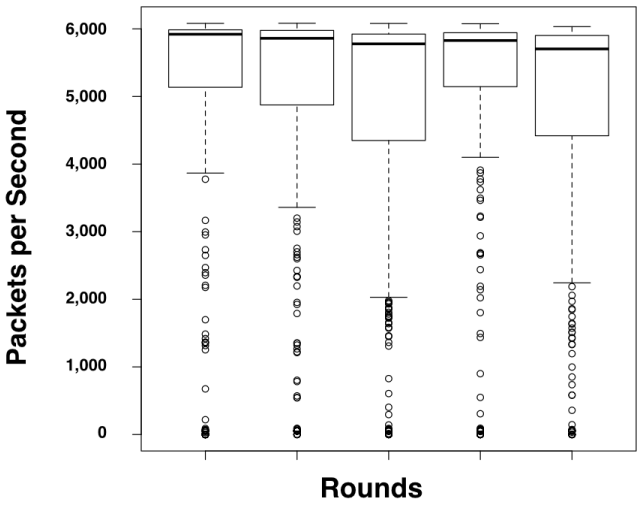


Figure 10. How many packets per second were sent or received by the domain controller across all five rounds of the first test. In each test, we captured approximately 1.8 million packets.

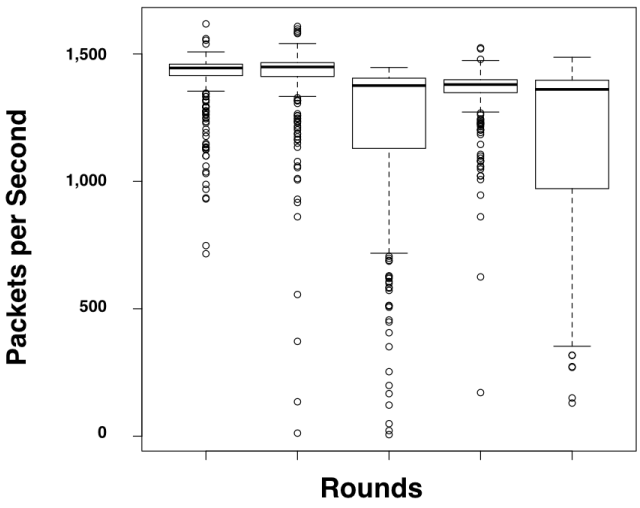


Figure 11. How many packets per second were sent or received by one of the clients across all five rounds of the first test.

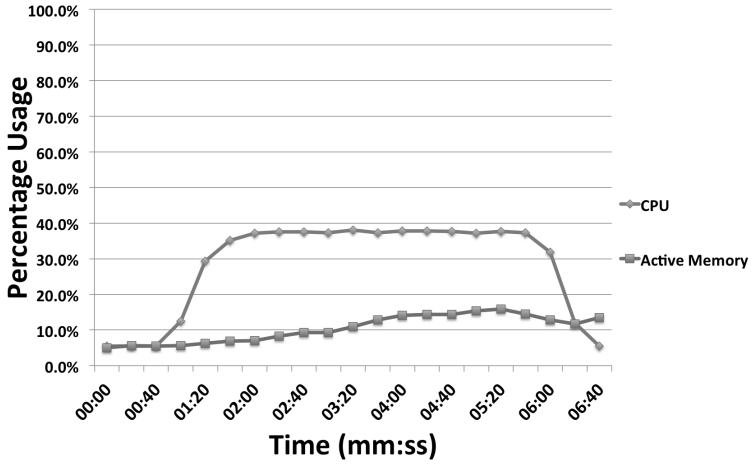


Figure 12. Domain controller CPU and memory utilization during the first test.

IV. Experimental Results and Analysis

This chapter reports results after conducting the experiments laid out in Chapter III. First, common reporting techniques and measures of performance are reviewed. These measures and reporting techniques are then used to report the results of the experiments conducted. The chapter concludes with a short summary.

4.1 Performance Measures

This section reviews the performance measures used in this chapter to demonstrate the efficacy and quality of the statistical 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 (OFP) methods written by Salfner et al. [1].

This research utilizes a technique called cross-validation in which a set of labelled training data are 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 (FN) 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 is the harmonic mean of precision and recall and represents a trade-off between the two [33]. 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 (FPR) 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.

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 FPR 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:

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

4.1.3 Negative Predictive Value (NPV) and Accuracy.

In some cases, it may be desirable 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. This predictor would be highly accurate, but ultimately ineffective.

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 13.

Another popular visualization is the Receiver Operating Characteristic (ROC) curve. By plotting True Positive Rate (TPR) over FPR one is able to see the predictors ability to accurately classify a failure. A sample ROC curve is shown in Figure 14.

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:

$$AUC = \int_0^1 TPR(FPR)\, d\, FPR \in [0, 1]$$

.

A purely 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.

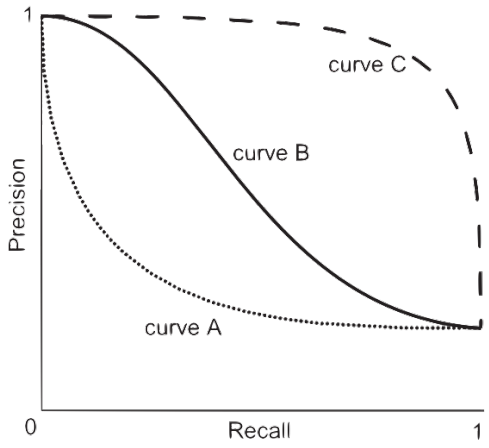


Figure 13. Sample precision/recall curves [1]. Curve A represents a poorly performing predictor, curve B an average predictor, and curve C an exceptional predictor.

The results of the experiments conducted in this research report all of the above described measures of performance in the next section.

4.2 Results

The experiments designed in Chapter III were executed in a virtual environment to produce failure data. The failure data generated was used to train statistical learning models using the open source statistical learning software suite: R. The parameters used to train each model were selected using cross-validation on a subset of the failures generated. Finally, each model was evaluated using a held-out test set. The results of this evaluation for each fault load are reported here.

The rest of this chapter is organized first by the target system, then by the different fault loads that were used to generate failure data on the corresponding target. In each sub-section, the results after training a machine learning model on failure data generated using that type of fault are detailed. Finally, this chapter is concluded with a summary of these results.

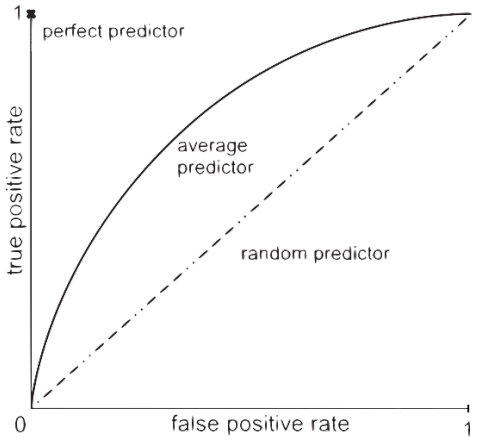


Figure 14. ROC plots of perfect, average, and random predictors [1].

4.2.1 Microsoft (MS) Domain Controller (DC).

4.2.1.1 Fault Injection.

This fault load was effective at creating a failure, but unfortunately, each failure observed occurred immediately after introducing the fault. Because there was no delay between injection and failure, there did not exist any indicators of failure. Consequently, machine learning cannot help in this situation. According to Russinovich, et al. [32] the *lsass.exe* process, as well as other critical system processes, are at the top of the structured exception handling stack and do not handle exceptions. When faced with exceptions, the processes exit and the system reboots.

4.2.1.2 Under-Resourced Central Processing Unit (CPU).

This fault load never resulted in failure. To test this fault load, the virtual domain controllers resources were reduced. The CPU went from a dual-core to a single virtual CPU, and the memory was reduced from 2 Gb to 512 Mb. This reduction was well beneath the recommended capacity [31] for a domain controller. The workload

generator was then allowed to run against this configuration for seven days. For the duration of the test, the CPU load was 100%, and physical memory was 90% utilized on average. While the service did experience reduced response time, failure did not occur.

Another test was conducted to test this fault load by allowing a third-party application to slowly consume all CPU time. Much like the previous test, this test never resulted in failure. Consequently, the learning was not attempted for fault load.

4.2.1.3 Under-Resourced Memory.

The under-resourced memory fault load was the first that created observable indicators of failure with any lead time. This fault load produced the best performing predictors and the largest sliding time window for prediction of sixty seconds. According to James, et al. [34], there can be advantages and trade-offs between parametric and non-parametric models. For this reason, this experiment explores the use of two machine learning models: the weighted Support Vector Machine (SVM), and boosted decision trees using the multinomial distribution.

4.2.1.4 Weighted SVM.

For this prediction method, the *e1071* package was used to train an SVM. The *tune* function was used to run a 5-fold cross-validation a total of 48 times to select the optimal parameters (gamma, cost, and degree polynomial) using: four kernels, four sliding data/prediction windows, and three training/test data splits. The classification weights were set to roughly equal the proportion of failure prone to non-failure prone data windows 0.8 for failure, and 0.2 for non-failure.

The optimal model was selected with the Radial kernel with $\gamma = 0.1$, $cost = 1$, time window = 60 seconds, and the split of data = 4 of the observed failures used for

training.

Initial test performance was poor so the test data was then evaluated in sequential order using a threshold. After two sequential windows were predicted as failure-prone, the next w windows were also predicted as failure-prone, where $w = \textit{window size} - \textit{threshold size}$. For threshold = 2, the resulting confusion matrix for the optimal F-Measure, the ROC curve, and the precision/recall curve are shown in Table 9, and Figure 15 respectively.

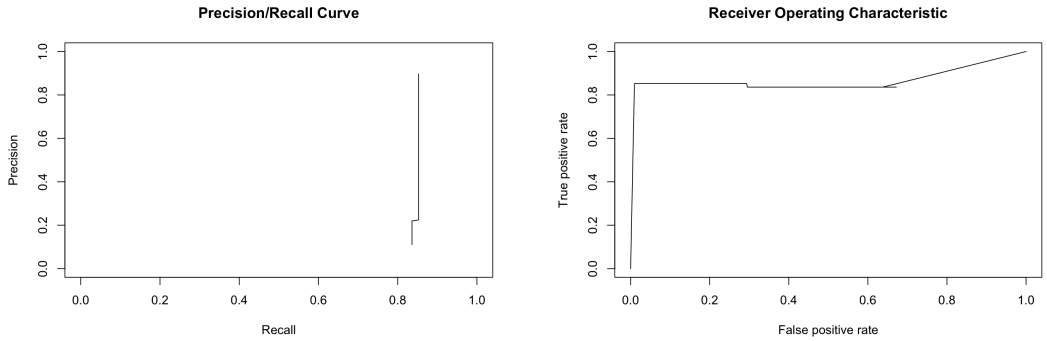
After the software update, the same model was used on a new set of generated failures. The old model did not accurately classify a single failure prone time window. A new model was then trained with the newly generated failure data. Unfortunately, after this software update, with all other things held constant, the weighted SVM model was unable to achieve the same level of performance as before.

4.2.1.5 Boosted Decision Trees.

For this prediction model, the *gbm* package was used to train a boosted decision tree. Cross-validation was used to select $\lambda = 0.03$, the interaction depth of = 4, and the number of trees = 1000. The multinomial distribution was used to perform classification. This was chosen instead of Bernoulli given that the two distributions are the same except multinomial is capable of classification with more than two classes. While this flexibility is not required for this experiment, it may be useful in the future to predict additional system states like 'degraded', or 'idle'.

The precision/recall, and ROC curves on a sixty second data/prediction window are shown in Figure 16. The confusion matrix at the optimal threshold for F-measure is shown in Table 10.

After the software update, the same prediction model was used new set of generated failures. A list of updates that were applied are shown in Appendix C. The



(a) Precision/Recall Curve. (b) ROC Curve (AUC = 0.8664).

Figure 15. Test data performance of the SVM prediction method on failure data obtained by consuming all available memory until target application fails.

Table 9. Confusion matrix on test data created before software updates on threshold with highest F-Measure (0.8739) using SVM.

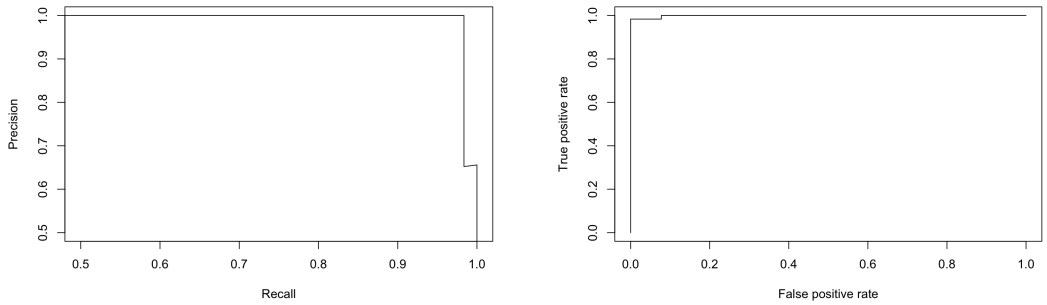
		Actual	
		Fail	**No-Fail**
Predicted	**Fail**	52	6
	No-Fail	9	607

precision/recall and ROC curves on data generated after the software update using the old model are shown in Figure 17. The confusion matrix at the optimal threshold for F-measure is shown in Table 10

Finally, a new predictor was trained using more generated failures as was done before the update. The precision/recall, and ROC curves on the held-out test data are shown in Figure 18 and the confusion matrix at the optimal threshold for F-measure is shown in Table 12.

4.2.1.6 Heap Space Corruption.

Just as with fault injection, this fault load was able to produce failures, but these failures were not preceded by any indicators. To increase realism in this fault load, the focus of the corruption was on the user database. The user database is incrementally cached as authentication requests are received [32]. To test this fault



(a) Precision/Recall Curve. (b) ROC Curve (AUC = 0.9984).

Figure 16. Test data performance of the boosting prediction method on failure data obtained by consuming all available memory until target application fails.

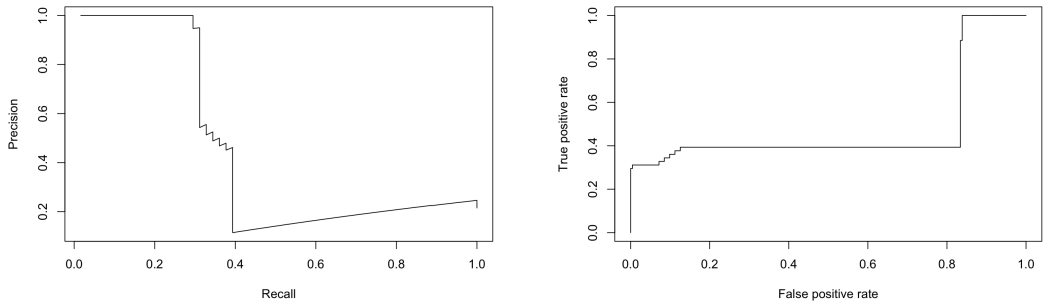
Table 10. Confusion matrix on test data created before software updates on threshold with highest F-Measure (0.9917) using boosting.

		Actual	
		Fail	**No-Fail**
Predicted	**Fail**	60	0
	No-Fail	1	412

load, the Adaptive Failure Prediction (AFP) execution phase was run as normal. After the workload generator reached a steady state, a single user in the database on disk was corrupted followed immediately by the same user being corrupted in process memory. If the disk was not corrupted along with memory, the process would treated the corruption as a cache miss, and re-cached the user from disk. When both were corrupted simultaneously, the process crashed and forced system reboot the very next time that user requested authentication. Unfortunately, exactly as with fault injection, there were no preceding indicators of failure and thus training a prediction model was unsuccessful.

4.2.2 Web Server.

To validate the approach and implementation of the AFP framework in this experiment, it was also tested against an Apache web server. The underlying system



(a) Precision/Recall Curve. (b) ROC Curve (AUC = 0.4854).

Figure 17. Performance of the boosting prediction method trained on failure data created before the software update obtained by consuming all available memory until target application fails.

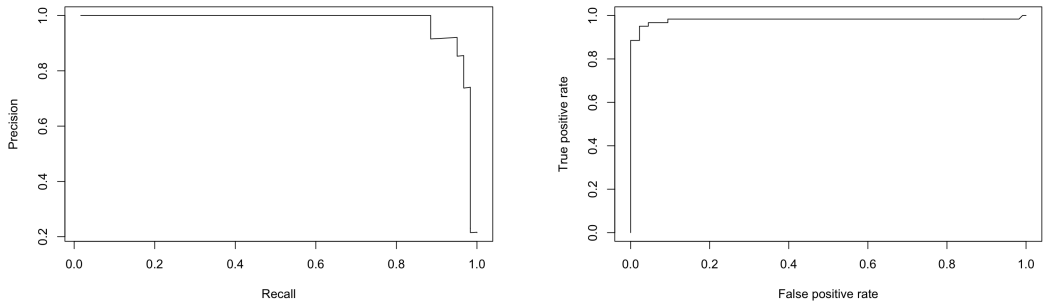
Table 11. Post-update failure data confusion matrix on threshold with highest F-Measure (0.4691) using model trained on failure data generated before software update.

		Actual	
		Fail	**No-Fail**
Predicted	**Fail**	19	1
	No-Fail	42	222

change in this experiment was simulated by upgrading Apache from version 2.2.31x64 to version 2.4.20x64. Results for the web server were almost identical to those for the DC for each fault load. The only predictable failure was in the case of the memory leak. The following sub-sections outline specific results after testing each fault load.

4.2.2.1 Fault Injection.

In the case of the web server, each library loaded by the Apache server process *httpd.exe* was targeted for fault injection. In every case, faults were injected until failure occurred. Much like the DC, for each failure observed, no preceding indications of failure were visible in the log messages.



(a) Precision/Recall Curve. (b) ROC Curve (AUC = 0.9801).

Figure 18. Performance of the boosting prediction method trained on failure data created after the software update obtained by consuming all available memory until target application fails.

Table 12. Post-update failure data confusion matrix on threshold with highest F-Measure (0.9355) using model trained on failure data generated after software update.

		Actual	
		Fail	**No-Fail**
Predicted	**Fail**	58	5
	No-Fail	3	218

4.2.2.2 Under-Resourced CPU.

Much like with the DC, both methods of creating this situation resulted in no failure. The client machines did experience delayed responses, but the server continued to run.

4.2.2.3 Under-Resourced Memory.

As with the DC, this was the only fault load that could be used to predict failure given only reported errors. However, machine learning was not necessary given the low number of log messages produced. Since Apache stores access requests in a separate file, they were essentially pre-filtered. Apache also by default, stores error messages in an external log. There were no messages reported in this file in any of the failure runs conducted. The only indicators produced, were reported by Windows

and recorded by the rsyslog server. An average number of 15 messages were reported during each round of the execution phase and the indicators of failure were easy to see. In this case, simple rules could be used to predict failure in this process so a learning algorithm was not trained.

After the Apache software update was applied, the indicators of failure did not change and there were no additional messages reported in the separate error log. For this reason, the same updates were applied to the operating system as was done for the DC target. After these updates, the indicators changed slightly but were still very few and could be used to write a few simple rules.

These results do not diminish the utility of the AFP framework. Without the framework, the indicators would still be unknown until after a failure. Moreover, there would be no way to tell how long a set of rules would be effective after being written.

4.2.2.4 Heap Space Corruption.

This fault load was tested against the Apache server by targeting the actual web page stored in memory. Much like was done by the DC with users, this was treated as a cache miss and the content was retrieved from disk. Again, to simulate a disk failure, this file was made inaccessible. The result was an immediate failure to serve the content. As with the DC, there were no preceding indications of failure.

4.2.3 Summary.

In summary, the only fault load usable for training a statistical model to predict failure based only on reported errors was the memory leak. As expected, the software update did drastically reduce the effectiveness of a model trained with failure data before the software update. The boosted decision tree was able to be re-trained after

the software update, but the SVM was not. This suggests that both models should be used to ensure the AFP framework is able to adapt to the underlying system changes and maintain at least one useful predictor.

V. Conclusion and Future Work

This chapter outlines several lines of future work based on the outcomes of this research. The future work is then followed by the conclusions drawn from this work and a discussion of their impact.

5.1 Future Work

Several lines of research following this work are presented in this section. First and foremost, in order to put this technique into use on production systems, the proof of concept Windows Software Fault Injection Tool (W-SWFIT) application must be completed. Furthermore, while automation was a consideration while conducting this research, it was not implemented. To be effective in a production environment, the entire Adaptive Failure Prediction (AFP) process must be automated.

One especially relevant and interesting line of effort that should follow this work is to better identify when the underlying system has changed enough to require re-training. While the process is automated, it will unlikely be necessary after every software update. In order to avoid unnecessary use of resources, this process could be explored.

As was demonstrated with the boosted decision trees, other statistical classifiers could be explored. The AFP framework is not limited to a single predictor [2]. A series of prediction models can be used to vote on the state of a system, the output being the majority. In addition to exploring other statistical learning models, additional states (or classes) could be explored. For example, instead of a failure state, a classification model could be used to predict when a system would be idle to know when best to install software updates. Further, a classification model may be able to automate the classification and prediction of when a target was under a malicious attack in a

method similar to the AFP framework.

An additional area of exploration should be to better identify how fault injection actually affects the underlying system. This research has shown that in some cases, it can be extremely difficult to identify areas that will create realistic failure conditions with any preceding indicators. Even when constrained, a single library can have hundreds of injection points. Furthermore, in some cases, even when all injection points are tested, none may lead to a realistic failure. For this reason, the additional fault loads play an integral role.

Finally, the integration of actual failure data with the AFP framework should be explored. Bootstrapping could be used to better integrate actual failure data into the training phase if it is observed.

5.2 Conclusion

This research explored the use of the AFP framework with additional fault loads to predict failure using reported errors in the Microsoft (MS) Domain Controller (DC)s. It has been shown that it is possible to predict failure in modern MS enterprise authentication architecture given a representative fault load. Unfortunately, at the time of writing, two out of the three fault loads introduced in this research were not successful in generating useful failure data. The new fault loads are not useless however. As was demonstrated with the Support Vector Machine (SVM) predictor, the underlying system changes can introduce or eliminate an applications vulnerability to certain types of faults. For this reason, if the AFP framework is implemented on MS DCs, all fault loads should be used in the execution and training phases.

Perhaps more interestingly, fault injection as was used in the original AFP framework implementation, had two outcomes: no failure occurred, or failure occurred immediately. In the controlled virtual environment, failure was predictable using polled

system health information, but perhaps the indicators used to predict the failure were not actual errors but the fault injection tool itself consuming resources. Clearly more work must be done to validate using fault injection alone in the AFP framework.

In addition to the new fault loads introduced in this work, a load generator has also been presented: Distributed PowerShell Load Generator (D-PLG), capable of sufficiently simulating peak usage of a MS enterprise DC. Additional uses for D-PLG outside of use in the AFP framework include capacity planning/sizing, network security testing and auditing, and software testing. This research also introduced W-SWFIT which can be used to perform fault injection for a variety of additional uses like software testing and auditing.

The impact of this research should not be over estimated. As mentioned, a major limitation of this technique is that it is not able to predict malicious acts or *Act of God* events. Furthermore, the data generated are still simulated data and as such, may not completely capture all possible failure events. The AFP framework as presented here will however provide more reliable predictions than are currently available today.

In conclusion, the modified AFP framework as presented here can be used to effectively predict failures that might occur in a production environment and is capable of adapting to underlying system changes using only reported errors. For these reasons, it is recommended that if the AFP framework is to be implemented as laid out in this research, all fault loads should be integrated to maximize the frameworks ability to adapt to system changes. To improve the efficacy of a predictor trained using this generated data, real failure data and additional predictors can easily be integrated if available. Finally, real failure data is difficult to obtain given how rare failure is in modern systems. Unfortunately, even after it is obtained, it can rapidly become deprecated by underlying system changes. Using the AFP with the fault loads introduced in this work to generate simulated failure data is the next best thing to having real

data and provides more useful predictions than are available with no failure data.

Appendix A. Windows Software Fault Injection Tool (W-SWFIT) Source Code

```
// 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();
char* GetAddressOfData(HANDLE process, const char *data, size_t len);

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;
}

// Fault Injection or Memory Corruptions?
string s_fivsmc = "";
cout << endl << "Would␣you␣like␣to␣inject␣Faults,␣or␣corrupt␣Memory?␣[f|m]:␣";
getline(cin, s_fivsmc);
if (s_fivsmc.find("m") != string::npos){

        string s_query = "";
        cout << endl << "What␣are␣you␣looking␣for␣in␣process␣memory?:␣";
        getline(cin, s_query);

        char* ret = GetAddressOfData(hTarget, s_query.c_str(), s_query.length());
        if (ret) {
                cout << "Found␣at␣addr:␣" << (void*)ret << endl;
                size_t bytesRead;
                size_t sizeToRead = s_query.size();
                char *buf = (char *)malloc(sizeToRead + 1);
                ReadProcessMemory(hTarget, ret, buf, sizeToRead, &bytesRead);
                buf[sizeToRead] = '\0';

                cout << "Num␣bytes␣read:␣" << bytesRead << endl;
                cout << "Contents:␣" << string(buf) << endl;

                // Overwrite it
                byte *null_array = (byte *)malloc(bytesRead);
                fill_n(null_array, bytesRead, 0x00);

                SIZE_T mem_bytes_written = 0;
                if (WriteProcessMemory(hTarget, (LPVOID)ret, null_array, bytesRead, &
                     mem_bytes_written) != 0) {
                        cout << "Bytes␣written:␣" << mem_bytes_written << endl;
                        cout << "Successful␣corruption." << endl;
                        return 0;
                } else {
                        cerr << "Failed␣to␣corrupt␣memory:␣" << GetLastError() << endl;
                        return -1;
                }
        } else {
                cout << "Not␣found" << endl;
        }
        return 0;
}

// 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;
}

char* GetAddressOfData(HANDLE process, const char *data, size_t len) {

        SYSTEM_INFO si;
        GetSystemInfo(&si);

        MEMORY_BASIC_INFORMATION info;
        vector<char> chunk;
        char* p = 0;
        while(p < si.lpMaximumApplicationAddress) {

          if(VirtualQueryEx(process, p, &info, sizeof(info)) == sizeof(info)) {

                p = (char*)info.BaseAddress;
                chunk.resize(info.RegionSize);
                SIZE_T bytesRead;

                if(ReadProcessMemory(process, p, &chunk[0], info.RegionSize, &bytesRead))
                  for(size_t i = 0; i < (bytesRead - len); ++i)
                        if(memcmp(data, &chunk[i], len) == 0)
                          return (char*)p + i;

                p += info.RegionSize;

          }
        }
        return 0;
}

// Class definition for the Funciton object

#ifndef FUNCTION_H
#define FUNCTION_H

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

#include "Operator.h"

#include <map>

using namespace std;

class Function {

        public:
                Function(HANDLE _target, DWORD64 _start, DWORD64 _end, byte *_code);
                ~Function();
```

```
		bool inject();

	private:
		map < DWORD64, Operator *> local_injection_points; // Address -> NOP Sequence
		DWORD64 start_addr = 0;
		DWORD64 end_addr = 0;
		byte *buf;
		DWORD64 size = 0;
		HANDLE hTarget; // Managed by Library (don't close it here)

		// Capstone Buffer
		cs_insn *code_buf;
		size_t cs_count = 0;
		csh cs_handle;

		bool build_injection_points();
		bool perform_injection(DWORD64 addr);
		bool inject(Operator *op, DWORD64 addr);

		// Build map of injectable points
		bool find_operators_mfc();
		bool find_operators_ompla();

};

#endif

// Class definition for the Library object (contains single DLL)

#ifndef LIBRARY_H
#define LIBRARY_H

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

#include "Operators.h"
#include "Operator.h"
#include "Function.h"

#include <vector>
#include <map>

using namespace std;

class Library {

	public:
		Library(HANDLE _target, DWORD64 _start, DWORD _size, string _path);
		~Library();

		bool write_library_to_disk(string path);
		bool inject();

	private:
		string name; // Name of library
		vector < Function * > functions; // Vector (list) of functions in library
		map < Operator *, Operator * > function_patterns; // Vector of function patterns
		byte *buf; // Buffer for memory contents
		DWORD64 start_addr = 0;
		DWORD image_size = 0;
		HANDLE hTarget;

		bool read_memory_into_bufer();
		bool build_operator_map();
		bool find_functions();
```

```
		bool find_pattern(Operator *op, DWORD64 start, DWORD64 stop, DWORD64 *location);
};

#endif

// Class definition for the Operator object
// This object contains a byte array and a size

#ifndef OPERATOR_H
#define OPERATOR_H

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

using namespace std;

class Operator {

	public:
		Operator(const byte *pattern, DWORD64 size);
		~Operator();

		DWORD64 size() { return _size; }
		const byte *pattern() { return (const byte *)_pattern; }

	private:
		byte *_pattern;
		DWORD64 _size;
};

#endif

// Operators.h : Defines the operators to search and replace
//

#ifndef OPERATORS_H
#define OPERATORS_H

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

const byte start_pattern_1[] = { 0x55, 0x48, 0x83, 0xec, 0x20, 0x48, 0x8b, 0xea };
const byte end_pattern_1[] = { 0x48, 0x83, 0xc4, 0x20, 0x5d, 0xc3 };
/*	Begin Function:
	PUSH RBP
	SUB RSP, 0x20
	MOV RBP, RDX

    End Function:
	ADD RSP, 0x20
	POP RBP
	RET
*/

const byte start_pattern_2[] = { 0xff, 0xf3, 0x48, 0x83, 0xec, 0x20, 0x48, 0x8b, 0xd9 };
const byte end_pattern_2[] = { 0x48, 0x83, 0xc4, 0x20, 0x5b, 0xc3 };
/*	Begin Function:
	PUSH RBX
	SUB RSP, 0x20
	MOV RBX, RCX

	End Function:
	ADD RSP, 0x20
	POP RBX
	RET
*/
```

```
const byte start_pattern_3[] = { 0xff, 0xf3, 0x48, 0x83, 0xec, 0x20, 0x8b, 0xd9 };
const byte end_pattern_3[] = { 0x48, 0x83, 0xc4, 0x20, 0x5b, 0xc3 };
/*     Begin Function:
       PUSH RBX
       SUB RSP, 0x20
       MOV EBX, ECX

       End Function:
       ADD RSP, 0x20
       POP RBX
       RET
*/

const byte start_pattern_4[] = { 0x57, 0x48, 0x83, 0xec, 0x20, 0x48, 0x8b, 0xf9 };
const byte end_pattern_4[] = { 0x48, 0x83, 0xc4, 0x20, 0x5f, 0xc3 };
/*     Begin Function:
       PUSH RDI
       SUB RSP, 0x20
       MOV RDI, RCX

       End Function:
       ADD RSP, 0x20
       POP RDI
       RET
*/

const byte start_pattern_5[] = { 0x57, 0x48, 0x83, 0xec, 0x20, 0x8b, 0xf9 };
const byte end_pattern_5[] = { 0x48, 0x83, 0xc4, 0x20, 0x5f, 0xc3 };
/*     Begin Function:
       PUSH RDI
       SUB RSP, 0x20
       MOV EDI, ECX

       End Function:
       ADD RSP, 0x20
       POP RDI
       RET
*/

const byte start_pattern_6[] = { 0x57, 0x48, 0x83, 0xec, 0x20, 0x8b, 0xf1 };
const byte end_pattern_6[] = { 0x48, 0x83, 0xc4, 0x20, 0x5f, 0xc3 };
/*     Begin Function:
       PUSH RDI
       SUB RSP, 0x20
       MOV ESI, ECX

       End Function:
       ADD RSP, 0x20
       POP RDI
       RET
*/

const byte start_pattern_7[] = { 0xff, 0xf3, 0x48, 0x83, 0xec, 0x20, 0x48, 0x8d, 0x0d };
const byte end_pattern_7[] = { 0x48, 0x83, 0xc4, 0x20, 0x5b, 0xc3 };
/*     Begin Function:
       PUSH RBX
       SUB RSP, 0x20
       LEA RCX, 'immed'

       End Function:
       ADD RSP, 0x20
       POP RBX
       RET
*/
```

```
const byte start_pattern_8[] = { 0x57, 0x48, 0x83, 0xec, 0x40, 0x48, 0x8b, 0xe9 };
const byte end_pattern_8[] = { 0x48, 0x83, 0xc4, 0x40, 0x5f, 0xc3 };
/*     Begin Function:
       PUSH RDI
       SUB RSP, 0x40
       MOV RBP, RCX

       End Function:
       ADD RSP, 0x40
       POP RDI
       RET
*/

const byte start_pattern_9[] = { 0x57, 0x48, 0x83, 0xec, 0x20, 0x48, 0x8b, 0xf1 };
const byte end_pattern_9[] = { 0x48, 0x83, 0xc4, 0x20, 0x5f, 0xc3 };
/*     Begin Function:
       PUSH RDI
       SUB RSP, 0x20
       MOV RSI, RCX

       End Function:
       ADD RSP, 0x20
       POP RDI
       RET
*/

 const byte start_pattern_10[] = { 0x57, 0x48, 0x83, 0xec, 0x20, 0x48, 0x8b, 0xe9 };
 const byte end_pattern_10[] = { 0x48, 0x83, 0xc4, 0x20, 0x5f, 0xc3 };
/*     Begin Function:
       PUSH RDI
       SUB RSP, 0x20
       MOV RBP, RCX

       End Function:
       ADD RSP, 0x20
       POP RDI
       RET
*/

 const byte start_pattern_11[] = { 0x57, 0x48, 0x83, 0xec, 0x30, 0x48, 0x8b, 0xe9 };
 const byte end_pattern_11[] = { 0x48, 0x83, 0xc4, 0x30, 0x5f, 0xc3 };
/*     Begin Function:
       PUSH RDI
       SUB RSP, 0x30
       MOV RBP, RCX

       End Function:
       ADD RSP, 0x30
       POP RDI
       RET
*/

 const byte start_pattern_12[] = { 0x57, 0x48, 0x83, 0xec, 0x20, 0x48, 0x8b, 0x05 };
 const byte end_pattern_12[] = { 0x48, 0x83, 0xc4, 0x20, 0x5f, 0xc3 };
/*     Begin Function:
       PUSH RDI
       SUB RSP, 0x20
       MOV RAX, 'immed'

       End Function:
       ADD RSP, 0x20
       POP RDI
       RET
*/

 const byte omva_1[] = { 0x48, 0x8b, 0x5c, 0x24, 0x30 }; // MOV RBX, [RSP+0x30]
```

```
 const byte omva_2[] = { 0x48, 0x8b, 0x74, 0x24, 0x38 }; // MOV RSI, [RSP+0x38]

#endif


#ifndef GLOBALS_H
#define GLOBALS_H

#include <stdio.h>
#include <tchar.h>

#include <windows.h>
#include <string>
#include <psapi.h>
#include <iostream>
#include <fstream>
#include <io.h>

#include <capstone.h>
#include <inttypes.h>

#endif

// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, but
// are changed infrequently
//

#pragma once

#include "targetver.h"

#include <WinSock2.h>
#include <Ws2tcpip.h>
#pragma comment(lib, "Ws2_32.lib")

#include <stdio.h>
#include <tchar.h>

#include "Library.h"
#include "Function.h"
#include "Operator.h"


#pragma once

// Including SDKDDKVer.h defines the highest available Windows platform.

// If you wish to build your application for a previous Windows platform, include WinSDKVer.h and
// set the _WIN32_WINNT macro to the platform you wish to support before including SDKDDKVer.h.

#include <SDKDDKVer.h>

// 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();
```

```
char* GetAddressOfData(HANDLE process, const char *data, size_t len);

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;
	}

	// Fault Injection or Memory Corruptions?
	string s_fivsmc = "";
	cout << endl << "Would␣you␣like␣to␣inject␣Faults,␣or␣corrupt␣Memory?␣[f|m]:␣";
	getline(cin, s_fivsmc);
	if (s_fivsmc.find("m") != string::npos){

		string s_query = "";
		cout << endl << "What␣are␣you␣looking␣for␣in␣process␣memory?:␣";
		getline(cin, s_query);

		char* ret = GetAddressOfData(hTarget, s_query.c_str(), s_query.length());
		if (ret) {
```

```
                cout << "Found␣at␣addr:␣" << (void*)ret << endl;
                size_t bytesRead;
                size_t sizeToRead = s_query.size();
                char *buf = (char *)malloc(sizeToRead + 1);
                ReadProcessMemory(hTarget, ret, buf, sizeToRead, &bytesRead);
                buf[sizeToRead] = '\0';

                cout << "Num␣bytes␣read:␣" << bytesRead << endl;
                cout << "Contents:␣" << string(buf) << endl;

                // Overwrite it
                byte *null_array = (byte *)malloc(bytesRead);
                fill_n(null_array, bytesRead, 0x00);

                SIZE_T mem_bytes_written = 0;
                if (WriteProcessMemory(hTarget, (LPVOID)ret, null_array, bytesRead, &
                     mem_bytes_written) != 0) {
                        cout << "Bytes␣written:␣" << mem_bytes_written << endl;
                        cout << "Successful␣corruption." << endl;
                        return 0;
                } else {
                        cerr << "Failed␣to␣corrupt␣memory:␣" << GetLastError() << endl;
                        return -1;
                }
        } else {
                cout << "Not␣found" << endl;
        }
        return 0;
}

// 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;
}
```

```
char* GetAddressOfData(HANDLE process, const char *data, size_t len) {

        SYSTEM_INFO si;
        GetSystemInfo(&si);

        MEMORY_BASIC_INFORMATION info;
        vector<char> chunk;
        char* p = 0;
        while(p < si.lpMaximumApplicationAddress) {

          if(VirtualQueryEx(process, p, &info, sizeof(info)) == sizeof(info)) {

                p = (char*)info.BaseAddress;
                chunk.resize(info.RegionSize);
                SIZE_T bytesRead;

                if(ReadProcessMemory(process, p, &chunk[0], info.RegionSize, &bytesRead))
                  for(size_t i = 0; i < (bytesRead - len); ++i)
                        if(memcmp(data, &chunk[i], len) == 0)
                          return (char*)p + i;

                p += info.RegionSize;

          }
        }
        return 0;
}


#include "stdafx.h"
#include "Function.h"
#include "Operators.h"

Function::Function(HANDLE _target, DWORD64 _start, DWORD64 _end, byte *_code) {
        hTarget = _target;
        start_addr = _start;
        end_addr = _end;
        size = end_addr - start_addr;
        buf = _code;
        local_injection_points = map < DWORD64, Operator *>();

        // Build Capstone (CS) Array of code
        if (cs_open(CS_ARCH_X86, CS_MODE_64, &cs_handle) != CS_ERR_OK)
                cerr << "Error␣disassembling␣code." << endl;

        // Enable op details
        cs_option(cs_handle, CS_OPT_DETAIL, CS_OPT_ON);
        //cs_option(cs_handle, CS_OP_DETAIL, CS_OPT_ON);

        cs_count = cs_disasm(cs_handle, buf, size, start_addr, 0, &code_buf);
        if (cs_count == 0)
                cerr << "Error␣disassembling␣code." << endl;


        // Build Injection points based on disassembled code
        build_injection_points();
}

Function::~Function() {
        cs_close(&cs_handle);
        cs_free(code_buf, size);
}

// Public Functions
bool Function::inject() {
```

```
	// For each injection point in the funciton, ask the user if they would like to inject
	for (map<DWORD64, Operator *>::iterator it = local_injection_points.begin();
		it != local_injection_points.end(); ++it) {

		// If the user injects, return true;
		//if (inject(it->second, it->first))
			//return true;
		inject(it->second, it->first);
		Sleep(100);
	}
	return false;
}

// Private Functions
bool Function::build_injection_points() {
	find_operators_mfc();
	//find_operators_ompla();
	return true;
}

// Returns address of injection point for "Operator of Missing Localize Part of the Algorithm (OMPLA)"
bool Function::find_operators_ompla() {

	// Constraint 2 (C02): Call must not be only statement in the block
	if (cs_count < 10) return false;

	for (size_t j = 0; j < cs_count; j++) {
		if (string(code_buf[j].mnemonic).find("mov") != string::npos){

			// Constraint 10: Statements must be in the same block and do not include loops
			size_t c = 0;
			for (size_t i = j + 1; i < cs_count; i++)
				if (string(code_buf[j].mnemonic).find("mov") != string::npos) {
					c++;
					continue;
				}

			if (c > 2 && c <= 5) {
				// Doesn't violate any of the OMPLA constraints, add it
				Operator *op = new Operator(code_buf[j].bytes, code_buf[j].size);
				local_injection_points[code_buf[j].address] = op;
			}
		}
	}
	return true;
}

// Returns address of injection point for "Operator for Missing Function Call (OMFC)"
bool Function::find_operators_mfc() {

	// Constraint 2 (C02): Call must not be only statement in the block
	if (cs_count < 6) return false;

	for (size_t j = 0; j < cs_count; j++) {
		if (string(code_buf[j].mnemonic).find("call") != string::npos){
			// Constraint 1 (C01): Return value of the function (EAX/RAX) must not be used.
			bool constraint01 = false;
			for (size_t i = j + 1; i < cs_count; i++) {
				cs_detail *details = code_buf[i].detail;
				if (code_buf[i].detail) {
					for (size_t k = 0; k < details->regs_read_count; k++) {
						string modreg = string(cs_reg_name(cs_handle, details->
							regs_read[k]));
						if (modreg.find("eax") != string::npos || modreg.find("rax
							") != string::npos)
							constraint01 = true;
```

```
                                        }
                                }
                        }

                        // Doesn't violate any of the OMFC constraints, add it
                        if (!constraint01) {
                                Operator *op = new Operator(code_buf[j].bytes, code_buf[j].size);
                                local_injection_points[code_buf[j].address] = op;
                        }
                }
        }
        return true;
}

bool Function::inject(Operator *op, DWORD64 addr) {

        // Ready to continue?
        //string cont = "";
        //printf("Ready to inject %d bytes at: 0x%X\n\n", op->size(), addr);
        //cout << "Continue? [Y/n]: ";
        //getline(cin, cont);

        //if (cont.find("n") != string::npos || cont.find("N") != string::npos) {
                //cout << "Aborting" << endl;
                //return false;
        //}

        byte *nop_array = (byte *)malloc(op->size());
        byte *tmp_buf = (byte *)malloc(op->size());
        fill_n(nop_array, op->size(), 0x90);

        SIZE_T mem_bytes_written = 0;
        if (WriteProcessMemory(hTarget, (LPVOID)addr, nop_array, op->size(), &mem_bytes_written) != 0)
              {
                cout << "Attemting␣injection..." << endl;

                // Check to make sure the OS allowed the operation
                SIZE_T num_bytes_read = 0;
                int count = 0;

                while (true) {
                        if (ReadProcessMemory(hTarget, (DWORD64 *)addr, tmp_buf, op->size(), &
                             num_bytes_read) != 0) {
                                cout << string((char *)tmp_buf) << endl;
                                int i;
                                for (i = 0; i < op->size(); i++) {
                                        if (tmp_buf[i] != nop_array[i])
                                                break;
                                }

                                if (i >= op->size() - 1) {
                                        cout << "Bytes␣written:␣" << mem_bytes_written << endl;
                                        cout << "Successful␣injection." << endl;
                                        return true;
                                }
                        }
                }
        } else {
                cerr << "Failed␣to␣inject␣fault␣into␣memory:␣" << GetLastError() << endl;
                return false;
        }
        return false;
}

#include "stdafx.h"
#include "Process.h"
```

```
Library::Library(HANDLE _target, DWORD64 _start, DWORD _size, string _path) {
	hTarget = _target;
	start_addr = _start;
	image_size = _size;
	name = _path;
	buf = (byte *)malloc(image_size);
	function_patterns = map < Operator *, Operator * >();
	functions = vector < Function *>();

	if (!buf) {
		cerr << "Failed␣to␣allocate␣space␣for␣memory␣contents:␣" << GetLastError() << endl;
		CloseHandle(hTarget);
		return;
	}
	read_memory_into_bufer();
	build_operator_map();
	find_functions();
}


Library::~Library() {
	free(buf);
	CloseHandle(hTarget);
}

// PUBLIC FUNCTIONS
bool Library::write_library_to_disk(string path) {

	cout << "Writing␣static␣copy␣of␣memory␣contents␣for␣analysis␣to␣" << path << endl;
	FILE *fp;
	fopen_s(&fp, path.c_str(), "w");
	SIZE_T bytes_written = 0;
	while (bytes_written < image_size) {
		bytes_written += fwrite(buf, 1, image_size, fp);
	}
	fclose(fp);
	cout << "Wrote␣" << bytes_written << "␣bytes." << endl << endl;
	return true;
}

bool Library::inject() {
	// For each function in the module, call public inject funciton
	while (true) {
		for (vector< Function *>::iterator it = functions.begin(); it != functions.end(); ++it)
		      {
			if ((*it)->inject())
				return true;
		}
	}
	return true;
}

// PRIVATE FUNCTIONS
bool Library::read_memory_into_bufer() {
	SIZE_T num_bytes_read = 0;
	int count = 0;

	if (ReadProcessMemory(hTarget, (DWORD64 *)start_addr, buf, image_size, &num_bytes_read) != 0)
	     {
		cout << "Buffered␣memory␣contents.␣Got␣" << num_bytes_read << "␣bytes." << endl << endl
		     ;
		return true;
	}
	else {
		cout << "Failed␣to␣read␣memory:␣" << GetLastError() << endl;
```

```
                return false;
        }
        return false;
}

bool Library::build_operator_map() {
        function_patterns[new Operator(start_pattern_1, sizeof(start_pattern_1))] =
                new Operator(end_pattern_1, sizeof(end_pattern_1));
        function_patterns[new Operator(start_pattern_2, sizeof(start_pattern_2))] =
                new Operator(end_pattern_2, sizeof(end_pattern_2));
        function_patterns[new Operator(start_pattern_3, sizeof(start_pattern_3))] =
                new Operator(end_pattern_3, sizeof(end_pattern_3));
        function_patterns[new Operator(start_pattern_4, sizeof(start_pattern_4))] =
                new Operator(end_pattern_4, sizeof(end_pattern_4));
        function_patterns[new Operator(start_pattern_5, sizeof(start_pattern_5))] =
                new Operator(end_pattern_5, sizeof(end_pattern_5));
        function_patterns[new Operator(start_pattern_6, sizeof(start_pattern_6))] =
                new Operator(end_pattern_6, sizeof(end_pattern_6));
        function_patterns[new Operator(start_pattern_7, sizeof(start_pattern_7))] =
                new Operator(end_pattern_7, sizeof(end_pattern_7));
        function_patterns[new Operator(start_pattern_8, sizeof(start_pattern_8))] =
                new Operator(end_pattern_8, sizeof(end_pattern_8));
        function_patterns[new Operator(start_pattern_9, sizeof(start_pattern_9))] =
                new Operator(end_pattern_9, sizeof(end_pattern_9));
        function_patterns[new Operator(start_pattern_10, sizeof(start_pattern_10))] =
                new Operator(end_pattern_10, sizeof(end_pattern_10));
        function_patterns[new Operator(start_pattern_11, sizeof(start_pattern_11))] =
                new Operator(end_pattern_11, sizeof(end_pattern_11));
        function_patterns[new Operator(start_pattern_12, sizeof(start_pattern_12))] =
                new Operator(end_pattern_12, sizeof(end_pattern_12));
        return true;
}

bool Library::find_functions() {
        for (map < Operator *, Operator * >::iterator it = function_patterns.begin();
                        it != function_patterns.end(); ++it ) {

                DWORD64 begin = 0;
                while (find_pattern(it->first, begin, image_size, &begin)) {

                        DWORD64 end = 0;
                        if (find_pattern(it->second, begin, image_size, &end)) {
                                functions.push_back(new Function(hTarget, start_addr + begin + (it->first
                                     )->size(),
                                                                                        start_addr +
                                                                                             end - (it
                                                                                             ->second)
                                                                                             ->size(),
                                                                                             &buf[begin
                                                                                             ]));
                        }
                        begin++;
                }
        }
        return true;
}

// Search 'buf' for 'pattern' at 'start'. If found, sets 'offset', and returns true.
bool Library::find_pattern(Operator *op, DWORD64 start, DWORD64 stop, DWORD64 *location) {

        const byte *pattern = op->pattern();
        for (DWORD64 i = start; i < stop; i++) {
                if (buf[i] == pattern[0]) {
                        for (int j = 1; j < op->size(); j++) {
                                if (buf[i + j] != pattern[j])
                                        break;
```

```
                        if (j < op->size() - 1)
                                continue;

                        *location = i;
                        return true;
                }
        }
    }
    return false;
}

#include "stdafx.h"
#include "Operator.h"

Operator::Operator(const byte *pattern, DWORD64 size) {
    _size = size;
    _pattern = (byte *)malloc(_size);
    memcpy(_pattern, pattern, size);
}

Operator::~Operator() {
    free(_pattern);
}

// stdafx.cpp : source file that includes just the standard includes
// FaultInjection.pch will be the pre-compiled header
// stdafx.obj will contain the pre-compiled type information

#include "stdafx.h"

// TODO: reference any additional headers you need in STDAFX.H
// and not in this file
```

Appendix B. ResourceLeak Source Code

```
##############################################################################
#                                                                            #
# W-SWFIT: Resource Leak                                                     #
# Authors: Paul Jordan                                                       #
# Date Created: 8 May 2016                                                   #
# Description: Makefile for the W-SWFIT Resource Leak project.               #
#                                                                            #
# Copyright (c) 2016                                                         #
#                                                                            #
##############################################################################
PROGRAM=resourceleak
C_FILES:=$(shell find . -iname "*.cpp")
OBJS:=$(patsubst %.cpp, %.o, $(C_FILES))
CFLAGS=-Wall -ffast-math -O3 -std=c++11 -I./incl/
LDFLAGS=
SRC=src

native: CC=g++
windows: CC=/usr/local/gcc-4.8.0-qt-4.8.4-for-mingw32/win32-gcc/bin/i586-mingw32-g++

all: native

windows: $(OBJS)
	$(CC) $(CFLAGS) $(OBJS) $(LDFLAGS) -o $(PROGRAM).exe

native: $(OBJS)
	$(CC) $(CFLAGS) $(OBJS) $(LDFLAGS) -o $(PROGRAM)

%.o: %.cpp
	$(CC) $(CFLAGS) -c $< -o $@

%: %.cpp
	$(CC) $(CFLAGS) -o $@ $<

clean:
	$(RM) $(PROGRAM) *.o $(SRC)/*.o $(PROGRAM).exe
```

```
/*****************************************************************************/
/*                                                                           */
/* 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;
}

/*****************************************************************************/
/*                                                                           */
/* cpu.cpp                                                                */
/* Project: W-SWFIT: Resource Leak                                           */
/* Authors: Paul Jordan                                                      */
/* Date Created: 8 May 2016                                                  */
/* Description: Implementation file for the CPU leak class.               */
/*                                                                           */
/* Copyright (c) 2016                                                        */
/*                                                                           */
/*****************************************************************************/

#include "cpu.hpp"

bool CPU::start(int rate) {
  _running = true;
  _rate = rate;
  __rate = rate; // mutable (degrading) rate
  _leak = thread(&CPU::leak, this);
  return true;
}

void CPU::leak() {
  while(_running) {

    if (__rate > 1) { __rate *= .99; }
    else { __rate = 0; }

    this_thread::sleep_for(chrono::milliseconds((int)__rate));
  }
}

bool CPU::stop() {
  _running = false;
  return true;
```

```
}

/*****************************************************************************/
/*                                                                           */
/* Memory.cpp                                                                */
/* Project: W-SWFIT: Resource Leak                                           */
/* Authors: Paul Jordan                                                      */
/* Date Created: 8 May 2016                                                  */
/* Description: Implementation file for the Memory leak class.               */
/*                                                                           */
/* Copyright (c) 2016                                                        */
/*                                                                           */
/*****************************************************************************/

#include <math.h>
#include "memory.hpp"

Memory::Memory() {
  storage = vector<void *>();
}

Memory::~Memory() {
  storage.clear();
}

bool Memory::start(int rate) {
  _running = true;
  _rate = rate;
  _leak = thread(&Memory::leak, this);
  return true;
}

void Memory::leak() {
  while(_running) {
    void *buf = malloc(pow(10,6)); // Allocate 1MB at rate
    storage.push_back(buf);
    this_thread::sleep_for(chrono::milliseconds(_rate));
  }
}

bool Memory::stop() {
  _running = false;
  return true;
}

/*****************************************************************************/
/*                                                                           */
/* cpu.hpp                                                                */
/* Project: W-SWFIT: Resource Leak                                           */
/* Authors: Paul Jordan                                                      */
/* Date Created: 8 May 2016                                                  */
/* Description: Header file for the CPU leak class.                       */
/*                                                                           */
/* Copyright (c) 2016                                                        */
/*                                                                           */
/*****************************************************************************/

#ifndef CPU_H
#define CPU_H

#include "globals.hpp"
#include "resource.hpp"

using namespace std;

class CPU : public Resource {
```

```
  public:
    CPU() {}
    ~CPU() {}

    bool start(int rate); // smaller number = faster leak
    bool stop();

    bool running() const { return _running; }
    int rate() const { return _rate; }

  private:
    void leak();

    bool _running = false;
    int _rate = 0;
    double __rate = 0;
    thread _leak;

};

#endif

#ifndef GLOBALS_H
#define GLOBALS_H

#ifdef __MINGW32__
#include "mingw.thread.h"
#endif

#include <stdlib.h>
#include <chrono>
#include <vector>
#include <thread>
#include <iostream>
#include <sstream>

#endif

/*****************************************************************************/
/*                                                                           */
/* Memory.hpp                                                                */
/* Project: W-SWFIT: Resource Leak                                           */
/* Authors: Paul Jordan                                                      */
/* Date Created: 8 May 2016                                                  */
/* Description: Header file for the Memory leak class.                       */
/*                                                                           */
/* Copyright (c) 2016                                                        */
/*                                                                           */
/*****************************************************************************/

#ifndef MEMORY_H
#define MEMORY_H

#include "globals.hpp"
#include "resource.hpp"

using namespace std;

class Memory : public Resource {
  public:
    Memory();
    ~Memory();

    bool start(int rate); // # of milliseconds to sleep
                          // before allocating more memory
                          // (smaller number = faster leak)
```

```
    bool stop();

    bool running() const { return _running; }
    int rate() const { return _rate; }

  private:
    void leak();

    vector<void *> storage;
    bool _running = false;
    int _rate = 0;
    thread _leak;

};

#endif

/*****************************************************************************/
/*                                                                           */
/* Resource.hpp                                                              */
/* Project: W-SWFIT: Resource Leak                                           */
/* Authors: Paul Jordan                                                      */
/* Date Created: 8 May 2016                                                  */
/* Description: Abstract resource header file. Each resource implements */
/* abstract class.                                                           */
/*                                                                           */
/* Copyright (c) 2016                                                        */
/*                                                                           */
/*****************************************************************************/

#ifndef RESOURCE_H
#define RESOURCE_H

#include "globals.hpp"

class Resource {
  public:
    virtual bool start(int rate) = 0;
    virtual bool stop() = 0;
    bool running() const { return _running; }
    int rate() const { return _rate; }

  private:
    bool _running = false;
    int _rate = 0;

};

#endif
```

Appendix C. Windows Updates

Table 13. Updates applied to Windows Server 2008 R2 x64 Edition.

Description	HotFixID	Description	HotFixID
Update	982861	Security Update	KB2676562
Security Update	KB2032276	Security Update	KB2685939
Security Update	KB2207559	Security Update	KB2690533
Security Update	KB2296011	Security Update	KB2691442
Security Update	KB2305420	Security Update	KB2698365
Update	KB2345886	Security Update	KB2705219
Security Update	KB2347290	Security Update	KB2706045
Security Update	KB2387149	Security Update	KB2712808
Security Update	KB2393802	Update	KB2718704
Security Update	KB2419640	Security Update	KB2729451
Security Update	KB2423089	Security Update	KB2736418
Security Update	KB2425227	Security Update	KB2742598
Security Update	KB2442962	Security Update	KB2743555
Update	KB2454826	Update	KB2748349
Security Update	KB2483614	Update	KB2749655
Update	KB2506014	Security Update	KB2753842
Security Update	KB2506212	Security Update	KB2756920
Security Update	KB2509553	Security Update	KB2757638
Security Update	KB2511455	Security Update	KB2758857
Update	KB2533552	Security Update	KB2765809
Security Update	KB2535512	Security Update	KB2769369
Security Update	KB2536275	Security Update	KB2770660
Security Update	KB2536276	Security Update	KB2772930

Security Update	KB2544893	Update	KB2779562
Update	KB2552343	Security Update	KB2785220
Security Update	KB2560656	Security Update	KB2789644
Security Update	KB2564958	Security Update	KB2790113
Security Update	KB2570947	Security Update	KB2790655
Security Update	KB2584146	Security Update	KB2807986
Security Update	KB2585542	Security Update	KB2813170
Security Update	KB2604114	Security Update	KB2813347
Security Update	KB2618451	Security Update	KB2840149
Security Update	KB2620704	Security Update	KB972270
Security Update	KB2621440	Update	KB974431
Security Update	KB2631813	Security Update	KB974571
Security Update	KB2643719	Hotfix	KB975467
Security Update	KB2644615	Security Update	KB975560
Security Update	KB2645640	Update	KB977074
Security Update	KB2647170	Security Update	KB978542
Security Update	KB2653956	Security Update	KB978601
Security Update	KB2654428	Security Update	KB979309
Security Update	KB2655992	Security Update	KB979482
Security Update	KB2656355	Security Update	KB979687
Security Update	KB2656410	Security Update	KB979688
Security Update	KB2658846	Update	KB979900
Security Update	KB2659262	Update	KB980408
Update	KB2661254	Security Update	KB982132
Security Update	KB2667402	Security Update	KB982799

Appendix D. List of Abbreviations

AD	Active Directory
AFP	Adaptive Failure Prediction
API	Application Programming Interface
ASLR	Address Space Layout Randomization
AUC	Area Under the Curve
CPU	Central Processing Unit
CRISP-DM	Cross Industry Standard Process for Data Mining
CSCS	Cyber Security and Control System
D-PLG	Distributed PowerShell Load Generator
DC	Domain Controller
DNS	Domain Name System
DOD	Department of Defense
FN	False Negative
FP	False Positive
FPR	False Positive Rate
G-SWFIT	General Software Fault Injection Technique
GB	Gigabyte
GHSMM	Generalized Hidden Semi-Markov Model
GHz	Gigahertz
IP	Internet Protocol
MS	Microsoft

NOS	Network Operation Squadrons
NPV	Negative Predictive Value
ODC	Orthogonal Defect Classification
OFP	Online Failure Prediction
PFM	Proactive Fault Management
RDP	Remote Desktop Protocol
ROC	Receiver Operating Characteristic
SQL	Structured Query Language
SVM	Support Vector Machine
TN	True Negative
TP	True Positive
TPR	True Positive Rate
VM	Virtual Machine
W-SWFIT	Windows Software Fault Injection Tool

Bibliography

1. F. Salfner, M. Lenk, and M. Malek, "A survey of online failure prediction methods," *ACM Computing Surveys (CSUR)*, vol. 42, no. 3, 2010.

2. I. Irrera, M. Vieira, and J. Duraes, "Adaptive failure prediction for computer systems: A framework and a case study," in *Proceedings of the 2015 IEEE 16th International Symposium on High Assurance Systems Engineering (HASE 2015)*, pp. 142–149, 2015.

3. J. Duraes and H. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Transactions on Software Engineering*, vol. 32, pp. 849–867, Nov. 2006.

4. C. Schmidt, *Agile Software Development Teams*. Progress in Information Systems, Springer International Publishing, 2016.

5. E. Bauer and R. Adams, *Reliability and Availability of Cloud Computing*. John Wiley & Sons, 2012.

6. G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot - a technique for cheap recovery," in *Proceedings of the 6th USENIX Symposium on Operating System Design and Implementation (OSDI)*, vol. 4, pp. 31–44, 2004.

7. A. Avižienis, J. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

8. F. Salfner, M. Schieschke, and M. Malek, "Predicting failures of computer systems: A case study for a telecommunication system," in *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2006)*, Apr. 2006.

9. F. Salfner and M. Malek, "Using hidden semi-markov models for effective online failure prediction," in *Proceedings of the 2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pp. 161–174, 2007.

10. C. Domeniconi, C. Perng, R. Vilalta, and S. Ma, "A classification approach for prediction of target events in temporal sequences," in *Proceedings of the 6th European Conference for Principles of Data Mining and Knowledge Discovery*, pp. 125–137, 2002.

11. 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*, vol. 86, no. 1, pp. 2–11, 2013.

12. E. Fulp, G. Fink, and J. Haack, "Predicting computer system failures using support vector machines," in *Proceedings of the 1st USENIX Conference on Analysis of System Logs*, 2008.

13. J. Murray, G. Hughes, and K. Kreutz-Delgado, "Machine learning methods for predicting failures in hard drives: A multiple-instance application," *Journal of Machine Learning Research*, vol. 6, pp. 783–816, 2005.

14. M. Sonoda, Y. Watanabe, and Y. Matsumoto, "Prediction of failure occurrence time based on system log message pattern learning," in *Proceedings of the 2012 IEEE Network Operations and Management Symposium (NOMS 2012)*, pp. 578–581, Apr. 2012.

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

16. Y. Watanabe, "Online failure prediction in cloud datacenters," *Fujitsu Scientific and Technical Journal*, vol. 50, no. 1, pp. 66–71, 2014.

17. I. Irrera, J. Duraes, H. Madeira, and M. Vieira, "Assessing the impact of virtualization on the generation of failure prediction data," in *Proceedings of the 2013 Sixth Latin-American Symposium on Dependable Computing (LADC 2013)*, pp. 92–97, 2013.

18. I. Irrera and M. Vieira, "A practical approach for generating failure data for assessing and comparing failure prediction algorithms," in *Proceedings of the 2014 IEEE 20th Pacific Rim International Symposium on Dependable Computing (PRDC 2014)*, pp. 86–95, 2014.

19. I. Irrera, J. Duraes, M. Vieira, and H. Madeira, "Towards identifying the best variables for failure prediction using injection of realistic software faults," in *Proceedings of the 2010 IEEE 16th Pacific Rim International Symposium on Dependable Computing (PRDC 2010)*, pp. 3–10, 2010.

20. N. Kikuchi, T. Yoshimura, R. Sakuma, and K. Kono, "Do injected faults cause real failures? a case study of linux," in *Proceedings of the 25th IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW 2014)*, pp. 174–179, 2014.

21. I. Irrera, C. Pereira, and M. Vieira, "The time dimension in predicting failures: A case study," in *Proceedings of the 2013 Sixth Latin-American Symposium on Dependable Computing (LADC 2013)*, pp. 86–91, 2013.

22. R. Vaarandi, "Sec - a lightweight event correlation tool," in *Proceedings of the 2002 IEEE Workshop on IP Operations and Management*, pp. 111–115, IEEE, 2002.

23. 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," tech. rep., The CRISP-DM consortium, Aug. 2000.

24. D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa, "Experimental analysis of binary-level software fault injection in complex software," in *Proceedings of the 9th European Dependable Computing Conference*, pp. 162–172, 2012.

25. R. Natella, D. Cotroneo, J. Duraes, and H. Madeira, "Representativeness analysis of injected software faults in complex software," in *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pp. 437–446, 2010.

26. K. Umadevi and S. Rajakumari, "A review on software fault injection methods and tools," *International Journal of Innovative Research in Computer and Communication Engineering*, vol. 3, no. 3, pp. 1582–1587, 2015.

27. N. Bridge and C. Miller, "Orthogonal defect classification using defect data to improve software development," *Software Quality*, vol. 3, no. 1, pp. 1–8, 1998.

28. E. Martins, C. Rubira, and N. Leme, "Jaca: A reflective fault injection tool based on patterns," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2002)*, pp. 483–487, 2002.

29. B. Sanches, T. Basso, and R. Moraes, "J-SWFIT: A java software fault injection tool," in *Proceedings of the 5th Latin-American Symposium on Dependable Computing (LADC 2011)*, pp. 106–115, Apr. 2011.

30. P. Jordan, D. Van Patten, G. Peterson, and A. Sellers, "Distributed powershell load generator (D-PLG): A new tool for dynamically generating network traffic," in *Proceedings of the 6th International Conference on Simulation and Modeling Methodologies, Technologies, and Applications (SIMULTECH 2016)*, July 2016.

31. S. Makbulolu and G. Geelen, "Capacity planning for active directory domain services," tech. rep., Technical report, Microsoft Corp, 2012.

32. M. Russinovich and D. Solomon, *Windows Internals: Including Windows Server 2008 and Windows Vista*. Microsoft Press, 5th ed., 2009.

33. C. van Rijsbergen, *Information Retrieval*. Newton, MA, USA: Butterworth-Heinemann, 2nd ed., 1979.

34. G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
10–09–2016	Master's Thesis	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

As society becomes more dependent upon computer systems to perform increasingly critical tasks, ensuring those systems do not fail also becomes more important. Many organizations depend heavily on desktop computers for day to day operations. Unfortunately, the software that runs on these computers is still written by humans and as such, is still subject to human error and consequent failure. A natural solution is to use statistical machine learning to predict failure. However, since failure is still a relatively rare event, obtaining labelled training data to train these models is not trivial. This work explores new simulated fault loads with an automated framework to predict failure in the Microsoft enterprise authentication service in an effort to increase up-time in desktop computers and improve mission effectiveness. These new fault loads were successful in creating realistic failure conditions that could be accurately identified by statistical learning models.

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