To appear in *Enterprise Information Systems*
Vol. 00, No. 00, Month 20XX, 1–32

Data Driven Device Failure Prediction

P.L. Jordan[a] G.L. Peterson[a] A.C. Lin[a] M.J. Mendenhall[a] A.J. Sellers[b]

[a] *Air Force Institute of Technology, Dayton, OH, USA*;
[b] *United States Air Force Academy, Colorado Springs, CO, USA*

(*Received 00 Month 20XX; final version received 00 Month 20XX*)

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.

Keywords: online failure prediction; machine learning; fault injection; enterprise architecture

1. 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 enterprise services 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 (Salfner, Lenk, and Malek 2010). Unfortunately, much of this work has gone unused (Irrera, Vieira, and Duraes 2015).

Failure has been defined as the result of a software fault or error (Salfner, Lenk, and Malek 2010). 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 (Schmidt 2016). 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 (Bauer and Adams 2012). 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. Salfner, Lenk, and Malek (2010) 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, Vieira, and Duraes (2015) presented the Adaptive Failure Prediction (AFP) framework and case study to automate the process of dynamically generating failure data and using it to train a predictor after an underlying system change. Unfortunately, the AFP framework required substantial changes and modernizations to be used on modern enterprise systems. Furthermore, the types of failures simulated within the framework were not completely representative of real failures (Kikuchi et al. 2014). This research explores a modernized implementation of the AFP framework with a more represantive fault laod on a Microsoft (MS) Windows Server Domain Controller (DC). This implementation is then validated by running the same experiments on the Apache web server.

2. Overview of 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 section reviews current research regarding OFP and its many approaches to build a foundation for this research. The rest of this section 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. This overview is then followed by a brief description of the AFP framework. This section then concludes with a brief summary.

2.1. *Background*

Salfner, Lenk, and Malek (2010) 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. Proactive Fault Management (PFM)

Salfner, Lenk, and Malek (2010) 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.

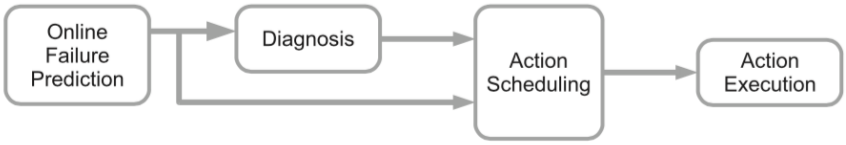


Figure 1. The stages of proactive fault management Salfner, Lenk, and Malek (2010).

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. Finally, in order to avoid failure, a system must execute the scheduled remediation action or let an operator know which actions can be taken in a stage called *action execution.*

2.1.1.1. Faults, Errors, Symptoms, and Failures. This research uses the definitions defined by Avižienis et al. (2004) as interpreted and extended by Salfner, Lenk, and Malek (2010) 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. 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, Lenk, and Malek (2010) introduce a taxonomy of OFP approaches and classify 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:

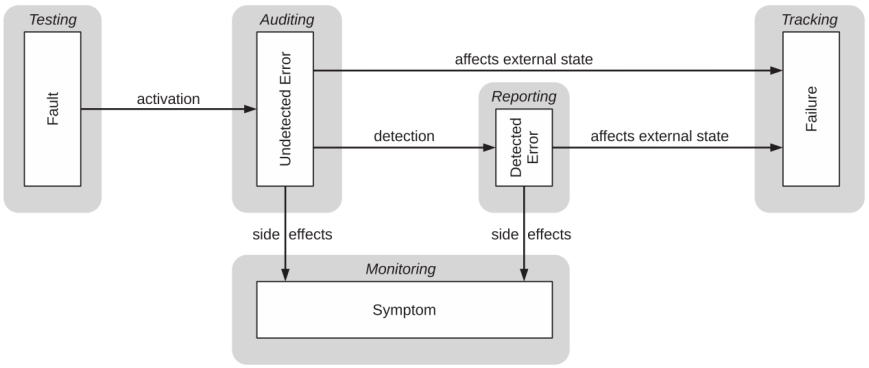


Figure 2. How faults and errors evolve into failure with the associated methods for detection represented by enclosing gray boxes Salfner, Lenk, and Malek (2010).

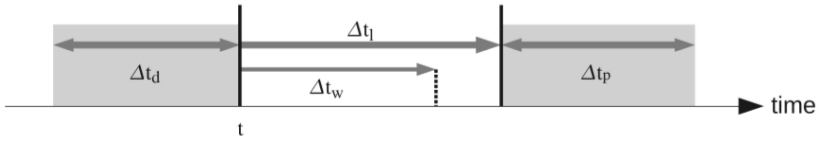


Figure 3. The timeline for OFP Salfner, Lenk, and Malek (2010).

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

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

2.2. *AFP Framework*

The AFP framework by Irrera, et al. Irrera, Vieira, and Duraes (2015) shown in Figure 4, 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. (2013); Irrera and Vieira (2014) to generate failure data by injecting software faults using a tool based on General Software Fault Injection Technique (G-SWFIT) (Duraes and Madeira 2006) in a virtual environment for comparing and automatically re-training predictors. With the introduction of the framework, Irrera, Vieira, and Duraes (2015) report results of a case-study. After

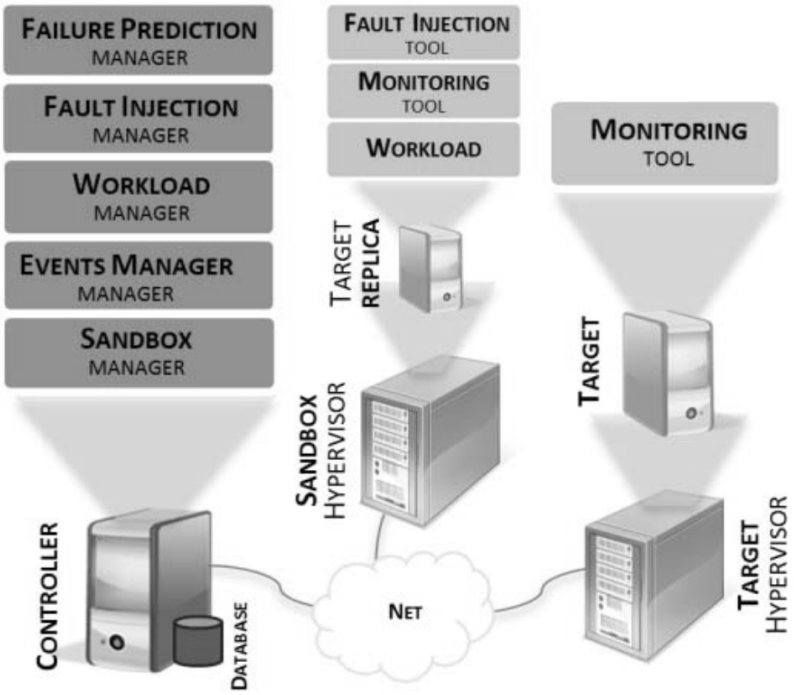


Figure 4. How the AFP framework is implemented Irrera, Vieira, and Duraes (2015).

implementing the AFP framework using a web server and an Support Vector Machine (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 Irrera et al. (2010); Irrera and Vieira (2014) report evidence supporting the claim that simulated failure data is representative of real failure data. Further, the authors suggest that since systems are so frequently updated and failures are in general rare events, real failure data is often not available. Moreover, the literature shows that even if there is a certain type of failure in training data and a predictor can detect and predict that type of error accurately, it will still miss failures not present in the training data. By injecting the types of faults that one can expect, each failure type is represented in the training data.

Irrera, Vieira, and Duraes (2015) 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 MS Windows enterprise network. Furthermore, the fault load previously explored does not completely represent all possible failures (Kikuchi et al. 2014).

2.3. *Summary*

This section covered the definitions, measures of performance, and the AFP framework. 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. Irrera, Vieira, and Duraes (2015) 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 extend the AFP framework and further generalize the approach.

3. Methodology

The purpose of the AFP framework is to automate the generation of realistic labelled failure data for the purposes of automatically training a failure prediction algorithm. The framework breaks down into modules so that it can be more easily adapted for different applications. This section 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 section outlines the implementation and extensions to the AFP framework Irrera, Vieira, and Duraes (2015) 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 Irrera, Vieira, and Duraes (2015) by presenting results after conducting another case study with an MS Windows Server acting as an Active Directory (AD) service with a more representative fault load as well as a new implementation of the 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. Salfner, Lenk, and Malek (2010) 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 SVM. The weighted SVM was used because of it performs well on imbalanced data and it is popular in the OFP community Salfner, Lenk, and Malek (2010). 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. Fulp, Fink, and Haack (2008), on a sliding time window as is done by Irrera, et al. Irrera, Pereira, and Vieira (2013) and Vaarandi Vaarandi (2002).

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 Irrera, Vieira, and Duraes (2015): preparation phase, execution phase, and training phase.

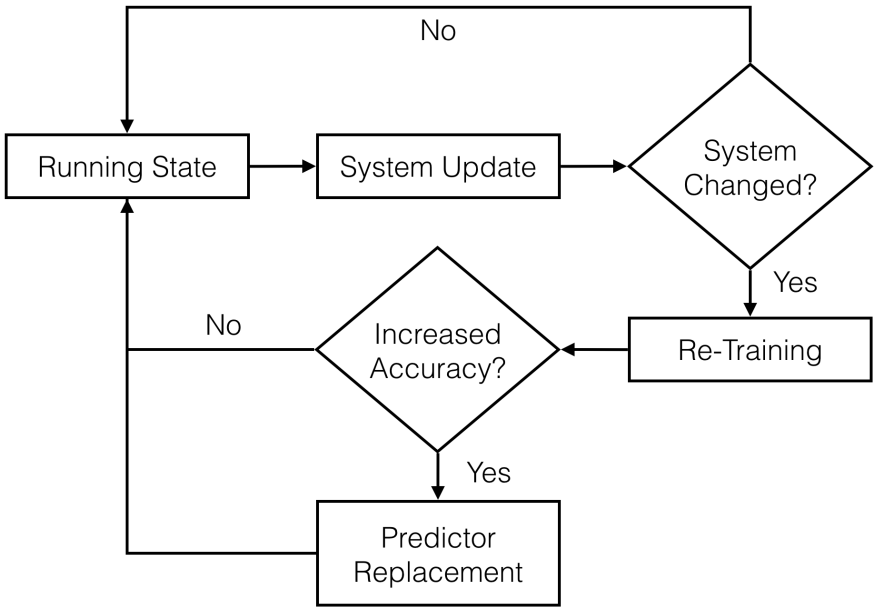


Figure 5. The flow of the major steps involved in the AFP framework execution phase Irrera, Vieira, and Duraes (2015).

3.1.1. Preparation Phase

In this phase the AFP framework is prepared to run for the first time as described in Irrera, Vieira, and Duraes (2015). The Cross Industry Standard Process for Data Mining (CRISP-DM) Chapman et al. (2000) 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 **??**.

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 5. 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.

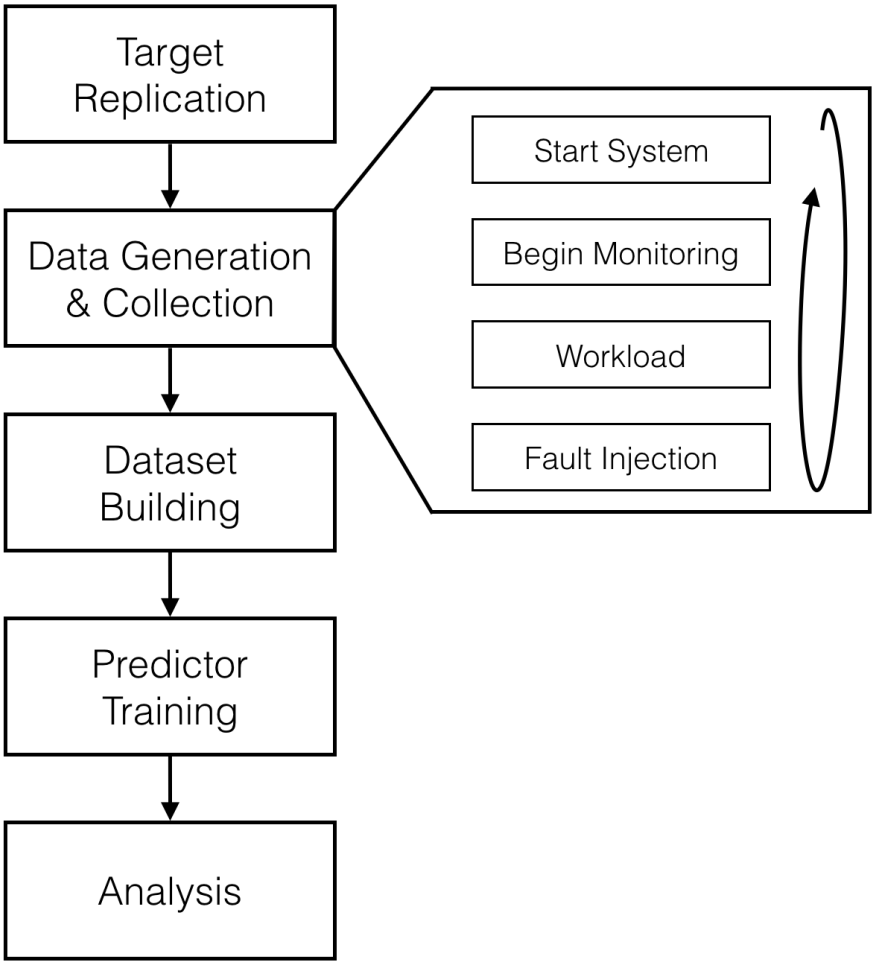


Figure 6. The flow of the major steps involved in the AFP framework training phase Irrera, Vieira, and Duraes (2015).

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 6. 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. Irrera, Vieira, and Duraes (2015) 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 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. Fulp, Fink, and Haack (2008) 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 Vaarandi (2002). 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. Irrera, Vieira, and Duraes (2015). This encoding enables the

[1] https://support.microsoft.com/en-us/kb/977519
[2] http://wiki.rsyslog.com/index.php/Snare_and_rsyslog

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.

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

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

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

used to conduct a $k-1$-fold cross validation training and evaluation 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.

3.1.3.5. 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 Irrera, Vieira, and Duraes (2015) 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 Domeniconi et al. (2002); Fulp, Fink, and Haack (2008); Salfner and Malek (2007); Watanabe (2014). 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 7 with the parts that were modified in this work highlighted.

3.2.2. AFP Modules

Irrera, et al. Irrera, Vieira, and Duraes (2015) 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.

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

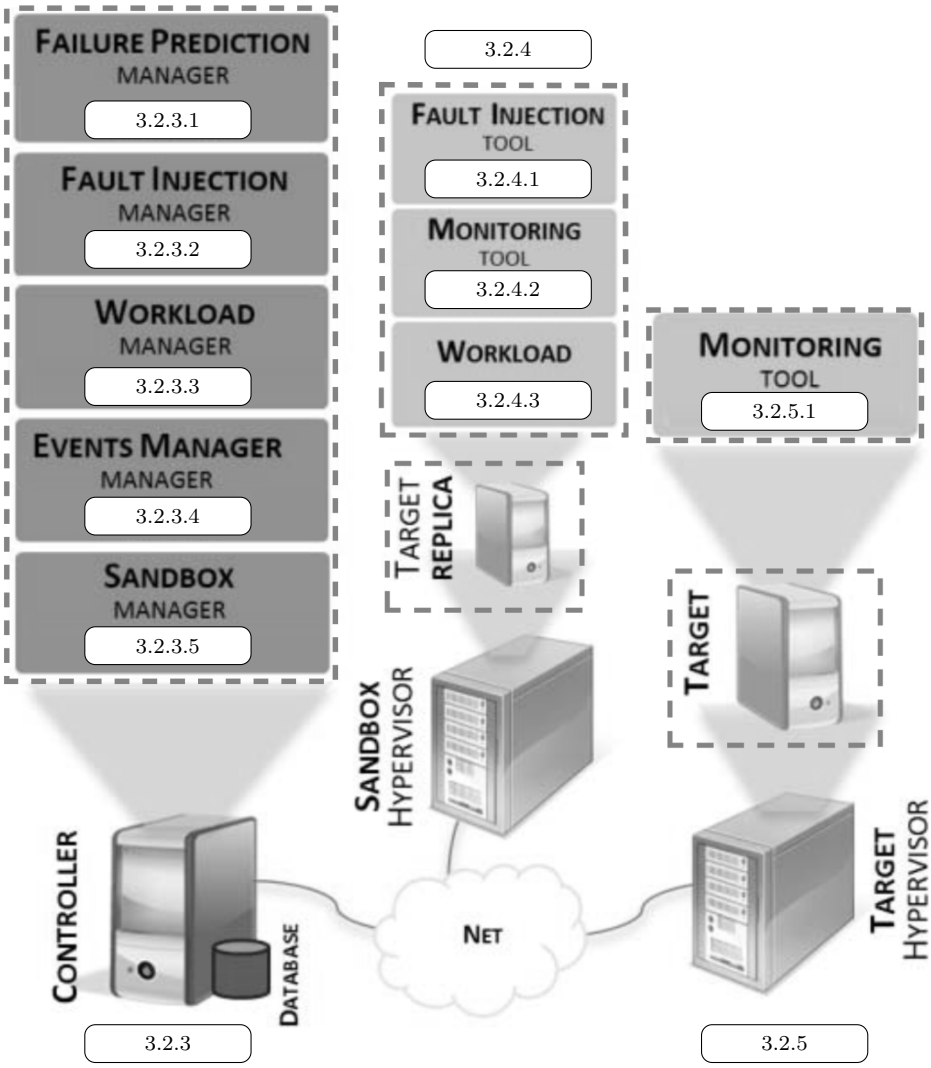


Figure 7. The AFP framework implementation Irrera, Vieira, and Duraes (2015) with modified components highlighted.

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

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.

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. Irrera, Vieira, and Duraes (2015) 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. Duraes and Madeira (2006) 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 Cotroneo et al. (2012); Irrera and Vieira (2014); Natella et al. (2010); Umadevi and Rajakumari (2015).

Recently, studies have questioned the representativeness of the failures generated by G-SWFIT Cotroneo et al. (2012); Kikuchi et al. (2014). 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 Natella et al. (2010). 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 **??**. 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 Duraes and Madeira (2006). For completeness, the source is also included in Appendix **??**.

For this research, the original plan was to use the same fault injection tool used in the original case study by Irrera, et al. Irrera, Vieira, and Duraes (2015). 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

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

Table 6. Fault operators used for fault injection Duraes and Madeira (2006).

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

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. Irrera, Pereira, and Vieira (2013), 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) Bridge and Miller (1998) and are shown in Table 6. The point of this mutation is that failure is ultimately the result of developer error Irrera, Vieira, and Duraes (2015); Salfner, Lenk, and Malek (2010), and that fault injection accurately simulates those errors Duraes and Madeira (2006). Unfortunately, G-SWFIT has only previously been implemented for Java applications Martins, Rubira, and Leme (2002); Sanches, Basso, and Moraes (2011), and the IA32 instruction set Duraes and Madeira (2006); Natella et al. (2010). 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.

Table 7. Function entry/exit patterns in IA32 bytecode Duraes and Madeira (2006).

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	

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

Table 8. Function entry/exit patterns in x86-64 bytecode Duraes and Madeira (2006).

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	

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 Irrera, Vieira, and Duraes (2015). 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 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 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 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 Jordan et al. (2016). 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.

[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`

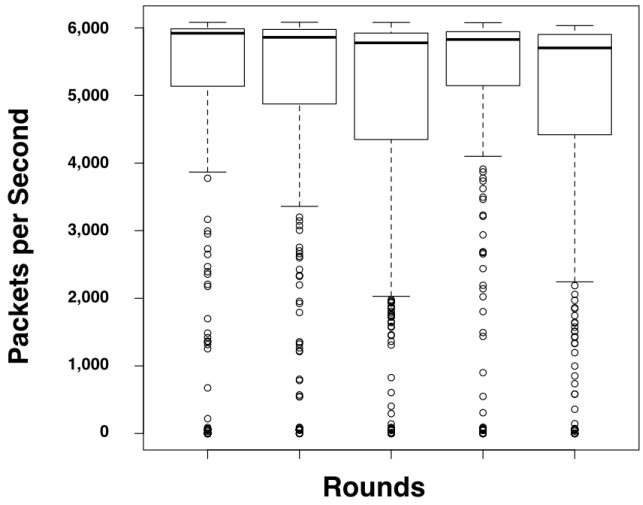


Figure 8. How many packets per second were sent or received by the domain controller across all five rounds of the first test. In each test, we captured approximately 1.8 million packets.

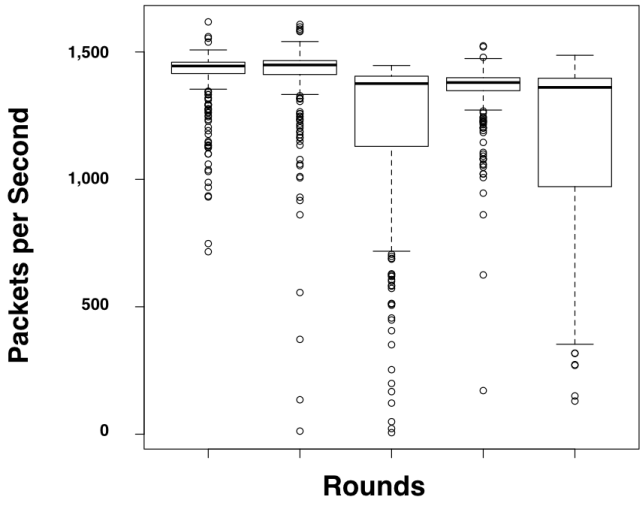


Figure 9. How many packets per second were sent or received by one of the clients across all five rounds of the first test.

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 Makbulolu and Geelen (2012) 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 8, 9, 10.

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

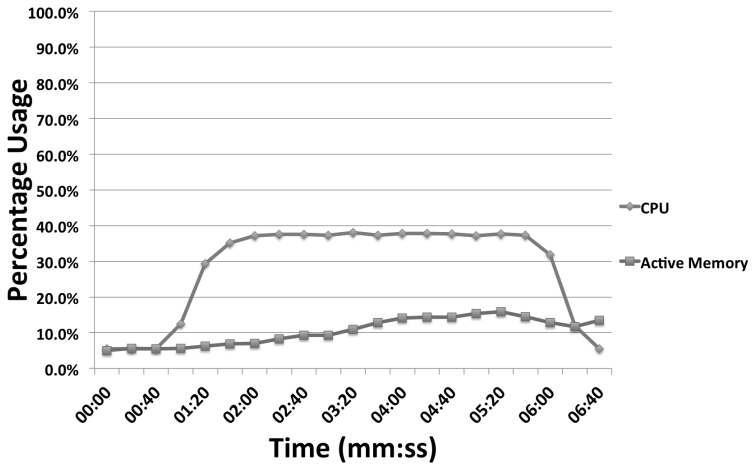


Figure 10. Domain controller CPU and memory utilization during the first test.

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 Makbulolu and Geelen (2012), 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. Irrera, Vieira, and Duraes (2015) 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.

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.

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

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. Irrera et al. (2013); Irrera, Vieira, and Duraes (2015) 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 Irrera et al. (2013).

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. Duraes and Madeira (2006) 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. Irrera, Vieira, and Duraes (2015) 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 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.

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

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 Cotroneo et al. (2012); Irrera, Vieira, and Duraes (2015); Kikuchi et al. (2014).

Irrera, et al. Irrera, Vieira, and Duraes (2015) 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.

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 Kikuchi et al. (2014), 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 section 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 **??**.

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

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 **??**.

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 Russinovich and Solomon (2009). 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 **??**.

3.3.4. Reported Errors

Finally, this research focusses on reported errors instead of system information using the *Logman* tool in the original study Irrera, Vieira, and Duraes (2015). As pointed out by Salfner, et al. Salfner, Lenk, and Malek (2010), 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. Russinovich and Solomon (2009), 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.

4. Experimental Results and Analysis

This section reports results after conducting the experiments laid out in Chapter 3. 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 section concludes with a short summary.

4.1. *Performance Measures*

This section reviews the performance measures used in this section 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 OFP methods written by Salfner et al. Salfner, Lenk, and Malek (2010).

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 van Rijsbergen (1979). 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 11.

Another popular visualization is the 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 12.

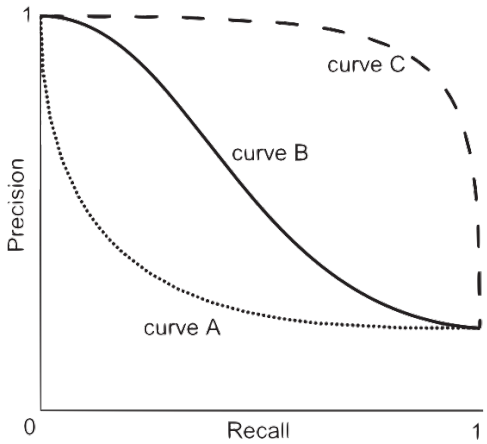


Figure 11. Sample precision/recall curves Salfner, Lenk, and Malek (2010). Curve A represents a poorly performing predictor, curve B an average predictor, and curve C an exceptional predictor.

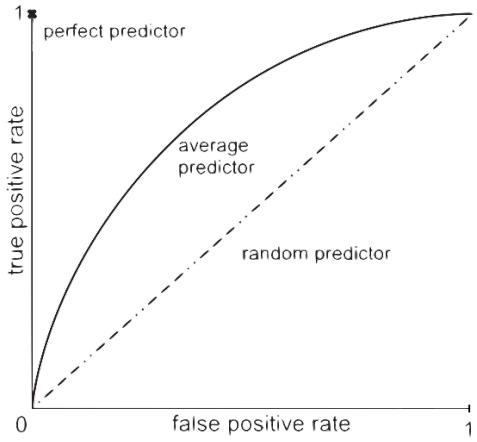


Figure 12. ROC plots of perfect, average, and random predictors Salfner, Lenk, and Malek (2010).

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.

The results of the experiments conducted in this research report all of the above de-

scribed measures of performance in the next section.

4.2. *Results*

The experiments designed in Chapter 3 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 section 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 section is concluded with a summary of these results.

4.2.1. MS DC

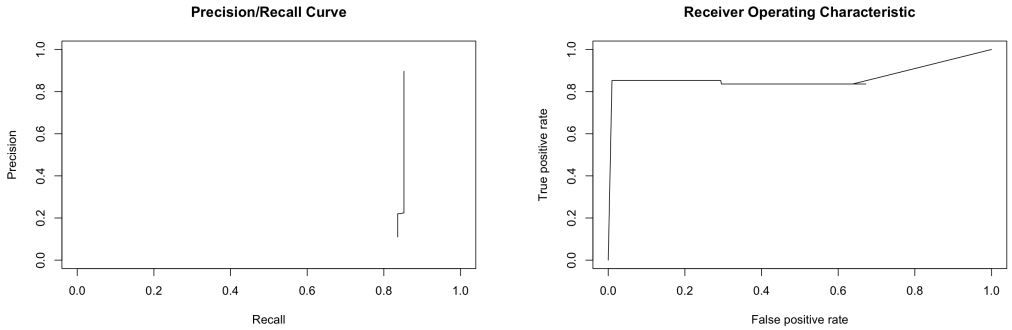
Fault Injection. This fault load was effective at creating a failure, but unfortunately, each failure observed occurred immediately after introducing the fault. Because there was no delay between injection and failure, there did not exist any indicators of failure. Consequently, machine learning cannot help in this situation. According to Russinovich, et al. Russinovich and Solomon (2009) 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.

Under-Resourced 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 Makbulolu and Geelen (2012) 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.

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. James et al. (2014), 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 SVM, and boosted decision trees using the multinomial distribution.

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 classifi-



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

Figure 13. 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

cation 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.

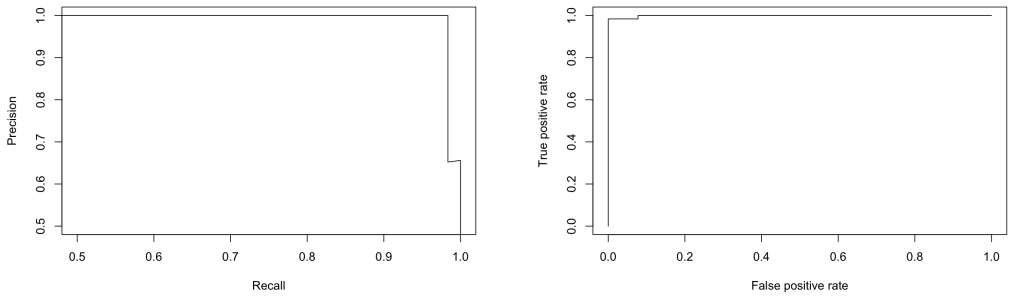
Initial test performance was poor so the test data was then evaluated in sequential order using a threshold. After two sequential windows were predicted as failure-prone, the next w windows were also predicted as failure-prone, where $w = window\ size - threshold\ size$. For threshold = 2, the resulting confusion matrix for the optimal F-Measure, the ROC curve, and the precision/recall curve are shown in Table 9, and Figure 13 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.

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 14. 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 **??**. The precision/recall and ROC curves on data generated after the software update using the old model are shown in Figure 15. The confusion matrix at the optimal threshold for F-measure is

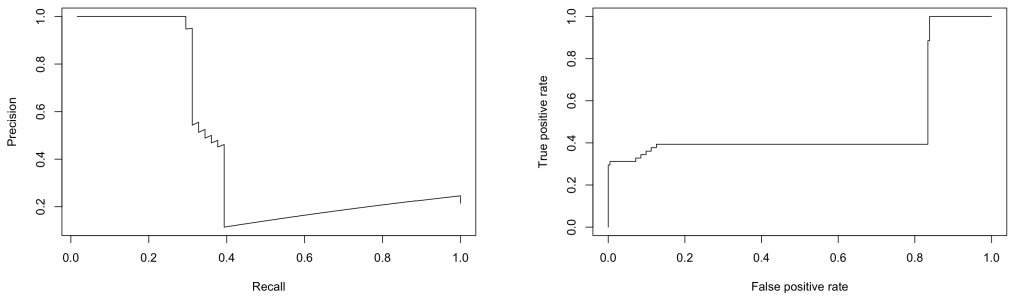


(a) Precision/Recall Curve. (b) ROC Curve (AUC = 0.9984).

Figure 14. 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



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

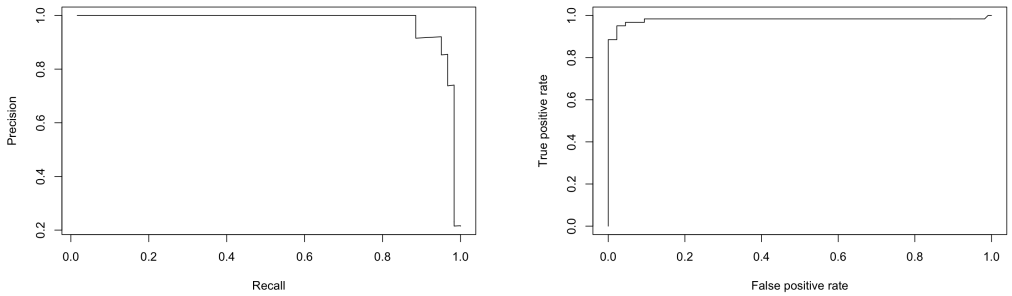
Figure 15. Performance of the boosting prediction method trained on failure data created before the software update obtained by consuming all available memory until target application fails.

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

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 16 and the confusion matrix at the optimal threshold for F-measure is shown in Table 12.



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

Figure 16. 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

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 Russinovich and Solomon (2009). To test this fault load, the 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 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.

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.

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.

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.

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.

5. Conclusion and Future Work

This section 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 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 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 retraining. 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 Irrera, Vieira, and Duraes (2015). 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 MS DCs. 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 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: 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.

References

Avižienis, A., J. Laprie, B. Randell, and C. Landwehr. 2004. "Basic Concepts and Taxonomy of Dependable and Secure Computing." *IEEE Transactions on Dependable and Secure Computing* 1 (1): 11–33.

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

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

Chapman, P., J. Clinton, R. Kerber, T. Khabaza, T. Reinartz, C. Shearer, and R. Wirth. 2000. *CRISP-DM 1.0 Step-by-Step Data Mining Guide*. Tech. rep.. The CRISP-DM consortium. `http://www.crisp-dm.org/CRISPWP-0800.pdf`.

Cotroneo, D., A. Lanzaro, R. Natella, and R. Barbosa. 2012. "Experimental Analysis of Binary-Level Software Fault Injection in Complex Software." In *Proceedings of the 9th European Dependable Computing Conference,* 162–172. `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6214771`.

Domeniconi, C., C. Perng, R. Vilalta, and S. Ma. 2002. "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,* 125–137.

Duraes, J., and H. Madeira. 2006. "Emulation of Software Faults: A Field Data Study and a Practical Approach." *IEEE Transactions on Software Engineering* 32 (11): 849–867.

Fulp, E., G. Fink, and J. Haack. 2008. "Predicting Computer System Failures Using Support Vector Machines." In *Proceedings of the 1st USENIX Conference on Analysis of System Logs,* `http://static.usenix.org/event/wasl/tech/full{_}papers/fulp/fulp{_}html/wasl08f.html`.

Irrera, I., J. Duraes, H. Madeira, and M. Vieira. 2013. "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),* 92–97.

Irrera, I., J. Duraes, M. Vieira, and H. Madeira. 2010. "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),* 3–10. `http://dl.acm.org/citation.cfm?id=1935935.1935974`.

Irrera, I., C. Pereira, and M. Vieira. 2013. "The Time Dimension in Predicting Failures: A Case Study." In *Proceedings of the 2013 Sixth Latin-American Symposium on Dependable Computing (LADC 2013),* 86–91. `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6542609`.

Irrera, I., and M. Vieira. 2014. "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),* 86–95.

http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6974775.

Irrera, I., M. Vieira, and J. Duraes. 2015. "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),* 142–149. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7027425.

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

Jordan, P., D. Van Patten, G. Peterson, and A. Sellers. 2016. "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),* Jul..

Kikuchi, N., T. Yoshimura, R. Sakuma, and K. Kono. 2014. "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),* 174–179.

Makbulolu, S., and G. Geelen. 2012. *Capacity Planning for Active Directory Domain Services*. Tech. rep.. Technical report, Microsoft Corp.

Martins, E., C. Rubira, and N. Leme. 2002. "Jaca: A Reflective Fault Injection Tool Based on Patterns." In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2002),* 483–487.

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

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

Salfner, F., M. Lenk, and M. Malek. 2010. "A Survey of Online Failure Prediction Methods." *ACM Computing Surveys (CSUR)* 42 (3).

Salfner, F., and M. Malek. 2007. "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),* 161–174. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4365693.

Sanches, B., T. Basso, and R. Moraes. 2011. "J-SWFIT: A Java Software Fault Injection Tool." In *Proceedings of the 5th Latin-American Symposium on Dependable Computing (LADC 2011),* 106–115. Apr..

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

Umadevi, K., and S. Rajakumari. 2015. "A Review on Software Fault Injection Methods and Tools." *International Journal of Innovative Research in Computer and Communication Engineering* 3 (3): 1582–1587.

Vaarandi, R. 2002. "SEC - A Lightweight Event Correlation Tool." In *Proceedings of the 2002 IEEE Workshop on IP Operations and Management,* 111–115. IEEE.

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

Watanabe, Y. 2014. "Online Failure Prediction in Cloud Datacenters." *Fujitsu Scientific and Technical Journal* 50 (1): 66–71.