To appear in *Enterprise Information Systems*
Vol. 00, No. 00, Month 20XX, 1–16

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 presents new simulated fault loads with an automated framework to predict failure in the Microsoft enterprise authentication service and Apache web server in an effort to increase up-time and improve mission effectiveness. These new fault loads were successful in creating realistic failure conditions that are accurately identified by statistical learning models.

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

1. Introduction

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 with potentially catastrophic consequences.

Being able to predict pending failure in those systems can offer tremendous, and potentially life-saving applications. While being able to accurately predict failure has unfortunately not been proven possible, there has been work over the past several decades attempting to make 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).

In this case, failure is 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, the trending solution to this problem is configuring massively redundant systems that can withstand failure (Bauer and Adams 2012). While effective, redundant systems incur a high cost and enterprise design may limit their implementation. Consequently, this research focuses on an area of

The views expressed herein are solely those of the authors and do not reflect the official policy or position of the U.S. Air Force, the Department of Defense, or the U.S. Government.

reliable computing called 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).

Training a prediction model requires training data, which is limited due to the rarity of failure events and the complex and manual training process. To address this problem, Irrera, Vieira, and Duraes (2015) presented the Adaptive Failure Prediction (AFP) framework that automates the process of dynamically generating failure data and using it to train a predictor after an underlying system change. Unfortunately, the types of failures simulated within the framework were not completely representative of failures which might actually occur (Kikuchi et al. 2014).

This research presents an AFP framework with a more representative fault load including focused software fault injection, third party memory leaks, third party Central Processing Unit (CPU) over-utilization, and heap-space corruption. The implementation is then validated on a Microsoft (MS) Windows Server Domain Controller (DC), and on an Apache web server. Results showed that targeted fault loads could create realistic failure conditions on Windows Server 2008 and that software fault injection did not. Furthermore, these failures were identifiable by Support Vector Machine (SVM) and boosted decision tree statistical learning models with an average area under the Receiver Operating Characteristic (ROC) curve of 0.98.

2. Overview of Online Failure Prediction (OFP)

OFP is the act of evaluating a running system in real time to make a prediction about whether a failure in a future state is imminent (Salfner, Lenk, and Malek 2010). Traditionally, failure is predicted using statistical information about past failures offline before a system is fielded. Unfortunately, the complexity of modern computer systems and the infinite number of ways in which they can be configured, limits the usefulness of offline analysis.

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. *Proactive Fault Management (PFM)*

Salfner, Lenk, and Malek (2010) define Proactive Fault Management (PFM) as the process by which faults are handled in a proactive way, analogous with *fault tolerance* and consisting of four steps: OFP, diagnosis, action scheduling, and action execution. 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 demonstrates the timeline associated with OFP. The parameters used by the community to define a predictor are as follows:

- Present Time: t
- Data Window: Δt_d, represents the time window of data used for a predictor to make its assessment.
- Lead Time: Δt_l, represents the time between when failure is predicted and when

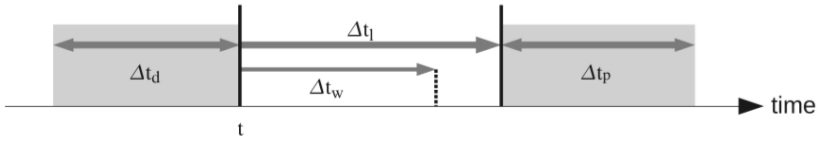


Figure 1. The timeline for OFP (Salfner, Lenk, and Malek 2010).

that failure will occur.

- 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.

2.2. *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. Second, *undetected errors* are errors that have not been identified by an error detector. Undetected errors are things like memory leaks. Finally, 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.

2.3. *Adaptive Failure Prediction (AFP) Framework*

The AFP framework by Irrera, Vieira, and Duraes (2015) presents a new approach to maintaining the efficacy of failure predictors given underlying system changes.

The framework generates 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. After implementing the AFP framework using a web server and an SVM predictor, they report that their findings demonstrate the 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 faults, there is an increased likelihood potential failure types are represented in the training data.

Table 1. 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

Irrera, Vieira, and Duraes (2015) reported good results and concluded that the AFP framework is an effective tool. Unfortunately, the fault load used does not completely represent all possible failures (Kikuchi et al. 2014).

3. Extended Adaptive Failure Prediction (AFP)

The extended AFP framework is an automated framework for generating realistic failure data for the purpose of training statistical failure prediction models. To do this requires representative fault loads, a workload generator, and a modern fault injection tool. Figure 2 shows the original AFP architecture with the modules updated in this work highlighted.

This section outlines the implementation and extensions to the original AFP framework (Irrera, Vieira, and Duraes 2015). The AFP framework was originally tested on a single system running Windows XP, which has been deprecated. Consequently, the extended AFP framework presented has been updated to run on the Windows Server 2008 operating system and tested against the DC services and an Apache web server.

3.1. *The Base Adaptive Failure Prediction (AFP)*

3.1.1. Adaptive Failure Prediction (AFP) Framework Implementation

This experiment replicated the experiment by 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, this research explored three additional fault loads: 1) third party memory leaks, 2) third party CPU over-utilization, and 3) heap-space corruption. In addition to using the SVM model, boosted decision trees were evaluated. Finally, in addition to the Apache webserver, the primary target was the MS Windows Server running Active Directory (AD) Domain Services. The purpose of Apache web server was to validate the approach and additional fault loads.

3.1.2. Adaptive Failure Prediction (AFP) Modules

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 1, and 2.

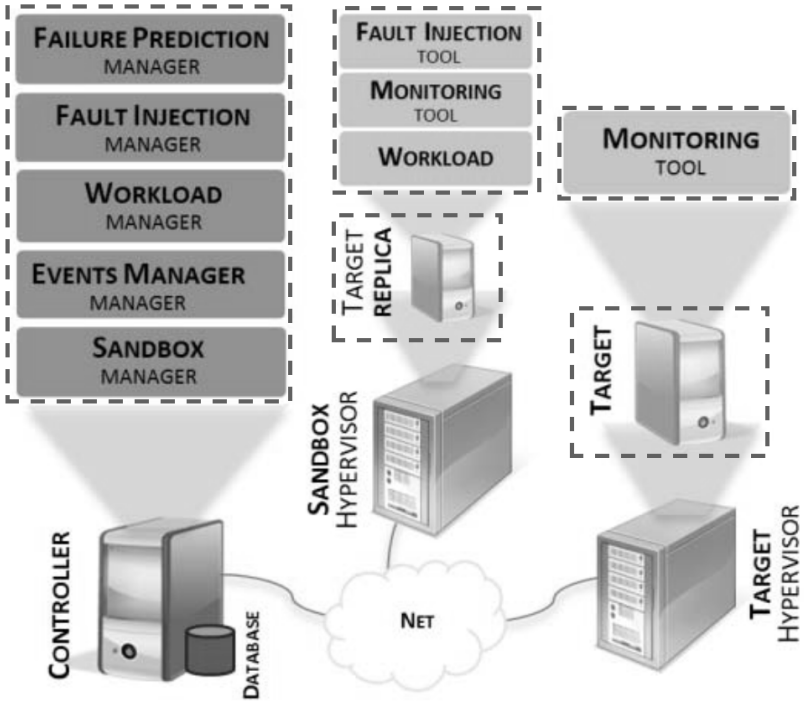


Figure 2. The AFP framework implementation (Irrera, Vieira, and Duraes 2015).

Table 2. Hypervisor 2 configuration (controller).

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

3.1.3. Controller Hypervisor

The controller responsibilities in this experiment were split between two systems on a single hypervisor shown in Table 2. 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.

3.1.3.1. 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 this experiment, the statistical models were trained using the popular statistical learning software suite *R*.

3.1.3.2. Fault Injection. This module is responsible for managing the fault load used to create realistic failure data. 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 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).

Unfortunately, previous G-SWFIT tools were incapable of injecting faults into elevated modern windows processes. Older tools were written for Java or x86 architectures (Duraes and Madeira 2006; Martins, Rubira, and Leme 2002; Natella et al. 2010; Sanches, Basso, and Moraes 2011). For this reason, this work introduces a modernized fault injection tool capable of injecting into x86-64 elevated system process (such as the 'lsass.exe' process). This tool is called Windows Software Fault Injection Tool (W-SWFIT) and its source code has been published as open source on Github[2] so that others may use it for any of the reasons cited in the original G-SWFIT paper (Duraes and Madeira 2006).

Because of the concerns with fault injection (Cotroneo et al. 2012; Kikuchi et al. 2014; Natella et al. 2010), this research generated failure data using fault injection in conjunction with three new fault loads, covered in Section 3.2.

3.1.3.3. 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. The purpose of this module is to accelerate the evolution of a fault 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 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) (Jordan et al. 2016). Realistic workload is critical in the implementation of the AFP framework. Consequently, D-PLG has been designed and shown to provide a realistic and sufficient workload for implementing the AFP framework for a MS DC. The client portion of D-PLG was used installed on five client machines and used as the sandbox workload generator as discussed in Section 3.1.4.3.

3.1.3.4. 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, 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.

3.1.3.5. 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. (2013); Irrera, Vieira, and Duraes (2015) point out, it is typically inappropriate to inject faults and cause failures in production systems, so a

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

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.

3.1.4. 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 1. The following sections outline each module within this module.

3.1.4.1. 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 3.1.3.2, W-SWFIT has been developed to serve this purpose and implements the G-SWFIT technique developed by 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. Since many of the critical functions performed by the AD services processes are performed in the 'ntdsa.dll'[3] library loaded by the 'lsass.exe' process, it was the focus of fault injection.

As mentioned, this work, introduces new fault loads. These new loads are discussed in Section 3.2.

3.1.4.2. Monitoring. The purpose of this module is to capture indicators of pending failure at the target host level so that it may be used to train a statistical prediction model. 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[4] 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.

3.1.4.3. Sandbox Workload. The purpose of this module is to create realistic work for the target application to do before faults are injected. In this experiment, D-PLG was used as the work load generator for both the DC and web requests. This module was implemented using the client portion of D-PLG installed on five workstations and managed by the central workload manager as discussed in Section 3.1.3.3.

[3] `https://technet.microsoft.com/en-us/library/cc780455(v=ws.10).aspx`
[4] `http://www.solarwinds.com/free-tools/event-log-forwarder-for-windows/`

3.1.5. Target Hypervisor

The target hypervisor was constructed as a clone of the sandbox hypervisor shown in Table 1. The following section outlines the monitoring tool installed on both the DC and web server on this hypervisor.

3.1.5.1. Monitoring. The target monitoring module was implemented exactly as the sandbox monitoring module was, using the *Solar Winds* syslog forwarding tool. The only modification worth noting here is that to ensure the messages were uniquely identifiable by the controller, the hostname of the target machine was changed after cloning.

3.2. *Fault Load Generators*

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. Finally, an outline of the changes in how data was collected from the target is presented.

3.2.1. 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: 1) by downsizing the number of virtual CPUs available to the target VM, and 2) by introducing a third-party application that ran at 100% CPU.

3.2.2. 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 assigned work.

To test this fault load, this experiment created both conditions outlined above by reducing the amount of memory available to the target VM, and by running a third-party application with an intentional memory leak on the target system.

3.2.3. Heap Space Corruption

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, 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.

3.2.4. 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, 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 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.

4. Experimental Results and Analysis

To test the extended AFP framework, failure data was generated before a series of major software updates using software fault injection, under-resourced CPU, under-resourced memory, and heap space corruption, on two Windows Server 2008 machines: the DC, and the Apache web server. The failure data was used to train two statistical prediction models: an SVM classifier, and a boosted decision tree. Following the software updates, more failure data was generated and the old statistical models were used to predict failure in the new data. Finally, new statistical models were trained using the new data. To compare each fault load both before and after the software updates, performance was measured using the Area Under the Curve (AUC) and F-Measure.

In general, the AFP framework works by virtually cloning the target production system after it has determined that system has changed. In this case, this determination is made after several important software updates. The framework then generates realistic work for the cloned service to perform, which accelerates the activation of an injected fault. When the cloned system is sufficiently loaded, faults are injected until failure occurs. Once failure has occurred, the recorded data is used to train a statistical learning model. The new model then replaces the existing model if it performs better.

Since log messages were used to train the statistical model, they needed to be transformed to numerical data. During execution, event messages were stored in a flat file on the Ubuntu machine by the syslog server daemon in the *Snare*[5] MSWinEventLog format. The first element in each message 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 3. Of these features, Criticality, EventLogSource, EventID, SourceName, and CategoryString were selected for further encoding.

Events were filtered by EventID as is done by Fulp, Fink, and Haack (2008) to reduce the noise generated by successful login attempts. Log messages with IDs shown in Table 4 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) (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_l was labelled as failure prone (Irrera, Vieira, and Duraes 2015). This encoding enables the use of classification

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

Table 3. 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 4. Microsoft log message IDs[6].

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 5. 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

algorithms in the training phase. An example of the final encoding is shown in Table 5.

Feature reduction was performed for both learning algorithms on a sliding time window (Fulp, Fink, and Haack 2008; Irrera, Pereira, and Vieira 2013; Vaarandi 2002). This transformed data was then used to train SVM and boosted decision tree models using cross validation on 5 recorded failure runs for each fault load for both systems before and after the software updates. Upon completion of the data generation and model training, several performance measures were calculated on held out test data.

4.1. *Microsoft (MS) Domain Controller (DC) Results*

The MS DC was configured in the virtual environment to host a $30,000$ user database and perform Domain Name System (DNS) and authentication for all workstations. The target of the fault injection was the *lsass.exe* process, and specifically the *ntdsa.dll* library. This library is responsible for processing authentication requests and handles interaction with the user database.

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 ($\Delta t_l \approx 0$), there did not exist any indicators

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

Table 6. SVM test data confusion matrix.

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

of failure. Consequently, machine learning cannot help in this situation. According to 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. While this fault load resulted in authentication requests that took longer, this fault never led to 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 for a domain controller (Makbulolu and Geelen 2012). 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. For this reason, this experiment explores the use of two machine learning models: the weighted SVM, and boosted decision trees using the multinomial distribution.

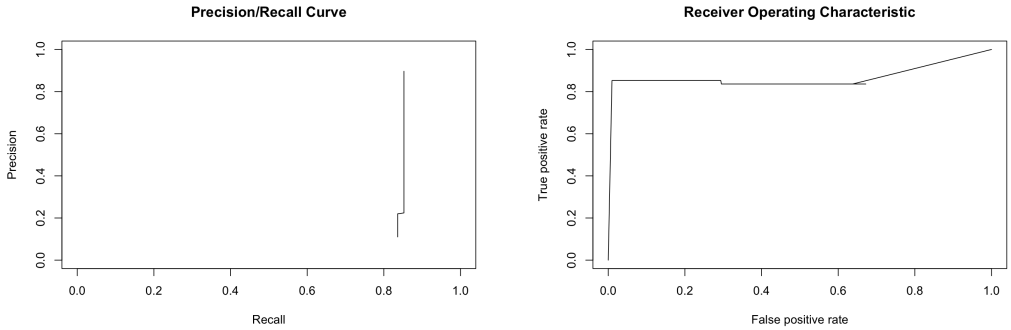
Weighted SVM. For this prediction method, the *e1071* package in R was used to train an SVM. The *tune* function was used to run a 5-fold cross-validation a total of 48 times to select the best performing parameters (gamma, cost, and degree polynomial) using: four kernels, four sliding data/prediction 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 best performing parameters were the Radial kernel with $\gamma = 0.1$, $c = 1$, time window = 60 seconds, and the split of data = 4 of the observed failures used for training, with the remaining used for test.

Test data is evaluated in temporal order over two data windows. The resulting precision/recall and ROC curves are shown in Figure 3. Table 6 shows the confusion matrix on test data created before software updates on threshold with highest F-Measure = 0.8739.

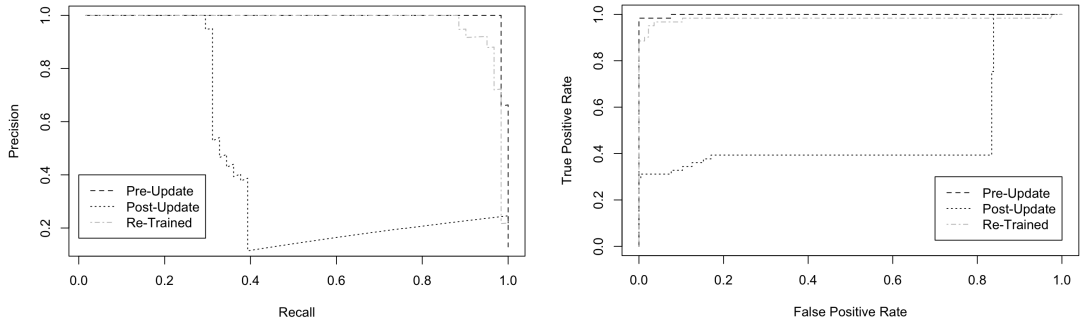
After the software update, the same model was used on a new set of generated failures. The old model did not accurately classify a single failure prone time window. A new model was then trained with the newly generated failure data. Unfortunately, after this software update, with all other things held constant, the weighted SVM model was unable to achieve the same level of performance as before resulting in a maximum F-Measure of 0.4380.

Boosted Decision Trees. For this prediction model, the *gbm* package in R was used to train a boosted decision tree. Cross-validation was used to select $\lambda = 0.03$, the interaction



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

Figure 3. Test data performance of the SVM prediction method on failure data obtained by consuming all available memory until target application fails.



(a) Precision/Recall Curves. (b) ROC Curves.

Figure 4. Performance of the boosting prediction method on data generated by consuming all available memory until target application fails.

Table 7. Boosting pre-update test data confusion matrix.

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

Table 8. Boosting post-update old model test data confusion matrix.

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

depth of = 4, and the number of trees = 1000. The multinomial distribution was used to perform classification.

The precision/recall, and ROC curves on a sixty second data/prediction window are shown in Figure 4. The confusion matrix at the optimal threshold for F-measure is shown in Table 7.

After the software update, the same prediction model was used new set of generated failures. The precision/recall and ROC curves on data generated after the software update using the old model are shown in Figure 4. The confusion matrix at the optimal threshold for F-measure is shown in Table 7

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 4 and the confusion matrix at the optimal threshold for F-measure is shown in

Table 9. Boosting post-update new model test data confusion matrix.

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

Table 9.

In summary, before the software update, the boosted decision tree performed using test data with an AUC of 0.9984. After the software update, the test AUC dropped to 0.4854 but was retrained to achieve an AUC of 0.9801.

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.

4.1.1. 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 *2.2.31 x64* to version *2.4.20 x64*. 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.

4.1.2. Summary

In summary, the memory leak was the only fault load usable for training a statistical model to predict failure based only on reported errors. 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 re-trainable after the software update whereas the SVM was not. This suggests that both models should be used to ensure the AFP framework maintains at least one useful predictor and is adaptable to underlying system changes.

Perhaps most interestingly, fault injection as was used in the original AFP framework implementation, had two extreme outcomes: 1) no failure, or 2) immediate failure. 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 injecting faults. Since during the golden runs, the fault injection tool never wrote to another processes memory, it is possible that a predictor could identify these operations if system health statistics are used as features instead of reported errors. Furthermore, even only using the Operator for Missing Function Call (OMFC), there were still thousands of injection points in the Windows Server 2008 operating system. Identifying the handful that may activate in a realistic way without crashing the target service immediately is not trivial. Clearly more work must be done to validate using fault injection alone in the AFP framework.

5. Conclusion and Future Work

The presented AFP framework extends the current AFP framework with additional fault loads which 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. 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 extended AFP framework is implemented on MS Windows 2008, all fault loads should be used in the execution and training phases.

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.

Acknowledgements

This work was supported by the the U.S. National Security Agency, National Information Assurance Education and Training Program (Alice Shafer and Glenn Ellisonn, Program Managers).

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.

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.

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 Workshop on Analysis of System Logs (WASL 2008),* 1–8.

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.

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.

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.

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.

Jordan, P., C. 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),* 195–202. 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.

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.

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