Anomaly-based Fault Detection in Pervasive Computing System

Byoung Uk Kim, Youssif Al-Nashif, Samer Fayssal, Salim Hariri
NSF Center for Autonomic Computing
The University of Arizona
Tucson, AZ 85721 USA
byoung, alnashif, sfayssal, hariri@ece.arizona.edu

ABSTRACT
The increased complexity of hardware and software resources and the asynchronous interaction among components (such as servers, end devices, network, services and software) make fault detection and recovery very challenging. In this paper, we present innovative concepts for fault detection, root cause analysis and self-healing architectures analyzing the duration of pattern transition sequences during an execution window. In this approach, all interactions among components of Pervasive Computing Systems (PCS) are monitored and analyzed. We use three-dimensional array of features to capture spatial and temporal variability to be used by an anomaly analysis engine to immediately generate an alert when abnormal behavior pattern is captured indicating some kind of software or hardware failure. The main contributions of this paper include the innovative analysis methodology and feature selection to detect and identify anomalous behavior. Evaluating the effectiveness of this approach to detect faults injected asynchronously shows a detection rate of above 99.9% with no occurrences of false alarms for a wide range of scenarios.

Categories and Subject Descriptors
C.4 [Performance of Systems]: Fault Tolerance

General Terms

Keywords
Abnormality detection, faults, interaction analysis, pattern profiling, performance objectives

1. INTRODUCTION
Fault detection, analysis and recovery in PCS is a challenging research problem due to the exponential growth in scale and complexity of resources (e.g., computers, PDAs, and smart phones), applications (e.g., Microsoft Outlook and Internet Explorer), and services (e.g., authentication, email and maps).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

One pervasive service example currently deployed is healthcare[1], where the system stores daily records of patients by monitoring and automatically providing orders for regular assistance, as well as possibly for urgent support. Disruption of this service for any reason such as faulty device or network can result in dangerous and possibly loss of life. For example, there was a massive computer outage resulting in seventy two primary care and eight acute hospital trusts to be out of commission. Therefore, PCS, especially those deployed in healthcare, police, safety or critical environmental situations are expected to continuously operate despite the presence of faults.

The fault detection and analysis approach presented in this paper is based on hardware/software fault tolerance [2] [3] [4] and data mining techniques including regression trees [5] [6], neural networks [7] [8], multivariate linear regression [9] [10] [11], fuzzy classification [12], logistic regression [13] [14] [15], classification tree [16], naïve bayes [17], and sequential minimal optimization [18]. In most fault-tolerant systems runtime properties are traced to extract the state of the system using the above mentioned methods and infer dependencies to identify possible causes of failures. This approach differs considerably from our approach as it relies on extracting dependency from runtime information. Our approach is a typical black box approach that first does not rely on any dependency information and second is able to achieve much higher detection rate along with lower false alarm rate.

In this paper[1], we propose innovative concepts for self-healing architecture to detect software and hardware faults and identify their source. Our anomaly-based approach includes online monitoring mechanism collecting significant behavior characteristics of application such as system calls and process information and behavior interactions among system state components such as CPU, memory, I/O, and network interface in real time among all components of pervasive system. In our pattern based analysis approach, we view application operation as being a finite state machine through which we can flag illegal sequences of state transitions as being anomalous. So, we focus on analyzing pattern transition sequences of length n during a window interval by recording and tracing these runtime properties which are stored in our three-dimensional array of features.

---

1 The research presented in this paper is supported in part by National Science Foundation via grants numbers CNS-0615323 and IIP-0758579, and by Electronic Warfare Associates, Inc. via the grant number ewagi-07-LS-0002, and it is conducted as part of the NSF Center for Autonomic Computing at University of Arizona.
related to as AppFlow, to capture both spatial and temporal variability. We have implemented an anomaly-based fault detection engine and used it to detect faults in a typical multi-tier web-based e-commerce pervasive computing environment that implements e-commerce transactions based on the TPC-W benchmark [19].

The rest of this paper is organized as follows. In section 2, we review related work and classify fault detection techniques. Section 3 presents self-healing system architecture. Section 4 illustrates anomaly analysis methodology and its functional components. Experimental environment, data source, fault classes and results that evaluate the effectiveness and performance of our approach are discussed in section 5. Finally, in section 6, we conclude the paper and discuss future research.

2. RELATED WORKS
Fault detection and analysis has always been an active research area in distributed systems and applications. In this section, we classify fault detection techniques based on criteria such as hardware and software techniques and the type of detection schemes such as statistical methods, distance-based, model-based and profiling methods.

2.1 Hardware/Software Based
Reinhardt and Mukherjee [2] proposed Simultaneous and Redundant Threading (SRT) to provide transient fault coverage taking advantage of the multiple hardware contexts of Simultaneous Multithreading (SMT). This scheme achieves good performance through using active scheduling of its hardware components among the redundant replicas and reduces the overhead of validation by eliminating cache misses. Ray et al., [20] proposed modern superscalar out-of-order data path by modifying a superscalar processor’s micro-architectural components and validating the redundant outcomes of actively duplicated threads of execution. Their fault recovery plan uses the branch-rewind mechanism to restart at a place where error happened. Commercial fault-tolerant systems combine several techniques such as error correcting codes, parity bits and replicated hardware. For example, HP’s (originally Compaq and before that Tandem) Non-Stop Himalaya [3] employs lockstepping which runs the same program on two processors and compares the results by a checker circuit. Reis et al., [21] introduced PROFIT technique regulating the stage of reliability at fine granularities by using software control. This profile-guided fault tolerance determines the weakness and performance trade-offs for each program region and decide where to turn on/off redundant execution using a program profile. Oh et al., [22] proposed Error Detection by Duplicated Instructions (EDDI) which copies all instructions and inserts check instructions for validation. Software-based mechanisms present better reliability at low hardware cost and high fault coverage. However the performance degradation and failure to directly check micro-architectural components result in another trend of fault detection, hybrid redundancy techniques [23] such as CompileR-Assisted Fault Tolerance (CRAFT).

2.2 Detection Scheme Based
We classify detection and analysis strategies based on the following criteria: statistical, profiling, model-based, and distance-based methods.

Statistical methods trace the system behavior or user activity by gauging variables over time such as the event messages between components, system resources consumption, and login/logout time of each session. They maintain averages of these variables and detect anomaly behaviors by making a decision whether thresholds are exceeded based on given standard deviation of the variables monitored. It also compares profiles of short/long term user activities using complex statistical models. Ye and Chen [24] employ chi-square statistics to detect anomalies. In this approach, the activities on a system are monitored through a stream of events and are distinguished by event type. For each event type, the normal data from audit events are categorized and then used to get chi-square for the difference between the normal data and testing data. It considers large deviations as abnormal data.

One of the limitations of statistical approaches is that they become inaccurate and difficult to calculate the multidimensional distributions of the data points when outlier detection points exist in higher dimensional spaces [25]. Distance-based methods try to overcome this limitation and identify outliers by computing distances among points. For example, Cohen et al., [26] present an approach using usual clustering algorithms such as k-mean and k-median to get the system status. The difference is others focus on clustering algorithm to create a signature and show the efficacy for clustering and connection-based recovery by means of distinguished techniques such as pattern recognition and information retrieval.

Model-based methods describe the normal activity of the monitored system by using different types of models and identify anomalies as divergences from the model that characterizes the normal activity. For example, Maxion and Tan [27] obtain sequential data streams from a monitored procedure and employ Markov models to decide whether the states are normal or abnormal. They calculate the probabilities of transitions between the states using the training data set and utilize these probabilities to evaluate the transitions between states in test data set.

Profiling methods build profiles of normal behavior for diverse types of systems, users and applications, and consider variations from the profiles as anomalous behaviors. These profiling methods usually deploy different data mining techniques or they could be heuristic-based. In data mining methods, each case in training data set is configured as normal or abnormal and a data mining learning algorithm is trained over the configured data set. By using these methods, new kinds of abnormalities can be detected in fault detection models with retraining [25]. Lane and Brodley [28] use a temporal sequence learning technique in profiling UNIX user commands for normal and abnormal scenarios. It then uses these profiles to detect any anomalous user activity. Other algorithms for fault detection include regression trees [5], multivariate linear regression [10] [9], logistic regression [13] [14], fuzzy classification [12], neural networks [8] [7] and decision trees [30].
3. SYSTEM FRAMEWORK

The importance of a self-healing framework in PCS is exemplified in its ability to tolerate a wide range of faults. It dynamically builds self-healing mechanisms according to the type of detected fault. As a typical autonomic computing paradigm, our approach is based on continuous monitoring, analysis of the system state, then planning and executing the appropriate actions if it is determined that the system is deviating significantly from its expected normal behaviors. By monitoring the system state, we collect measurement attributes about the CPU, IO, memory, operating system, and network operations. Analysis of such collected data can reveal any anomalous behavior that might have been triggered by failures. In this section, we provide the overall framework and highlight the core modules for self-healing in PCS.

3.1 Self-Healing Engine

The framework achieves fault-tolerant services for components of a PCS such as applications, nodes and services. It achieves anomaly detection through tracing the interaction among components during runtime, identifying the source of faults and then planning recovery actions without any user interference, as shown in Figure 1.

![Figure 1. Self-Healing System Architecture](image)

Our framework consists of several core modules such as Self-Healing Engine (SHE), Application Fault Manager (AFM), and Component Fault Manager (CFM). Through Application Fault Management Editor (AFME), the user specifies the fault tolerance requirements associated with the type of applications or nodes involved in the self-healing system. SHE receives this request and builds an appropriate Self-Healing Strategy (SHS). The composition of SHS by SHE is policy driven. Policies are a set of predicates that define a course of action. SHS is selected when user-defined attributes and fault type match SHS in knowledge repository in the SHE. Through this process, SHE can identify static configuration and cater to users’ needs before runtime. Once SHS is completely created, AFM which is responsible for executing and monitoring the SHS gets an instance of SHS.

3.2 Application Fault Manager

AFM is a core module in our framework and is responsible for several activities such as monitoring, anomaly analysis, root-cause analysis, and recovery. Runtime behavior interactions of components and applications are stored in knowledge repository following the AppFlow datagram format which is three-dimensional array of features capturing spatial and temporal variability for each application. Once we get behavior information, we can analyze whether a deviation of application behavior shows anomaly behavior (or not) which is acquired by training experiment. To increase the accurate modeling of anomaly behavior, AFM also have fault injector and workload generator in training environment. If the behavior of application evaluated by AFM shows abnormal states, AFM identifies the source of faults by tracing and pinpointing the key in AppFlow. Once a fault is detected, the next step is to identify the appropriate fault recovery strategy defined in SHS to bring the system back into a fault-free state. In this paper, we focus on monitoring and analyzing the behavior interactions among these components to detect any hardware or software failures. The activities such as monitoring, anomaly analysis, root-cause analysis, and recovery exist in a hierarchical fashion so that failure of any one level does not lead to failure of the entire system.

3.3 Component Fault Manager

The first step for anomaly detection is to identify a set of measurement attributes that can be used to define the normal behaviors of these components as well as their interactions with other components within the distributed system. For example, when a user runs a QuickTime application, one can observe certain well defined CPU, Memory and I/O behaviors. These operations will be significantly different when the application experiences un-expected failure that leads to application termination. One can observe that the application, although consuming CPU and memory resources, does not interact normally with its I/O components. CFM resides in each node and traces all set of measurement attributes for applications and nodes. Once it is collected, it sends these monitored data to knowledge repository following the AppFlow datagram format with keys and features. Also, CFM which is subordinate to the AFM focuses on the execution and maintenance of the SHS associated with the component running on a node. During runtime phase, AFM and CFM maintain the operations of each resource and component according to the policies specified in SHS through the Component Management Interface (CMI) [31] [29] [37]. CMI provides the appropriate data structures (ports) to specify the control and management requirements associated with each software component and/or hardware resource. It includes four management ports including: configuration port; function port; control port; and operation port. The configuration port defines the configuration attributes that are required to automate the process of setting up the execution environment of a software component and/or a hardware resource. The function port specifies the functions to be provided by a component or a resource. It defines the syntax to be used to invoke each function. The control port defines all the data that need to be monitored and the control functions that can be invoked to self-control and manage a component or resource. This control port consists of two parts: sensor and action parts. The operation port defines the policies that must be enforced to govern the operations of the component and/or a resource as well as its interactions with other components or resources. We integrate our system with CMI to effectively handle all the management services required during
setting up the execution environment of any component or a resource as well as during runtime.

4. APPLICATION EXECUTION ENVIRONMENT

In this section, we propose a methodology to characterize the dynamic behaviors of applications and components in PCS, and analyze the behavior in training and testing using gram generation and pattern profiling.

4.1 AppFlow

AppFlow is a three-dimensional array of features capturing temporal variability and spatial variability for applications and systems. It characterizes the dynamic behaviors of applications and systems simultaneously with respect to key features in pervasive systems. This concept is very effective in anomaly detection and root-cause analysis since it remains consistent as we move from one hierarchy to another over time. For example, for EB (Emulated Browser)’s transactions going through the web, application and database servers, AppFlows can show the servers, applications and network behavior in one flow for each transaction. Through this Appflow procedure, we detect anomalies by characterizing the dynamic behaviors of the environment and immediately identify sources of fault by analyzing AppFlow key features showing the interactions among hardware and software for each instance.

4.2 AppFlow Data Structure

We capture the interactions behavior by using a set of core features for the applications and systems. These core features can be categorized into three distinct classes: Hardware Flow (HWFlow); Software Flow (SWFlow); and Network Flow (NETFlow). Each distinct class includes keys and features shown in Figure 2. Keys can be seen as links connecting different spaces. For example, if there are thousands of application instances over various system nodes including servers and clients, we need information allowing us to differentiate each instance after fault detection. The necessity of this information results in building key data structure. By classifying and tracing the keys from the AppFlow knowledge repository, we can instantaneously identify the source of faults once anomaly behavior is detected. Features in each distinct class indicate the dynamic capacity of the applications/systems that changes based on the requirements of the application at runtime.

4.3 AppFlow Information Exchange and Behavior Analysis

AppFlow is instantiated by CFM to model the behavior of applications and sent to AFM to detect anomaly states. The information of keys is further used by AFM in the stage of root cause analysis to pinpoint the source of faults. Figure 3 explains abnormality identification with AppFlow drift.

Normal execution has a certain trajectory with respect to system interfaces defined in AppFlow. When we assume large rectangle as a safe operating zone and several small rectangles as an anomalous operating zone in application operation shown in Figure 3, it initially shows steady state behavior over time in normal transaction. But if fault happened, we can see the transient behavior with respect to steady state behavior over AppFlow’s functional features and time. Notice that, we maintain the AppFlow functional features within the safe operating zone. This zone defines the safe and normal operation for the application. If the AppFlow functional features vary outside the safe operating zone to any of the anomalous operating zones, AFM analyzes whether the state of application meets the condition of anomaly state. Once AFM classifies the state as abnormal state, it pinpoints the source of fault and takes an action/strategy to recovery/reconfigure the application so that the features stay within the safe operating zone. We can apply this approach to real example such as QuickTime. When it runs normally, one can observe several normal behaviors such as normal CPU usage, normal memory usage, normal I/O repeat retry access, normal access time and no error message in system call. These operations will be different when the application experiences un-expected failure that leads to, let’s say, disk failure. One can observe several suspicious symptoms; I/O repeat/retry accesses are increased, access time takes longer, error messages in system call, no I/O read or write bytes, unusual system call patterns, etc. These suspicious symptoms are captured within AppFlow features and AFM analyzes and decides the state of application. AppFlow defines and classify these behaviors appropriately in each normal and abnormal transaction. Figure 4 shows the ability of AppFlow functional features to differentiate the abnormal pattern from normal pattern. When we trace the patterns within AppFlow functional features using system call, it shows the distinct behavior between abnormal transaction and normal transaction.
from the point of fault injection. With this, AppFlow concepts and our proposed approach could achieve 100% detection rates with no occurrences of false alarm in most of scenarios. Related experiments are explained in section 5.

4.4 Gram Generator and Framing
Pattern profiling for anomaly-based fault detection involves building a complete profile of normal traffic through our AppFlow functional feature datagram. In this work, we apply our approach to analyze the anomalous behavior of TPC-W, an industry standard e-commerce web application. It is an ideal application to emulate complex pervasive computing environment engaging several computational units and systems concurrently. Our anomaly-based approach has two phases: training and testing, shown in Figure 5. In the training phase, we capture behavior interactions by using AppFlow functional features for the applications and systems that are categorized into three distinct classes, as explained in section 4.2. This allows us to build a normal profile. We then filter and collect AppFlow core functional features such as system call for every connection established within a specified time window. The sequence of AppFlow for every connection within a specified time window is then framed into higher order n-grams. This allows us to examine the state transitions for each connection at different granularities. And then, we represent a set of elements and support membership queries by using a Bloom Filter [32], a probabilistic data structure that is space as well as memory efficient. These data are used in testing phase to detect anomaly flow.

In our pattern behavior analysis approach we focus on analyzing pattern transition sequences of length \( n \) during a window interval. Our heuristic approach follows that when an application (e.g., TPC-W) fault happens. It generally generates unusual state transitions that can be identified by our pattern transition analysis. For example, in TPC-W transactions, the browsing transaction should follow the following sequence in order to complete an EB. Home \textendash{} Search Request \textendash{} Search Result \textendash{} Search Request \textendash{} Search Result \textendash{} Product Detail. With a TPC-W failure (e.g., DB failure – access denied), several repeat/retry Search Request sequence are observed generating abnormal patterns of system calls. Another example is wrong input/output in sequence interactions. For example, Search Result containing the list of items that match a given search criteria is invoked by an HTTP request only from Search Request. But if the input comes from Admin Request due to application malfunction, it will show different sequence transitions resulting in distinct behavior patterns that can be captured within AppFlow’s functional features.

We utilize n-gram analysis [33] [34] [35] to study these transitions and a sliding time-window to increase the velocity of anomaly detection as soon as they happen. One motivating study for n-gram is Anagram [36]. They used a mixture of higher order n-grams to detect anomalies in the packet payload. In our study, we use n-gram analysis over a specified time window to reveal the distinct class in behavior interaction patterns of applications/systems. N-gram analysis is a methodology that uses a sub-sequence of \( n \) items from a give sequence. In our system, we generate n-grams by sliding windows of length \( n \) over AppFlow functional features (e.g., system calls). The window-based training is performed using a window size of seconds, which was found to be optimal for detecting the anomaly behavior of applications. Analysis of the sequence transitions that occur during a given window allows us to accurately detect various types of faults explained in section 6.1. By using different window sizes and varying the gram length, it is possible to detect fast or slow-occurring transition violations.

4.5 Pattern Profiling
One of the significant issues in pattern profiling after gram generation with sliding windows is memory overhead since our data set is huge over 25 millions. So, we have extended the use of a Bloom Filter [38] to efficiently detect the occurrence of faults in fault management the memory issue. It is a probabilistic data structure having a strong space gain over other data structures for representing sets as well as memory gain. Its primary function is to represent a set of elements and to efficiently support membership queries. Given a set of elements \( Y \) with \( m \) keys,
for each element $y'_i$ in $Y$, it is a bit-vector of length $\ell$ using $h$ hash functions. If one of the $h$ bits is no less than 0, the element is stated to be a non-member of the set. If all bits are 1, the element has a probability to be a member of the set. If all $h$ bits are 1 and $y'_i$ is not a member of $Y$, then it can be declared as a false positive. There is a clear trade-off between the size of the bloom filter and the probability of a false positive. The probability that a random bit of the $\ell$ bit vector is 1 by a hash function, $P_{x}$, is $1/\ell$. The probability that it is not set, $P_{ns}$, is:

$$P_{ns} = \left(1 - \frac{1}{\ell}\right)$$

The probability that it is not set by any of the $m$ keys with element set $h$ bits, $P_{nsh}$, is:

$$P_{nsh} = \left(1 - \frac{1}{\ell}\right)^m h$$

Therefore, the probability of the occurrence of a false positive is:

$$P_{f} = \left(1 - \left(1 - \frac{1}{\ell}\right)^m h\right)^h$$

This value can be approximated to:

$$P_{f} = \left(1 - e^{-\frac{m h}{\ell}}\right)^h$$

This value is minimized for a value of:

$$h = \frac{\ell}{m} \ln 2$$

The value of $k$ determines the number of hash functions to be used, and it needs to be an integer. In order to calculate the number of hash functions needed to efficiently model our normal traffic, while still maintaining a high detection rate, we chose the false probability rate of the bloom filter to be 0.01%. We used seven distinct hash functions based on the number of elements in our training data set, and our desired false positive rate of 0.01%. We chose seven one-way additive and rotatable general purpose string hashing algorithms in the training as well as the testing phases of our system [38] [32].

5. RESULT AND EVALUATION
In this section, we illustrate data source, abnormal loads and various scenarios and evaluate the anomaly detection capabilities of our approach using abnormality extent degree with AppFlow analysis.

5.1 Experimental Setup and Data
In our evaluation, we use an industry standard web e-commerce application, TPC-W [4] to emulate the complex environment of PCS. TPC-W is an ideal application to establish PCS engaging several computational units and systems concurrently. The system is used to run a typical end user e-commerce activity initiated through a web browser and consisting of several TPC-W transactions. It defines 3 types of traffic mixes such as browsing mix, shopping mix and ordering mix and specifies 14 unique web transactions. In our environments, the database is configured for 288,000 customers and 10,000 items. According to the TPC-W specification, the number of concurrent sessions is executed throughout the experiment to emulate concurrent users. The maximum number of Apache clients and maximum Tomcat threads are set to 90. The workloads are generated by the workload generator that varies the number of concurrent sessions and running time from 90 to 300 seconds.

We also developed abnormal workload generator allowing us to make and track the system abnormally behavior. To enlighten our variety in abnormal loads, several papers [12] [20] [31] are considered as a previous study in faults injected in their experiments. Some of them focus on triggering only application level failures; others inject the faults concentrated on problems that cause program crashes or Byzantine faults. We believe that there are system interaction symptoms that characterize how system will respond to a fault injected. Thus, we have confidently decided to include anomaly activities showing the symptom of software failures as well as hardware failures in complex PCS. Table 1 shows the types of fault classes to be used by our anomaly analysis experiments.

<table>
<thead>
<tr>
<th>Fault Scenario</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault Type All (FTA)</td>
<td>Random based – All of the faults</td>
</tr>
<tr>
<td>Fault Type 1 (FT1)</td>
<td>Declared exceptions and undeclared exceptions such as Unknown host Exception in TPC-W transactions</td>
</tr>
<tr>
<td>Fault Type 2 (FT2)</td>
<td>DB failure – Access denied in TPC-W transactions</td>
</tr>
<tr>
<td>Fault Type 3 (FT3)</td>
<td>I/O random failure in TPC-W ordering transaction</td>
</tr>
<tr>
<td>Fault Type 4 (FT4)</td>
<td>Disk failure - Not enough space in TPC-W browsing and ordering transaction</td>
</tr>
<tr>
<td>Fault Type 5 (FT5)</td>
<td>Connection refused</td>
</tr>
</tbody>
</table>

In these experiments, we inject five different types of faults explained in Table 1. We model these various faults that are triggered by the interfaces including interactions between an application and the operating system or between an application and other function libraries. It allows us to isolate the node by removing the connection from the network interfaces. FT1 and FT2 are about database/server related failure such as access denial and application exceptions such as declared exceptions regarding 3 different types of TPC-W traffics including browsing, ordering, and shopping. Because java based e-commerce application engenders various different kinds of failures from programmer faults to IO faults, an injection of exception faults is apposite to reveal the abnormal behavior of e-commerce application by tracking system interactions. Here, we injected declared exceptions which are often handled and masked by the application itself such as unknown host exception. FT3, FT4, and FT5 are injected from shared libraries into applications to test the capability of applications to handle faults from library routines.
referring the fault injection technique [10]. We inject the faults using various system calls such as read(), write(), and recv() and observe the effect of injected faults. All these faults happen in a process of TPC-W transaction. We believe that the selected faults span the axes from expected to unexpected/undesirable behaviors and divulge the relationship of system interaction for the problems that can occur in a real life.

5.2 Trustworthiness and Validation for Fault Types
In this experiment, we have six different fault scenarios explained in table 1 and section 5.1. We categorize and inject faults by building six different scenarios. In scenario 1 and 2, we chose to inject particular faults such as declared exceptions and database access denied in regarding 3 different types of TPC-W traffic such as browsing mix, ordering mix and shopping mix. In scenario 3, 4 and 5, the faults injected are from shared libraries into applications to test the capability of applications to handle fault from library routines. We inject the faults with various different system calls such as read(), write(), and recv() and utilized TPC-W application with three different functions randomly to observe the effect of injected faults with interfaces. The last scenario includes all faults explained and they are randomly injected in experiments. For the scenarios explained, we evaluate the cross-validated false positive rate (the percentage of normal flows incorrectly classified as abnormal) and false negative rate (the percentage of abnormal flows incorrectly classified as normal) to evaluate our methodology. We have implemented gram generator using all possible n grams from three to nine and it allows us to find optimal solutions for various faults in PCS.

FT1, FT2, FT3, FT4, FT5 and FTA that utilize all previous five scenarios based on a random mix are given with the number of grams. Figure 6 shows the false positive rate and false negative rate for all studied scenarios. From the graphs, it is noticeable that fault scenarios from one to five achieve 0% false positive rate and 0% false negative rate in n grams from three to seven. These results are the best results that we can get. One might think that FTA would have the worst result compare to the other scenarios since FTA has all of the generated faults that might result in complex and intricate interactions when compared to single fault type. But it is the other way around. Our results for FSA show that the worst result is less than 0.09% in false positive rate and 0.075% in false negative rate from three to eight grams. From the experiment, we find the optimal grams to be five, six and seven. In general, low order n grams produce less optimal detections since relative few n grams are not long enough to capture accurate behavior interactions. This fact is also applied in high order n grams since they are too long and may miss correct patterns of application behavior.

5.3 Trustworthiness and Validation for Data Sizes
We classify and compose data based on the specification supplied with the multi-tier web benchmark by building four different scenarios in this experiment. These four scenarios reveal the impact of bulk training data set.

Data scenario 1 (DS1) consists of the abnormal set and normal set containing more than 16 million records maintaining the ratio of
Table 2. Detection rate and missed alarm rate of all scenarios under applying the AppFlow approach

<table>
<thead>
<tr>
<th></th>
<th>Detection Rate (4 gram)</th>
<th>Missed Alarm (4 gram)</th>
<th>Detection Rate (5 gram)</th>
<th>Missed Alarm (5 gram)</th>
<th>Detection Rate (6 gram)</th>
<th>Missed Alarm (6 gram)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FS1</td>
<td>100 %</td>
<td>0 %</td>
<td>100 %</td>
<td>0 %</td>
<td>100 %</td>
<td>0 %</td>
</tr>
<tr>
<td>FS2</td>
<td>100 %</td>
<td>0 %</td>
<td>100 %</td>
<td>0 %</td>
<td>100 %</td>
<td>0 %</td>
</tr>
<tr>
<td>FS3</td>
<td>100 %</td>
<td>0 %</td>
<td>100 %</td>
<td>0 %</td>
<td>100 %</td>
<td>0 %</td>
</tr>
<tr>
<td>FS4</td>
<td>100 %</td>
<td>0 %</td>
<td>100 %</td>
<td>0 %</td>
<td>100 %</td>
<td>0 %</td>
</tr>
<tr>
<td>FS5</td>
<td>100 %</td>
<td>0 %</td>
<td>100 %</td>
<td>0 %</td>
<td>100 %</td>
<td>0 %</td>
</tr>
<tr>
<td>FSA</td>
<td>99.9545 %</td>
<td>0 %</td>
<td>99.955 %</td>
<td>0 %</td>
<td>99.9772 %</td>
<td>0 %</td>
</tr>
<tr>
<td>DS1</td>
<td>99.9545 %</td>
<td>0 %</td>
<td>99.9775 %</td>
<td>0 %</td>
<td>100 %</td>
<td>0 %</td>
</tr>
<tr>
<td>DS2</td>
<td>100 %</td>
<td>0 %</td>
<td>99.9775 %</td>
<td>0 %</td>
<td>100 %</td>
<td>0 %</td>
</tr>
<tr>
<td>DS3</td>
<td>99.9545 %</td>
<td>0 %</td>
<td>99.955 %</td>
<td>0 %</td>
<td>99.9772 %</td>
<td>0 %</td>
</tr>
<tr>
<td>DS4</td>
<td>99.9 %</td>
<td>0 %</td>
<td>99.99 %</td>
<td>0 %</td>
<td>99.9089 %</td>
<td>0 %</td>
</tr>
</tbody>
</table>

All these data scenarios are given with the size of the data flows and the number of grams and the results are shown in Figure 7. All scenarios have good results even in low and high order $n$ grams. However, there is slight impact in DS4 regarding to false positive rate and in DS2 regarding to false negative rate when compared to the other scenarios because of the ratio in data set. DS4 which has more abnormal flows and less normal flows compare to the other scenarios shows higher false positive rate and DS2 which has more normal flows and less abnormal flows shows higher false negative rate. These experiments allow us to know that there is slight impact of bulk training data set. But all scenarios still have very good results showing at most 0.15% in false positive rate and 0.023% in false negative rates. All scenarios from three gram to seven gram achieve 0% false negative rates.

5.4 Performance Validation for Testing Data

Based on the confidence about the results showed before, we have experiments by using optimal $n$ grams such as four, five and six. Table 2 shows the detection rate and missed alarm for all scenarios we did for different number of grams.

As expected, the detection rate and missed alarm rate are very good for most of scenarios in huge data set. The ratio of data size in abnormal and normal flows and the complex interactions influences the detection rate in FSA, DS2 and DS4. The worst detection rate in all scenarios is 99.9%, but is still very high. Missed alarm rate is extraordinarily 0% in all scenarios. We have achieved no less than 99.9% (100% in most of scenarios) detection rate with no occurrences of false alarm for a wide range of scenarios. It proves that our approach is superior in various fault scenarios.

6. Conclusion

Our In this paper, we presented an innovative concept, based on AppFlow, to capture abnormal behaviors of applications triggered by hardware and/or software failures. We also developed an effective experimental framework to detect any type of faults for PCS and evaluated detection rates and missed alarm rate for various scenarios with varying the number of grams. Our experiment results show a detection rate of above 99.9% with no occurrences of false alarm for a wide range of scenarios. Based on these results, we are confident of our approach and are currently implementing root-cause analysis not only to detect the faults once they occur, but also to identify the source of fault allowing us to perform automatic fault recovery.

7. REFERENCES


