Diagnosis of Software Failures Using Computational Geometry

Edward Stehle, Kevin Lynch, Maxim Shevertalov, Chris Rorres, and Spiros Mancoridis
Department of Computer Science, College of Engineering, Drexel University
3141 Chestnut Street, Philadelphia, PA 19104, USA
{evs23, kml43, max, spiros}@drexel.edu, crorres@cs.drexel.edu

***Abstract*—Complex software systems have become commonplace in modern organizations and are considered critical to their daily operations. They are expected to run on a diverse set of platforms while interoperating with a wide variety of other applications. Although there have been advances in the discipline of software engineering, software faults, and malicious attacks still regularly cause system downtime [1]. Downtime of critical applications can create additional work, cause delays, and lead to financial loss [2].**

This paper presents a computational geometry technique to tackle the problem of timely failure diagnosis during the execution of a software application. Our approach to failure diagnosis involves collecting a set of software metrics and building a geometric enclosures corresponding to known classes of faults. The geometric enclosures are then used to partition the state space defined by the metrics

I. Introduction

Many of the software bugs and security attacks that a software system experiences are due to classes of faults that the software has previously encountered. For example, problems caused by memory leaks may be seen multiple times during the lifecycle of a software system. This paper proposes a technique that constructs models of failures due to known faults. These models can be used to automatically diagnose future occurrences of similar faults. For example, if system metrics are captured while a software system is experiencing a memory leak, a model of a memory leak can be constructed to identify future memory leaks.

In this paper the term failure refers to a state where a software system is no longer as operating as intended. The term does not necessarily imply that the software has already crashed or has become non-responsive. Often a failing state indicates that software is approaching a state that will likely cause a crash. The term fault refers to a coding bug or malicious attack that may cause a failure.

Our previous work [3] proposed a technique for detecting software failures using computational geometry. This paper proposes an technique for diagnosing software failures that extends the geometric approach from our previous work. After a failure is detected, diagnosis attempts to characterize the failure. Failures are characterized by determining if they are members of known fault classes, such as for example, a memory leak or an infinite loop. This will aid in the recovery of failing software systems by providing insight into the cause of the failure. This insight can be used to determine a strategy to mitigate the effects of the failure.

The proposed diagnosis technique uses geometric enclosures to partition a state space defined by software metrics. For example, there could be one enclosure for memory leaks, one enclosure for infinite loops, etc. If a point falls inside of an enclosure, it is labeled as a failure due the fault class corresponding to the enclosure. If a point falls outside of the enclosure for the safe state and outside of all of the enclosures for fault classes, it is labeled as an unknown failure. Our technique is illustrated in Figure 1.

II. Related Work

A failure diagnosis is a description of a detected failure intended to give insight into the cause of the failure. Chen *et al.* define failure diagnosis as “The task of of locating the source of a system fault once it is detected” [4]. Some diagnoses localize a failure in a system [5]. In a software system with a large number of components, knowing which component is experiencing a failure can be useful to the mitigation mechanism.

Bayesian Network classifiers have been applied to several previous approaches to diagnosis. Ghanbari *et al.* propose an approach to diagnosis based on Bayesian networks that are wholly or partially specified by a human user [6]. Tree augmented naïve bayesian classifiers are the basis for diagnosis in work by Cohen *et al.* [7]. Zhang *et al.* use ensembles of tree augmented naïve Bayesian classifiers to diagnose faults. [8]. Other classification techniques have been used to diagnose software faults. Chen *et al.* offered an approach to the diagnosis of failures in large-scale Internet service systems using decision trees [4]. Duan *et al.* proposed an approach to diagnosis of software failures that uses active learning to minimize the number of data points that a human administrator must label. [9].

III. The Aniketos Approach

Our approach to the detection and diagnosis of software faults is named Aniketos after the minor Olympian god, and son of Heracles, who protected the gates of Mount Olympus from attack. Aniketos is constructed from a collection of new techniques and tools that enable software systems to self-diagnose potential failures due to faults or security attacks.

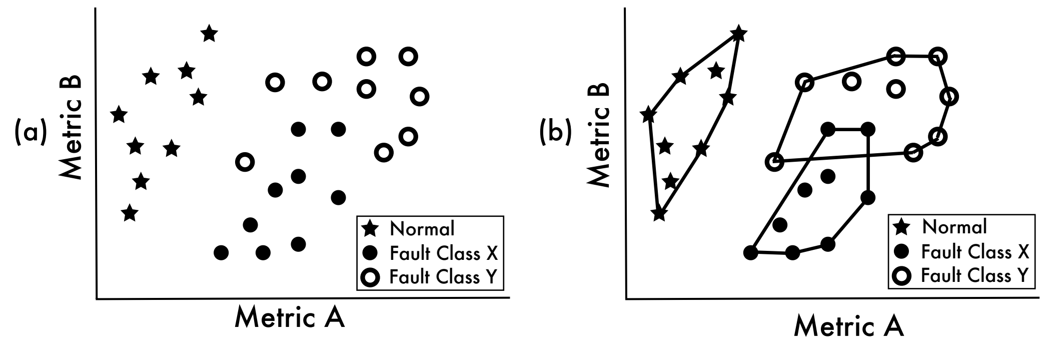

Fig. 1. Building convex hulls for diagnosis (a) sample points from the normal operating state of the software and two fault classes (b) convex hulls constructed around the samples. In (b) the state space has been partitioned into normal state, known faults, and unknown faults. Points that fall into a fault hull are diagnosed as the class of the corresponding hull. Points that do not fall into any hull are diagnosed as unknown faults.

The technical approach is based on a set of metrics that are computed using data obtained by software sensors that monitor the status of software at runtime. Previously we used these metrics to determine whether software is operating as expected or deviating from its expected operation. Software whose metric values deviate significantly from their expected values are considered potentially failing. The software may then be subjected to mitigation actions that can either attempt to avoid a failure or collect data to facilitate a postmortem forensic analysis. In order to mitigate the affects of a software fault it is first necessary to diagnose the fault.

The Aniketos failure diagnosis technique can be divided into two phases, a training phase and a monitoring phase. The training phase involves learning models of a software system's behavior while the system is executing known faults. The monitoring phase uses the models of fault classes to diagnose abnormal system behavior that may be symptomatic of faults occurring. Once a potential fault has been detected, the current system state is compared to the models of known faults to determine if the current fault is one that we have seen before.

A. Training Phase

During the training phase, the monitored system executes a set of known faults one at a time and collects measurements at regular time intervals. Once enough training points are collected for each type of fault, they are grouped into a training data sets. These training data sets are used to construct a set of n-dimensional convex hulls, where n is the number of distinct metrics used (i.e., CPU time, heap memory, etc.).

B. Diagnosis Phase

Diagnosis begins once a failure has been detected. Once diagnosis has started, there are three possibilities for an observed point. If a point falls into only one hull, the point is labeled as being a failure due to the fault whose convex hull it falls into. If a point does not fall into any of the hulls, the point is labeled as an unknown failure. If a point falls into more than one hull, we say that we have a tie. Ties are broken by finding the distance from the point we are trying to label to the centroids of each of the tied classes. The point we wish to diagnose will be labeled as the fault class corresponding to the closest centroid.

IV. Experimental Setup

A case study involving three server applications was performed to evaluate the Aniketos failure diagnosis technique. The servers used were the Java based HTTP servers NanoHTTPD and Jigsaw, and the Java based application server Apache Tomcat. Twenty-three metrics were collected while software systems executed faults injected into the source code. The metrics fall into six categories. These categories are memory, threads, classes, CPU, network, and stack.

A. Experimental Data

The case study used resources and access patterns of the Drexel University Computer Science Department website to generate realistic application data. Nine weeks worth of website access logs and all of the resources accessed in those logs were collected. The resources were extracted and converted into a static mirror of the Drexel University Computer Science website. Both NanoHTTP and Jigsaw were used to host the static version of the Drexel CS Website.

Nine weeks worth of website access logs were collected and all resources accessed in those logs were extracted and converted into static HTML pages. Out of the nine weeks of logs, three weeks were chosen at random to be used in the case study. These were replayed against the statically hosted version of the website and provided the case study with realistic workload access patterns.

One week of request logs was used to generate the convex hull enclosures representing the normal operating space. Another week of request logs was used to evaluate how well the enclosures classify fault-free data. A third week of request logs was used as background activity during the fault-injection experiments.

In addition to the HTTP servers hosting the static mirror of the Drexel CS website, data was collected from FaultBench, a e-commerce benchmarking application implemented on the Tomcat application server. The application uses probabilistic models of customers browsing, buying, and selling to create realistic use patterns. Training data was collected from the generated use patterns to represent the normal operational state of Tomcat.

B. Injected Faults

To test the Aniketos failure diagnosis technique, faults were injected into the server applications while they executed their normal behavior. For NanoHTTP and Jigsaw, faults were injected during the play back of HTTP requests, and for Tomcat, faults were injected while FaultBench simulated users accessing a commerce website. The injected faults capture coding errors as well as security vulnerabilities and attacks.

C. Failure Detection

The Aniketos failure diagnosis technique assumes that failures can be detected. For the case study, the detection technique from our previous work on failure detection is used. Like the Aniketos failure diagnosis technique, the failure detection technique uses software metrics to construct a model of the software state. The detection approach uses information collected by the runtime sensors to construct a geometric enclosure whose enclosing space represents the normal execution of the monitored application.[3]

V. Experimental Results

This section contains the results of the case study on the diagnosis of software failures. The failure diagnosis results include the results of experiments designed to test Aniketos's ability to diagnose known fault types. Section V-A details the results of diagnosis experiments

A. Failure Diagnosis Experiments

In the case study experiments, Aniketos attempts to diagnose occurrences of known faults. In each of the experiments, samples of measurements were collected while the server was executing known faults. These samples were used to train classifiers. The classifiers were then used to diagnose occurrences of faults of the same types as those in the training data. For example, the Tomcat failure-detection experiment uses a set of six faults. The faults used were database lock, infinite recursion, and a fast and slow versions of infinite loop and log explosion faults. Aniketos collected metrics from the Tomcat application server as Tomcat executed each of the six faults. Each fault was executed five separate times. The data collected from two of the five runs for each of the faults was used as training data. The remaining three runs were used to test the performance of failure diagnosis.

Four additional classifiers are compared with Aniketos. The additional classifiers used were Naive Bayes, voting feature intervals (VFI), C 4.5 decision trees and nearest neighbor. Detailed results of the Tomcat experiments can be seen in Table II. The percentage of points that were diagnosed correctly is shown in the Correct column under each diagnosis method in Table II . For example, when we used convex hulls to diagnose a database lock, we were able to diagnose 91% of the points correctly. The unknown column under Convex Hulls shows the percentage of points that were classified as unknown faults.

Because the other techniques are based on classifiers they always return a class. Therefore they do not have unknown columns. All of the techniques that were used correctly diagnosed all of the faults. The columns showing the percentage incorrect are included for comparison with convex hull approach. A summary of the diagnosis results for all servers in the case study can be seen in Table I.

TABLE I
Summary of the diagnosis results

	Points Correctly Classified			
	Average Over all Servers	Tomcat	Nano	Jigsaw
Aniketos	94%	92%	93%	96%
Naive Bayes	93%	92%	97%	90%
VFI	94%	97%	91%	94%
C 4.5	99%	99%	98%	99%
Nearest Neighbor	96%	91%	98%	99%

Although it is important that Aniketos recognize faults that it has seen before, it is also important for Aniketos determine if an observed fault is not of a type that Aniketos has not seem before. If Aniketos is used to monitor deployed software systems, it is likely to encounter new faults and security attacks, for example, a zero day virus. We do not want our diagnosis technique to always give us a best guess from a finite list of known faults. We would like our diagnosis to recognize faults that it has seen before, but also recognize if a fault is new. We need to know when we have something new that we need to analyze, and eventually include in our set of known faults

In our diagnosis approach we only say that the system is experiencing a specific fault if points collected from the current operating state fall inside the convex hull corresponding to the fault. If Aniketos has detected a fault, but the faulty points do not fall into a convex hull representing a known fault, Aniketos reports that the system is experiencing an unknown fault.

VI. Conclusions

This paper presents a novel technique for the diagnosis of runtime faults that uses methods based on computational geometry. The technique uses convex hulls to partition software state space. To detect failures, training points are collected while a software system is operating normally. A convex hull is constructed around these training points as a model of the normal operating state of the software. To diagnose failures convex hulls are built around training points generated by the execution of known faults. When a system is monitored, points that fall into a convex hull are classified as being in the state that the convex hull represents. For example, if a point falls into the hull for a memory leak, the point is labeled as a memory leak. The technique will detect failures caused to bugs (*e.g.*,

TABLE II
RESULTS OF FAILURE DIAGNOSIS EXPERIMENTS ON THE APACHE TOMCAT APPLICATION SERVER

	Convex Hulls			Naive Bayes		VFI		C 4.5		Nearest Neighbor	
Fault	**Correct**	**Incorrect**	**Unknown**	**Correct**	**Incorrect**	**Correct**	**Incorrect**	**Correct**	**Incorrect**	**Correct**	**Incorrect**
Database Lock	91%	0%	9%	99%	1%	100%	0%	100%	0%	78%	22%
Fast Infinite Loop	100%	0%	0%	100%	0%	100%	0%	100%	0%	100%	0%
Slow Log Explosion	95%	2%	3%	96%	4%	92%	8%	98%	2%	97%	3%
Fast Log Explosion	97%	0%	3%	97%	3%	99%	1%	99%	1%	100%	0%
Slow Infinite Loop	82%	8%	11%	91%	9%	94%	6%	100%	0%	84%	16%
Slow Infinite Recursion	92%	0%	8%	72%	28%	98%	2%	100%	0%	87%	13%

memory leak) and failures caused by malicious attacks (*e.g.*, denial of service attack).

The Aniketos technique it is easy to understand In concept and it covers a range of failure detection and diagnosis types in a single technique. It can use metrics that are dependent or independent. If the data is independent then our method will approximate range checking techniques and if there are dependancies in the data, detection and diagnosis can be improved by making use of these dependencies. Note that although our technique can take advantage of dependancies it does not require dependancies to work and it does not need to learn correlations.

VII. ACKNOWLEDGEMENTS

The authors would like to thank the Lockheed Martin Corporation for sponsoring this research.

REFERENCES

[1] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do internet services fail, and what can be done about it? In *USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[2] T. Sweeney. No Time for DOWNTIME – IT Managers feel the heat to prevent outages that can cost millions of dollars. *InternetWeek*, 807, 2000.

[3] E. Stehle, K. Lynch, M. Shevertalov, C. Rorres, and S. Mancoridis. On the use of computational geometry to detect software faults at runtime. In *Proceeding of the 7th international conference on Autonomic computing*, pages 109–118. ACM, 2010.

[4] Mike Chen, Alice X. Zheng, Jim Lloyd, Michael I. Jordan, and Eric Brewer. Failure diagnosis using decision trees. *Autonomic Computing, International Conference on*, 0:36–43, 2004.

[5] E. Kiciman and A. Fox. Detecting application-level failures in component-based internet services. *Neural Networks, IEEE Transactions on*, 16(5):1027–1041, Sept. 2005.

[6] S. Ghanbari and C. Amza. Semantic-driven model composition for accurate anomaly diagnosis. In *Autonomic Computing, 2008. ICAC '08. International Conference on*, pages 35–44, June 2008.

[7] Ira Cohen, Moises Goldszmidt, Terence Kelly, Julie Symons, and Jeffrey S. Chase. Correlating instrumentation data to system states: a building block for automated diagnosis and control. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.

[8] Steve Zhang, Ira Cohen, Moises Goldszmidt, Julie Symons, and Armando Fox. Ensembles of models for automated diagnosis of system performance problems. *Dependable Systems and Networks, International Conference on*, 0:644–653, 2005.

[9] Songyun Duan and S. Babu. Guided problem diagnosis through active learning. In *Autonomic Computing, 2008. ICAC '08. International Conference on*, pages 45–54, June 2008.