On the use of Computational Geometry to Detect Software Faults at Runtime

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

ABSTRACT

Despite advances in software engineering, software faults continue to cause system downtime. Software faults are difficult to detect before the system fails, especially since the first symptom of a fault is often system failure itself.

This paper presents a computational geometry technique and a supporting tool to tackle the problem of timely fault detection during the execution of a software application. The approach involves collecting a variety of runtime measurements and building a geometric enclosure, such as a convex hull, which represents the normal (i.e., non-failing) operating space of the application being monitored. When collected runtime measurements are classified as being outside of the enclosure, the application is considered to be in an anomalous (i.e., failing) state. This paper presents experimental results that illustrate the advantages of using a computational geometry approach over the distance based approaches of Chi-Squared and Mahalanobis distance. Additionally, we present results illustrating the advantages of using the convex-hull enclosure for fault detection in favor of a simpler enclosure such as a hyperrectangle

Categories and Subject Descriptors

D.2.7 [**Performance of Systems**]: Reliability, availability , and serviceability

General Terms

Reliability, Security

Keywords

Fault tolerance, system monitoring, failure detection

1. INTRODUCTION

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 and servers. Although there have been advances in the discipline of software engineering, faults still regularly cause system downtime. Downtime of critical applications can create additional work, cause delays, and lead to financial loss [20]. Faults are difficult to detect before an executing system reaches a point of failure, as the first symptom of a fault is often system failure itself. While it is unrealistic to expect software to be fault-free, actions such as resetting the software, quarantining specific software features, or logging the software's state prior to the failure for later analysis can be taken.

This paper describes a novel approach to solving the fault detection problem using computational geometry. The technique has a training and a detection phase. The training phase involves collecting a variety of runtime measurements, such as CPU time and heap memory, and uses these measurements to build a geometric enclosure that represents the normal operating space of the application being monitored. During the detection phase, the geometric enclosure is used to classify runtime measurements. The application is considered to be in an anomalous state when the collected measurements are outside of the geometric enclosure.

A case study is presented to demonstrate the advantage of using a convex-hull enclosure for fault detection over other geometric enclosures, such as a hyperrectangle, as well as the distance based measures of Chi-Square and Mahalanobis distance. This case study involves eight experiments and each experiment injects a different fault into an HTTP server. The convex hull method makes no assumption about any relationship among the different kinds of measurements, but takes advantage of dependence relationships if they exist. In the case study the convex-hull enclosure performed fault detection better than the hyperrectangle enclosure, which treats measurements as independent. In fact, in our experiments three of the eight faults were detected by the convex-hull method, but not by the hyperrectangle method. The Chi-squared approach failed to detect four of the faults and the Mahalanobis distance based approach failed to classify five of the faults.

Our implementation of this computational geometry approach to fault detection is named Aniketos after the minor Olympian god, and son of Heracles, who protected the gates of Mount Olympus from attack. Although our case study uses only the hyperrectangle and the convex-hull enclosures, Aniketos supports any type of geometric enclosure. For example, a sphere or an ellipsoid could be used in place of a hyperrectangle or a convex hull.

The remainder of the paper is organized as follows: Section 2 describes previous work in autonomic fault detection; Section 3

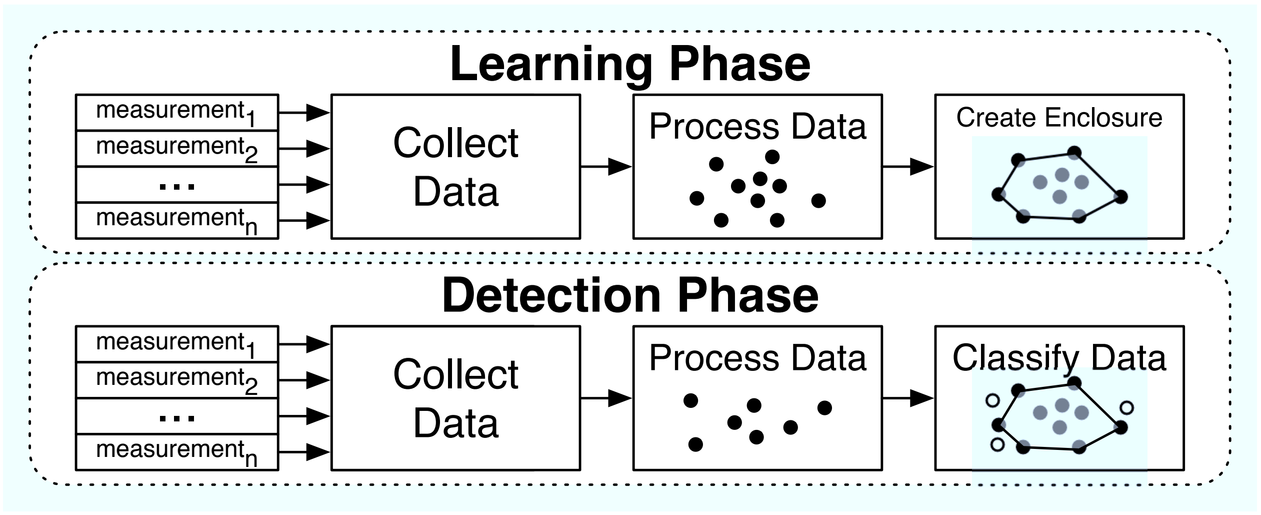

Figure 1: During the learning phase Aniketos collects and processes runtime measurements. It then constructs an enclosure representing the normal execution of the monitored system. During the detection phase Aniketos collects and processes runtime measurements. Then, each data point (n-tuple of metrics) is classified as normal (black points) if it is inside the enclosure, or anomalous (white points) if it is outside of the enclosure.

describes the Aniketos approach to fault detection; Section 4 describes the architecture of the Aniketos fault detection system; Section 5 describes a case study involving NanoHTTPD [9], a java-based web server; Section 6 summarizes the results of the case study; finally Section 7 states our conclusions and plans for future work.

2. PREVIOUS WORK

Existing approaches to the detection of software faults fall into two categories, signature-based and anomaly-based [1]. Signature-based methods detect faults by matching measurements to known fault signatures.These techniques are used in static fault-checking software such as the commercial antivirus software McAfee [15] and Symantec [21], as well as network intrusion detection systems such as Snort [19] and Netstat [22]. These techniques can also be used to detect recurring runtime faults [4].

If a set of known faults exists, then training a system to recognize these faults will typically lead to better fault detection. However, that system is unlikely to recognize faults it has not seen before. For example, a fault caused by a zero-day virus is unlikely to be detected by commercial antivirus software because there are no known patterns to recognize.

Anomaly-based methods learn to recognize the normal runtime behavior of the monitored system and classify anomalous behavior as potentially faulty. The advantage of using anomaly-based methods is that they can detect previously unseen faults. However, they risk incorrectly treating any newly encountered good states as faulty. This occurs if insufficient training data were supplied. The rest of the paper will focus on anomaly detection techniques, of which Aniketos is one.

Typically, anomaly-detection techniques begin by collecting sensor measurements of a normally behaving system. Then, they construct a representation of the monitored system and compare any future measurements against that representation. A naïve approach assumes that all metrics are independent and determines the safe operating range for each of them. In other words, during the learning phase, this method will record the maximum and minimum safe values of each metric and then classify the system as faulty when any of the measurements fall outside of the interval determined by these values.

While the naïve approach is capable of detecting some faults, it can fail when the assumption that the metrics are independent is incorrect. Therefore, more sophisticated fault detection techniques assume that there are relationships between metrics. Many of these techniques use statistical machine learning. A common approach is to use metric correlations to quantify a monitored system. During detection, if the correlations between metrics becomes significantly different from the learned correlations, the system is classified to be in a faulty state [10, 24, 8, 12].

Other fault-detection techniques consider the order of events and use statistical methods based on times series. These include Markov-based methods [5, 6] and N-gram analysis [11]. Furthermore, other anomaly-detection techniques use statistical methods based on decision trees [7, 8] or combine statistical models into an ensemble detection approach [23].

3. THE ANIKETOS APPROACH

Aniketos employs an anomaly-detection approach and therefore can detect faults it has not seen before. However, unlike the previously discussed anomaly-detection methods, Aniketos does not use statistical machine learning. Instead, it takes a novel approach that uses computational geometry and works well independent of whether the metrics are correlated or not.

Figure 1 illustrates the Aniketos fault-detection technique, which consists of a training phase and a detection phase. During the training phase, Aniketos learns to recognize the normal behavior of the monitored system by constructing a geometric enclosure around good measurements. During the detection phase, Aniketos observes the application measurements and uses the enclosure to detect anomalous behavior. If Aniketos determines that the monitored system is in a faulty state, it triggers a fault warning signal. This signal can be sent to a human system administrator or to an automated fault mitigation system that will attempt to correct the fault.

3.1 Training Phase

During the training phase, the monitored system executes its normal behavior and Aniketos collects measurements at one second time intervals. Once enough training points are collected, they are grouped into a training data set. The collection of measurements at runtime continues until Aniketos has enough training data points to adequately represent the normal behavior of the monitored system. This training data set is used to construct an n-dimensional geometric enclosure, where n is the number of distinct metrics used (i.e., CPU time, heap memory, etc.).

The problem of knowing when enough training data has been collected depends on the system being monitored and is related to the testing adequacy problem [17]. If a system has an extensive test suite that represents its normal behavior, then the execution of that test suite will produce a good training data set. It may also be beneficial to create the training data set from observations collected on a deployed system. Doing so creates a training set that captures how the system is typically used. To collect a representative training data set from a deployed system, the training period should include the full spectrum of conditions under which the system normally operates. For example, this paper's case study uses a static mirror of the Drexel University Computer Science Department website. To capture normal system behavior, Aniketos collected measurements while a test system served requests that were played back from logs that were collected when the production web server was being used over a one week period. Because the amount and rate of HTTP requests varied with the days of the week, a full week of execution was used to collect a representative training data set.

3.2 Detection Phase

Once a geometric enclosure is constructed, it is used to detect faulty behavior. Each data point (i.e., n-tuple of n metric measurements) outside of the enclosure is labeled anomalous. Data points outside of the enclosure when the system is in a normal state are referred to as *false positives*. The percentage of false positives during a given time period is referred to as the false positive rate. In the case study the false positive rate was between 2% and 3%.

To avoid sending false fault-warning signals, Aniketos uses a time smoothing method analogous to a capacitor. The capacitor method keeps track of a charge value which is initialized to 0. Each time a point is classified as anomalous, the charge value is incremented by 1. Additionally, once the charge value is greater than 0, it begins to slowly discharge. If the charge grows larger than a threshold, a fault-warning signal is issued. If the occurrences of anomalies is low during a time period, the charge will not grow enough to trigger a fault warning. Because Aniketos samples at discrete intervals, the charge is approximated using the following equation:

$$C_t = \sum_{i=t-p}^{t} d^{t-i} c_i$$

In this equation C_t is the charge at time t, c_i is 1 if the data point at time i is outside of the enclosure and 0 otherwise, p is the number of successive data points considered, and d is the discount rate. The discount rate must be greater than 0 and less than 1.

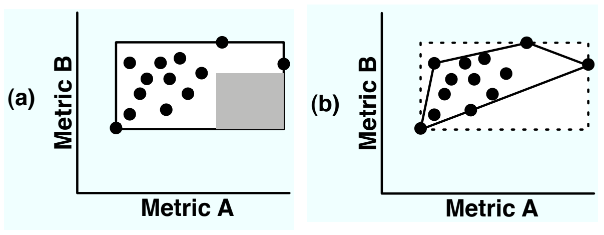

Figure 2: (a) presents the smallest rectangle constructed around a training data set. Note that the rectangle overestimates the actual shape of the normal space. (b) presents a convex hull constructed around the same training data set as in (a). Note that hull is a more accurate representation of the actual normal space.

3.3 Geometric Enclosures

The naïve method described in Section 2 generates a geometric enclosure that is a hyperrectangle in n-dimensional space where each dimension represents one of n metrics. For example, given 2 metrics, A and B, if the safe range for metric A is 5 to 10 and the safe range for metric B is 10 to 20 the normal behavior of the system can be represented as a 2-dimensional rectangle with the vertices (5,10), (5,20), (10,20), and (10,10). Any data point that falls within that rectangle, for example (7,15), is classified as normal. Any metric that falls outside of the rectangle, for example (15,15), is classified as anomalous. In practice, using n metrics results in n-dimensional enclosures embedded in n-dimensional Euclidean space.

Hyperrectangle enclosures often overestimate the normal operating region for an application, as demonstrated in Figure 2a. Better results can be achieved if a more compact enclosure containing the safe data points is used; for example, a convex hull, as demonstrated in Figure 2b. The convex hull of a point set is the smallest convex polygon that contains all of the points. A convex polygon is a polygon such that for every two points inside the polygon, the straight line segment between those points is also inside of the polygon. A convex hull for a set of points in one-dimensional space is a line segment between the largest and smallest values. In two-dimensional space one can think of each point as a peg on a board and the convex hull as the interior of a rubber band that snaps around all of those pegs. In three-dimensional space one can think of the convex hull as the interior of an elastic membrane stretched around points.

If there is a dependency between metrics then the convex hull will be smaller than the hyperrectangle and may produce better detection results. Any faults detected by the hyperrectangle will be detected by the convex hull because the convex hull is subsumed by the hyperrectangle. The convex hull may detect faults faster than the hyperrectangle but never slower and the convex hull may detect faults that the hyperrectangle cannot detect.

The convex-hull method also has advantages over existing methods that use only metric correlations. Those methods cannot make use of independent metrics, whereas the convex-hull method can make use of both dependent and independent metrics. Note that Aniketos does not require any prior knowledge that metrics are correlated. Correlation methods usually capture dependencies be-

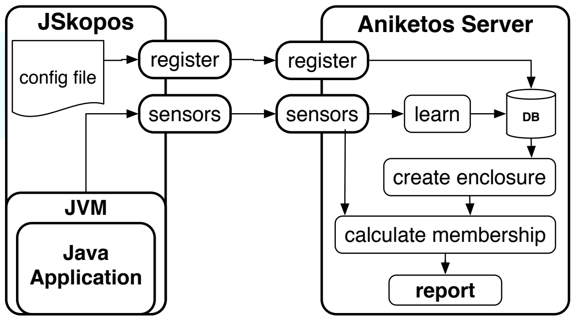

Figure 3: The current implementation of Aniketos has two components. JSkopos is a wrapper around Java applications that collects sensor measurements directly from the JVM using JMX. JSkopos registers with the Aniketos server and streams sensor measurements at a sample rate of one per second. The Aniketos server uses the measurement stream to learn the normal behavior of the system by constructing multidimensional geometric enclosures. Once the normal behavior is learned, the Aniketos server uses the incoming stream to classify the application being monitored as operating normally or not.

tween two or three metrics. Using the convex-hull method we have seen instances where fault detection relied on the dependencies between as many as five metrics.

If a correlation between a set of metrics is known to be useful for detection, it can be used as a dimension in the convex hull in addition to the dimensions that correspond to the other metrics.

4. ANIKETOS ARCHITECTURE

The Aniketos system comprises two components, the Skopos wrappers and the Aniketos server. The Skopos wrappers monitor applications and transmit collected measurements to the Aniketos server via the Skopos Protocol. The Aniketos server processes data and determines if the application being monitored is operating as expected.

Currently Aniketos provides JSkopos for monitoring applications running on the Sun Java Virtual Machine (JVM) [14]. Developers can provide sensors that are specific to the application being monitored and use the Skopos Protocol to create their own wrappers for any language.

Aniketos is designed to have minimal impact on monitored applications. The most computationally expensive operations, such as constructing multidimensional geometric enclosures, are handled by the Aniketos server, which can run on a separate machine. The Skopos wrappers must be executed on the same machines as the applications being monitored, and so they are developed to be lightweight processes. Depending on the usage patterns of NanoHTTPD, JSkopos adds anywhere from a 3.3% to 11.1% overhead with all sensors activated. When a single client sequentially requested a small text file thousands of times the overhead of JSkopos was 4.2% on average. This decreased to 3.3% on average when a large PDF file was requested. When the same experiments were run with 10 concurrent clients repeatedly requesting the same file, the overhead increased to 11.1% for the small text file and to 7.4% for the large PDF file. JSkopos automatically records when a new socket is created or closed. In NanoHTTPD, a new socket is opened for every HTTP request. As a result, sockets are opened and closed at a typically faster rate when smaller files are requested. This is evident in the increase in overhead when smaller files are requested.

In addition to the aforementioned benefits, Aniketos provides developers with control in choosing how their applications are monitored. While there may be some overlap, the set of sensors useful in monitoring a web server will be different from the set of sensors useful in monitoring an image-processing application. Developers are the ideal group to select the sensors most pertinent to their applications.

Figure 3 describes the interactions between the Aniketos server, the JSkopos wrapper, and a Java application being monitored. JSkopos uses Java Management Extensions (JMX) [13] to collect sensor measurements from the monitored JVM. After registering with the server, JSkopos streams the measurement data to the Aniketos server at a rate of one set of measurements per second. Currently, JSkopos is able to extract twenty-seven measurements for kinds of metrics. We chose the following seven most pertinent for monitoring a web server.

1. **Loaded Class Count**: Number of classes loaded in the JVM.
2. **Total Memory**: Amount of heap and non-heap memory used by the JVM.
3. **TCP Bytes Read and Written**: Number of bytes read and written using TCP across all sockets.
4. **TCP Accept Sockets Alive**: Number of accepted sockets open (clients connected to this application's server socket).
5. **CPU Time Average**: Average CPU time percentage across all threads executed by the JVM.
6. **Stack Depth Average**: Average stack depth of all threads currently running.
7. **Thread Count**: Number of threads currently running in the JVM.

The set of metrics had to be reduced to seven due to a limitation in the QHull [2] library used by Aniketos. Note that this is not a limit of the Aniketos detection technique. Many of the unused metrics were either irrelevant or redundant for the task of monitoring a web server. For example, UDP sensors are irrelevant because that protocol is never used by the web server being monitored. In addition, several CPU and memory metrics are highly correlated with one another and thus are redundant. The Aniketos technique can use as many metrics as the user wants, but currently it is artificially limited to seven metrics because of the limitations of the QHull library.

5. EXPERIMENTAL SETUP

To evaluate the technique presented in this paper a case study was conducted that involves monitoring NanoHTTPD, a Java-based web server. Figure 4 presents the design of the testbed used in this case study. One machine is used to host the Aniketos server and NanoHTTPD. and another machine manages clients that request resources from NanoHTTPD. JSkopos is used to monitor NanoHTTPD's execution and report measurements to the Aniketos server. The Aniketos server stores the gathered data and processes it using QHull.

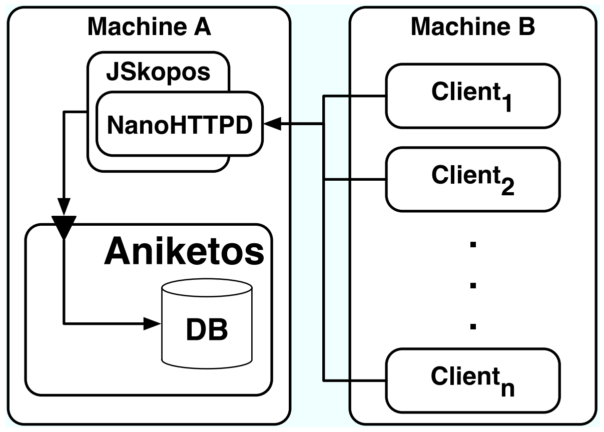

Figure 4: The experimental testbed contains two machines. The first is used to run Aniketos and NanoHTTPD. The second houses clients that access NanoHTTPD.

NanoHTTPD was chosen for this case study because it is open source and manageable in size, thus making it easy to modify and to inject with faults. It is a web server that hosts static content. NanoHTTPD spawns a new thread for each HTTP client request. If a thread crashes or goes into an infinite loop, it does not compromise NanoHTTPD's ability to serve other files.

The goal of this case study is to evaluate the performance of the Aniketos detection technique in a realistic scenario. To this end, it uses resources and access patterns of the Drexel University Computer Science Department 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 hyperrectangle and convex hull enclosures representing the normal operating space of NanoHTTPD. 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 thev fault injection experiments.

NanoHTTPD was injected with eight faults. These faults capture coding errors as well as security vulnerabilities and attacks. Two of the most common codding errors are the Infinite-Loop and the Infinite-Recursion faults. An Infinite-Loop fault is presented as a `while` loop that iterates indefinitely. Two versions of this fault were created. One in which each iteration of the loop does nothing, and the second in which each iteration of the loop performs a `sleep` operation for 100ms. The goal of the Slow-Infinite-Loop fault is to create a more realistic scenario in which an infinite-loop is not simply a drain on the CPU resource. Similar to the Infinite-Loop fault, the Infinite-Recursion fault also has two versions, a regular and a slow one that performs a `sleep` operation for 100ms. An infinite recursion is presented as a function calling itself until the thread running it crashes due to a Stack Overflow Error exception.

Another common fault injected into NanoHTTPD was the Memory-Leak fault. The fault performed a realistic memory leak and leaked strings containing the requested URLs by adding them to a `vector` stored in memory. The Memory-Leak fault doubled the size of the leak `vector` with each request.

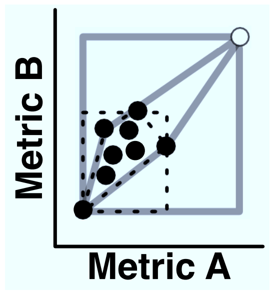

Figure 5: When a new point is introduced, it has a greater effect on the rectangle than the convex hull. Black points are the original points. White point is the most recently added point. Dashed lines represent the original enclosures. Solid lines represent new enclosures that include the new point.

Log-explosion [18] is another problem common to the server environment and was injected into NanoHTTPD. The log-explosion fault causes NanoHTTPD to continuously write to a log file until there is no more space left on the hard drive. While this does not cause the web server to crash, the Log-Explosion fault does cause a degradation in performance.

In addition to faults due to coding errors, two security attacks were perpetuated against NanoHTTPD. In the first, NanoHTTPD was injected with a spambot trojan [3]. Once triggered, the spambot began to send spam email message packets to an outside server at a rate of three emails per second. Each message was one of three spam messages chosen at random and varied in length between 166 and 2325 characters each.

The second attack perpetuated against NanoHTTPD was a Denial-of-Service (DOS) attack [16]. During the DOS attack several processes that continuously requested resources from NanoHTTPD were launched from two separate machines. The attack escalated with about 100 new processes created every second. It continued until NanoHTTPD could no longer process legitimate requests.

Each fault was triggered by a specific URL request. For example, the memory leak fault was triggered by accessing the "memleak" resource via a web browser. Each fault was triggered after a minimum of one minute of fault-free execution. Each experiment was performed on a clean system, meaning that the Aniketos server, JSkopos, and NanoHTTPD were all reset between each experiment. Each fault experiment was conducted five times, using a different segment of fault-free traffic each time. The entire case study took about three weeks to execute. Most of that time was spent executing fault-free experiments.

The following section contains the results of these experiments and demonstrates the relative superiority of the convex-hull method over the hyperrectangle method.

6. EXPERIMENTAL RESULTS

Two types of experimental results presented in this section. The first subsection presents the false-positive rates observed during the experiments and the second subsection presents performance of Aniketos with respect to fault detection. These experiments were designed to compare the performance of the convex-

Day	Number of Requests	Convex Hull False Positive Rate	Hyperrectangle False Positive Rate
Sunday	33,285	3.1%	2.39%
Monday	42,594	3.6%	2.79%
Tuesday	50,638	17.5%	2.70%
Wednesday	49,782	11.2%	2.90%
Thursday	44,544	19.0%	4.97%
Friday	29,718	47.0%	2.77%
Saturday	15,072	99.7%	2.27%

Table 1: False-positive results for enclosures constructed from seven individual twenty-four hour segments taken from one week of server traffic. Due to the variation in false-positive results, we concluded that a single day of measurements should not be relied on to construct an enclosure that represents normal system operation.

	Convex Hull			Hyperrectangle		
Number of Days	False Positive Rate	False Trigger Rate	False Trigger Periods	False Positive Rate	False Trigger Rate	False Trigger Periods
1	99.70%	99.62%	1	7.27%	4.94%	12
2	31.71%	28.50%	15	2.90%	2.46%	8
3	10.73%	8.50%	12	2.77%	2.40%	6
4	3.13%	2.59%	8	2.75%	2.40%	6
5	2.90%	2.47%	6	2.68%	2.38%	5
6	2.84%	2.45%	6	2.68%	2.38%	5
7	**2.58%**	**2.32%**	**3**	**2.35%**	**2.28%**	**1**

Table 2: False-positive results for multiple consecutive-day training sets. For all numbers of days n, all sets of n consecutive days were used. For example, for two-day data sets we built enclosures from Monday and Tuesday, Tuesday and Wednesday, Wednesday and Thursday and so on. The hyperrectangle approach yielded significantly better results than the convex hull in the experiments that use training data sets of one to three days. The two enclosures had similar performance in experiments that used four to seven day training data sets.

	Chi-Square	Mahalanobis	Hyperrectangle	Convex Hull
False Positive Rate	0.04%	0.03%	2.35%	2.58%
Fault	**Average Detection Time (sec)**			
Slow Infinite recursion	17	4	25	27
Fast Infinite Loop	0	0	26	15
DOS Attack	27	39	46	45
Log Explosion	5	No Detection	28	27
Memory Leak	No Detection	No Detection	218	53
Fast Infinite Recursion	No Detection	No Detection	No Detection	26
Slow Infinite Loop	No Detection	No Detection	No Detection	13
Spambot	No Detection	No Detection	No Detection	59

Table 3: Fault-detection results for enclosures constructed from a full week of measurements. Of the eight fault experiments, there are three (Slow Infinite Recursion, Log Explosion, and DOS Attack) where the convex-hull method does not significantly outperform the hyperrectangle method. The convex-hull method was able to detect three faults that the hyperrectangle method could not detect. Although the Chi-Square and Mahalanobis distance have better detection time for the faults that they successfully detect, Chi-Square fails to detect four of the faults and Mahalanobis distance fails to detect five of the faults.

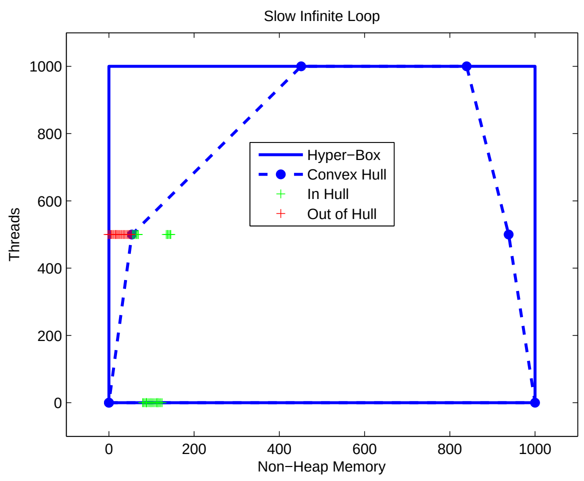

Figure 6: Because the Slow Infinite Loop can be detected using only two metrics, that detection process can be illustrated in a two dimensional figure. The convex-hull method has observed that when a new thread is created, memory is allocated to handle the request. Therefore, when a new thread is created and no new memory allocation is observed, the convex-hull method detects a fault.

hull method to the hyperrectangle method. In the worst case, it took 37 minutes to construct a convex hull and less than 1 second to construct a hyperrectangle. This is a reasonable amount of time to process seven days of measurement considering that processing is done offline and is a one-time setup requirement. In our case study we collected one sample per second and the worst case average detection time was 292 microseconds for the convex hull and 4.6 microseconds for the hyperrectangle. These rates are low enough to support a 1 second sample rate.

The fault-detection section includes results from two additional distance based approaches. Both of these approaches depend on a measure of distance from the mean of the training data. The first measure is the Chi-Squared statistic, which is defined as

$$\frac{(x-\mu)}{\mu}$$

where x is an observed value and μ is the mean of the training set. The second measure is the Mahalanobis distance, which is defined as

$$\sqrt{(x-\mu)^T S^{-1}(x-\mu)}$$

where x is an observed value, μ is the mean of the training data and S is the covariance matrix of the training data.

For both of these distance measures, the mean and covariance matrix are generated from one week of fault-free training data. We calculate a safe distance from the mean by finding the largest distance of any individual point in the training set from the mean. Any observed point beyond the safe distance is labeled as an anomaly.

The false-positive rate for the distance based approaches is determined by applying them to a second week of fault-free data.

Because the false-positive rates generated by the distance based approaches are low relative to the geometry based approaches, we do not apply a smoothing function.

6.1 False-Positive Rate Experiments

Because the convex hull lies inside the hyperrectangle the false-positive rates for the hyperrectangle will be no higher than the false-positive rates for the convex hull. Any data points that fall outside of the hyperrectangle must fall outside of the convex hull. Therefore, the convex hull false- positive rate cannot be lower than the hyperrectangle false-positive rate. The convex hull false-positive rate can be higher than the hyperrectangle false-positive rate because there may be regions inside the hyperrectangle that are not inside the convex hull.

Table 1 reports the false-positive results for enclosures constructed from seven individual twenty-four hour segments taken from one week of server traffic. The first column lists the days of the week, the second column shows the number of HTTP requests served each day of the week, and the third and forth columns show the percentage of non-fault data points that were outside of the enclosures. We expected individual twenty-four hour segments to be insufficient to capture the normal operating space of NanoHTTPD hosting the Drexel University Computer Science's website. This should have resulted in high false-positive rates for experiments using enclosures constructed from twenty-four hour segments.

False-positive rates for enclosures built using measurements from individual days varied from 3.1% to 99.7% for the convex hull and from 2.4% to 7.3% for hyperrectangle. The false-positive rates for the hyperrectangle were lower than the results for the convex hull. These results are due in part to the variance in the number of requests processed. On day 7 (Saturday) 15072 requests were processed, as opposed to day 3 (Tuesday) when there were 50638 requests. It is not surprising that the geometric enclosures built from the segment of measurements that contain the fewest requests produced the highest false positive rate when used to classify data points from a full spectrum of system behavior. However, the enclosures built from the segment of measurements that contain the most requests do not have the lowest false-positive rate. We concluded from these results that a single day of measurements should not be relied on to construct an enclosure that represents normal system operation.

Table 2 reports the false-positive results for multiple consecutive day training sets. The first column shows the number of days worth of data used to construct the geometric enclosures. For all numbers of days n, all sets of n consecutive days were used. For example, for two-day data sets we built enclosures from Monday and Tuesday, Tuesday and Wednesday, Wednesday and Thursday and so on. The results shown in Table 2 reflect the worst performance for any set of the corresponding number of days. For example 31.71% was the highest false-positive rate for any set of two days. We include the worst results in this table to illustrate the worst performance to be expected from randomly choosing n consecutive days as a training set. The false-trigger-rate column shows the percentage of time when the capacitor charge is over the threshold and would trigger a fault warning. The false-trigger-periods column shows the number of periods when the capacitor is over the threshold. A trigger period is a segment of time durning which the charge of the capacitor does not drop below the threshold for more than fifteen minutes. False triggers are caused by server behavior that the Aniketos fault-detection technique does not recognize and a false-trigger period represents a single stretch of time when Aniketos does not recognize the server behavior. A false-trigger period can be thought of as a single unrecognized event. The false-trigger period columns shows the number of unrecognized events and the false-trigger rate shows the percentage of time that these events covered. For example, for two day training data sets there were fifteen separate periods when the capacitor stayed over the threshold and these periods covered 28.5% of the experiment. These periods were separated by at least fifteen minutes of the capacitor staying below the threshold. For the one-day training data sets, the worst performing set had a single false-trigger period, but this period lasted for 99.62% of the experiment.

We expected seven days to be sufficient because we consider seven days to be a natural cycle of this website's usage patterns. Therefore we expected the false-positive rate to decline as more days were used for the construction of enclosures. The hyperrectangle approach yielded significantly better results than the convex-hull approach in the experiments that use training data sets of one to three days. The two enclosures had similar performance in experiments that used four to seven day training data sets. However, false-positive rates are only one part of evaluating a fault detection technique. Enclosures that produce lower false-positive rate, may not produce the best detection of true faults.

Another point that needs to be discussed is the variation in the false-positive rate between the hyperrectangle and convex-hull methods. The hyperrectangle method displayed less variation in the false-positive rate than the convex-hull method. This is because extreme points, those boundary points that define the enclosure, have a greater effect on the volume of the hyperrectangle than the volume of the convex hull. Figure 5 provides an example of this phenomenon. Notice how by introducing a single new point, the rectangle's area is effected significantly more than that of the convex hull.

6.2 Fault Detection Experiments

The results of the fault-detection experiments are listed in Table 3. All reported results are for enclosures constructed from a full week of measurements. Although the distance-based approaches provided faster detection of some faults, they failed to detect other faults that were detected by the geometric approaches. The difference in detection time is likely the result of the capacitor function applied to the geometric approaches.

We expected that the detection of some faults would depend on the values of a single metric while other faults would be detected because of dependancies between two or more metrics. Therefore, the convex-hull method should detect some faults as fast as the hyperrectangle method and others significantly faster.

The results documented in Table 3 demonstrate the promise of using the convex-hull method for fault detection. Of the eight fault experiments, there are three (Slow-Infinite-Recursion, Log-Explosion, and DOS-Attack) where the convex-hull method does not significantly outperform the hyperrectangle method. In all three cases a single metric was enough to determine the presence of a fault. Those metrics were Stack Depth Average in the case of the Slow Infinite Recursion fault, CPU Time Average in the case of the Log Explosion fault, and TCP Accept Sockets Alive or Thread Count in the case of the DOS Attack fault. In the remainder of the fault experiments, the convex-hull method outperformed the hyperrectangle method significantly.

The Memory-Leak and Infinite-Loop faults were detected by both the convex-hull and hyperrectangle techniques. However, the convex-hull technique detected the faults significantly faster than the hyperrectangle technique. This implies that even though there is a single metric that can be used as a predictor of the fault, a combination of metrics is more efficient for the detection of that fault.

To detect Memory-Leak and Infinite-Loop faults, the hyperrectangle depended on the Total-Memory and CPU-Time-Average metrics. This result was expected because each of these two faults strain a single resource and can, therefore, be detected by observing that resource. An interesting result in these experiments was that the convex-hull method detected the faults significantly faster by exploiting the relationship between metrics.

In addition to outperforming the hyperrectangle method in some cases, the convex-hull method was also able to detect three faults that the hyperrectangle method could not detect. These were the Slow-Infinite-Loop, Infinite-Recursion, and Spambot fault. It is clear that the relationship between metrics needs to be considered when detecting these faults. In the case of the Slow-Infinite-Loop fault, the Total-Memory metric and any one of the following metrics needed to be considered.

- Loaded Class Count
- TCP Accept Sockets Alive
- Stack Depth Average
- Thread Count

In the case of the Infinite-Recursion fault, the Total-Memory metric and any of the following metrics needed to be considered.

- Loaded Class Count
- TCP Accept Sockets Alive
- Stack Depth Average

In the case of the Spambot fault the relationship between five metrics had to be considered. Those five metrics were Loaded Class Count, Total Memory, TCP Bytes Read and Written, CPU Time Average, and either TCP Accept Sockets Alive or Thread Count.

Because the slow infinite loop can be detected using only two metrics, that detection process can be illustrated in a two-dimensional figure. Figure 6 demonstrates why the convex-hull method detected this fault while the hyperrectangle method did not. The fault is triggered by a URL request that causes NanoHTTPD to spawn a new thread. However, this thread does not handle the request and instead goes into a slow infinite loop. The convex-hull method has observed that when a new thread is created, memory is allocated to handle the request. Therefore, when a new thread is created and no new memory allocation is detected, the convex-hull method detects a fault. In contrast, the hyperrectangle method treats each metric independently and has previously detected both a state in which a new thread is created and a state in which memory allocation is minimal. Therefore, it incorrectly judges the faulty behavior as normal.

The Infinite-Recursion fault is similar in many respects to the Slow-Infinite-Loop fault. When the Infinite-Recursion fault is triggered, it quickly crashes the thread created to handle the request. This places NanoHTTPD in a state similar to the one observed during the Slow Infinite Loop fault, with one key exception. Once the thread crashes it is no longer alive and is therefore no longer counted by JSkopos. However, all other indicators, such as a live TCP accept socket, are present. Therefore, the convex-hull method is still able to detect the Infinite-Recursion fault.

Since the Spambot fault requires five metrics to be detected, it is impossible to visualize. However, it is interesting to note that that TCP Accept Sockets Alive and Thread Count can be interchanged. This is because they are both indicators that a new request is being processed.

The set of metrics used in the case study was chosen based on our opinion that these metrics would be sufficiently capture NanoHTTPD's execution. It is worth noting that each metric was necessary in detecting at least one of the faults presented in the case study. Overall, the case study demonstrates the advantages of the convex-hull method over the hyperrectangle method and the feasibility of a computational geometry approach to fault detection.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a novel approach to detection of runtime faults that uses methods based on computational geometry. We have implemented this approach in the Aniketos fault-detection system described in Section 4. We performed a case study, described in Section 5, and we have reported results, in Section 6, that demonstrate the promise of our approach.

We have shown that the Aniketos fault-detection technique is robust. Aniketos can use either a single metric or a variety of metrics which can be either dependent or independent. Aniketos can detect faults triggered by bugs (i.e., memory leak) or malicious attacks (i.e., denial of service attack). The Aniketos fault-detection technique is mathematically elegant. In concept it is easy to understand and it covers a range of fault-detection types in a single technique.

We conclude from our case study that the methods that assume independence among metrics will perform poorly for common faults. In addition, we conclude that the convex-hull method employed by Aniketos can capture relationships between metrics that can be used to detect some faults faster that methods that assume independence between metrics, and detect some faults that they cannot detect at all.

We plan to conduct experiments that compare our detection method to additional commonly used methods based in statistical machine learning. We also plan to experiment with including elements of correlation methods in our approach. For example, we would like to employ existing methods for detecting correlation between metrics and then include those correlations as metrics in our method. Currently we are limited to building convex hulls in seven dimensions. We would like to find a way around this technical limitation and test our method in higher dimensions. Finally, we will expand our case study to include different web servers and other kinds of system and application software.

8. ACKNOWLEDGEMENTS

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

9. REFERENCES

[1] Y. Al-Nashif, A. Kumar, S. Hariri, Y. Luo, F. Szidarovsky, and G. Qu. Multi-Level Intrusion Detection System (ML-IDS). In *Autonomic Computing, 2008. ICAC'08. International Conference on*, pages 131–140, 2008.

[2] C. Barber and H. Huhdanpaa. Qhull. *The Geometry Center, University of Minnesota, http://www.geom.umn.edu/software/qhull*.

[3] D. Barroso. Botnets–The Silent Threat. *European Network and Information Security Agency (ENISA)*, 2007.

[4] M. Brodie, S. Ma, G. Lohman, L. Mignet, M. Wilding, J. Champlin, and P. Sohn. Quickly finding known software problems via automated symptom matching. In *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 101–110, June 2005.

[5] M. Burgess. Probabilistic anomaly detection in distributed computer networks. *Sci. Comput. Program.*, 60(1):1–26, 2006.

[6] M. Burgess, H. Haugerud, S. Straumsnes, and T. Reitan. Measuring system normality. *ACM Trans. Comput. Syst.*, 20(2):125–160, 2002.

[7] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer. Failure diagnosis using decision trees. *Autonomic Computing, International Conference on*, 0:36–43, 2004.

[8] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. 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.

[9] J. Elonen. NanoHTTPD, 2007. `http://elonen.iki.fi/code/nanohttpd/`.

[10] Z. Guo, G. Jiang, H. Chen, and K. Yoshihira. Tracking probabilistic correlation of monitoring data for fault detection in complex systems. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 259–268, June 2006.

[11] G. Jiang, H. Chen, C. Ungureanu, and K. Yoshihira. Multi-resolution abnormal trace detection using varied-length n-grams and automata. *Autonomic Computing, International Conference on*, 0:111–122, 2005.

[12] M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. Ward. System monitoring with metric-correlation models: problems and solutions. In *ICAC '09: Proceedings of the 6th international conference on Autonomic computing*, pages 13–22, New York, NY, USA, 2009. ACM.

[13] H. Kreger. Java Management Extensions for application management. *Technology*, 40(1), 2001.

[14] T. Lindholm and F. Yellin. *Java virtual machine specification*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.

[15] McAfee. McAfee-Antivirus Software and Intrusion Prevension Solutions. `http://www.mcafee.com/us/`.

[16] D. Moore, C. Shannon, D. Brown, G. Voelker, and S. Savage. Inferring Internet denial-of-service activity. *ACM Transactions on Computer Systems (TOCS)*, 24(2):139, 2006.

[17] G. Myers, T. Badgett, T. Thomas, and C. Sandler. *The art of software testing*. Wiley, 2004.

[18] L. Peterson, A. Bavier, M. Fiuczynski, and S. Muir. Experiences building planetlab. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 351–366, November 2006.

[19] M. Roesch. Snort - lightweight intrusion detection for networks. In *LISA '99: Proceedings of the 13th USENIX conference on System administration*, pages 229–238, Berkeley, CA, USA, 1999. USENIX Association.

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

[21] Symantec. Symantec - AntiVirus, Anti-Spyware, Endpoint Security, Backup, Storage Solutions. `http://www.mcafee.com/us/`.

[22] G. Vigna and R. A. Kemmerer. Netstat: a network-based intrusion detection system. *J. Comput. Secur.*, 7(1):37–71, 1999.

[23] S. Zhang, I. Cohen, M. Goldszmidt, J. Symons, and A. Fox. Ensembles of models for automated diagnosis of system performance problems. *Dependable Systems and Networks, International Conference on*, 0:644–653, 2005.

[24] Y. Zhao, Y. Tan, Z. Gong, X. Gu, and M. Wamboldt. Self-correlating predictive information tracking for large-scale production systems. In *ICAC '09: Proceedings of the 6th international conference on Autonomic computing*, pages 33–42, New York, NY, USA, 2009. ACM.