Reverse Engineering Utility Functions using Genetic Programming to Detect Anomalous Behavior in Software

Sunny Wong, Melissa Aaron, Jeffrey Segall, Kevin Lynch, and Spiros Mancoridis

Department of Computer Science
Drexel University
Philadelphia, PA, USA
Email: {sunnyw, mea36, js572, kev, spiros}@drexel.edu

***Abstract*—Recent studies have shown the promise of using utility functions to detect anomalous behavior in software systems at runtime. However, it remains a challenge for software engineers to hand-craft a utility function that achieves both a high precision (i.e., few false alarms) and a high recall (i.e., few undetected faults). This paper describes a technique that uses genetic programming to automatically evolve a utility function for a specific system, set of resource usage metrics, and precision/recall preference. These metrics are computed using sensor values that monitor a variety of system resources (e.g., memory usage, processor usage, thread count). The technique allows users to specify the relative importance of precision and recall, and builds a utility function to meet those requirements. We evaluated the technique on the open source Jigsaw web server using ten resource usage metrics and five anomalous behaviors in the form of injected faults in the Jigsaw code and a security attack. To assess the effectiveness of the technique, the precision and recall of the evolved utility function was compared to that of a hand-crafted utility function that uses a simple thresholding scheme. The results show that the evolved function outperformed the hand-crafted function by 10 percent.**

***Index Terms*—autonomic computing, utility function, genetic programming, software fault tolerance**

I. Introduction

Recent published research [1]–[4] describes the promise of using utility functions to detect software faults and anomalous behavior at runtime. These functions often are based on various resource usage metrics (e.g., memory usage, processor usage, thread count) and are often derived by imposing a lower and an upper threshold on the values of each metric. With the growing popularity of self-healing autonomic systems [5], utility functions play a critical role in detecting and anticipating software faults. In addition, the security needs of different systems present different requirements on the behavior of a utility function. In a system that demands high security, the utility function must achieve a high recall (i.e., fewer undetected faults), even if a large number of false alarms are reported. Other systems, however, may require high precision (i.e., fewer false alarms) as a convenience to the system administrators, even if some faults are undetected.

Every software system has a resource usage signature that can be defined by a set of utility functions highlighting different characteristics of the system. Because it is difficult to know these signatures *a priori*, it is necessary to reverse engineer the signature using empirical resource usage measurements. However, it remains a challenge to construct a utility function that achieves both high precision and high recall without using automation. To address this issue, we propose a technique that uses evolutionary computing to automatically construct a utility function for a specific system, set of resource usage metrics, and precision/recall preference.

A difficulty in constructing a useful utility function stems from the large number of available resource usage metrics. While some metrics are effective in detecting some types of faults, others metrics offer no apparent indication of failure. Selecting an appropriate set of sensors to monitor the behavior of a specific system presents a challenge. Due to the large number of possible metrics that can be computed from the values of these sensors, reporting a fault whenever any single metric exceeds a pre-defined threshold leads to many false alarms. On the other hand, the large number of metrics may result in waiting for all metrics to exceed their threshold, thus possibly leaving actual faults undetected. Several approaches have used domain experts to select the most appropriate metrics, based on the target application, in building a utility function. However, such an approach is tedious, costly, and error-prone, making it difficult to manually construct utility functions for a large number of systems, especially ones with specific precision/recall trade-offs.

To address the issue that manually constructed utility functions are often inefficient or ineffective, we present a technique that uses genetic programming (GP) [6] to automatically compute a utility function from a set of metric data. GP has been used to evolve computer programs and mathematical functions to solve various optimization problems. Given a set of resource usage metric values for the normal behavior of a system and a set of metrics for the system in a faulty or failing state, our technique derives a utility function that biases towards precision and recall as desired.

Our technique uses the resource usage threshold values as the building blocks for constructing a utility function, combining various metrics and finding the ones that best achieves the desired precision and recall requirements. Through genetic programming, the candidate utility functions that perform the best are combined to produce new combinations of metrics. We evaluate our technique on the open source Jigsaw[1] web server, using ten resource usage metrics and five hand-crafted software faults and security attacks on the system. To assess the success of our technique we compare the performance of the resulting utility function with a simple threshold function. Our results show that our approach was able to construct a utility function that outperforms the threshold function by about 10 percent.

The rest of the paper is organized as follows: Section II reviews related work and how our technique differs from existing approaches. Section III describes our utility function evolution technique. Section IV presents our evaluation strategy and results. Section V discusses our results and threats to validity. Section VI concludes the paper.

II. Related Work

In this section, we review related work and differentiate our technique from existing approaches.

A. Utility Functions

Self-healing and self-protection are two self-* properties presented by Kephart and Chess [5] in their paper describing autonomic computing. Self-healing systems can automatically detect and repair hardware and software problems by monitoring their own performance. Self-protection is the ability to defend oneself from malicious attacks or software faults. Both self-healing and self-protecting systems often rely on utility functions when determining system state. A utility function maps a set of values (i.e., processor usage, thread count, memory usage) to an ordinal value that characterizes the extent to which a system is not failing. In this paper, we present a technique that constructs utility functions automatically.

Walsh et al. [7] show how utility functions can replace an administrator and enable multiple autonomic systems to optimize resource allocations in dynamic environments. They showed that utility functions can translate quality of service needs, constructed at the business level, to dynamic resource allocation. Using this, an autonomic system can determine the best way to reach a desired state. While utility functions have their advantages, they can become increasingly more complex as systems scale in size. This problem does not occur when using genetic programming because the natural selection process efficiently determines the best solution.

Various researchers (e.g., Jiang et al. [1], deGrandis and Valetto [2], Archarya and Kommineni [3], Stehle et al. [4], Burgess et al. [8], Garlan et al. [9]) have studied the use of utility functions in detecting software faults. For example, Archarya and Kommineni [3] proposed an approach for filtering redundant and irrelevant threshold values, and use domain experts to manually remove the remaining irrelevant resource usage metrics. Jiang et al. [1] show that using metric-correlation models, such as the generalized least squares regression, can lead to better detection of errors than the current linear metric models. Principal component analysis [10] has also been applied to identify the most relevant resource metrics for detecting software faults. These papers corroborate our hypothesis that current methods for obtaining utility functions can be improved upon. Additionally they show that these functions can be extracted automatically. Our approach uses evolutionary computing to automatically construct a utility function without domain expertise. We can therefore apply it to commercial off-the-shelf (COTS) components, where we are not experts on the design or implementation of the software.

B. Genetic Algorithms/Programming

The work presented in this paper is part of an emerging research area called *search-based software engineering* (SBSE) [11]. SBSE involves applying search-based techniques such as genetic algorithms, genetic programming, and simulated annealing to automate various aspects of software engineering processes including cost estimation and planning, requirements analysis, testing, deployment, as well as the maintenance and evolution of legacy systems. Our research group has applied search-based methods to solve various software engineering problems, most notably in the areas of automatic modularization [12], autonomic computing [13], software forensics [14], and network application detection [15].

In the context of autonomic computing, there is a growing interest in using search-based methods. For example, Ramirez et al. [16] use genetic algorithms to show that reconfiguration can be done at runtime. Systems that are non-self-reconfiguring must either allow for downtime to respond to environmental changes or must be designed with reconfiguration strategies. These approaches are not optimal because failsafe systems cannot afford to have downtime and not all reconfigurations can be anticipated at design time. Ramirez et al.'s genetic algorithm uses system and environmental monitoring such that the algorithm is directly affected by changes in the environment. While their work uses genetic algorithms to create new configurations based on environmental changes, the intent of our approach is not to derive optimal configurations for self-adaptation, but rather to construct utility functions for detecting software faults in self-healing systems.

In another example of using search-based methods in the area of autonomic computing, Shevertalov et al. [13], represent measurements in an n-dimensional space and use genetic programming to combine metrics and reduce the dimensionality of their measurements while minimizing the loss of information. Instead of using one metric per dimension, each dimension is evolved to be an arithmetic expression that is a function of multiple metrics. These evolved dimensions are then used to determine whether a measurement is in a "safe" or "unsafe" region. They show that these expressions perform better than naïvely picking a subset of the metrics.

[1] http://www.w3.org/Jigsaw/

They compare their generated expressions to a subset of the metrics selected by a genetic algorithm, a subset hand-picked by a software engineer, and a subset hand-picked by a system administrator. Our approach, instead, evolves a utility function that is directly used to determine whether a fault is occurring rather than minimizing metric dimensionality.

III. Technique

In this section, we describe a motivating example and an overview of our technique of using evolutionary computing to generate utility functions. We also describe background concepts in evolutionary computing and the monitoring facilities in the Java Virtual Machine (JVM).

A. Motivating Example

Alice is a manager in an organization that is using a COTS web server as the platform for their software product. Since she does not fully know the security capabilities of the web server, Alice is concerned someone may exploit the server with malware (e.g., turn it into a spam-bot). However, her organization cannot afford to develop and maintain their own web server. To address this issue, Alice employs a system to monitor the resource usage of the web server. This monitoring system uses a utility function that reports suspicious behavior based on resource usage metrics. For example, if Alice's server turns into a spam-bot and starts using a large number of threads to send spam, the monitoring system will detect the increase in thread count and report this potential problem to Alice.

However, using simple threshold values for detecting malicious behavior may be insufficient. For example, the number of threads in a system may suddenly increase due to a dramatic increase in the number of clients, so detecting threshold violations in individual sensors may lead to a large number of false positives. However, it is difficult to manually determine which set of sensors are most important in detecting anomalous behavior.

To apply our technique, Alice first executes the normal behavior of her software and records the resource usage data. Then she runs the software with seeded malware to record changes in sensor values. She runs our utility function evolution algorithm to automatically generate a utility function based on the correlation of the sensors to the malware, and embeds this function into the monitoring system. Whenever the running system fails to satisfy this utility function, the system reports it as a potential software fault. Alice only needs to evolve the utility function once, offline, every time the system's resource usage changes.

B. Overview of Technique

Figure 1 shows an overview of our utility function construction approach. Our approach uses two test suites, which execute realistic behavior of the target system, as input. One of the test suites, denoted as Test Suite A in the diagram, is executed on the target system to produce a threshold function involving all desired resource usage metrics. The threshold function checks if each resource usage value is within an anticipated minimum and maximum range. The threshold values in this utility function are used by the GP as building blocks for constructing new utility functions. The set of resource usage values from running this test suite, denoted as Clean Metrics in the diagram, is then used by the GP to determine if the constructed utility function reports any false alarms (false positives).

For instance, going back to our motivating example, possible software faults for a web server may occur due to a denial of service attack or faulty code that causes a memory leak. As such, it is worthwhile for Alice to introduce these types of faults into the server for the purposes of obtaining data on the expected behavior resulting from such attacks. A second test suite, denoted as Test Suite B in the diagram, is executed on the target system with the seeded software faults or anomalous behavior to produce a set of resource usage values for when the system is in a faulty state, denoted as Faulty Metrics in the diagram. The GP uses these metric values to see if a constructed utility function fails to detect a faulty state (false negative).

With metric values for the target system in both stable and faulty states, our technique uses GP to construct a utility function for detecting the faulty state. Our GP generates a logical expression, or *predicate*, as a utility function. This predicate, can be embedded into a runtime monitoring system to detect anomalous behavior as they occur.

The goal of our GP is to provide tighter constraints on the bounds of normal operation. Considering the resource usage metrics as an n-dimensional space, the simple threshold function encloses the majority points of the system stable state in an n-dimensional hyper-rectangle. However, such a region may enclose the majority of points in the stable state but may also include a large number of points from the faulty state. To improve upon this, Stehle et al. [4] construct a convex hull around the n-dimensional points. However, this technique may still include a large number of points from the faulty state. Our complex logical expressions also improve on the hyper-rectangle by forming a tighter fit around the points of the stable state. These expressions may potentially form convex hulls represented by half-plane intersections, tighter concave polygons, or an unclosed shape. In doing so, we aim to achieve better accuracy in detecting faulty states.

C. Monitoring

In order to collect resource usage values, we use the Managed Beans, or MBeans, framework provided by the Java HotSpot virtual machine. The MBeans framework provides the ability to monitor an existing Java application without the need to make changes to source code, known as *passive sensors*. MXBeans are Java objects that represent managed resources (e.g., heap memory usage, daemon thread count, number of loaded classes) in the MBeans framework. This non-intrusive framework allows for auditing of the following resources:

- *Loaded Class Count*: This sensor reports the number of classes currently loaded in the JVM.

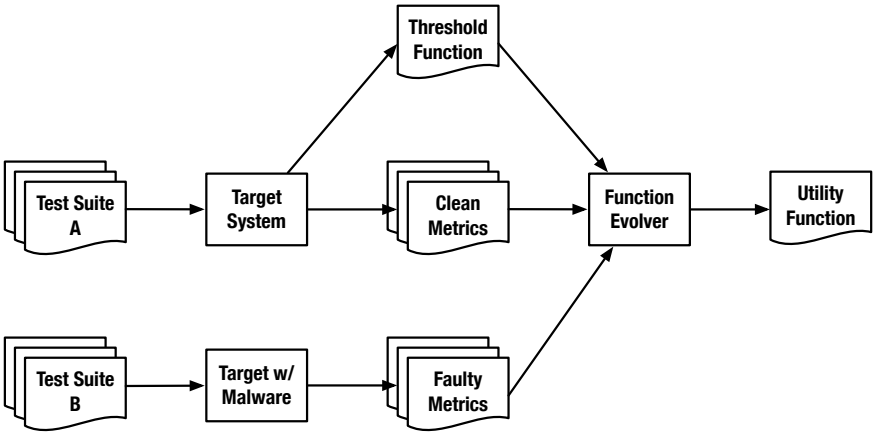

Fig. 1. Overview of Technique

- *Unloaded Class Count*: This sensor reports the number of classes unloaded since the start of the the JVM.
- *Total Class Count*: This sensor reports the number of classes loaded since the start of the the JVM.
- *Heap Memory*: This sensor reports the kilobytes of heap memory currently used.
- *Non-Heap Memory*: This sensor reports the kilobytes of non-heap memory currently used.
- *CPU Time*: This sensor reports the number of nanoseconds each live thread has executed. We record this sensor data as the total amount for all live threads.
- *User Time*: This sensor reports the number of nanoseconds each live thread has executed in user mode. We record this sensor data as the total amount for all live threads.
- *Stack Depth*: This sensor reports the stack depth of all live threads. We record this sensor data as the average stack depth of all live threads.
- *Thread Count*: This sensor reports the number of live daemon and non-daemon threads.
- *Daemon Thread Count*: This sensor reports the number of live daemon threads.

For each of these resource usage metrics, we keep track of the minimum, maximum, and average values. We also compute the change in each resource usage metric between each time period of monitoring, and record the minimum, maximum, and average of these changes.

Certain sensors intuitively seem more significant than others in detecting faulty state. For example, the rapidly changing and periodic nature of heap memory usage, as depicted in Figure 2, suggests that this resource's usage information may be difficult for detecting abnormal behavior. On the other hand, the number of daemons threads, as depicted in Figure 3, shows a steady trend that may be easilyer for detecting faulty state. However, the lack of upper bound on the number of possible daemon threads simultaneously running limits the effectiveness of this sensor as well. Although the behavior of these resources may suggest their effectiveness in detecting faulty state, their actual significance may greatly differ.

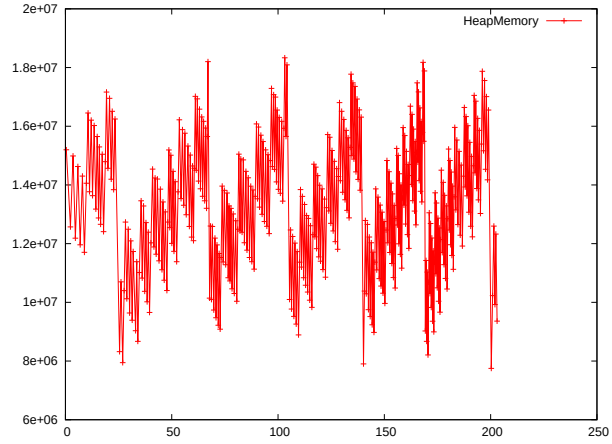

Fig. 2. Example Heap Memory Usage

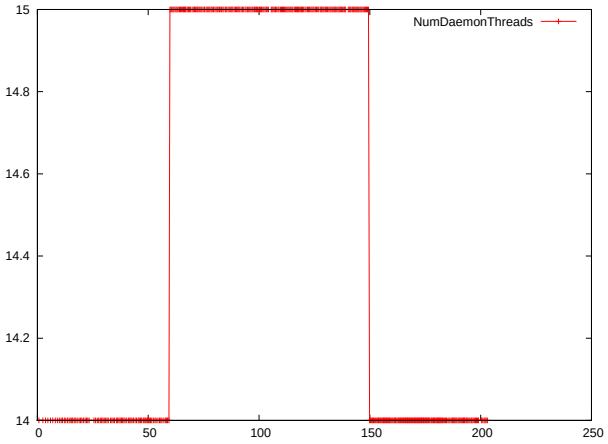

Fig. 3. Example Daemon Thread Count

D. Genetic Programming (GP)

Genetic algorithms (GAs) [17] describe a class of search algorithms based on evolutionary biology. A solution to a given problem is determined by evolving individuals from a pool of possible solutions based on a definition of solution

fitness. During each iteration, individuals are selected for reproduction and form the next generation of possible solutions. Genetic programming (GP) [6] expands upon the methodology of genetic algorithms by using evolutionary theory to create computer programs. Unlike genetic algorithms, where chromosomes can have a multitude of representations, genetic programming chromosomes are tree structures, representing mathematical expressions. Initial populations are usually randomly generated and the algorithm for generating these individuals can significantly impact the GA's performance.

During each iteration, the individuals with the greatest fitness, and a random sampling of less fit individuals are selected to form the next generation of solutions. Reproduction is handled through crossover and mutation. During crossover, selected individuals are paired into parents that then produce a child. The child shares traits with its parents, but is often different. Several techniques exist for performing crossover, including variations on the number and positions of points selected and variations on the expected chromosome length of children based on chromosome length of parents. Mutation occurs randomly and introduces an element of randomness into each generation. This randomness helps avoid getting stuck at local minima and maxima as well as provides the only outlet for the removal of unnecessary traits from the population once they have been introduced. Removing traits allows the resulting solution to be more efficient. Furthermore, mutation alleviates issues that arise when populations become too similar. Through successful reproduction, new generations are expected to obtain increasing fitness, as the most beneficial traits are passed down.

A GP process can be terminated when a solution attains a desired fitness level, when no further improvements are being made (the variance in fitness values approaches zero), or when a predetermined amount of time or generations has elapsed. At this point, the solution with the highest fitness level becomes the solution to the individual problem. There are a wide variety of applications for which genetic algorithms are appropriate, including many search and optimization problems. In these cases, genetic algorithms are often useful for approximating solutions for otherwise unsolvable problems.

As with most GP frameworks, chromosomes are represented by tree structures with each node being a piece of a logical expression. Logical expressions can take the form of binary expressions or single terms. The initial population is generated randomly, while ensuring that trees do not exceed a preset maximum depth. Each expression follows the grammar as defined in Figure 4.

Furthermore, our GP employs an island model [18]. With this model, we consider the evolutionary process taking place on many isolated islands. The process continues as described above and each island evolves independently. In the interest of diversity, occasionally and at random, an island will send a copy of its best individual to another island and will replace its least fit individual with the best individual from another island. Such an algorithm increases the probability of including all of the best traits from each isolated evolutionary process, assuming that in isolation, important traits may be unintentionally removed from the gene pool.

$$
\begin{array}{rcl}
\langle expr \rangle & \Rightarrow & \langle expr \rangle \ \langle logic\_op \rangle \ \langle expr \rangle \\
& | & \langle term \rangle \\
\langle logic\_op \rangle & \Rightarrow & \texttt{and} \\
& | & \texttt{or} \\
& | & \texttt{xor} \\
& | & \texttt{nand} \\
& | & \texttt{nor} \\
\langle term \rangle & \Rightarrow & \langle var \rangle \ \langle comp\_op \rangle \ number \\
\langle comp\_op \rangle & \Rightarrow & < \\
& | & > \\
& | & \leq \\
& | & \geq \\
\langle var \rangle & \Rightarrow & ident \\
& | & \Delta \, ident
\end{array}
$$

Fig. 4. Predicate Grammar

Not only does the island model provide benefits from a strictly biological viewpoint, it also provides a method for the simple parallelization of GP. Using this methodology, our algorithm can be easily parallelized not only to multiple threads on the same machine, but also across multiple machines, to achieve results faster.

1) Crossover: New generations are created by choosing two parents from the current population and swapping a random subtree from each parent's expression tree. We note that only compatible subtrees can be swapped. In other word, only subtrees with the same type in the grammar can be swapped. The creation of a new generation propagates the best features of the current generation, while at the same time creating a potentially diverse population.

2) Mutation: In order to introduce randomness into the evolutionary process, we use a system of expression mutation. With a degree of probability, any generation may undergo one of three mutations. These mutations include the addition or deletion of a subtree or the random changing of expression terminals. The addition of mutations into the algorithm adds another level of defense against stagnation at local minima and maxima, and also allows for the evaluation of more efficient traits by the deletion of unnecessary or duplicate genes.

3) Fitness: Termination of the algorithm occurs when a logical expression meets a defined fitness value. In our evaluation, we define our fitness function as follows:

$$F_\beta = \frac{(1+\beta^2) \cdot pos}{((1+\beta^2) \cdot pos + \beta^2 \cdot fneg + fpos)}$$

The above function, also known as the F_β score [19], is a weighted function of both false positives ($fpos$) and false negatives ($fneg$) compared to the number of true positives (pos). The minimization of both of these discrepancies yields a greater fitness for anomalous behavior detection. The F_β score falls within the range of 0 to 1—with a fitness of 0

representing a poor solution to a problem and a fitness of 1 representing an excellent solution.

The value of β is configurable based on precision/recall requirements of the system's security. Running with $\beta = 1$ weighs false positives and false false negatives equally. This β value compromises between convenience and security. However, β can be varied depending on the specific security needs of each individual application. Choosing $\beta < 1$ gives a preference to false positives over false negatives. On the other hand, choosing $\beta > 1$ gives a preference to false negatives over false positives. For example, an application that can sacrifice security for the convenience of fewer false positives may opt to choose $\beta = 0.5$, which prefers fewer false positives at twice the weight of fewer false negatives. On the contrary, an application that demands high security may opt to choose $\beta = 2$, to ensure higher recall.

IV. Evaluation

In this section, we present an evaluation of our utility function construction technique.

A. Evaluation Strategy

For the evaluation, we applied our technique to the open source Jigsaw web server, which is the W3C's Java-based web server platform. The latest version, 2.2.6, contains about 850 files and over 100 KSLOC. For our evaluation, we ask the following questions:

- Q1: Does the automatically constructed utility function outperform a hand-crafted threshold function in detecting faulty state?
- Q2: Does varying the value of β produce utility functions that use different resource usage metrics?

To answer these questions, we developed two realistic test suites and generated utility functions for three different β values. Our evaluation used ten resource usage metrics and five hand-crafted faults and attacks on Jigsaw.

The following subsections describe the test suites, the simulated anomalous behaviors, the GP implementation, and the evaluation results.

B. Evaluation Environment

To evaluate our technique, we gathered resource usage measurements from a realistic scenario. Our evaluation environment used resources and access patterns from the Drexel University Computer Science Department website.[2] Nine weeks worth of website access logs were collected and all resources accessed in those logs were extracted and converted into static HTML pages. Jigsaw was configured to serve the retrieved pages. Out of the nine weeks of logs, we randomly selected contiguous time periods for traffic workflow generation in our case study.

To establish a baseline for normal activity, a 30 hour time period was randomly selected to replay traffic to our site without any faults running (Test Suite A). For each of the five faults and attacks on the server, a 10 hour time period was randomly selected to be replayed while the fault was activated. These were replayed against the statically hosted version of the website and provided the case study with realistic workload access patterns (Test Suite B). Only one fault was active for each scenario. After each time period finished, the server was restarted.

[2]http://www.cs.drexel.edu/

C. Anomalous Behavior

By using passive sensors, we did not alter Jigsaw's source code nor needed to gain domain expertise of its implementation. To invoke the faulty behavior we manually injected a thread into Jigsaw that listened on a network socket for the specific fault to run. To avoid having this additional thread and socket skew the usage metric values and keep the same baseline between the faulty and non-faulty versions of Jigsaw, we also ran this thread in the non-faulty version of Jigsaw. Below, we describe each injected faults and security attacks.

The five attacks and faults on the Jigsaw server intended to either deny Jigsaw the ability to perform its normal hosting functions or add additional, malicious functionality to the running server. The following are all common attacks and software faults faced by web servers and, thus, were prime candidates for examining anomalous behavior exhibited by a server in distress.

a) Denial of Service: A denial of service attack floods a server with false requests so that the server is too busy to handle legitimate client requests. Our attack continuously sends a large number of HTTP GET requests to the server.

b) Infinite Loop: Faulty software components occasionally enter into an infinite loop. To create such a scenario, we implemented a component that runs in an infinite loop, generating a pseudorandom number per iteration. We generate a random number in each loop iteration to prevent the optimizing compiler from removing our loop.

c) Log File Explosion: Like many web servers, Jigsaw provides feature to log error information; for example, if a component encounters a programmatic exception, it can record the error information to a file. Exploiting this feature, we implemented a component that simulates faulty code that continuously encounters an exception and logs the error. The resulting scenario is one in which the log file rapidly grows in size and fills the available disk space. Our component creates a deeply nested exception and, in an infinite loop, writes the exception's stack trace to the log file.

d) Memory Leak: Although the JVM uses a garbage collector to reclaim unused memory, memory leaks still occur when references are retained to objects that are no longer used. While approaches have been proposed to identify such scenarios (e.g., Xu and Rountev [20]), memory leaks can still accumulate and hinder the performance of a system. To create a memory leak software fault, we implemented a component that continuously requests a 10KB block of heap memory every second. The memory is referenced via a *soft reference*, which allows the JVM to reclaim some of the leaked memory when memory becomes scarce. Memory usage is

hence kept high without running out of memory completely. Additionally, by checking for an `OutOfMemoryError`, the faulty component keeps itself from crashing.

e) Recursion: Incorrectly implemented recursive algorithms often enter a state where the recursive base case is never satisfied and recursion occurs indefinitely (until the maximum stack depth is reached). To create such a faulty component, we implemented a recursive function that continues recursing until the maximum stack depth is reached (i.e., by checking for a `StackOverflowError`). After reaching the maximum allowable stack depth, our component runs in an infinite loop to ensure the fault persists and the stack memory is not released.

D. Threshold Predicate

Figure 5 shows the threshold predicate determined by using the data obtained from running 30 hours of test data and observing resource usage under normal operation. Using this utility function, a monitoring system would have a 18.609% false negative rate for a F_1 fitness value of 0.914874. This false negative rate indicates that the utility function was unable to detect all of the faults and attacks.

E. Evolving Utility Functions

This subsection describes the configuration of our experiments and the evaluation results.

1) Experimental Setup: Our GP is run with four islands, each with 256 individuals. During execution there is a 5% migration rate to other islands and a 5% mutation rate. A maximum of 40 generations are evolved and the algorithm terminates when either this maximum number of generations is reached, a solution reaches a maximum fitness of 1, or the standard deviation of the fitness in a single island is less than the smallest representable value of a floating point number.

The population is initialized using a modified version of the PTC1 algorithm proposed by Luke [21]. Individuals are randomly generated given an expected and maximum tree depth. With a given probability, computed as described by Luke, each node in the expression tree is chosen to be either a terminal or a non-terminal. Both of these choices are simple in our approach as the grammar used by our genetic program has only one type of non-terminal. If a node is chosen to be a non-terminal, each expression within the non-terminal's production is recursively generated until the tree is completed or reaches its maximum depth (in our case 16).

Our algorithm also ensures that the outcome of the GP performs no worse than the threshold predicate. During initialization the threshold predicate is inserted, as an individual, into the population of each island. During each stage of evolution, our algorithm employs an elitism-one approach, ensuring that the most fit individual from each generation is propagated to the next generation. This ensures that best fitness of the population is monotonically increasing.

2) Results: For each of three β value, 1, 0.5, and 2, we ran a GP four times. Each GP ran for 8-12 hours before stopping. Figures 6, 7, and 8 show the best automatically constructed utility functions from these experiments. The F_β fitness function scores for the evolved predicates are shown in Table I. For each of these functions, all numerical values are shown as percentages of the resource usage values in the threshold predicate. Each value ranges from 0, representing the minimum value for the given metric in the threshold predicate, to 1, representing the maximum value for the given metric in the threshold predicate. We use the standard mathematical symbols of $\oplus$ for `xor`, $\mid$ for `nand`, and $\perp$ for `nor`.

$$
\begin{aligned}
& usr\_time\_minimum = 0 \\
\wedge\ & \Delta usr\_time\_minimum = 0 \\
\wedge\ & 16 \times 10^7 \leq usr\_time\_maximum \leq 12338 \times 10^7 \\
\wedge\ & 0 \leq \Delta usr\_time\_maximum \leq 11944 \times 10^7 \\
\wedge\ & 2564102.56 \leq usr\_time\_average \leq 2634125 \times 10^3 \\
\wedge\ & 0 \leq \Delta usr\_time\_average \leq 1528417721.52 \\
\wedge\ & -1 \leq cpu\_time\_minimum \leq 0 \\
\wedge\ & 0 \leq \Delta cpu\_time\_minimum \leq 1 \\
\wedge\ & 43 \times 10^8 \leq cpu\_time\_maximum \leq 810171 \times 10^7 \\
\wedge\ & 0 \leq \Delta cpu\_time\_maximum \leq 407234 \times 10^7 \\
\wedge\ & 60512820.51 \leq cpu\_time\_average \leq 166170506329.12 \\
\wedge\ & 382992.53 \leq \Delta cpu\_time\_average \leq 100450742088.61 \\
\wedge\ & 1449 \leq classes\_loaded \leq 1579 \\
\wedge\ & 0 \leq \Delta classes\_loaded \leq 47 \\
\wedge\ & 0 \leq classes\_unloaded \leq 8 \\
\wedge\ & 0 \leq \Delta classes\_unloaded \leq 4 \\
\wedge\ & 1451 \leq classes\_total \leq 1571 \\
\wedge\ & 0 \leq \Delta classes\_total \leq 45 \\
\wedge\ & 4.19 \leq stacks\_minimum \leq 4.64 \\
\wedge\ & 0 \leq \Delta stacks\_minimum \leq 0.37 \\
\wedge\ & 4.56 \leq stacks\_maximum \leq 31.56 \\
\wedge\ & 0 \leq \Delta stacks\_maximum \leq 0.62 \\
\wedge\ & 4.53 \leq stacks\_average \leq 5.46 \\
\wedge\ & 0 \leq \Delta stacks\_average \leq 0.87 \\
\wedge\ & 8380288 \leq memory\_heap \leq 512443816 \\
\wedge\ & 0 \leq \Delta memory\_heap \leq 495101608 \\
\wedge\ & 18670168 \leq memory\_non\_heap \leq 21301632 \\
\wedge\ & 0 \leq \Delta memory\_non\_heap \leq 277880 \\
\wedge\ & 78 \leq threads\_total \leq 81 \\
\wedge\ & 0 \leq \Delta threads\_total \leq 1 \\
\wedge\ & 72 \leq threads\_daemons \leq 75 \\
\wedge\ & 0 \leq \Delta threads\_daemons \leq 1
\end{aligned}
$$

Fig. 5. Threshold Predicate

TABLE I
F_β FITNESS FUNCTION SCORES FOR THE EVOLVED PREDICATES

β value	F_β score	$fpos$ rate	$fneg$ rate
0.5	0.996344	0.000%	1.835%
1	0.998327	0.000%	0.335%
2	0.998786	0.539%	0.016%

After executing the GP, the best utility function we found is shown in Figure 6. The automatically constructed utility func-

tion was able to improve the false negative rate of the threshold function, from 18.609% to 0.335%, without increasing the false positive rate. This is about a 10% improvement over the fitness value of the threshold predicate. With this result, we can answer evaluation question *Q1*: yes, the automatically constructed utility function can more accurately detect a faulty state than the threshold predicate.

$$\begin{gathered}((cpu\_time\_maximum < 0.8956985388949147 \\ \wedge\ threads\_total \leq 0.6131020761032768) \\ \oplus\ (threads\_total \leq 0.9873018396192027 \\ \wedge\ threads\_daemons < 0.590277498349674)) \\ \oplus\ threads\_total \leq 0.763775881645383 \\ \oplus\ ((\Delta stacks\_average < 0.13942112989901811 \\ \vee\ \Delta cpu\_time\_maximum > 0.579209835184787) \\ \wedge\ threads\_daemons \geq 0.9795736847762824)\end{gathered}$$

Fig. 6. Evolved Predicate $\beta = 1$

In an attempt to explore alternate parameters, the GP was also run with $\beta = 0.5$ and $\beta = 2$. Figure 7 shows the best solution found with a fitness function having $\beta = 0.5$. Figure 8 shows the best solution found, with a fitness function having $\beta = 2$. Table I shows the fitness of the best evolved predicates for each β value.

$$\begin{gathered}classes\_unloaded \leq 0.6457107819682341 \\ \wedge\ (cpu\_time\_average \leq 0.5727762258083509 \\ \wedge\ (classes\_unloaded > 0.6928451352159278 \\ \vee\ (threads\_total \leq 0.9764865537324401 \\ \wedge\ classes\_total \geq 0.35937482288003575)))\end{gathered}$$

Fig. 7. Evolved Predicate $\beta = 0.5$

$$\begin{gathered}((threads\_daemons \geq 0.6267680140791754 \\ \wedge\ threads\_total \leq 0.8312147645132364) \\ \vee\ threads\_total \leq 0.6082492464245868) \\ \wedge\ stacks\_maximum > 0.04801365463774654\end{gathered}$$

Fig. 8. Evolved Predicate $\beta = 2$

We see from the differences in Figure 7 and Figure 8 that varying the β value produces predicates with differing resource usage metrics—hence, answering our evaluation question *Q2*. Therefore, we conclude that some resource usage metrics are better indicators of faulty states but may not detect all anomalous behaviors. All of the evolved predicates improve upon the threshold predicate fitness score. An evolved predicate with β greater than 1 gives preference to false negatives over false positives. As expected, for $\beta = 2$ the false negative rate improves at the expensive of an increased false positive rate. Likewise, an evolved predicate with β less than 1 gives preference to false positives over false negatives. In our experiment, there was no room for improvement for the false positive rate when $\beta = 1$; however, the false negative rate increased slightly to 1.835%.

V. Discussion

In this section, we discuss our evaluation results and threats to validity of our utility construction technique.

Although the 0.998327 fitness value obtained by our GP is not theoretically optimal, the theoretical optimal may not be realistically obtainable. Though a perfect solution would, ideally, have a fitness value of 1, to obtain a perfect solution would require a perfect application, test suite, and genetic algorithm. Of the multiple runs of our GP, most results approached the 0.998327 fitness, though at times we recorded solutions with fitness values as low as 0.957775. The probabilistic nature of genetic programming may occasionally skew results and, as such, optimal results may not be recorded during every experiment. However, recent research (e.g. Harik et al. [22]) in evolutionary computing has built probabilistic models of GAs that allow for more predictable results. Applying these models to our GP remains a future work.

A. Threats to Validity

Since we only applied our technique to a single target system, we cannot conclude that the effectiveness of our utility function construction technique generalizes to all virtual machines and software systems.

The training/test data for our technique play a significant role in affecting the results. Though our training data were substantial, our algorithm may have yielded a more accurate detection predicate had we used a larger training set. The results of Stehle et al. [4] corroborate this idea that increasing amount of training data can improve the constructed utility function. They showed that insufficient data, which inaccurately represented normal software usage, produced higher false positive rates. Furthermore, our attacks may not include all possible attacks on Jigsaw and the actual false negative rate during runtime may be higher. Additionally, the requirement to provide training/test data may hinder the applicability of our technique to certain COTS components, since the definition of normal use varies by application. Though our component, as a web server, provides a well-defined metric for usage, other COTS components may not be afforded the same luxury. We had access to real-world usage statistics and were able to employ these to train our algorithm. With other software systems, even if a model of normal usage can be determined, the data used for training may still be biased unless it comes from actual use.

Our technique assumes that the normal behavior of a target system can be fully captured by a test suite. However, as Burgess et al. [8] show, it is difficult to define the normal behavior of a system over a short duration. By using a more sophisticated approach to identify thresholds during normal behavior, these collected resource usage metrics can be given as input to our GP for constructing more accurate utility functions.

B. Future Work

We plan to evaluate our technique further by applying it to several target systems of varying sizes and characteristics, and from various domains.

In this paper, we only compared our automatically constructed utility function to a threshold predicate. It remains an open future work to compare the accuracy of our automatically constructed utility function with utility functions constructed with other existing techniques (e.g., manual selection by domain experts [3], principal component analysis [10], computational geometry [4]).

The inclusion of various other resource monitors is a future work. Additionally, we only considered *passive sensors* (i.e., without modifying the system being monitored) in this paper. A possible future work is to explore the use of *active sensors* for collecting additional resource usage metrics. In contrast to passive sensors that can monitor a system in the absence of the system's code, active sensors require modifying the original code to collect resource usage information. Using aspect-oriented programming [23] may reduce the effort for injecting active sensors into existing software systems.

As future work one might consider not simply taking a weighted sum of true positives, negatives and false positives and negatives, but instead to treat these as four independent criteria to be optimized using search-based methods. Using a Pareto optimal approach, the engineer could then study the trade offs.

Although our technique can construct an accurate utility function, determining the root cause of the problem and mitigating the fault remains active areas of research.

VI. Conclusion

Since utility functions for detecting software faults are difficult and costly to construct manually, we created a technique that uses genetic programming to construct a utility function automatically for a specific system, set of resource usage metrics, and precision/recall preference. Our approach included a configurable parameter for meeting the user's precision and recall preferences in detecting faults. We evaluated our technique on the open source Jigsaw web server, using ten resource usage metrics and five faults. To assess the success of our technique, we compared the performance of the resulting utility function to that of a hand-crafted utility function that uses simple thresholding of the system metrics to detect faults and failures. The results of our study show that our technique can automatically construct a utility function that can outperform a hand-crafted utility function by 10 percent.

References

[1] M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. S. Ward, "System monitoring with metric-correlation models: Problems and solutions," in *Proc. 6th International Conference on Autonomic Computing*, Jun. 2009, pp. 13–22.

[2] P. deGrandis and G. Valetto, "Elicitation and utilization of application-level utility functions," in *Proc. 6th International Conference on Autonomic Computing*, Jun. 2009, pp. 107–116.

[3] M. Acharya and V. Kommineni, "Mining health models for performance monitoring services," in *Proc. 24th IEEE/ACM International Conference on Automated Software Engineering*, Nov. 2009, pp. 409–420.

[4] E. Stehle, K. Lynch, M. Shevertalov, C. Rorres, and S. Mancoridis, "On the use of computational geometry to detect software faults at runtime," in *Proc. 7th International Conference on Autonomic Computing*, Jun. 2010.

[5] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[6] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[7] W. E. Walsh, G. Resauro, J. O. Kephart, and R. Das, "Utility functions in autonomic systems," in *Proc. 1st International Conference on Autonomic Computing*, May 2004, pp. 70–77.

[8] M. Burgess, H. Haugerud, S. Straumsnes, and T. Reitan, "Measuring system normality," *ACM Transactions on Computer Systems*, vol. 20, no. 2, p. 125160, 2002.

[9] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self adaptation with reusable infrastructure," *IEEE Computer*, vol. 37, no. 10, pp. 41–50, Oct. 2004.

[10] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley and Sons, 1991.

[11] J. A. Clark, J. J. Dolado, M. Harman, R. M. Hierons, B. F. Jones, M. Lumkin, B. S. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. J. Shepperd, "Formulating software engineering as a search problem," *IEE Proceedings - Software*, vol. 150, no. 3, pp. 161–175, Jun. 2003.

[12] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the Bunch tool," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193–208, 2006.

[13] M. Shevertalov, K. Lynch, E. Stehle, C. Rorres, and S. Mancoridis, "Using search methods for selecting and combining software sensors to improve fault detection in autonomic systems," in *Proc. 2nd International Symposium on Search Based Software Engineering*, Sep. 2010.

[14] R. C. Lange and S. Mancoridis, "Using code metric histograms and genetic algorithms to perform author identification for software forensics," in *Proc. Genetic and Evolutionary Computation Conference*, Jul. 2007, pp. 2082–2089.

[15] M. Shevertalov, E. Stehle, and S. Mancoridis, "A genetic algorithm for solving the binning problem in networked applications detection," in *Proc. IEEE Congress on Evolutionary Computation*, Sep. 2007, pp. 713–720.

[16] A. J. Ramirez, D. B. Knoester, B. H. C. Cheng, and P. K. McKinley, "Applying genetic algorithms to decision making in autonomic computing systems," in *Proc. 6th International Conference on Autonomic Computing*, Jun. 2009, pp. 97–106.

[17] J. H. Holland, *Adaptation in Natural and Artificial Systems*. MIT Press, 1992.

[18] R. Tanese, "Distributed genetic algorithms," in *Proc. 3rd International Conference on Genetic Algorithms*, Jun. 1989, pp. 434–439.

[19] S. M. Beitzel, "On understanding and classifying web queries," Ph.D. dissertation, Illinois Institute of Technology, May 2006.

[20] G. Xu and A. Rountev, "Precise memory leak detection for Java software using container profiling," in *Proc. 30th International Conference on Software Engineering*, May 2008, pp. 151–160.

[21] S. Luke, "Two fast tree-creation algorithms for genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 4, no. 3, pp. 274–283.

[22] G. R. Harik, F. G. Lobo, and K. Sastry, "Linkage learning via probabilistic modeling in the extended compact genetic algorithm," in *Studies in Computational Intelligence*, J. Kacprzyk, Ed. Springer, 2006, vol. 33, pp. 39–61.

[23] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of the European Conference on Object-Oriented Programming*, Jun. 1997.