Using Evolutionary Diversity to Identify Problematic Software Parameters

Errin W. Fulp, H. Donald Gage, and Matthew R. McNiece
Department of Computer Science
Wake Forest University
Winston-Salem, NC 27109, USA
Email: fulp@wfu.edu

Abstract—Many computer performance and security issues can be attributed to poor software configuration settings (misconfigurations). However, finding and resolving errors is complicated by the number of parameters to set and the complexity (unplanned interdependencies) of these configurations. This problem only becomes more difficult as software systems become more integrated and complex.

This paper introduces an evolutionary technique to identify and resolve parameter errors found in software configurations. The approach discovers characteristics of correct and incorrect parameter settings by comparing and contrasting a set of configurations. The composition of the configuration set is important and must contain sufficient information to identify and resolve the errors. Given the difficulty of forming this set, an evolutionary algorithm is used to discover configurations that have these characteristics. The effectiveness of this approach is analyzed experimentally through a study of various configurations with various security issues. Experimental results indicate the approach is able to identify and determine secure parameter settings when confronted with a variety of simulated attacks.

Index Terms—software configuration; misconfiguration; computer security; automated management; genetic algorithm;

I. INTRODUCTION

Software configuration errors remain a continual source of computer system problems. For example, Google reported that a significant number of software failures were not the result of software bugs (code errors), rather failures were more likely due to poor configuration settings that control the software [12]. Similarly, the majority of the software failures for Yahoo’s Zookeeper (a service for coordinating processes of distributed applications) were caused by misconfigurations [11]. These examples highlight how misconfigurations can result in software failures (application crashes or incorrect output generation); however, misconfigurations can also introduce security vulnerabilities. For example, misconfigurations have been reported to be the most prevalent vulnerability for web-services [13].

Given the prevalence of the problem, several research efforts have focused on resolving software misconfigurations. The difficulty of the problem arises from the potential size (large number of parameters) and complexity (interdependencies) of software configurations [5], [12]. A variety of approaches have been developed, such as considering the configuration as a Boolean expression and using SAT-solvers to find correct settings [6]. An alternative is to leverage evolutionary search algorithms to discover secure software configurations [10].

Evolutionary approaches, also used in this paper, have the advantage of managing multiple solutions to a problem. This solution diversity also improves the system resiliency, since alternative secure configurations are available if the computing environment changes (new threats discovered) [4]. While these approaches can typically discover a secure configuration, they are unable to identify which parameters were at issue. Often an administrator may not know exactly which parameters are causing a problem, and as a result, a larger set of parameters is unnecessarily configured in the process. If the specific parameters causing (or contributing) to the problem could be identified, the administrator would have the ability to set the other parameters based on other criteria, such as improving performance.

This paper describes how evolutionary-based algorithms can be used to discover secure configurations and identify which parameters were at fault. This approach models software configurations (application and/or operating system) as chromosomes, where configuration parameters are individual chromosome traits or alleles. Mimicking different forms of selection, crossover, and mutation processes observed in nature, such an approach maintains a set (pool) of configurations that will address the problem. Given these configurations (observed over multiple iterations of the algorithm), it is possible to compare and contrast settings to determine which contributed to the problem or resolved the problem. For example, an insecure setting should be consistent across vulnerable configurations. This list of potential problem parameters and the associated good and bad settings, are refined over subsequent iterations of the algorithm.

Simulation results presented in this paper will demonstrate the ability the proposed evolutionary-based approach to discover secure configurations and identify the problematic parameters. Experiments considered a variety of factors that increase the difficulty of finding secure configurations (configuration size and parameter types). The size of the configuration had no effect on the number of generations required to identify and resolve misconfigurations. However, the number of possible values that could be assigned to a parameter (alternative settings) did linearly increase the number of generations the evolutionary approach required.
The remainder of this paper is structured as follows. Section II describes the compositions of software configurations. A technique for identifying problem parameters based on labeled configurations is described in Section III, while Section III-C discusses how evolutionary algorithms are used to find configurations needed by the identification process. Section IV provides experimental results indicating the proposed approach is suitable for a variety of configurations and parameters. Finally, Section V reviews this paper and discusses some areas of future work.

II. SOFTWARE CONFIGURATIONS

As mentioned in the Introduction, poorly chosen parameter settings (misconfigurations) are the cause of many computing system problems. This includes poor performance, software failures and security issues [9]. Guidelines are available to help avoid misconfigurations. For example, the Federal Desktop Core Configuration settings and the Consensus Security Configuration Checklist managed by NIST and CIS (Center for Internet Security) provide guidance and configuration settings of various applications and operating systems [3], [8]. Many of the configuration parameters individually impact security. For example, the NIST guidelines for a Debian operating system installation requires a SSH client and server to only use version 2 of the protocol (set in the /etc/ssh/ssh_config file) due to known security issues with version 1.

Although best practice guidelines can help to improve the system’s operation, they may not apply to all computer installations. A system may need certain parameter settings to remain functional, even though such settings may be deemed insecure. Installing many applications and software patches may have unwanted consequences for functionality and security [1], [5], [9], [12]. In addition, future security threats may render current guidelines useless.

III. IDENTIFYING AND RESOLVING PROBLEMATIC CONFIGURATION PARAMETERS

The configuration management approach described in this paper compares and contrasts the settings of configurations to determine what settings configurations have, or do not have, in common. Using this information it is possible to identify the problematic parameter settings as well as how to resolve the issue.

Consider a software configuration as an ordered list of \( m \) parameter settings \( \{s_0, s_1, ..., s_{m-1}\} \), where \( s_i \) is the setting for the \( i^{th} \) parameter. Likewise, the \( i^{th} \) parameter has a finite set of possible values \( \{v_{i0}, v_{i1}, ...\} \) from which the parameter setting is assigned, \( s_i \in \{v_{i0}, v_{i1}, ...\} \). For example, a binary parameter has two possible values. Note, it has been observed in [4] that Linux parameters settings are from either: an integer range defined over an interval, a specified lists (e.g. lists of allowable strings), or a bit vector (e.g., file permissions). Therefore modeling possible settings with a finite set can be done without loss of generality. A parameter can be of one of two types. It may be a problem parameter, in which case it has one or more possible settings that will cause an error and at least one setting that will resolve it. Or it can be a passive parameter, i.e., having no settings that will cause an error.

Given the parameters and alternative settings, \( \prod_{i=0}^{m-1} |v_i| \) configurations exist for the software system where each configuration has the same list of parameters but differ with respect to their settings. Furthermore, assume configurations can be categorized as either good or bad. A good configuration is one that does not have any problem parameters, while a bad configuration is one that does. Note, the error could be performance or security related. The label associated with a configuration is determined based on observation, for example measuring system performance or vulnerability to attacks after the configuration is instantiated.

A. Using Counters for Parameter Discovery

Given a set of labeled configurations, it may be possible to determine the parameter settings that are the cause of a problem by noting what settings are associated with good or bad configurations. Let \( z_i \) be the counter for \( v_i \) (the \( j^{th} \) possible value of the \( i^{th} \) parameter). Initially, each counter is set to zero and is updated based on if the value occurred with a good or bad configuration. For every good configuration the associated setting counter is incremented, and for every bad configuration the associated setting counter is decremented. As the number of configurations are processed, settings with the lowest counts are associated with configuration errors, while the highest counts are associated with settings that resolve errors. For example if \( n \) bad configurations (containing the same setting error) are processed, then the counter for the problematic settings should be \( -n \). It is possible to normalize the setting counts by the number of configurations that contained each setting. As result, the normalized values will be 1 for good settings and -1 for bad settings. Passive parameters will have a normalized value between -1 and 1.

Consider identifying the problematic parameter setting for the three configurations presented in Figure 1. In this example each configuration, \( c_i \), consists of four binary parameters. Settings are either \( D \) or \( E \) for each parameter, and a misconfiguration occurs if the fourth parameter is set to \( E \) (when \( s_3 = E_3 \)). As the configurations are processed, the counters for each setting are increased (for configurations \( c_0 \) and \( c_1 \)) or decreased (for configuration \( c_2 \)). After the configurations are processed, setting \( E_3 \) has the lowest value thus indicating this setting causes the error. The setting \( D_3 \) has the highest value and corresponds with the error resolution.

The counter based approach can also detect situations where the error can be caused or resolved by multiple settings. Consider the set of configurations given in Figure 2. In this example, parameters have three possible settings, \( D, E, \) or \( F \). There are two settings \( E_2 \) and \( F_2 \) that can cause the error and one setting \( D_2 \) that will resolve the error. The bad-setting \( E_2 \) is used in configuration \( c_2 \) and the corresponding counter is -1. Once configuration \( c_3 \) is processed, the alternative bad-setting,
$F_2$ occurs and the corresponding setting is also -1. Both bad-settings have the lowest counter value and correctly indicate these are the problematic settings.

### B. Configuration Sets and Generation

The composition of configurations processed by the identification approach will affect how quickly problematic settings are discovered. Consider the impact of processing good or bad configurations on setting counters. Counters for bad settings will be less than or equal to zero and will only decrease since these settings can only be associated with bad configurations. Similarly, counters for good settings will always be positive and can only increase. As a result, a distinction between these two sets can occur when processing a small number of configurations.

Passive setting counters can increase or decrease depending on if the setting is associated with a good or bad configuration. This is because these parameters do not contribute to or resolve an error. After processing examples of good and bad configurations, the counter values for passive settings should be greater than the values for bad settings and less than the values for good settings. Processing more diverse configurations (trying as many passive settings as possible) will increase this distinction. Therefore, obtaining a set of configurations that have these characteristics is important.

### C. Evolutionary Strategies for Discovering Configurations

Finding configurations that are different yet provide the same functionality and security has been the subject of recent research [4], [5], [12]. These configuration alternatives have been used to create a moving target defense [7], as well as a technique to improve resiliency. Finding alternatives has been shown to be a difficult problem, primarily due to the size (number of parameters and settings) and complexity (unknown interdependencies) of configurations. One approach that has shown promise in this area is the use of evolutionary algorithms.

Evolutionary Algorithms (EAs) are a type of search heuristic that mimic evolution. EAs represent potential solutions as a chromosome consisting of multiple traits, or parts of the solution. When applied to searching for software configurations, the configuration is considered a chromosome where the individual configuration settings are regarded as traits. A measure of fitness is also important for evolutionary algorithms to ensure fitter chromosomes are more likely to survive and influence the next generation. For this application, fitness corresponds to the number of misconfigurations, where higher fitness values are given to configurations with fewer errors.

### D. Evolutionary Processes for Discovering Configurations

An EA progresses by maintaining a group of potential solutions to the problem. This set of solutions is referred to as the chromosome pool. As depicted in Figure 3, a new pool (set) of configurations are created for each generation based on the previous generation. This is done using a series of selection, recombination, and mutation processes (mimicking processes observed in nature) [2].

<table>
<thead>
<tr>
<th>Configurations \ c_i = {s_0, s_1, s_2, s_3}</th>
<th>Counter-Based Approach</th>
<th>Counter-Based Identification</th>
</tr>
</thead>
<tbody>
<tr>
<td>{D_0, D_1, D_2, D_3}</td>
<td>{s_0, s_1, s_2}</td>
<td>{(D_0, E_0) \ (D_1, E_1) \ (D_2, E_2) \ (D_3, E_3)}</td>
</tr>
<tr>
<td>{E_0, E_1, E_2}</td>
<td>{s_0, s_1}</td>
<td>{(D_0, E_0, F_0) \ (D_1, E_1, F_1) \ (D_2, E_2, F_2)}</td>
</tr>
<tr>
<td>{D_0, D_1, D_2}</td>
<td>{s_2}</td>
<td>{F_2}</td>
</tr>
</tbody>
</table>

Fig. 1. Example identification of a problematic parameter setting. Each configuration, $c_i$, consists of 4 parameters, where each parameter is either set to $D$ or $E$ ($s_i \in \{D_i, E_i\}$) and $s_3 = E_3$ is the misconfiguration.

Fig. 2. Example identification of a problematic parameter setting. Each configuration, $c_i$, consists of 3 parameters, where each parameter is either set to $D, E$, or $F$ ($s_i \in \{D_i, E_i, F_i\}$), and $s_2 \in \{E_2, F_2\}$ is the misconfiguration.
The selection process probabilistically identifies candidates from the current pool. For this application, two different candidate configurations are randomly selected from the pool based on their fitness. This is a form of fitness proportionate selection or roulette-wheel selection, where the probability of selection is based on the fitness (fit chromosomes are more likely to be selected) [2].

Recombination (also known as crossover in the Genetic Algorithm literature) is applied to the pair of selected configurations. This is done to exploit the combination of the pair of selected configurations to form a new offspring chromosome. Recombination combines portions of existing configurations to create a new configuration. A variety of recombination processes exist and can be differentiated largely on how portions of the candidate chromosome are selected. Multi-point recombination was used for this research; random portions of the parameter settings, based on a uniform distribution function, are identified and copied to the offspring chromosome (the new configuration). Therefore the new chromosome traits are from one of the two parents.

The mutation process provides the ability to explore new regions of the problem space by randomly changing parameter settings in the off-spring created from the recombination process. The purpose of mutation is to maintain diversity across the generations of configurations, which is important for identifying problematic parameter settings. Given the new configuration, each parameter setting will be mutated with a certain probability. If mutation occurs then the parameter setting is randomly modified using a uniform distribution and according to its type. The new configuration can then be enacted using a Virtual Machine (VM) and tested to ensure the necessary functionality is provided. Once the assessment has taken place, the new configuration can join the pool.

This series of reproduction, recombination, and mutation processes are repeated until a new pool (set) of configurations has been generated. The EA continues creating new chromosome pools for every generation until a condition is met. For this application, the EA continues to identify new configurations until the problem parameters have been identified and resolved.

IV. EXPERIMENTAL RESULTS

This section evaluates the performance of the counter-based parameter identification and resolution approach using simulation. As described in Section III-C, the approach uses an evolutionary algorithm to search for software configurations that are compared and contrasted. The probabilities associated with the evolutionary algorithm processes of recombination and mutation was 0.05 per parameter and the population (pool) size was 100 configurations. The effect of configuration size (number of parameters) and number of settings were investigated. Performance was measured based on whether or not the approach was able to identify and resolve a set of misconfigured parameters and the number of generations needed. Note, identifying the misconfigurations occurs when the problematic parameters are successfully identified. Resolving the misconfigurations represents when the approach successfully discovers a configuration that does not have the misconfigurations (correct settings); however, the identity of which parameters are involved is unknown. Once a miscon-figured parameter is both identified and resolved, then the specific parameter and the setting required to fix the error is known. The values presented are the average of 50 independent experiments with the same initial configuration properties.

A. Configuration Size

Given the potential large size of configurations (number of parameters), one area of interest is the effect of configuration size on the identification approach. In these experiments, configurations consisted of 100, 500, or 1000 parameters. Each parameter had 6 possible settings; therefore, for a configuration with 100 parameters, there are at least $6^{100}$ possible configurations. Configurations had 6 parameters with possible misconfigurations, with one setting that resolved the issue and the remaining settings causing an error. While this situation
is potentially too pessimistic, it was done to observe how the approach worked in a worse case scenario. Figure 4(a) depicts the number of suspected parameters as configurations are processed by the proposed approach. Initially, all parameters are suspect; however after approximately 12 generations, the list of suspected parameters drops significantly and the six problematic parameter settings are identified after 30 generations regardless of the number of parameters in the configuration. This is in part due to multi-point crossover and mutation processes used. Each parameter is given the same chance to change regardless of the number of parameters. Therefore the number of configuration parameters does not impact identification performance. Similar results were observed for resolving the misconfigured parameters, as seen in Figure 4(b). Recall, resolving indicates a configuration that does not have the six misconfigurations has been identified, but not which parameters were at issue. The approach quickly discovers configurations that resolve the six configuration errors using approximately 10 generations. This is the same regardless of the number of parameters in the configuration. The approach is able to identify good configurations (resolving the misconfigurations) before determining which parameters caused the errors. Therefore, more information (processing additional configurations) are needed to identify the specific misconfigurations.

**B. Values per Parameter**

Given that parameters may have a large number of potential values (alternative settings), the effect of these alternatives is also of interest. In these experiments configurations consisted of 500 parameters, where every parameter in a configuration had either 6, 12, or 24 possible values. Figure 5(a) depicts the average number of generations required to identify the six parameter errors in the configurations. Each configuration type started with 500 suspected parameters; however, the number of possible values a parameter could be assigned did affect the number of generations needed. Configurations that had parameters with more possible values required more generations to successfully identify the misconfigured parameters. A configuration with 24 values/parameter required 75 generations as compared to 33 generations for configurations with 12 values/parameter and 15 generations for configurations with 6 possible values/parameter. This indicates that more alternative settings does increase the search. Similar results were observed for the number of generations required to resolve the misconfigurations. Configurations with 24 values/parameter required 40 generations as compared to 12 generations for configurations with 12 values/parameter and 10 generations for configurations with 6 possible values/parameter. Again, the approach is able to identify good configurations before determining which parameters caused the errors.

V. CONCLUSIONS

This paper introduced an approach for identifying and resolving misconfigurations. Configurations for the same software system are labeled based on the presence of a misconfiguration then compared and contrasted to understand the commonalities they may or may not share. Using counters that are incremented or decremented based on if a setting was associated with a correct or incorrect configuration, the list of suspected parameters becomes shorter until the actual problematic settings are revealed. An important component of the approach is the availability of alternative configurations, which is done using an evolutionary algorithm.
Simulation results demonstrated the ability of the approach to quickly discover and identify misconfigurations for a variety of configuration types. The approach was able to scale as the number of parameters in the configuration increased and identified six misconfigurations in configurations of 100, 500, and 1000 parameters within the same number of generations. The number of possible values a parameter could be assigned did impact the number of generations required to identify and resolve misconfigurations. As the number of possible values increased, the number of generations required increased.

The experimental results for the proposed misconfiguration resolution approach are promising, since the technique was capable of identifying and resolving problematic parameters in a variety of examples. However, there are several additional related research areas that warrant additional investigation. For example, a growing concern is the existence of parameter interdependencies. In these cases, parameters must be set in concert in order to correct a misconfiguration. Future work will examine the ability of the approach to identify and correct parameter chains.

VI. ACKNOWLEDGMENTS

This material is based upon work supported by Cisco Systems, Inc. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of Cisco Systems, Inc.

REFERENCES


