# Using Probability Densities to Evolve more Secure Software Configurations

Caroline A. Odell
Dept. of Computer Science
Wake Forest University
Winston-Salem, NC, USA
odelca14@wfu.edu

Matthew R. McNiece
Dept. of Computer Science
Wake Forest University
Winston-Salem, NC, USA
mcnimr13@wfu.edu

Sarah K. Gage
Dept. of Mathematics
Indiana University
Bloomington, IN, USA
skgage@indiana.edu

H. Donald Gage
Dept. of Computer Science
Wake Forest University
Winston-Salem, NC, USA
gagehd@wfu.edu

Errin W. Fulp
Dept. of Computer Science
Wake Forest University
Winston-Salem, NC, USA
fulp@wfu.edu

## ABSTRACT

The use of Evolutionary Algorithms (EAs) is one method for securing software configurations in a changing environment. Using this approach, configurations are modeled as biological chromosomes, and a continual sequence of selection, recombination, and mutation processes is performed. While this approach can evolve secure configurations based on current conditions, it is also possible to inadvertently lose solutions to previous threats during the evolution process.

This paper improves the performance of EA-based configuration management by incorporating parameter-setting history. Over the generations (EA iterations), counts are maintained regarding the parameter-settings and the security of the configuration. Probability densities are then developed and used during mutation to encourage the selection of previously secure settings. As a result, these secure settings are likely to be maintained as attacks alternate between vulnerabilities. Experimental results using configuration parameters from RedHat Linux installed Apache web-servers indicate the addition of parameter history significantly improves the ability to maintain secure settings as an attacker alternates between different threats.

## 1. INTRODUCTION

Computer applications often have an associated configuration that directs and governs operation. These configurations typically consist of a set of parameters, each with a certain setting that impacts how the application functions. Configuration parameters include operating system settings such as file permissions or enabling address randomization, as well as application settings like those found in the Apache web-server `httpd.conf` file. Configuration parameters are normally set to ensure functionality, improve performance, such as response times [15, 17, 23], and to provide system security [2, 7, 11].

While maintaining parameter-settings that optimize system performance is important, determining settings that improve the security of the system (i.e. fixing vulnerabilities) is critical. Although best practice guidelines for setting configuration parameters can help to improve system security [5, 10], they may not apply to all computer installations. Guidelines often do not consider all the different combinations of applications that may reside and interact on a computer. In addition, future security threats may render current guidelines useless. Configuration parameters can also be interdependent, forming a configuration parameter chain that often requires a complex combination of parameter-settings to improve security [8].

One promising approach for securing software configurations uses Evolutionary Algorithms (EAs) to search for secure configurations [2, 7]. Evolutionary Algorithms (EAs) are a type of search heuristic that mimics evolution. They seek better (more fit) solutions by discovering, emphasizing, and recombining portions of current solutions. By modeling a configuration as a chromosome, where individual parameters are traits, the processes of selection, recombination, and mutation can be used to discover more secure configurations. An important advantage of an EA, as compared to other search methods, is the ability to adapt to current environments and threats, which can yield a more resilient approach for securing configurations. However as both the attacks and the computing environment change, it is likely that an EA will not maintain previously secure settings.

This paper investigates the use of a parameter-setting history to improve the performance of EAs when searching for secure configurations. As the EA evolves new configurations, information about previous settings and whether they were associated with successfully attacked configurations is maintained. As a result, the probability of each parameter setting being secure or insecure can be estimated allowing for parameter specific density functions to be approximated. The estimation of each parameter density function is further refined as more generations evolve. Note, the speed of estimation development can be altered by increasing the number of chromosomes in the pool (resulting in more samples

*SafeConfig'15,* October 12, 2015, Denver, Colorado, USA.
© 2015 ACM. ISBN 978-1-4503-3821-9/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2809826.2809831.

are acquired per generation). These estimations can be associated with different attack vectors and reinstated once a previous threat reappears. The density functions are then used during mutation to bias the EA to choose previously secure settings; however, other settings are possible. The result is that the EA is able to quickly adjust to different attacks and potentially new variations.

The remainder of this paper is structured as follows. Section 2 briefly reviews software configurations, vulnerabilities, and the use of security checklists. Evolution of configurations is discussed in Section 3. Section 4 introduces the use of history to influence mutation, while in Section 5 simulation results demonstrate the performance of history-based mutation with server configurations similar to Apache 2.2 and Red-Hat 5. Finally, Section 6 reviews the evolutionary-based approach and discusses areas of future work.

## 2. SOFTWARE CONFIGURATIONS AND VULNERABILITIES

A computer's operation, or application state, is often governed by its software configuration. This can include transient data, such as information found in the Linux `/proc` file system, and persistent data that is stored in non-volatile memory such as the Windows Registry and Unix resource files. Regardless of the OS, there is a large number of possible configuration parameters to consider. For example, the Windows Registry has over 77,000 entries initially, but after loading a common suite of applications can have on average 200,000 settings [18]. These large and complex software configurations are often difficult for administrators to secure. In response, several guidelines have been issued to help administrators select parameters that have been associated with secure systems.

The Federal Desktop Core Configuration, Consensus Security Configuration Checklist (developed by NIST), and Center for Internet Security (CIS) provide lists of parameter settings for a variety of operating systems and applications [10]. 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 (found in the `/etc/ssh/ssh_config` file), due to known security issues with version 1. While these checklists, and others, are extensive and provide a reasonable start for securing configurations, they are not comprehensive with regards to all possible operating system application combinations. For example, it is possible that a secure OS parameter-setting described in a checklist may not be plausible given the existence of an application. Furthermore, it is likely that parameters are interdependent, forming a parameter chain, that must be collectively resolved in order to improve security. Parameter chains have recently gained attention since they are often difficult to recognize and manage [4] and will be an important component of the next Common Vulnerability Scoring System (CVSS) version 3 [6].

## 3. EVOLVING CONFIGURATIONS

The complexity of software configurations and persistent threat of new attacks makes finding and maintaining secure configurations a difficult and continual problem. It is possible to find secure configurations using search algorithms and machine learning approaches suited for complex spaces [1, 17, 21]. However, these methods were developed for tuning parameters to improve system performance (user response times), not securing applications. When securing a configuration, the administrator may not know which parameters are vulnerable. In addition, securing a system requires continual updates as the operational environment (software installed) and threats change over time.

An Evolutionary Algorithm (EA) can provide a resilient configuration management approach by modeling configurations as chromosomes (candidate solutions) and applying a series of selection, recombination, and mutation processes (mimicking processes observed in nature) [2, 7]. As depicted in Figure 1, a population of chromosomes continually evolves to become more secure with respect to the current threat environment. System administrators can then apply these configurations to improve security and resiliency as needed.

### 3.1 Modeling Configurations

When applying an EA to a problem, candidate solutions must be mapped to chromosomes. For this EA application, a candidate solution is a configuration and the individual parameters are considered typed chromosome traits. For example, Linux and Apache server configuration parameter-settings are generally one of three types: *integer range*, defined over the interval $[0, \text{MaxInt}]$, *specified lists* (e.g. lists of allowable strings), and *bit vector* (e.g., file permissions) [7, 14]. These three types have been consistently observed with other Linux-oriented application parameters [7, 14].

#### 3.1.1 *Chromosome Feasibility and Fitness*

Establishing a notion of feasibility and fitness is required for creating better chromosomes from current chromosomes. For computer configurations, a chromosome is considered *feasible* if the configuration it represents provides the necessary functionality for the computer. For example, if the computer is a web server, then any configuration that blocks network access would be infeasible, and therefore not enacted on a computer. Previous research has developed various methods for identifying and correcting misconfigurations that can be used for feasibility testing [8, 19, 20, 22]. In addition, automated software testing approaches can be borrowed to test feasibility. For example, predefined predicates can create a test oracle to determine if application services are adequately provided [13].

The fitness of a chromosome is a heuristic measure of the effectiveness of the corresponding configuration with respect to security. For this EA-based approach, configuration fitness corresponds to the number of different successful attacks detected during a period of time. Penetration-testing software, instantiating the configurations in a honeynet [16], and replaying recent unresolved attacks [12] are all possible methods for observing how well configurations withstand attacks. This information is used to rank configurations, where configurations that were vulnerable to fewer attack types are deemed better. No additional information about the individual parameter fitnesses or which parameters were involved with the attack is provided to the algorithm.

### 3.2 EA Processes

Using the chromosome feasibility and fitness models, the EA uses a series of *selection*, *recombination*, and *mutation* operators. These processes create a new chromosome that is added to a new pool. A pool is a set of chromosomes that evolve after each generation (iteration of the EA).

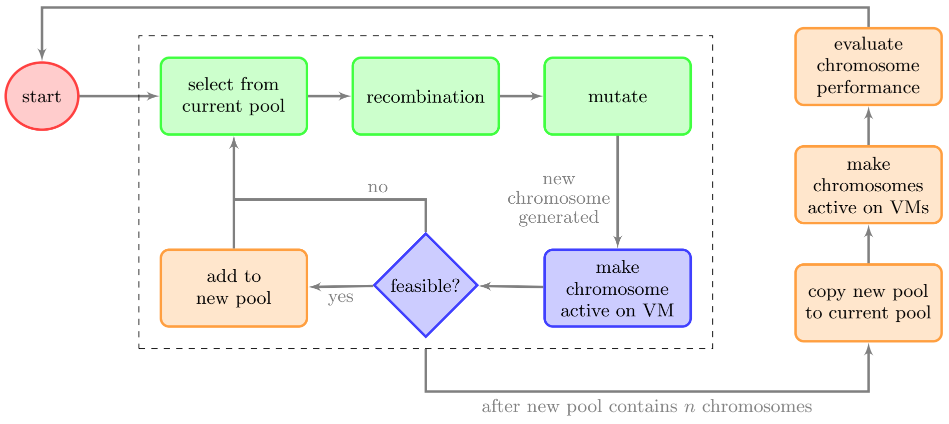

start
select from current pool
recombination
mutate
new chromosome generated
make chromosome active on VM
feasible?
no
yes
add to new pool
evaluate chromosome performance
make chromosomes active on VMs
copy new pool to current pool
after new pool contains $n$ chromosomes

**Figure 1: Flow chart of the tasks associated with the establishing a new pool of secure software configurations.**

The selection process probabilistically identifies candidates from the current pool. For this application, two different candidate chromosomes 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) [3]. Recombination (also known as crossover in the Genetic Algorithm literature) is applied to the pairs of selected chromosomes. This is done to exploit the combination of the pairs of selected chromosomes to form new offspring chromosomes. 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. Uniform recombination is used in this research; random portions of the parameter settings, based on a uniform distribution function, are identified and swapped between a pair of chromosomes.

Finally, 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 chromosomes and avoid permanent fixation at any particular locus. Given the new chromosome, each parameter setting will be mutated with a certain probability. If mutation occurs then the parameter setting is randomly modified according to its type. There is no *a priori* knowledge of a parameter's true probability density function, which ideally could enhance the random mutation changes. Instead, often a uniform distribution is applied to modify the current setting. However, more parameter centric probability densities have the potential to enhance the effectiveness of mutation, especially if the nature of attacks changes over time. This is the problem addressed by this paper and further described in Section 4.

### 3.3 Managing the Chromosome Pool

The series of selection, recombination, and mutation processes will create an offspring chromosome (configuration). The offspring chromosome is tested for feasibility then added to a new pool. Once the pool consists of $n$ configurations, it replaces the current pool, as depicted in Figure 1. Once the current pool has been established, the chromosomes can be instantiated on VMs that are non-production systems and the security performance evaluated as described in Section 3.1.1. The overall process continually repeats and evolves another pool of chromosomes from the current set. This provides a pool of updated configurations that can be applied to production computers when the administrator deems it is necessary [14]. Additional implementation details about this approach can be found in [9].

## 4. HISTORY-BASED MUTATION

As previously described, the attacks encountered over time will change as new vulnerabilities are discovered and the computing environment is updated. Regardless, it is important to recall parameter-settings that are associated with past secure configurations, since it is not uncommon for past threats to reappear. Furthermore, given the complexity of configurations and a constantly changing computing environment, it is desirable to have this process automated and not require the administrator to intervene. For the proposed EA, maintaining previous secure settings is done via the mutation process. When evoked, this process chooses parameter values that have been associated with secure configurations in the past with a higher likelihood. Allowing alternative setting choices (with a lower likelihood) may be desirable since attacks may change (new variations) over time.

### 4.1 Estimating Setting Densities

This desired mutation effect can be provided by developing a density estimation based on past choices. For each threat assume there is a unique attack signature, for example developed from log files, sensor settings, and/or the attack result. It is not necessary to know how the attack was accomplished (i.e., what parameters were involved). It is only necessary to know the attack was different from others, which will allow the development of attack specific parameter setting density functions.

Density functions will be estimated using counters associated with parameter settings. For each parameter, associate a counter for each possible parameter-setting. These counters cover all configurations within the pool. Each parameter-setting counter is initialized to zero. If a configuration is scored as fit (suffered no detectable attacks in a period of time), then each currently used setting counter is incre-

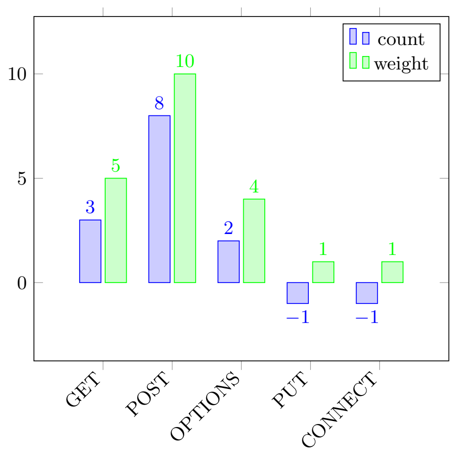


**Figure 2: An example configuration parameter with possible settings `GET`, `POST`, `OPTIONS`, `PUT`, or `CONNECT`. Counts represent the number of times a setting was associated with a secure or insecure configuration, while weights are adjusted counts used for mutation.**

mented by one. If the configuration is scored as unfit, then all the current setting counters are decremented by one. Note for parameters with a large number of possible settings (e.g. bit vector), it may be possible to bin the settings to a smaller number of alternatives without loss of generality. Figure 2 depicts example counts for an Apache web server parameter with the settings `GET`, `POST`, `OPTIONS`, `PUT`, or `CONNECT`. In this example the settings `GET`, `POST`, and `OPTIONS` were associated with secure configurations, while `PUT` and `CONNECT` were associated with insecure configurations.

Collecting data for the parameter-setting distribution estimation will initially occur within a *training phase*, where the counters for the settings are adjusted based on the attacks. The appropriate length of the training phase depends on the number of attacks encountered and the size of the configuration pool. Every configuration subjected to an attack provides information for the parameter-setting estimation. If a certain number of instances are necessary, then this can be achieved by lengthening the duration of the training phase and/or increasing the pool size. This changes the temporal and spatial components to provide the desired feedback. In addition, updating parameter-setting counts will continue after the training phase, which will allow the system to adjust to variations of existing attacks.

The collected setting counts are then adjusted to form weights. Weights are calculated per parameter, the absolute value of the lowest count is added to each setting plus one. For example, using the weight based histogram shown in Figure 2, if mutation is applied, the mutation operator would choose the `POST` setting 48% of the time. Note that in this example, settings that were associated with vulnerable configurations still maintain a non-zero weight. This is done to allow for reconsideration of these settings in the future. If a setting is known to always be insecure, then the administrator can choose to omit it from the settings list [14]. Over time (multiple generations), settings that resolve a vulnerability would be expected to have a higher count than the alternative settings. If a parameter does not affect a vulnerability, then it is expected that the counts will be approximately equal for all the alternative settings. The parameter-setting distribution is associated with a certain attack; therefore, once an attack returns it can be used to reestablish secure settings.

After the training-phase, histogram-based mutation can be used to create a desired percentage of the new configurations per generation. Configurations generated with histogram-based mutation will have a higher likelihood of reestablishing previously known secure settings. The remaining configurations in the generation are created using standard mutation and can provide a wider search for alternative secure parameter settings. Furthermore, these setting results can be used to update the histograms and allow the system adapt as the computing environment and threats change.

### 4.2 Distribution Estimations per Attack

The parameter-setting estimation described in the previous section encourages the EA to recall previous secure parameter-settings. Since the estimations are developed based on certain attacks, it is possible to reestablish the estimations if an attack returns. Furthermore, if the parameters of different attacks are not in conflict (require different settings), it is possible to merge the densities.

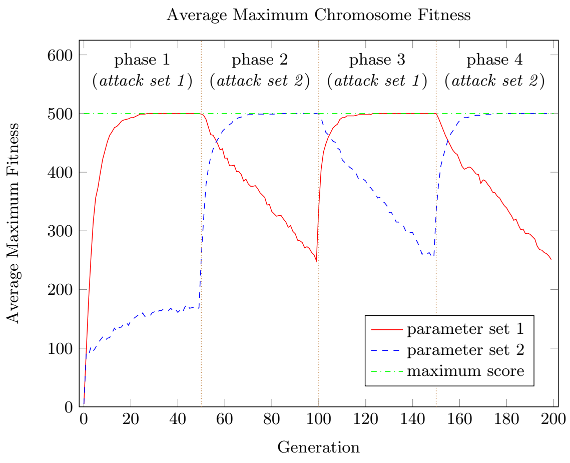
## 5. EXPERIMENTAL RESULTS

In this section the performance of EA-based configuration management using history-based mutation is compared to using standard mutation (all settings are equally likely). For the history-based mutation experiments, only 50% of each new generation was generated using history-based mutation. All experiments used configurations consisting of 102 RedHat 5 and Apache 2.2 parameters that were a mixture of integer ranges, specified lists, and bit vectors [7, 14]. Recombination and mutation probability was 0.05 per parameter. The EA pool size was 200 configurations and parameters were initially assigned to poor (vulnerable) settings.

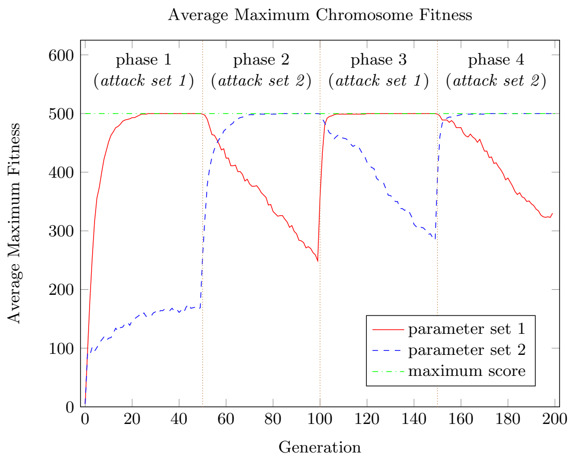
For an experiment, the configurations were subjected to two alternating attacks over 4 equal length phases (50 generations). Each attack targeted a set of five non-overlapping configuration parameters to demonstrate how the defense would respond to different attack vectors. The parameter sets consisted of sets and ranges, as described in Section 3.1. The first attack occurred during phases 1 and 3, while the second attack occurred during phases 2 and 4. Repeating attacks provides an opportunity to observe how quickly the defense can reinstate previously discovered secure settings.

As described in Section 3.1.1, configurations were scored based on the number of successful exploits. Each successful defense was assigned a score of 100 and each unsuccessful defense a score of 1. Since each attack targets five parameters, the fittest configuration score during a phase was 500 (all 5 attacked vulnerabilities were resolved) and the most unfit score was 5. Each experiment was performed 100 times and the fitness values of each parameter set were recorded.

The average maximum scores for the experiments are depicted in Figure 3. For the EA using standard mutation, the score of the attacked parameters increases during their corresponding attack phase. For example in Figure 3(a), parameter set 1 is attacked during phases 1 and 3. During both phases, the score of parameter set 1 reaches 500 (fittest score possible) after generation 20 of the phase. When an attack reappears the EA using standard mutation is able to



(a) Standard mutation.



(b) History-based mutation.

**Figure 3: Attack experiments consisting of four phases. Phases 1 and 3 attacked parameters from set 1, while phases 2 and 4 attacked parameters from set 2. Fitness values are the average maximum score for the pool.**

reestablish secure settings; however, it again requires multiple generations.

In contrast, the EA with history-based mutation is able to quickly reinstate secure configuration settings as the attacks change, as seen in Figure 3(b). In this experiment, phases 1 and 2 are the training phases for the respective parameter-setting estimations. It is the first occurrence of either attack and the setting counts are only collected. As a result, the performance is similar to the EA with standard mutation during these phases, seen in Figure 3(a). However during phase 3, the setting estimations developed during phase 1 are reinstated and the score for parameter set 1 is quickly returned to the maximum value. The same event occurs during phase 4, where the setting estimations developed during phase 2 were reinstated resulting in higher scores.

Similar trends were observed for the average pool score, which suggests the general health of all the configurations. As seen in Figure 4, the average score of the attacked parameters steadily improves during the corresponding attack phase. During the same time, the score for other non-attacked set of parameters lowers. During phases 1 and 2, standard and history-based mutation provided essentially the same performance. However during phases 3 and 4, average fitness for the history-based mutation is higher.

## 6. CONCLUSIONS AND FUTURE WORK

Evolutionary Algorithms (EAs) offer a unique approach to maintaining secure configurations. Using a series of selection, recombination, and mutation processes, configurations can evolve to address many misconfiguration issues. Unfortunately this continual evolution may cause EAs to lose previously secure configuration settings. This paper introduced a modified mutation process that chooses parameter settings based on mutation guided by probability densities reflecting the resolution of insecure parameter settings. Settings that were deemed secure in previous generations will be more likely to be used when past threats are reintroduced.

Although the preliminary results indicate history-based mutation has merit, there are several areas for future work. For example, better understanding of the trade-offs between training duration and pool size would benefit implementation. History-based mutation has demonstrated the ability to determine secure parameter settings, but additional research to identify exactly the targeted parameters would be useful for administrators. This could allow merging of multiple density estimators that could be used to safeguard against multiple attacks.

## 7. ACKNOWLEDGEMENTS

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.

## 8. REFERENCES

[1] Xiangping Bu, Jia Rao, and Cheng-Zhong Xu. A reinforcement learning approach to online web systems auto-configuration. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, pages 2–11, 2009.

[2] Michael B. Crouse and Errin W. Fulp. A moving target environment for computer configurations using genetic algorithms. In *Proceedings of the 4th Symposium on Configuration Analytics and Automation (SafeConfig 2011)*, 2011.

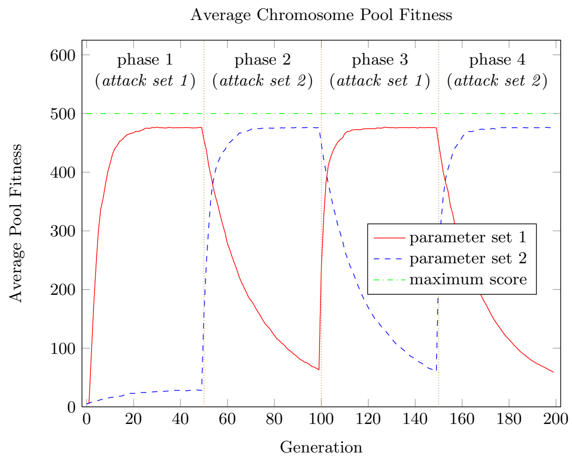
[3] Kenneth Alan De Jong. *Evolutionary Computation: A Unified Approach.* MIT Press, 2006.

[4] Carsten Eiram and Brian Martin. The CVSSv2 shortcomings, faults, and failures formulation. Technical report, Forum of Incident Response and Security Teams (FIRST), 2013.

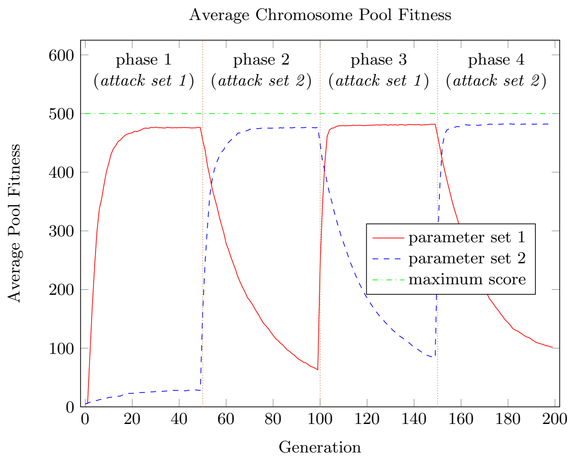
[5] Center for Internet Security. Security and assessment benchmarktools. `http://www.cisecurity.org/resources-publications/`.

[6] Seth Hanford. Common vulnerability scoring system, v3 development update. Technical report, Forum of Incident Response and Security Teams (FIRST), 2013.

[7] David J. John, Robert W. Smith, William H. Turkett, Daniel Cañas, and Errin W. Fulp. Evolutionary based moving target cyber defense. In *Proceedings of the*



(a) Standard mutation.



(b) History-based mutation.

**Figure 4: Attack experiments consisting of four phases. Phases 1 and 3 attacked parameters from set 1, while phases 2 and 4 attacked parameters from set 2. Fitness values are the average score for the pool.**

*Genetic and Evolutionary Computation Conference (GECCO) Workshop on Genetic and Evolutionary Computation in Defense, Security and Risk Management (SecDef)*, 2014.

[8] Emre Kycyman. Discovering correctness constraints for self-management of system configuration. In *Proceedings of the First International Conference on Autonomic Computing*, pages 28–35, Washington, DC, USA, 2004. IEEE Computer Society.

[9] Brian Lucas, Errin W. Fulp, David J. John, and Daniel Cañas. An initial framework for evolving computer configurations as a moving target defense. In *Proceedings of the 9th Annual Cyber and Information Security Research Conference (CISRC)*, 2014.

[10] NIST. Consensus security configuration checklist. `http://web.nvd.nist.gov/view/ncp/repository`.

[11] Jon Oberheide, Evan Cooke, and Farnam Jahanian. If it ain't broke, don't fix it: challenges and new directions for inferring the impact of software patches. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, 2009.

[12] P. Pal, R. Schantz, A. Paulos, and B. Benyo. Managed execution environment as a moving-target defense infrastructure. *Security Privacy, IEEE*, 12(2):51–59, Mar 2014.

[13] Mauro Pezzè and Cheng Zhang. Automated test oracles: A survey. *Advances in Computers*, 95:1–48, 2014.

[14] Robert Smith. Evolutionary strategies for secure moving target configuration discovery. Master's thesis, Computer Science, Wake Forest University, 2014.

[15] Monchai Sopitkamol. A method for evaluating the impact of software configuration parameters on e-commerce sites. In *In Proceedings of the ACM 5th International Workshop on Software and Performance*, pages 53–64, 2005.

[16] L. Spitzner. The honeynet project: trapping the hackers. *Security & Privacy, IEEE*, 1(2):15–23, Mar 2003.

[17] R. Thonangi, V. Thummala, and S. Babu. Finding good configurations in high-dimensional spaces: Doing more with less. In *Modeling, Analysis and Simulation of Computers and Telecommunication Systems, 2008. MASCOTS 2008. IEEE International Symposium on*, pages 1–10, Sept 2008.

[18] Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J. Wang, Chun Yuan, and Zheng Zhang. Strider: A black-box, state-based approach to change and configuration management and support. In *Proceedings of the 17th USENIX conference on System Administration*, pages 159–172, 2003.

[19] Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J. Wang, Chun Yuan, and Zheng Zhang. Strider: a black-box, state-based approach to change and configuration management and support. *Science of Computer Programming*, 53(2):143 – 164, 2004.

[20] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Configuration debugging as search: finding the needle in the haystack. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6*, 2004.

[21] Bowei Xi, Zhen Liu, Mukund Raghavachari, Cathy H Xia, and Li Zhang. A smart hill-climbing algorithm for application server configuration. In *Proceedings of the 13th International Conference on World Wide Web*, pages 287–296, 2004.

[22] Ding Yuan, Yinglian Xie, Rina Panigrahy, Junfeng Yang, Chad Verbowski, and Arunvijay Kumar. Context-based online configuration-error detection. In *Proceedings of the USENIX Annual Technical Conference (USENIX '11)*, June 2011.

[23] Wei Zheng, Ricardo Bianchini, and Thu D Nguyen. Massconf: automatic configuration tuning by leveraging user community information. *ACM SIGSOFT Software Engineering Notes*, 36(5):283–288, 2011.