8-16-2021

# Automated Statistical Structural Testing Techniques and Applications

Yang Shi
*Portland State University*

# Automated Statistical Structural Testing

# Techniques and Applications

by

Yang Shi

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Electrical and Computer Engineering

Portland State University
2021

## ABSTRACT

Statistical structural testing(SST) is an effective testing technique that produces random test inputs from probability distributions. SST shows superiority in fault-revealing power over random testing and deterministic approaches since it heritages the merits from both of them. SST ensures testing thoroughness by setting up a probability lower-bound criterion for each structural cover element and test inputs that exercise a structural cover element sampled from the probability distribution, ensuring testing randomness. Despite the advantages, SST is not a widely used approach in practice. There are two major limitations. First, to construct probability distributions, a tester must understand the underlying software's structure, which is difficult to obtain in the real-world testing scenario. Although automated search is able to construct probability distributions iteratively, the efficiency remains unsatisfiable. Second, SST is limited to unit testing or programs with small structural complexity.

The first research objective is to analyze the root cause of the unsatisfiable efficiency. It turns out that a strong statistical structural coverage criterion results in a high impact of the noisy fitness estimation. Hence, we proposed a weakened criterion that can significantly reduce the search time without the loss of substantial fault-detecting power. We also developed a search algorithm called $CACO_{\mathcal{R}}$ that resists the noisy fitness influence. The input distribution model harnesses a set of weighted uniform distributions over the input domain space, which is enumerated effectively by the constrained ant colony optimization strategies. Experimental studies demonstrate the excellent search performance of the $CACO_{\mathcal{R}}$ algorithm and the high-grade fault-detecting ability of the input distributions produced by the algorithm.

The second research objective is to apply the SST strategy to the real-world testing industries. The mutation-based fuzzing technique (e.g., AFL) has enjoyed great success due to the automation of the testing process and, more importantly, the ability to discover critical vulnerabilities. The role of SST in the fuzzer is a test input provider when AFL is stuck in discovering new program paths. We noticed that AFL is often stuck in testing the code structures with deeply nested conditions and scanty input sub-domain spaces. AFL's mutation strategy often produces inputs that stay in the same or upper condition level and hardly trigger the deeper level. For scanty input subdomain space, AFL acts as random testing. We perform a comprehensive search to construct input distribution such that both of the outgoing edges of a conditional statement are guaranteed to be triggered with a probability lower bounds threshold. The experimental study demonstrates that the lead time to discover bugs with SST is significantly decreased.

## DEDICATION

**This work is dedicated to my daughter Jiayan who joined us when I was writing my dissertation, for giving me unlimited happiness and pleasure.**

## ACKNOWLEDGMENTS

Throughout my Ph.D. career, I have received a great deal of support and assistance. I cannot make this dissertation possible without their supports.

I would first like to thank my supervisor, Dr. Xiaoyu Song, whose expertise was invaluable in formulating the research questions and methodology and provided guidance throughout my studies. I would also like to thank my dissertation committee members, Dr. Jingke Li, Dr. Fu Li, and Dr. Marek Perkowski, for serving on my committee.

I sincerely thank my parents for their unconditional trust, timely encouragement, and endless patience. I thank with love to Siying and Jiayan, my wife and one-year-old daughter. Finally, I deeply thank my best friends, Yansheng Li, Yiwei Li, Ding Luo, and Xiao Li, who spent an incredible journey with me in Portland, OR.

TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# Chapter 1

# INTRODUCTION

## PROBLEM STATEMENT

Software testing plays a critical role in the software development process. In general, it takes more than 40%-50% of the total development costs [1]. To increase the quality of a test data set, researchers have conducted extensive investigations in both testing methodologies and algorithms. This dissertation is concerned with two testing techniques.

Statistical Structural Testing(SST) is a generalized random testing method. Instead of applying the uniform distribution in random testing, SST constructs a probability distribution(a.k.a input distribution) such that every structural cover element (e.g, branch cover element) can be triggered at least with equal probability. In this way, test randomness characteristic is preserved meanwhile achieving a high level of code coverage. It is demonstrated that test randomness is crucial to discover errors that can be triggered with a scanty test input domain space [2]. Despite the advantages, SST is not a widely used approach in practice. To construct probability distributions, a tester requires to understand the underlying software's structure which is difficult to obtain in the real-world testing scenario. Although Clark [3] demonstrated that automated search is able to derive such probability distributions, the efficiency is unsatisfiable. Search is suitable only for unit testing or programs with elementary structural complexity.

Fuzzing is an automated testing technique to discover vulnerabilities by feeding managed random inputs to the system under test(SUT). With the rapid growth of cloud computing, fuzzing is employed in the cloud platform like Google's OSS-Fuzz platform, which continuously tests open-source applications and found over 1000 bugs in 5 months [4]. The mutation-based fuzzing technique is one of the most popular fuzzing strategies at present. The fuzzers such as AFL apply genetic operators to mutate and reproduce test inputs. AFL employs a feedback loop to assess a test input's goodness. A test input that discovers a new path is retained and will be mutated further in the next cycles. AFL relies on a huge amount of mutated test inputs in the hope to discover new paths. This blind mutation strategy without the structural knowledge of the SUT makes AFL inefficient.

This dissertation is concerned with the search-based SST's(SBSST) efficiency as well as the fuzzer's mutation strategy. We observe the following key challenges in the two testing methodologies.

- **Curse of uncertainty in SBSST**. The curse of uncertainty in SBSST originates from the fitness calculation, which is an estimation of the cover element's triggering probability. The under/overestimate of a probability distribution leads the search algorithm in the wrong direction, causing an expansion in the search's lead time.

- **Path discovery deficiency in AFL**. Due to the blind mutation strategy, AFL performs badly in code structural characteristics with deeply nested conditions and conditions with scanty input subdomain spaces. For deeply nested conditions, AFL executes the same or above level of conditions in the majority of cycles. For conditions with scanty input subdomain spaces, AFL's mutation strategy acts as random testing and finds the input subdomain space with sheer luck.

## PROPOSED APPROACH

We have proposed a scalable framework that generates test inputs as a complementary for mutation-based fuzzing. The framework leverages improved SBSST's technique that randomly produces test inputs with prioritization. The key components are listed in the following.

- **Improved SBSST**. We mitigate the noisy fitness estimation impact from two perspectives. First, we analyze how the coverage criterion's strength affects the noisy fitness estimation. It turns out that a strong statistical structural coverage criterion results in a high impact of the noisy fitness estimation. Hence, we proposed a weakened criterion that can significantly reduce the search time but without loss of substantial fault-detecting power. Second, we investigate search algorithms that resist noisy fitness estimation. Based on the existing Ant Colony Optimization, we developed a constrained ACO algorithm ($\text{CACO}_{\mathbb{R}}$) that is dedicated to the SST problem. Experimental studies demonstrate the excellent search performance of the ($\text{CACO}_{\mathbb{R}}$) algorithm and the high-grade fault-detecting ability of the input distributions produced by the algorithm.

- **Enhanced AFL**. We developed a test input generator called SBSFuzz with the search-based SST technique for AFL. The test input generator works as complementary to mutation-based fuzzing when AFL is stuck in discovering new program paths. In the test input generator, we maintain a trace graph and a set of optimized probability distributions. To efficiently discovering new paths, we prioritize test inputs in terms of the exercised program traces. The *input-sensitivity prioritization* allows test inputs in the scanty input subdomain spaces to be sampled with priority. The *path-depth prioritization* allows test inputs that execute paths with deeper depth to be sampled with priority. The sampled test input is then fed back to AFL for testing.

More details of these components are summarized in the following.

**p-L1-Max criterion**. The proposed criterion is called *fairness-enhanced-sum-of-triggering-probability*, short for p-L1-Max. The p-L1-Max criterion evaluates an input distribution based on the estimated sum of triggering probabilities on branches, and the input distribution must satisfy the fairness property which measures the L2 distance from the input distribution to the uniform distribution. We use a parameter p to manually adjust the importance of the two objectives. In this way, the input distribution generation process can be tuned to bias on either L1-Max or fairness. The traditional criterion uses the lowest triggering probability among all branches (Tri-Low-Bound) which is considered a strong criterion. Our experiments show that by using a relaxed constraint p-L1-Max as an evaluation criterion, the time consumption on search is significantly reduced. However, the fault-detecting ability remains at the same level.

**Fault-detecting ability**. We leverage the mutation testing technique to produces templated buggy binaries for measuring the fault-detecting ability of input distributions. However, what is the definition of an input distribution's fault-detecting ability? We introduce a definition of expected faults found in the effective test set size region. To measure the effective test set size region, we present a theoretical analysis of the expected faults found in terms of test set sizes. We use the uniform distribution as a baseline to derive the effective test set size region's definition.

**The CACO$_{\mathbb{R}}$ algorithm** We modeled the input distribution construction process as a resource allocation optimization problem and utilizes the ant colony metaphor to solve the problem. We modeled the input distribution as a sum of weighted one-dimensional uniform distributions. Each uniform distribution occupies a non-overlapped, consecutive input sub-domain space (a.k.a a bin). The objective is to assign the proper amount of weight and input sub-domain size to each uniform component. The ant colony optimization algorithm(ACO) is an ideal algorithm to solve such optimization problems with its extension generalizes the search domain space from discrete to continuous. ACO algorithm can minimize the negative impact of noisy fitness estimation. Since the fitness of an individual is aggregated in the long run, the noisy fitness estimation influence will decrease as search iteration increases.

**SBSFuzz**. The SBSFuzz consists of three components: a modified AFL that can communicate with other processes via an inter-process protocol, a Trace analyzer that maintains a trace graph and produces test inputs, and a comprehensive search engine that produces input distributions. SBSFuzz utilizes a modified AFL as the main fuzz engine. The trace analyzer runs its instrumentation version of the target binary with the test input to obtain a piece of *fine-grained* path information. It also starts a thread to initiate the comprehensive search engine when necessary.

## DISSERTATION OUTLINE

Chapter one provides an introduction. Chapter two provides background knowledge. Chapter three describes the noisy fitness impact and the proposed criteria. Chapter four introduces the proposed search algorithm. Chapter five introduces the application of SST in mutation-based fuzzing. Chapter six introduces other search-based SST methods. Chapter seven briefly describes the related works. Chapter eight makes the conclusion and describes the future works.

# Chapter 2

# BACKGROUND

## STATISTICAL STRUCTURAL TESTING

An SUT usually consists of $n \geq 1$ inputs. Each input is associated with a domain space. In this article, we assume the inputs are independent. We use a domain $\mathcal{D} = [L, H]$ representing the cross product of the $n$ domain spaces. $L$ denotes the lowest element, and $H$ denotes the highest element in $\mathcal{D}$.

**Input Distribution:** An input distribution, denoted as $P(x)$ is a discrete probability distribution over an input domain space $\mathcal{D}$, where $x$ represents a test input, $x \in \mathcal{D}$.

In SST, an SUT is essentially treated as a control flow graph. In the context of this dissertation, an edge is also known as a branch cover element (BCE). A BCE is said to be triggered by an input $x$ if the path in a control flow graph executed by $x$ contains the corresponding edge. Hence, for the entire input domain space $\mathcal{D}$, there exists a subset of the input domain space that triggers each branch cover element.

**Triggering Probability:** Suppose that an SUT has $C = \{c_1, \ldots, c_m\}$ BCEs. Let $c_i$ denote the i-th BCE and $\mathcal{D}_{c_i}$ denotes the subset of the input domain space that triggers $c_i$. The probability of triggering a cover element $c_i$ is defined as the sum of the probabilities of each input in $\mathcal{D}_{c_i}$. Formally,

$$tri_i = \sum_{x \in \mathcal{D}_{c_i}} P(x) \tag{2.1}$$

where $tri_i$ is the probability of a randomly sampled input from the input distribution triggering $c_i$.

However, for SUTs with a sizeable input domain space, it is often infeasible to calculate the triggering probabilities using this equation. In these scenarios, the use of samples to estimate the triggering probabilities becomes necessary. Suppose, given a sampled test input set, the number of times that each BCE is triggered is $\{k_1, \ldots, k_m\}$ respectively. It is known that the number of triggered $c_i$ follows a binomial distribution with the triggering probability $tri_i$[5]. In this case, the unbiased estimation of triggering probability for $c_i$ is

$$\hat{tri}_i = \frac{k_i}{\sum_{i=1}^{m} k_i} \tag{2.2}$$

### 2.1.1 Input Distribution Model

In this dissertation, we mainly consider the input distribution model as a sum of weighted uniform distributions, where each uniform distribution is applied to an input sub-domain space, also known as a bin. For an input distribution model of $k$ non-overlapping consecutive bins, the input domain space is split by the bins into intervals

$$\mathcal{D} = \Big\{ \lfloor \delta_0, \delta_1 \rceil, \lfloor \delta_1, \delta_2 \rceil, \ldots, \lfloor \delta_{k-1}, \delta_k \rceil \Big\} \tag{2.3}$$

where $\delta_0 = L$ and $\delta_k = H$. Other $\delta_i s$ denotes the boundary value between the i-th bin and the (i+1)-th bin. Since bins are non-overlapping, $\delta_{i-1} \leq \delta_i$. Let

$w = \{w_1, \ldots, w_k\}$ be the set of weights applied to each uniform distribution. The weight vector $w$ satisfies the following constraints:

$$\sum_{i=1}^{k} w_i = 1,\ w_i \geq 0\ \forall w_i \in w \tag{2.4}$$

Given a set of values for the weights $\{w_1, \ldots, w_k\}$ and the boundaries $\{\delta_1, \ldots, \delta_k\}$, the probability of selecting an input $x \in \mathcal{D}$ is

$$P(x) = \sum_{i=1}^{k} w_i * U(x), x \in \lfloor \delta_{i-1}, \delta_i \rceil \tag{2.5}$$

where $U(x)$ represents a uniform distribution component in the model.

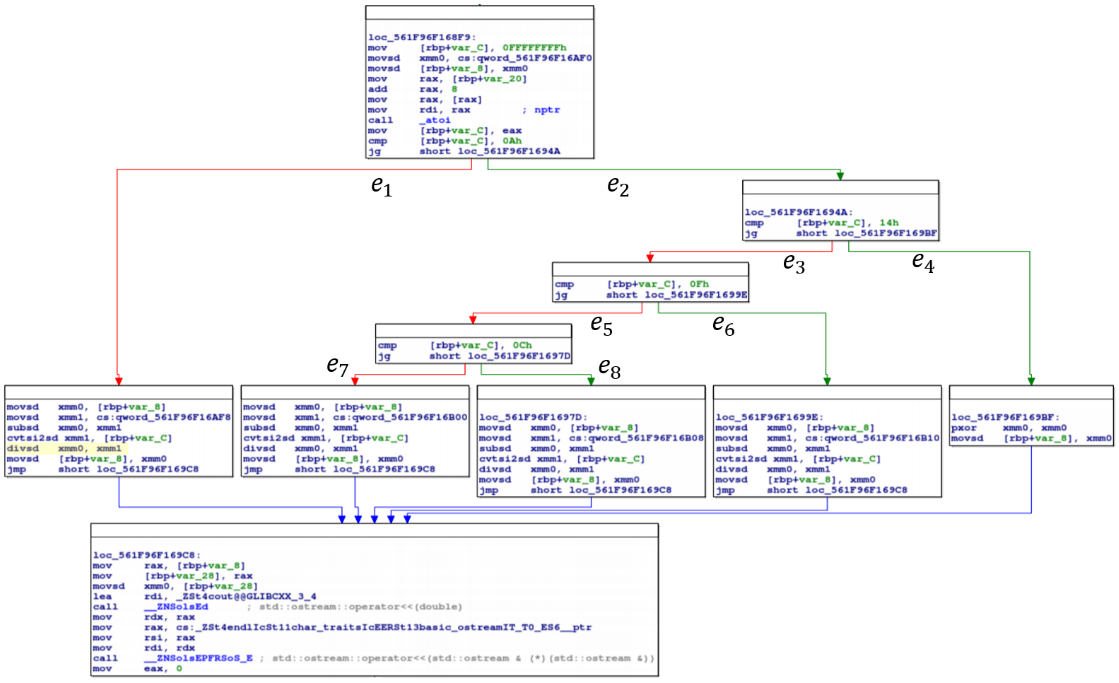
As an example, consider a SUT that takes an input $x$ whose domain space ranges from 0 to 99. Its control flow graph is shown in Figure 2.1. The SUT has 8 branch cover elements (BCEs) $\{e_1, \ldots, e_8\}$ which forms 5 linearly independent paths, shown in Table 2.1. To ensure all BCE's triggering probabilities are maximized, each linearly independent path should be triggered with equal probability. A test input's probability$P(x)$ is then assigned to the reciprocal of its subdomain's cardinality multiply by $\frac{1}{5}$ to ensure the 20% BCEs' triggering probabilities.

| **Linearly Independent Paths** | **Input Sub-domains** | **Sub-domain Size** | $P(x)$ |
|---|---|---|---|
| $e_1$ | $x < 10$ | 10 | 0.02 |
| $e_2$ $e_4$ | $x > 20$ | 80 | 0.0025 |
| $e_2$ $e_3$ $e_6$ | $x > 15$ & $x \leq 20$ | 5 | 0.04 |
| $e_2$ $e_3$ $e_5$ $e_7$ | $x > 10$ & $x \leq 12$ | 2 | 0.1 |
| $e_2$ $e_3$ $e_5$ $e_8$ | $x > 12$ & $x \leq 15$ | 3 | 0.067 |

Table 2.1: An Example of Input Distribution

### 2.1.2 Fault Discovery Ability Model

This dissertation leverages Duran's fault discovery ability model. Suppose that an input domain space is re-organized into many consecutive, non-overlapped subsets.



```
loc_561F96F168F9:
mov      [rbp+var_C], 0FFFFFFFFh
movsd    xmm0, cs:qword_561F96F16AF0
movsd    [rbp+var_8], xmm0
mov      rax, [rbp+var_20]
add      rax, 8
mov      rax, [rax]
mov      rdi, rax         ; nptr
call     _atoi
mov      [rbp+var_C], eax
cmp      [rbp+var_C], 0Ah
jg       short loc_561F96F1694A
```

```
loc_561F96F1694A:
cmp      [rbp+var_C], 14h
jg       short loc_561F96F169BF
```

```
cmp      [rbp+var_C], 0Fh
jg       short loc_561F96F1699E
```

```
cmp      [rbp+var_C], 0Ch
jg       short loc_561F96F1697D
```

```
movsd    xmm0, [rbp+var_8]
movsd    xmm1, cs:qword_561F96F16AF8
subsd    xmm0, xmm1
cvtsi2sd xmm1, [rbp+var_C]
divsd    xmm0, xmm1
movsd    [rbp+var_8], xmm0
jmp      short loc_561F96F169C8
```

```
movsd    xmm0, [rbp+var_8]
movsd    xmm1, cs:qword_561F96F16B00
subsd    xmm0, xmm1
cvtsi2sd xmm1, [rbp+var_C]
divsd    xmm0, xmm1
movsd    [rbp+var_8], xmm0
jmp      short loc_561F96F169C8
```

```
loc_561F96F1697D:
movsd    xmm0, [rbp+var_8]
movsd    xmm1, cs:qword_561F96F16B08
subsd    xmm0, xmm1
cvtsi2sd xmm1, [rbp+var_C]
divsd    xmm0, xmm1
movsd    [rbp+var_8], xmm0
jmp      short loc_561F96F169C8
```

```
loc_561F96F1699E:
movsd    xmm0, [rbp+var_8]
movsd    xmm1, cs:qword_561F96F16B10
subsd    xmm0, xmm1
cvtsi2sd xmm1, [rbp+var_C]
divsd    xmm0, xmm1
movsd    [rbp+var_8], xmm0
jmp      short loc_561F96F169C8
```

```
loc_561F96F169BF:
pxor     xmm0, xmm0
movsd    [rbp+var_8], xmm0
```

```
loc_561F96F169C8:
mov      rax, [rbp+var_8]
mov      [rbp+var_28], rax
movsd    xmm0, [rbp+var_28]
lea      rdi, _ZSt4cout@@GLIBCXX_3_4
call     __ZNSolsEd       ; std::ostream::operator<<(double)
mov      rdx, rax
mov      rax, cs:_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6__ptr
mov      rsi, rax
mov      rdi, rdx
call     __ZNSolsEPFRSoS_E ; std::ostream::operator<<(std::ostream & (*)(std::ostream &))
mov      eax, 0
```

Figure 2.1: An example of control flow graph

In each subset, the test inputs are uniformly selected. Let $\theta_i$ be the failure rate of the i-th partition, which refers to the probability that a randomly selected input triggers the system failure. Let pi be the probability of selecting the i-th partition, k be the total number of partitions and n be the test set size. The expected number of errors found under the assumption that each partition contains one error is formulated as follows:

$$E(k, n, \theta, \mathbf{p}) = k - \sum_{i=1}^{k} (1 - p_i \theta_i)^n$$

## EVOLUTIONARY ALGORITHMS

Evolutionary Algorithm(EA) are heuristic algorithms which solve problems with rules of thumb or common sense approaches. Heuristic algorithms usually are not expected to find the best answer to a problem but are only expected to find solutions that are "close enough" to the best. In this dissertation, we consider two EA, the Genetic algorithm (GA) and the Ant Colony Optimization (ACO) algorithm.

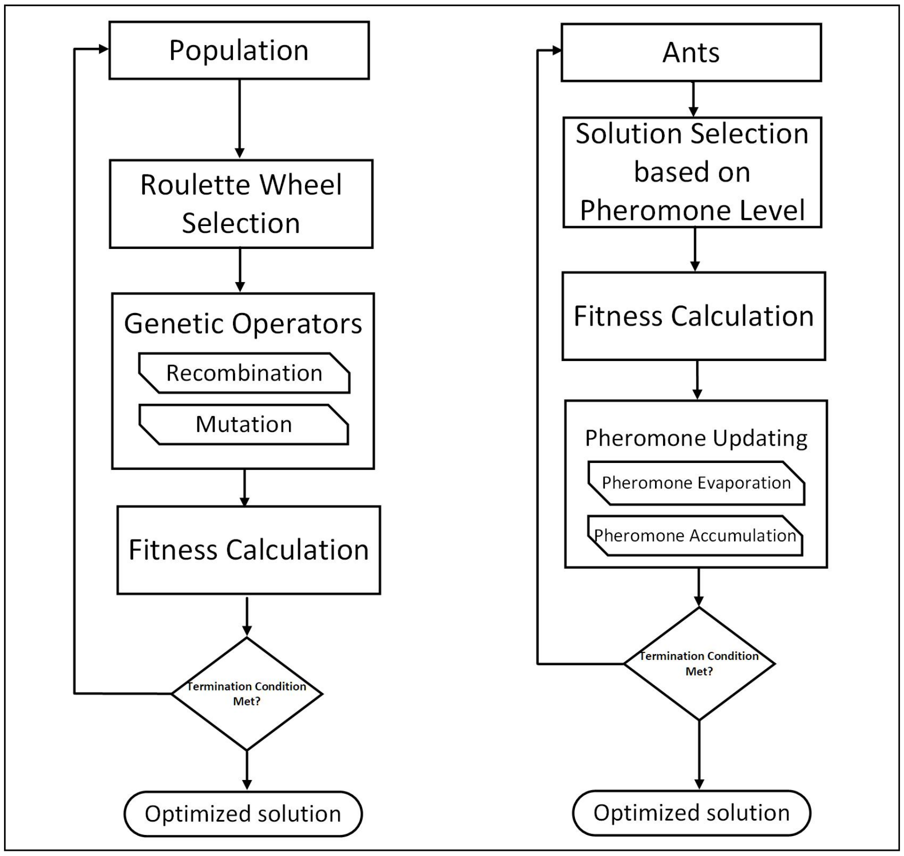


Figure 2.2: Workflow of Genetic Algorithm(left) and Ant Colony Optimization(right)

### 2.2.1 Genetic Algorithm

The workflow of GA is shown on the left side of Figure 2.2. GA starts from initializing a pool of solutions. Then, it selects individuals from the pool with *roulette-wheel slection*. Then its genetic operation produces child solutions to form a new solution pool from the chosen solutions. The fitness function then evaluates all the child solutions. If the termination condition does not meet, the new solution pool undergoes the same process. GA usually stops with three termination conditions. First, the iteration reaches the maximum limit. Second, the fitness of the best solution reaches the target value. Third, the fitness of the best solution does not improve over the last $n$ iterations.

The Genetic operations are the essence of GA. The conventional Genetic Algorithm(GA) is used to optimize a bit string. The process to recombine two bit-strings is called recombination. A standard recombination method is called *one-point-crossover*, in which the operator swaps two portions of bit-strings from a *crossover point*. The crossover point is a randomly selected value between 0 and the maximum index of a bit string. The process to flip a bit of the bit-string is called mutation. A general method is to select a small mutation probability $m$. After the crossover process, each bit in the child bit-string has a $m$ probability of flipping to the opposite value. The probability value generally ranges from [0, 1]. Too high of a mutation probability makes the GA acting as a random search; Too low of a mutation probability results in the search falls into local optima.

### 2.2.2 ACO Algorithm

The ACO algorithm was first developed to solve the traveling salesman problem. Each ant travels from one city to another until all cities are passed and then deposits pheromone on the path. Pheromones are not only deposited, but they also evaporate. The probability that an ant travels from one city to another is proportional to the pheromone levels between the cities. The workflow of the ACO algorithm is shown on the right side of Figure 2.2. The algorithm begins by initializing a pool of ants. All the ants start traveling simultaneously. Each ant randomly selects a city to travel with *pheromone-level-proportionate-selection*. After finishing traveling, the fitness evaluation function calculates the exercised path distance and derives the pheromone amount. Then on each visited path, the pheromone levels are accumulated based on the amount. In the meantime, pheromone levels on each unvisited path are evaporated at a constant rate. This process continues until the shorted path distance does not change over the last $n$ iterations.

There are many variations of the ACO algorithm. In this dissertation, we adopt the one from [6] that applies the ACO algorithm to the real-domain space (ACO$_R$. The essence of ACO is the updating pheromone process. In ACO$_R$, pheromone accumulation and evaporation functions are simulated by a weighted Gaussian distribution, which we will give a detailed description in chapter 4.

## SEARCH-BASED SOFTWARE TESTING

Search-based software testing (SBST), as the name implies, utilizes heuristic search algorithms to solve software testing problems. Generally, two issues need to be considered in order to apply the SBST methods. First, problem encoding. The candidate solutions should be encoded in a way that the search algorithm is able to manipulate. Second, fitness evaluation. The fitness function guides the search algorithm towards a better solution. The fitness function is problem-specific and needs to be defined accordingly. For instance, *Temporal Testing* tries to find a system's best-case and worst-case execution times. The fitness function is simply a measure of the system's wall-time [7; 8]. *Structural Testing* tries to utilize a test set to maximize a structural coverage criterion. A commonly used fitness function proposed by Wegener et al. [9] incorporates two metrics, known as the *approach level* and the *branch distance*. In this dissertation, we focus on the SBST's scheme for the SST problems.

# Chapter 3

# EFFECTIVENESS ASSESSMENT OF SEARCH-BASED STATISTICAL STRUCTURAL TESTING

Search-based statistical structural testing (SBSST) is a promising technique that uses automated search to construct input distributions for statistical structural testing. It has been proved that a simple search algorithm, for example, the hill-climber is able to optimize an input distribution. However, due to the noisy fitness estimation of the minimum triggering probability among all cover elements (Tri-Low-Bound), the existing approach does not show a satisfactory efficiency. Constructing input distributions to satisfy the Tri-Low-Bound criterion requires an extensive computation time. Tri-Low-Bound is considered a strong criterion, and it is demonstrated to sustain a high fault-detecting ability. In this chapter, we try to answer the following question: if we use a relaxed constraint that significantly reduces the time consumption on search, can the optimized input distribution still be effective in fault-detecting ability? We propose a type of criterion called fairness-enhanced-sum-of-triggering-probability (p-L1-Max). The criterion utilizes the sum of triggering probabilities as the fitness value and leverages a parameter $p$ to adjust the uniformness of test data generation. We conducted extensive experiments to compare the computation time and the fault-detecting ability between the two criteria. The result shows that the 1.0-L1-Max criterion has the highest efficiency, and it is more practical to use than the Tri-Low-Bound criterion. To measure a criterion's fault-detecting ability, we introduce a definition of expected faults found in the effective test set size region. To measure the effective test set size region, we present a theoretical analysis of the expected faults found with respect to various test set sizes and use the uniform distribution as a baseline to derive the effective test set size region's definition.

## OVERVIEW

Statistical structural testing has been studied for decades. In SST, test inputs are sampled from probability distributions (a.k.a, input distributions) over the input domain space. The distributions guarantee that a sampled test input has a probability greater than a threshold to trigger each branch cover element (BCE) under test. This criterion increases the chance of triggering BCEs associated with a small input sub-domain space, resulting in a higher fault-detecting ability than random testing [10]. Constructing such distributions is not a trivial work. A tester needs to know the input sub-domain space associated with each cover element and then assign the right probabilities to each space to create an optimal input distribution. Fortunately, this process can be automated by the search-based software testing framework. Search-based SST(SBSST) is similar to the traditional search-based coverage-driven approaches where a test input set is refined during the system under test (SUT's) run-time. However, SBSST optimizes an input distribution's parameter values and uses sampled test input sets to evaluate fitness. A general evaluation criterion is the Triggering Probability Lower Bound (Tri-Low-Bound), where the minimum triggering probability among all BCEs under test is used as the fitness value. Poulding and Clark in [11] demonstrate the effectiveness of using the hill-climbing algorithm to search input distributions with the Tri-Low-Bound criterion. However, the time consumption on search is a significant concern. The critical issue is that the estimated triggering probabilities cause over/underestimation, which significantly misleads the search direction. Moreover, if a BCE under test is associated with a diminutive input sub-domain space, triggering the BCE is considered a rare event. The probability estimation of a rare event is usually inaccurate. We conducted a small experiment to show the problem: Our synthetic

SUT has two inputs, with each consist of 30 elements. A cover element C can be triggered by 4 non-consecutive test inputs, and the sample set used to estimate fitness has 90 test inputs. We use the hill-climbing algorithm with a Tabu list to search for an input distribution that maximizes C's triggering probability. The fitness is estimated with the Wilson Score approach with continuity correction [12]. Over 5000 iterations, fitness swings around 0.01, and the confidence band ranges from near 0 to an average around 0.15, which could not provide helpful information to guide the search direction moving forward. Tri-Low-Bound is considered a strong criterion since every BCE's triggering probability is constrained. In this chapter, we answer the following question: **If we use a relaxed constraint that significantly reduces the time consumption on search, can the optimized input distribution still be effective in fault-detecting ability?** We propose a new criterion called fairness-enhanced-sum-of-triggering-probability (p-L1-Max). Instead of Tri-Low-Bound, the sum of triggering probabilities could reduce the noisy fitness influence by estimating the group of events. However, it causes the search direction biasing to one input sub-domain space, whereas the rests take zero chances to be sampled. Hence, we also take fairness a parameter $p$ into consideration, which tunes the distribution to be uniform. A question raised is how to compare two criteria. In SST, test inputs are sampled from distributions, and the test set size is proportional to fault-detecting ability. In this chapter, we provide a theoretical analysis of the fault-detecting ability in terms of various test set sizes. We use the uniform distribution as a baseline to derive the effective test set size region $\mathscr{R}$ where SST outperforms random testing and determine the expected faults found in $\mathscr{R}$ as the effectiveness measure of criteria. To compare two criteria, we use the effectiveness-to-cost ratio, where cost is the wall-time on search. This chapter starts from presenting a method called effectiveness-to-cost ratio to evaluate the fault-detecting ability of criteria for SST problems. Then, we introduce a new criterion(p-L1-Max) and conducted a series of experiments to compare the proposed and traditional criteria. Our results show that the proposed criterion has a better effectiveness-to-cost ratio, and it is more realistic in practical uses.

## EFFECTIVENESS ESTIMATION OF INPUT DISTRIBUTIONS

An optimized input distribution is a biased uniform distribution. The point of biasing the uniform distribution is to detect faults more effectively than random testing. Given a test set size, if the number of faults found by the uniform distribution outperforms or equal to the biased input distribution, the biased input distribution should have no effectiveness, since random testing does not require the input distribution construction process. Hence, to investigate an input distribution's effectiveness, we should determine the effective test set sizes.

### 3.2.1 Effective Test Set Size

To find effective test set size theoretically, we adopt Duran's fault revealing ability model. Suppose that an input domain space is re-organized into many consecutive, non-overlapped subsets. In each subset, the test inputs are uniformly selected. Let $\theta_i$ be the failure rate of the i-th partition, which refers to the probability that a randomly selected input triggers the system failure. Let $p_i$ be the probability of selecting the i-th partition, $k$ be the total number of partitions and $n$ be the test set size. The expected number of errors found under the assumption that each partition contains one error is formulated as follows:

$$E(k, n, \theta, \mathbf{p}) = k - \sum_{i=1}^{k} (1 - p_i\theta_i)^n \tag{1}$$

For a uniform input distribution, $p_i = \frac{1}{k},\ \forall i \in \{1, \ldots, k\}$. Then, the effectiveness of a biased input distribution is the following:

$$\mathscr{E} = E_b - E_r = \sum_{i=1}^{k}(1 - \frac{1}{k}\theta_i)^n - \sum_{j=1}^{k}(1 - p_j\theta_j)^n \tag{2}$$

We are interested in the maximum and minimum of $\mathscr{E}$ with respect to various $n$. This function is a summation over exponential functions. An intuitive re-formation can be done as follows: Suppose that a set $\mathcal{U} = \{(1 - \frac{1}{k}\theta_1), \ldots, (1 - \frac{1}{k}\theta_k)\}$, A set $\mathcal{B} = \{(1 - p_1\theta_1), \ldots, (1 - p_k\theta_k\}$ and $C(x)$ is an indication function that $C(x) = 1$ if $x \in \mathcal{U}$; $C(x) = -1$ if $x \in \mathcal{B}$. Then, Equation 2 can be written as follows:

$$\mathscr{E} = \sum_{\alpha \in \mathcal{U} \cup \mathcal{B}} C(\alpha)\alpha^n \tag{3}$$

Shestopaloff in [13] proves the following corollary of the above function: "if there exists a sequence of $\alpha$ such that $0 < \alpha_N \ldots \alpha_0 < 1,\ C_0 > 0$. This series can change its algebraic sign a maximum of two times. It can have a maximum of two extrema. It monotonically converges to zero after the second extremum, which is always a maximum." $\alpha_0$ refers to the maximum fault-detecting rate of a partition, it can be either from $U$ and $B$. We analyze them separately. let $T_i$ denote index sets that stores indexes for $U$ and $B$. Specifically,

$$\begin{aligned} \mathcal{T}_1 &= \{t_i | p_i > \tfrac{1}{n}\} \\ \mathcal{T}_2 &= \{t_j | p_j < \tfrac{1}{n}\} \end{aligned} \tag{4}$$

Suppose that $\alpha_0$ is an element in $\{\mathcal{U}_i | i \in \mathcal{T}_1\}$. Forming an order over $\mathcal{U} \cup \mathcal{B}$ with one algebraic sign change is impossible. With two algebraic sign changes, the sequence should be the following:

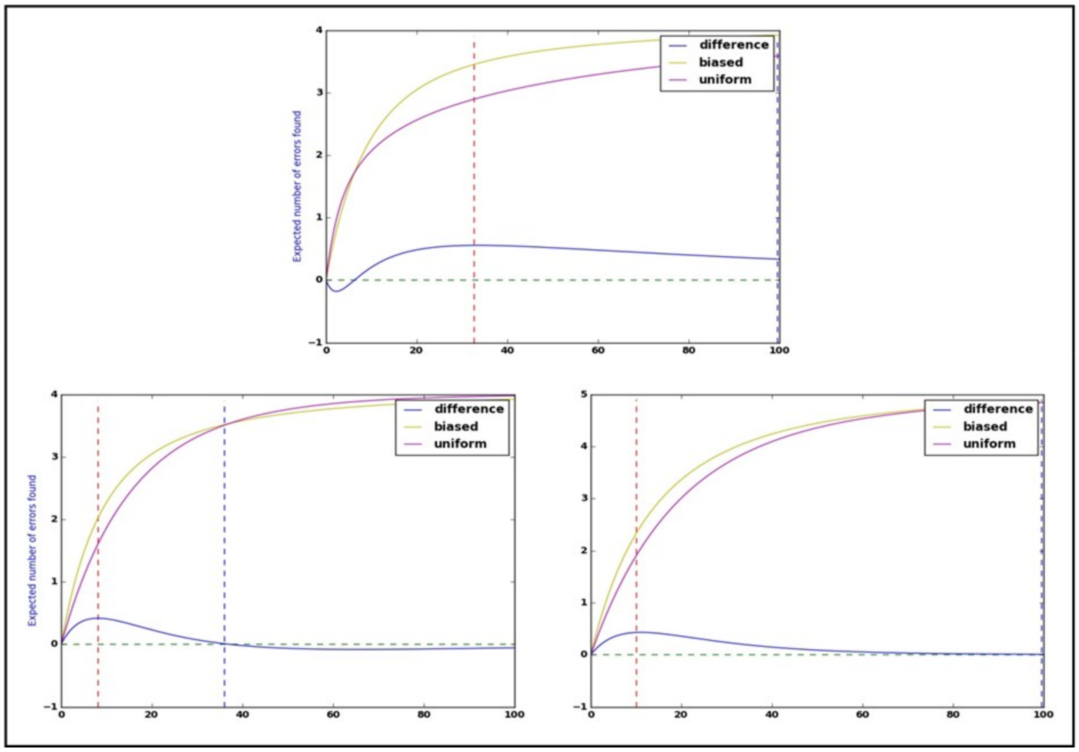


Figure 3.1: Three typical effectiveness functions from theoretical perspective

$$\begin{array}{ccccc} \{\mathcal{U}_i|\ i\in\mathcal{T}_1\} & > & \{\mathcal{B}_i|i\in\mathcal{T}_1\cup\mathcal{T}_2\} & > & \{\mathcal{U}_i|\ i\in\mathcal{T}_2\} \\ + & & - & & + \end{array} \qquad (5)$$

where " $>$ " indicates that any element in the left set is greater than any element in the right set. This ordering relation reflects a type of input distributions. The leftmost figure of Figure 3.1 shows an example of effectiveness function which satisfies the above sequence. The maximum effectiveness point is on the intersection of the effectiveness curve and the dashed red line and it monotonically converges to 0 after the maximum.

Suppose that the maximum value $\alpha_0$ is an element in $\{\mathcal{B}_i|i\in\mathcal{T}_2\}$. Then, the sequence with two algebraic sign changes is

$$\underset{-}{\{\mathcal{B}_i|\ i \in \mathcal{T}_2\}} > \underset{+}{\{\mathcal{U}_i|i \in \mathcal{T}_1 \cup \mathcal{T}_2\}} > \underset{-}{\{\mathcal{B}_i|\ i \in \mathcal{T}_1\}} \tag{6}$$

It is noted that $C_0$ is a negative number. To applying the shestopaloff's corollary, the indication function outputs the opposite number, and the output E should multiply -1 to coincide with the original output. The leftmost graph in Figure 3.1 shows an example of an effectiveness function that satisfies the above sequence. The zero-effectiveness test set size is marked by the blue dashed line. After this point, the uniform distribution outperforms the biased input distribution. The maximum effectiveness test set size is marked by the red dashed line. If a sequence contains more than two algebraic sign changes, a possible outcome can be depicted by the rightmost figure of 3.1. For this case, there is only one extreme, and it is a maximum.

Hence, for three situations in above, each effectiveness function shows a maximum effectiveness test set size. Further, we can conclude that there is a range of test set sizes that the effectiveness of biased input distribution outperforms the uniform distribution, and we call it the effective region. Formally, Let $n_m$ denotes the test set size at the maximum effectiveness an $n_s$, $n_s > n_m$ denotes the test set size at the zero or minimum effectiveness. The effective region, denoted by $\mathscr{R}$, ranges within $[n_m, n_s]$. Then the effectiveness for a coverage criterion, which measures the average number of faults found per test for each test set in the effective region is defined as follows:

$$\eta = \frac{1}{n_s - n_m + 1} \sum_{k=n_m}^{n_s} \frac{1}{k} * E_b \tag{7}$$

In the later assessments of criteria, for each system under test (SUT), we estimate $n_s, n_m$ and $E_b$, and calculate the effectiveness based on Equation 7.

### 3.2.2 Estimation of Expected Errors Found $E_b$

To estimate the expected errors found, we use 32 test sets sampled with replacement from the input distribution. Each test set runs against the mutation testing tool, named Milu [14], to retrieve mutation scores. The averaged 32 sets of mutation scores are calculated to estimate the expected errors found by an input distribution at each test set size. Mutation testing is a software testing method dedicated to evaluating the effectiveness of a test set. In mutation testing, a SUT is mutated into a set of mutants. Each mutant is a copy of the SUT injected with an artificial fault. A test input is said to kill a mutant if one of the mutant's execution results is different from the original SUT. A mutant that produces the same result as the original SUT is called an equivalent mutant. Mutation score, defined as the percentage of the killed but excepting the equivalent mutants, is an estimation of the expected errors found (i.e., fault-detecting ability) by a test set.

### 3.2.3 Estimation of the Effective Region $\mathscr{R}$

With the given sets of mutation scores at each test set size, we perform the least-square regression on the data set to create estimation functions of fault detecting ability with the test set size n for both biased input distributions and the uniform distributions. To find the effective test set region with strong confidence, we perform hypothesis testing on the data set. It is difficult to fit the data into the sum of exponential functions. Instead, we created an exponential function model presented in the following to best fit the averaged mutation scores at each test size:

$$p_l = a_2 + a_0^{a_1}(a_4 n + a_3)^{a_0 - 1} e^{-a_1(a_4 n + a_3)} \tag{8}$$

where $\{a_1, \ldots, a_4\}$ are the learning parameters, $p_l$,$n$ are the training variables where $p_l$ denote the percentage of living mutants left, which is equal to 1-ms and $n$ denotes the test set size. The reason not to directly applying mutation score is that the exponential functions are convex with $ms \geq 0$, whereas the 8 is concave. The function model is a gamma distribution without the normalization constant. $\{a_2, a_3, a_4\}$ are the parameters used to shift or scale the input and output. Hypothesis testing makes statistical inference on mutation score sets when comparing the effectiveness of the uniform and biased input distributions at each test set size. If there is no significant evidence showing either one performs better, even with the difference shown by the estimated curves, their effectiveness is treated as equal. We perform a one-tailed hypothesis testing with the Wilcoxon rank-sum test [15] on the two mutation score sets at each test set size. The confidence level is 0.05. Specifically, the hypotheses are:

- $H_0$**:** there is no significant difference between mutation scores that produced by random testing and input distributions constructed from an evaluation metric.

- $H_1$**:** The mutation scores produced by random testing is significantly different from mutation scores produced by input distributions constructed from an evaluation metric.

**Algorithm 1** Algorithm to determine $n_m,\ n_s$

```
procedure DETERMINETESTSETSIZE
    input: $f_b$ - learned function for biased distribution
           $f_u$ - learned function for uniform distribution
           $p_v$ - p-values

    output: $n_s,\ n_m$ - test set sizes

    if NumOfTestSetSize($p_v \leq 0.05$) then
        $n_m,\ n_s$ don't exist
    else
        $ts_{max} = \text{argmax}_n(f_b, f_u)$
        if $p_v(ts_{max}) \leq 0.05$ then
            $n_m = ts_{max}$
        end if
        $\{ts_{min}\} = argmin_n(f_b, f_u)$
        $n_s = \{\ t\ |t \in ts_{min},\ ts_{max} > ts_{min}\}$
    end if
    return $n_s,\ n_m$
end procedure
```

To determine the effective set size region, we present Algorithm 1. If there is no test set size such that the corresponding p-value is less than 0.05, the biased input distribution is indifferent from the uniform distribution on the fault detecting ability at any test set size that below the maximum set size. Otherwise, the learned functions $f_b$,$f_u$ are used to mathematically derive the test set size at the maximum effectiveness $ts_{max}$ and the zero-effectiveness set of test set sizes $ts_{min}$. If the p-value at $ts_{max}$ is less than 0.05, $n_m$ is determined to be $ts_{max}$. If there exists a test set size $t$ in $ts_{max}$ such that $t$ is greater than $n_m$, then $n_s$ is determined to be $t$.

## SEARCH-BASED STATISTICAL STRUCTURAL TESTING

In this section, we provide a formal representation of SBSST. In SST, a SUT is essentially treated as a control flow graph where each node represents a linear sequence of basic blocks, and each edge represents the flow of control between basic blocks [16]. In the context of structural testing, an edge is also known as a branch cover element (BCE). A BCE is said to be triggered by an input $x$ if the path in CFG executed by $x$ contains the corresponding edge. Hence, for the entire input domain space $D$, there exists a subset of the input domain space that triggers each BCE. Suppose the BCE $c_i$ is under test. $P(x)$ denotes a discrete probability distribution over the input domain space. The probability of triggering the BCE $c_i$ is the sum of the probabilities of each input in $D_{c_i}$. However, it is not possible to enumerate all inputs to derive the triggering probability. Therefore, we estimate the triggering probabilities derived from the sampled input set. Suppose the sampled set size is n, and the test size triggers $c_i$ is $n_{c_i}$. The estimated triggering probability of $c_i$ is $\frac{n_{c_i}}{n}$.

### 3.3.1 Input Distribution Model

We choose the sum of the weighted uniform distributions as the input distribution model, which is formally defined as follows:

$$P(x) = \sum_{i=1}^{k} w_i * U(x), x \in S_i \tag{9}$$

where the weight vector $w$ satisfies:

$$\sum_{i=1}^{k} w_i = 1,\ w_i \geq 0\ \forall w_i \in w \tag{10}$$

$U(x)$ is a multi-dimensional uniform distribution whose dimension equals the input domain space's dimension. The uniform distributions are applied on the consecutive, none-overlapped sub-input domain spaces, denoted as $\{S_1, \ldots, S_{k_u}\}$.

### 3.3.2 Input Distribution Construction

We view the input distribution construction shown in Figure 3.2 as a two-step process: First, we arrange sub-input domains to each uniform distribution's boundary.

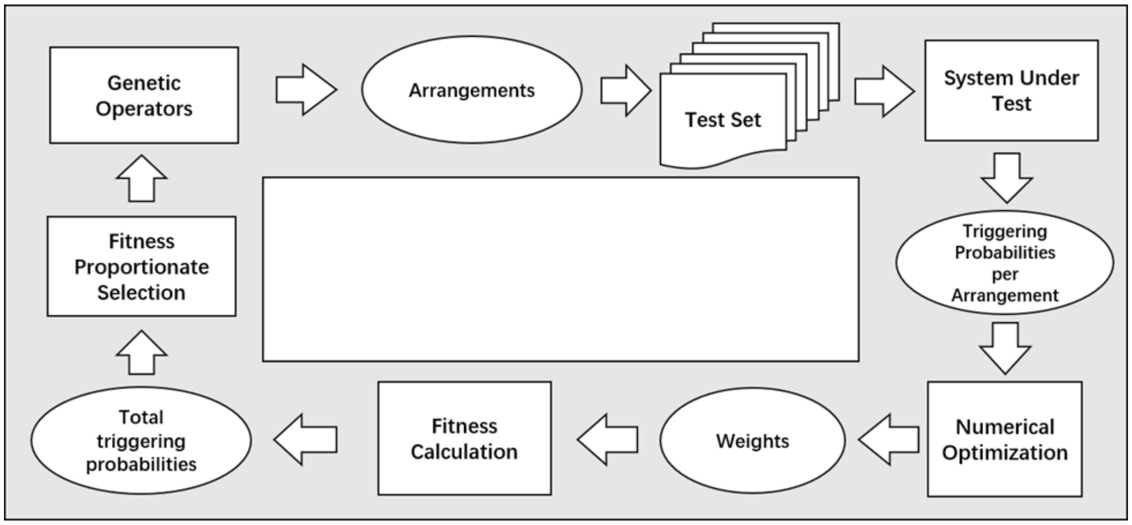


Figure 3.2: The overall workflow

Second, we assign weights to the uniform distributions. In each iteration, the genetic operators produce an arrangement to form a new set of uniform distributions. Each uniform distribution generates a sampled input set to run with SUT to estimate triggering probabilities. Then, we apply numerical optimization methods to derive the best weights from the estimated triggering probabilities to maximize the overall triggering probabilities. The purpose of adopting the Genetic Algorithm (G.A) is to search for the best arrangement. The detail of G.A is described as follows.

- **Encoding:** The chromosome is encoded as an array of integers. Each integer $\delta_i$ represents the size of an input sub-domain space. The lower boundary $l_i$ of the i-th uniform distribution equals $\sum_{k=0}^{i-1} \delta_k$. The upper boundary $u_i$ of the i-th uniform distribution equals $l_i + \delta_i$.

- **Recombination:** We adopt the two-point crossover strategy. The two-point crossover randomly selects two positions from two individuals and swaps the contents between them. The crossover rate setups to 0.9.

- **Mutation:** We adopt the uniform mutation strategy. The uniform mutation operator mutates a gene by randomly picking up an input set and assigning the index of the input set into the gene. Each gene has a probability of 0.8 to be mutated.
- **Selection:** We adopt the roulette-wheel selection strategy with elitism for reproduction. Elitism is applied to ensure the best solution in the current iteration is still available for reproduction in the next generations.
- **Fitness Evaluation:** The fitness function depends on the criterion, which is described in later section.
- **Stop criteria:** The main loop continues until one of the two stop conditions is satisfied. First, the fitness does not improve over the last 100 iterations. Second, the number of iterations reaches a pre-set maximum value.

## FITNESS CRITERIA

This section provides formal definitions of Tri-Low-Bound and p-L1-Max criteria and shows how to use numerical optimization methods to derive the weight vectors. Before start, we reformulate the triggering probabilities to matrix form. The triggering probabilities in all sub-domains can be written in a matrix form, denoted by $A$, where each column represents a sub-domain in the set S, and each row represents a BCE. The value $a_{ij}$ of the i-th row and the j-th column is the triggering probability of the BCE $c_i$ in $S_j$.

$$A = \begin{array}{c|cccc|}
 & S_1 & S_2 & \dots & S_{k_u} \\
\hline
tri_1 & a_{11} & a_{12} & \dots & a_{1k_u} \\
tri_2 & a_{21} & a_{22} & \dots & a_{2k_u} \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
tri_m & a_{m1} & a_{m2} & \dots & a_{mk_u} \\
\hline
\end{array}$$

Given a matrix $A$, the triggering probability vector $\mathbf{P_c}$ is the linear combination of the column vectors with scalar vector $w$:

$$\mathbf{P_c} = \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix} w_1 + \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix} w_2 + \cdots + \begin{bmatrix} a_{1k_u} \\ a_{2k_u} \\ \vdots \\ a_{mk_u} \end{bmatrix} w_{k_u} \tag{11}$$

where $k_u$ denotes the number of components in the input distribution model. $k_u$ is set to $m$ which is the number of branch cover points (i.e. the row counts of matrix $A$).

### 3.4.1 Tri-Low-Bound

The Tri-Low-Bound criterion for statistical structural testing originates from the definition of the statistical test set quality, which is defined as the minimum probability of triggering a cover element by a test set. Formally,

$$\text{Tri-Low-Bound} = \min\{P_{c_1}, \ldots, P_{c_m}\}$$

### 3.4.2 p-L1-Max

The proposed p-L1-Max criterion evaluates an input distribution based on the estimated sum of triggering probabilities, and the input distribution must satisfy the fairness property. According to Equation 11, the sum of triggering probabilities, denoted by $f_A(w)$ can be derived by adding up the dot product of each row vector of matrix $A$ and the weight vector:

$$f_A(w) = \sum_j^m w_j \sum_i a_{ij} \tag{12}$$

Since the weight vector is constrained by Equation 10, we can view Equation 12 as a m-Simplex. The maximum value *L1-Max* is equal to the maximum sum of column vectors of matrix $A$:

$$\text{L1-Max} = \max\left\{\sum_i a_{i1}, \ldots, \sum_i a_{im}\right\} \tag{13}$$

Hence, the weight associated with the maximum sum of column vectors equals 1.0. The weights associated with the rest columns are 0.0. This situation brings up the fairness issue, where only the bins selecting the element that is associated with the maximum weight has the ability to be sampled. Other bins have no chance to be sampled.

It is the fact that the uniform distribution is the fairest distribution, since each input has equal probability to be sampled. Therefore, we define the fairness property as the *L2-distance* from the input distribution to the uniform distribution. We use a tuning parameter $p$ to manually adjust the importance of the two objectives. In this way, the input distribution generation process can be directed to focus more on *L1-Max* or the fairness. The formal definition of criterion p-L1-Max is defined as follows:

$$p\text{-L1-Max} = \begin{cases} \ell^2 = \sum_{x \in \mathcal{D}} [P(x) - U(x)]^2 \text{ subject to:} \\ \qquad \sum_{i} P_{c_i} \geq \text{L1-Max} * p \end{cases} \tag{14}$$

In this study, we use the following values for the tuning parameter $p$:
$p \in \{0.2, 0.4, 0.6, 0.8, 1.0\}$. The optimization objective becomes to minimize the *L2-distance* of the input distribution and the uniform distribution, with constrains that the estimated sum of triggering probabilities should be at least greater than the proportion of the maximum sum of triggering probabilities.

### 3.4.3 Weights Calculation for Tri-Low-Bound

The fitness measure for *Tri-Low-Bound* is the minimum value in the triggering probability vector $\mathbf{P_c}$. Our objective is to optimize the weight vector such that the minimum value in the triggering probability vector is maximized. The problem of Maximizing the inner minimum is equivalent to the following linear programming problem where the objective is to maximize the variable $v$ with respect to weight vector $w$. Specifically, the optimization problem is defined as follows:

$$\begin{aligned} & \max v \quad \textit{subject to} \\ & v - \sum_{j=1}^{m} a_{ij} w_i \leq 0 \quad \forall i \in \{1, \ldots, m\} \\ & \sum_{j=1}^{k} w_i = 1 \\ & w_i \geq 0, \quad i \in \{w_1, ..., w_m\} \end{aligned} \tag{15}$$

This is a standard linear programming problem.We selected to use the active set method provided in ALGLIB [13] to solve the problem.

### 3.4.4 Weights Calculation for $p$-L1-Max

The fitness for criterion p-L1-Max is measured by the sum of estimated triggering probabilities. To calculate the sum of estimated triggering probabilities, the weight vector $w$ should be optimized according to the definition of p-L1-Max in Equation 14. This optimization problem uses the tuning parameter $p$ whose value can be categorized into three types:

- $p = 0$: Any feasible weight vector satisfies the constraint $\sum_i P_{c_i} \geq 0$. The *L2-distance* is 0. The derived input distribution is a uniform distribution.
- $p = 1$: The constraint $\sum_i P_{c_i} =$ L1-Max. Then, it is not necessary to minimize the L2-Distance. The fitness equals L1-Max.
- $p > 0$ & $p < 1$: see below.

The optimization problem defined in Equation 14 with $p > 0$ & $p < 1$ is a Constrained Quadratic Programming(CQP) problem which can be solved by the active set method. To form the problem as a CQP problem, we construct the quadratic matrix $Q$ and the linear vector $H$. After expanding the *L2-Distance* equation, matrix $Q$ is found to be a diagonal matrix with elements $\{q_{i,i} = 2 * |S_i|_{bin}^{-2}, \forall i \in \{1, \ldots, m\}\}$. The vector $H$ has elements $\{h_i = -2 * |S_i|_{bin}^{-1} * |S|_{bin}^{-1}, \forall i \in$

$\{1, \ldots, m\}\}$. The size of an input set $|S_i|_{bin}$ is equal to the number of bins in the input set.

The whole process to optimize the weight vector starts from checking the value of $p$. If $p$ equals 0, the process finishes and outputs the uniform distribution. If $p$ equals 1, the process ignores the fairness property and use *L1-Max* as the fitness. If $p$ is between 0 & 1, the process first forms matrix $Q$ and vector $H$. Then the process applies the active set method to derive the optimal weight vector $\mathbf{w}^*$.

## EXPERIMENTS

Table 3.1: SUT Characteristics

| SUT | NLOC | CCN | NBC | Domain Size | Param |
|---|---|---|---|---|---|
| Triangle | 27 | 22 | 28 | 1000 | 3 |
| BestMove | 91 | 28 | 42 | 262144 | 2 |
| Nichneu | 2344 | 626 | 502 | 1679616 | 8 |

Our experiment's objective is to compare the effectiveness-to-cost ratios of SSBST by adopting different criteria. Hence, we naturally divided the experiments into three sections: the effectiveness, the search run-time, and the effectiveness-to-cost ratio sections. For criterion p-L1-Max, we select p={0.2,0.4,0.6,0.8,1.0} for study. We implemented the G.A by C++ and C# under the Windows 10 environment. We use the mutation testing tool under Ubuntu 12. The hardware configuration is IntelCore i7-4770K 3.50GHz with 16GB DDR3 memory. We continue to use the benchmark programs provided by Poulding that are sufficient for our research purpose. The characteristics are shown in Tab. 1. NLOC denotes the number of lines of code. CCN denotes the Cyclomatic complexity of a SUT. NBC denotes the number of branches of a SUT. Param denotes the number of input variables. Triangle and Nichneu can be found in [17][18].

Table 3.2: RMSE Table

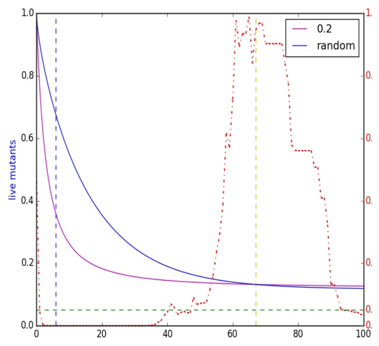
| **RMSE** | **rnd** | **0.1** | **0.2** | **0.3** | **0.4** | **0.5** |
|---|---|---|---|---|---|---|
| tri | 7.70E-03 | 7.67E-03 | 1.16E-02 | 4.27E-03 | 5.05E-03 | 4.90E-03 |
| bestMove | 1.43E-02 | 1.54E-02 | 1.88E-02 | 9.02E-03 | 1.35E-02 | 1.75E-02 |
| nichneu | 1.82E-03 | 3.40E-03 | 3.01E-03 | 3.17E-03 | 2.96E-03 | 2.52E-03 |
| **RMSE** | **0.6** | **0.7** | **0.8** | **0.9** | **1** | **lb** |
| tri | 7.36E-03 | 5.78E-03 | 4.53E-03 | 4.33E-03 | 3.37E-03 | 2.05E-03 |
| bestMove | 1.65E-02 | 1.73E-02 | 2.36E-02 | 1.50E-02 | 1.40E-02 | 8.38E-03 |
| nichneu | 5.18E-03 | 2.70E-03 | 1.95E-03 | 3.41E-03 | 1.41E-01 | 4.21E-03 |

### 3.5.1 Estimation Accuracy

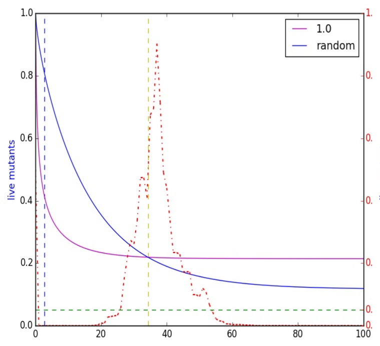
The estimation accuracy for the learned functions can be measured by the root mean square errors, which are presented in Table 3.2. In general, the smaller RMSE, the fitter the estimated model. If the RMSE value is smaller or equal to 0.3, the estimated model is acceptable [19]. In Table 3.2, the maximum value is 0.141, and the averaged value is 0.0117, which is far less than 0.3. Hence, our fitted curves can accurately predict the expected numbers of errors found as test set size varies.

### 3.5.2 Estimation Results

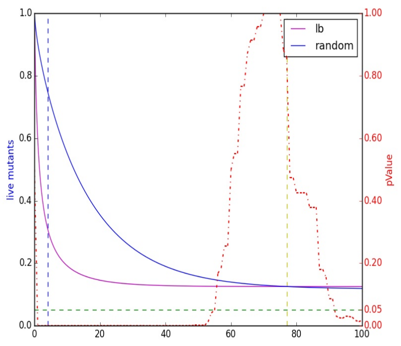
We select three coverage criteria including 0.2-L1-Max, 1.0-L1-Max, and Tri-Low-Bound for graphical representation. The detailed assessment data for all of the coverage criteria are shown in Table 3.3, Table 3.4 and Table 3.5 respectively. Figure 3.3 shows the learned function curves for the biased input distribution, marked as purple and the uniform input distribution, marked as blue and the p-values at each test set size for the three coverage criteria for all SUTs. The dashed blue line represents the test set size at $n_m$, whereas the dashed yellow line represents the test set size at $n_s$. The Triangle SUT is shown in the first row. The p-values for all coverage criteria have the pyramidal peak shape, which indicates that the mutation scores of biased and uniform distributions are significantly different on the left side and right side of the peak. The inflection points of the peaks, which p-values are at least greater than 0.9 are the test set sizes that $n_s$ is most possibly located. By solving the two learned functions mathematically, it can be seen that $n_s$ is very close to the inflection point. $n_m$ is calculated mathematically by solving the equation of the deviate of the difference of the two learned functions. It can be seen that the p-value at $n_m$ is less than 0.05.



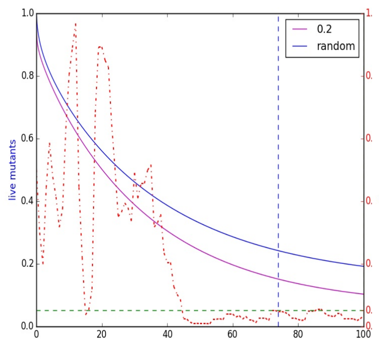
(a) 0.2-L1-Max on Tri



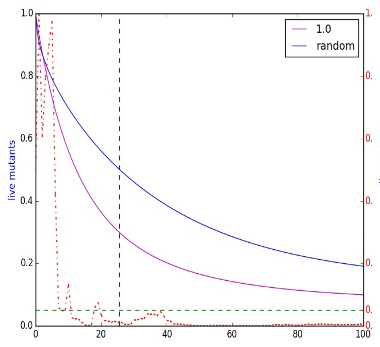
(b) 1.0-L1-Max on Tri



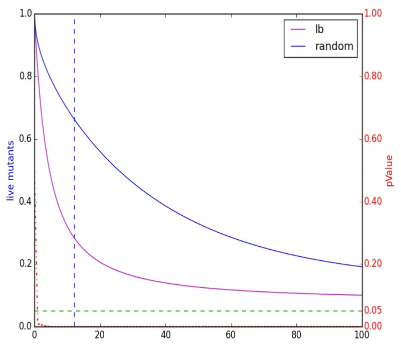
(c) Tri-Low-Bound on Tri



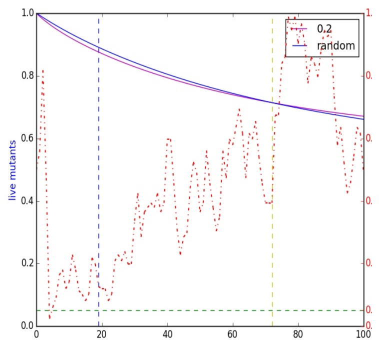
(d) 0.2-L1-Max on BestMove



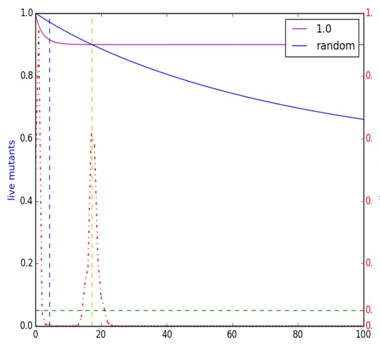
(e) 1.0-L1-Max on BestMove



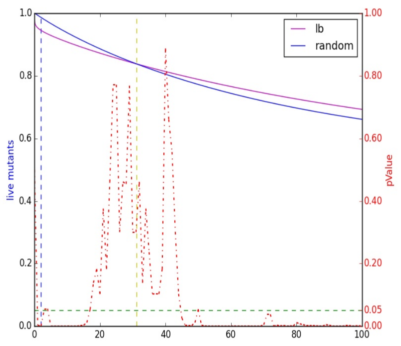
(f) Tri-Low-Bound on Best-Move



(g) 0.2-L1-Max on Nichneu



(h) 1.0-L1-Max on Nichneu



(i) Tri-Low-Bound on Nichneu

Figure 3.3: Figures for comparing Random Testing and Biased Input Distribution Testing

For BestMove, the p-values for all coverage criteria are decreasing as the test set size increases. The Tri-Low-Bound criterion shows the highest decreasing speed. Observing from the function curves, it can be seen that the difference of two functions is gradually decreasing after the test set size $n_m$. Hence, it can be inferred that after the maximum test set size, which is set to 100, there exists a pyramidal peak that is similar to the Triangle SUT. However, within the 100 test set size, $n_s$ is located at the maximum test set size. For Nichneu, the p-values behave differently for each coverage criteria. The p-values in 0.2-L1-Max are all above 0.05, which indicate that the biased input distribution is indifferent from the uniform distribution in detecting faults. The p-values in 1.0-L1-Max have a pyramidal peak shape, and $n_s$ is located near the inflection point. The p-values for Tri-Low-Bound have multiple local optima, which indicate that $n_s$ could possibility resides in a relatively large test set size interval than the unique global optima. Seen from Figure 3.3, the mathematical derived $n_s$ matches this observations.

Table 3.3, 3.4 and 3.5 provide the detailed data list for all coverage criteria. A cell in the column $p - value < 0.05$ *test size* is filled with TRUE if there exists an effective test set region. Otherwise, it is filled with FALSE. If no such region exists, there is no significant difference between a biased input distribution and the uniform distribution in the test set size ranges within $[0, n]$. For Tri and BestMove, such region exists on coverage criteria. For Nichneu, the biased input distributions begin to take effect when the $p$ value of the $p$-L1-Max coverage criteria increases to 0.8.

As described in Section 2.2, the effectiveness measures the difference of the fault detecting abilities between a biased input distribution and the uniform input distribution. The test set size $n_m$ provides the maximum effectiveness. The column *Improvement*% shows the percentage increment of mutation scores from the random testing method at the test set size $n_m$. The *Improvement* data reflects how a biased input distribution outperforms the uniform distribution at the test set size $n_m$. The column *p-value on* $n_m$ shows the p-values at the maximum effectiveness. For all of the SUTs and any coverage criteria, the p-values are less than 0.05, which provides confidences on the estimated maximum effectiveness.

The data in the *Improvement%* column shows that the Tri-Low-Bound is significantly superior to any $p$-L1-Max coverage criteria except for the Triangle SUT. In Triangle, the 1.0-L1-Max outperforms Tri-Low-Bound with 15% lead. We suspect that the estimation error on the test set size $n_m$, which are all round up to integers, is the potential cause of the problem. However, in general, one conclusion can be made on the improvement of biased input distribution over uniform at test set size $n_m$ for all coverage criteria. That is,

$$\textit{Tri-Low-Bound} \geq \textit{1.0-L1-Max} \geq \ldots \geq \textit{0.2-L1-Max}$$

The column $m_s$ *on* $n_s$ shows the mutation scores at test size $n_s$. The values in this column indicate the maximum fault detecting rate that can be achieved with the effectiveness equals 0. Therefore, the p-value at the test size $n_s$ should be greater than 0.05. For Tri and Nichneu, the p-values shown in the column *p-values on* $n_s$ are both greater than 0.4. For BestMove, as discussed previously, the maximum test set size is not large enough to show the "true" $n_s$. Therefore, the p-values for BestMove are both less than 0.05.

### 3.5.3 Search Results

The search results are shown in Table 3.6 and Table 3.7. The first five columns show the fitness statistics. The column *Fit. Of Unif. Dist* presents the fitness values for the uniform distribution. The column *Improv.* presents the increment in the percentage of the averaged fitness of a searched input distribution over the fitness of the uniform distribution. The fitness improvement is much more significant by using the Tri-Low-Bound criteria. The last four columns present the computation time statistics. It can be observed that for each SUT, the averaged computation time for the Tri-Low-Bound criteria is greater than the 1.0-L1-Max criteria.

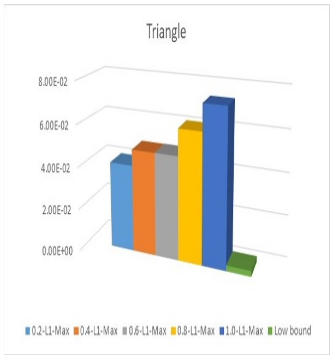
**Effectiveness**

Table 3.8: Effectiveness Table

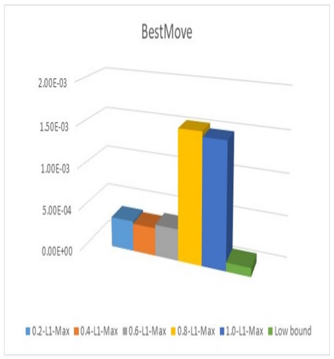
| **Effectiveness** | **Triangle** | **BestMove** | **Nichneu** |
|---|---|---|---|
| **0.2-L1-Max** | 3.20E-02 | 1.01E-02 | – |
| **0.4-L1-Max** | 3.80E-02 | 1.21E-02 | – |
| **0.6-L1-Max** | 3.81E-02 | 1.40E-02 | – |
| **0.8-L1-Max** | 4.79E-02 | 1.35E-02 | 7.41E-03 |
| **1.0-L1-Max** | 5.77E-02 | 1.50E-02 | 9.83E-03 |
| **Tri-Low-Bound** | 3.43E-02 | 2.03E-02 | 8.91E-03 |

The effectiveness measure $\eta$, which is defined in Equation 7, represents the averaged number of errors found per test in the effective test set size region. The effectiveness data is shown in Table 3.8. For Triangle and Nichneu, The 1.0-L1-Max coverage criterion outperforms the Tri-Low-Bound criteria by 68.22% and 10.32% accordingly. For BestMove, Tri-Low-Bound outperforms 1.0-L1-Max, since $n_s$ equals 100, which is not the "true" $n_s$. Based on the above observations, we conclude that 1.0-L1-Max outperforms Tri-Low-Bound in effectiveness. Another observation is that, as the cyclomatic complexity increases, the superiority of Tri-Low-Bound is decreasing.

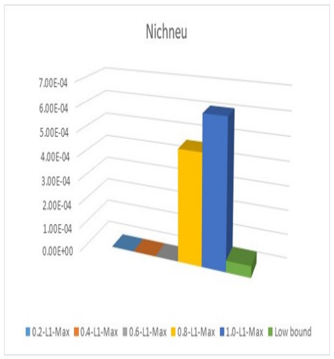
We also noticed that adding more fairness into the input distribution construction process does not benefit the fault-detecting ability. We believe this is due to the defect of using mutation testing as a tool to evaluate the fault-detecting ability. As the fairness increases, the estimated sum of triggering probabilities decreases. From the test thoroughness point of view, it is reasonable to believe the fault-detecting ability drops. However, the real faults sometimes is more difficult to discover than mutation testing tool which is automatically produced from a template. Hence, the fairness property must have its value in the real testing scenario.



(a) Effectiveness-to-Cost Ratio for Triangle



(b) Effectiveness-Cost Ratio for BestMove



(c) Effectiveness-Cost Ratio for Nichneu

Figure 3.4: The bar charts of effectiveness-to-cost ratio for investigated coverage criteria

### 3.5.4 Efficiency

The Efficiency which is expressed as the fraction of the effectiveness over the search time indicates the overall value of using a particular coverage criterion. Figure 3.4 shows the efficiency for each coverage criteria in the three SUTs. For all SUTs, the efficiency for 1.0-L1-Max is significantly greater than the Tri-Low-Bound. For $p$-L1-Max coverage criteria, the efficiency decreases as $p$ decreases. Except for Nichneu where the 0.2, 0.4, 0.6-L1-Max has no efficiency, $p$-L1-Max shows a higher efficiency value than Tri-Low-Bound. Hence, a conclusion can be made that the search algorithm by using the 1.0-L1-Max coverage criteria has the most efficiency in detecting software faults. The Tri-Low-Bound has the least efficiency, due to the large computation time it requires to search for the optimal input distribution.

## SUMMARY

The current search-based statistical structural testing has its limitations on the efficiency. The major reason is the noisy fitness estimation of triggering probabilities. This paper aims to improve the efficiency from the point view of criteria. We proposed a new criterion, called p-L1-Max, and conducted experiments to compare the efficiency of input distributions produced against the p-L1-Max and the traditional Tri-Low-Bound criterion. The experiments show that although Tri-Low-Bound criteria could give the highest effectiveness, but it also brings the significant increment on the computation time. Thus, the efficiency of using Tri-Low-Bound is much lower than others. On the other hand, 1.0-L1-Max has a bit less effectiveness than Tri-Low-Bound, however brings the highest efficiency. Hence, our conclusion is that in the practical applications, we recommend to use the 1.0-L1-Max criterion.

Table 3.3: Effective Test Set Size Region on Tri.

| **Metric** | **p-value <0.05 points** | $\mathbf{n_m}$ | **p-value on** $\mathbf{n_m}$ | **m.s. on** $\mathbf{n_m}$ | **Effectiveness on** $\mathbf{n_m}$ | **Improvement %** | $\mathbf{n_s}$ | **p-value on** $\mathbf{n_s}$ | **m.s. on** $\mathbf{n_s}$ |
|---|---|---|---|---|---|---|---|---|---|
| 0.2 | TRUE | 6 | 5.87E-06 | 6.42E-01 | 3.17E-01 | 97.64% | 67 | 9.46E-01 | 8.68E-01 |
| 0.4 | TRUE | 4 | 4.53E-06 | 6.00E-01 | 3.54E-01 | 144.26% | 59 | 9.89E-01 | 8.59E-01 |
| 0.6 | TRUE | 4 | 1.57E-06 | 6.24E-01 | 3.60E-01 | 136.89% | 60 | 4.17E-01 | 8.61E-01 |
| 0.8 | TRUE | 3 | 2.67E-06 | 5.97E-01 | 3.76E-01 | 170.07% | 46 | 5.70E-01 | 8.31E-01 |
| 1 | TRUE | 3 | 7.85E-07 | 5.85E-01 | 3.94E-01 | 206.63% | 34 | 4.09E-01 | 7.81E-01 |
| lb | TRUE | 4 | 1.92E-07 | 6.89E-01 | 4.40E-01 | 175.86% | 77 | 8.60E-01 | 8.75E-01 |

Table 3.4: Effective Test Set Size Region on BestMove.

| **Metric** | **p-value <0.05 points** | $\mathbf{n_m}$ | **p-value on** $\mathbf{n_m}$ | **m.s. on** $\mathbf{n_m}$ | **Effectiveness on** $\mathbf{n_m}$ | **Improvement %** | $\mathbf{n_s}$ | **p-value on** $\mathbf{n_s}$ | **m.s. on** $\mathbf{n_s}$ |
|---|---|---|---|---|---|---|---|---|---|
| 0.2 | TRUE | 74 | 4.99E-02 | 8.49E-01 | 9.09E-02 | 11.99% | 100 | 3.37E-02 | 8.98E-01 |
| 0.4 | TRUE | 45 | 5.65E-02 | 7.52E-01 | 1.08E-01 | 16.69% | 100 | 7.20E-02 | 8.80E-01 |
| 0.6 | TRUE | 33 | 4.38E-02 | 7.08E-01 | 1.46E-01 | 25.92% | 100 | 1.38E-02 | 8.76E-01 |
| 0.8 | TRUE | 31 | 2.22E-02 | 7.23E-01 | 1.76E-01 | 32.27% | 100 | 4.32E-03 | 8.89E-01 |
| 1 | TRUE | 26 | 1.14E-02 | 7.01E-01 | 2.02E-01 | 40.42% | 100 | 5.56E-03 | 9.00E-01 |
| lb | TRUE | 12 | 1.36E-05 | 7.15E-01 | 3.80E-01 | 113.11% | 100 | 8.25E-05 | 8.99E-01 |

Table 3.5: Effective Test Set Size Region on Nichneu.

| **Metric** | **p-value <0.05 points** | **$n_m$** | **p-value on $n_m$** | **m.s. on $n_m$** | **Effectiveness on $n_m$** | **Improvement %** | **$n_s$** | **p-value on $n_s$** | **m.s. on $n_s$** |
|---|---|---|---|---|---|---|---|---|---|
| 0.2 | FALSE | – | – | – | – | – | – | – | – |
| 0.4 | FALSE | – | – | – | – | – | – | – | – |
| 0.6 | FALSE | – | – | – | – | – | – | – | – |
| 0.8 | TRUE | 8 | 9.20E-04 | 9.40E-02 | 4.22E-02 | 81.67% | 33 | 9.25E-01 | 2.39E-01 |
| 1 | TRUE | 4 | 5.96E-03 | 8.40E-02 | 5.65E-02 | 205.71% | 17 | 6.16E-01 | 1.00E-01 |
| lb | TRUE | 2 | 2.46E-03 | 5.42E-02 | 4.13E-02 | 321.65% | 31 | 3.01E-01 | 1.62E-01 |

Table 3.6: Search Results for 1.0-L1-Max

| **1.0-L1-Max** | **Min Fitness** | **Avg. Fitness** | **Max Fitness** | **Fit. Of Unif. Dist.** | **Improv.** | **Min RunTime** | **Avg. Runtime** | **Max Runtime** |
|---|---|---|---|---|---|---|---|---|
| **Triangle** | 9.41E+00 | 1.08E+01 | 1.30E+01 | 2.54E+00 | 325.34% | 5.98E-01 | 7.83E-01 | 9.42E-01 |
| **BestMove** | 8.41E+00 | 8.63E+00 | 8.88E+00 | 7.65E+00 | 12.77% | 5.57E+00 | 6.16E+00 | 6.97E+00 |
| **Nichneu** | 2.49E+02 | 2.49E+02 | 2.49E+02 | 1.39E+02 | 78.29% | 1.48E+01 | 1.61E+01 | 1.79E+01 |

Table 3.7: Search Results for Tri-Low-Bound

| **Tri-Low-B.** | **Min Fitness** | **Avg. Fitness** | **Max Fitness** | **Fit. Of Unif. Dist.** | **Improv.** | **Min RunTime** | **Avg. Runtime** | **Max Runtime** |
|---|---|---|---|---|---|---|---|---|
| **Triangle** | 1.11E-01 | 1.21E-01 | 1.25E-01 | 6.00E-03 | 1909.71% | 5.14E+00 | 1.28E+01 | 3.23E+01 |
| **BestMove** | 8.33E-03 | 1.40E-02 | 1.88E-02 | 6.10E-05 | 22881.30% | 9.33E+01 | 1.06E+02 | 1.78E+02 |
| **Nichneu** | 1.11E-02 | 1.11E-02 | 1.11E-02 | 2.00E-04 | 5455.56% | 1.58E+02 | 1.87E+02 | 2.73E+02 |

# Chapter 4

# SOFTWARE STATISTICAL STRUCTURAL TESTING: AN ACO-BASED APPROACH

Statistical structural testing(SST) has been proved to be more effective than random testing and deterministic coverage-driven testing in fault-detecting ability. Automated search is able to automate the input distribution construction process for SST. However, due to the impact of the noisy environment, the existing search algorithm is still ineffective. In this dissertation, we present a novel statistical search algorithm called constrained ant colony optimization on the real domain ($\text{CACO}_{\mathbb{R}}$) for deriving effective input distributions. The input distribution model harnesses a set of weighted uniform distributions over the input domain space, which is enumerated effectively by the constrained ant colony optimization strategies. Experimental studies demonstrate the excellent search performance of the $\text{CACO}_{\mathbb{R}}$ algorithm and the high-grade fault-detecting ability of the input distributions produced by the $\text{CACO}_{\mathbb{R}}$ algorithm.

## OVERVIEW

Search-based software testing (SBST) refers to a testing framework that uses meta-heuristic optimization techniques to automate testing tasks. SBST is a dynamic testing process where a test object is refined during the system under test's (SUT's) run-time until it satisfies a coverage criterion. A classical implementation proposed by Xanthakis [20] applies Genetic Algorithms (G.A.) to refine a test input set. They aimed to cover all the branches with the least amount of test inputs. For each targeted branch, they conducted the G.A. to search a test input that triggers it. The SBST framework has been successfully applied to the real-world testing industry. In particular, EvoSuite[21] applies a novel hybrid approach that generates and optimizes whole test suites towards satisfying a coverage criterion for classes written in Java code.

Statistical structural testing (SST) is a traditional testing method that uses a probability distribution over the system under test's input domain space (a.k.a, input distribution) to generate test inputs. SST inspects a SUT as a control flow graph. On the graph, an edge, a node, or a path are all types of structural cover elements. An input distribution guarantees that the cover elements can be triggered at least greater than a target probability value. This requirement ensures that every cover element has a "good" chance to be triggered by a randomly selected test input from an input distribution. Compared with random testing, which selects test inputs from a uniform distribution, SST is more "balanced" in terms of SUT's structure. Thevenod-Fosse et al. in [22] provide empirical results demonstrating the superior efficacy of test sets generated using structural statistical testing, compared to test sets of the same size generated using random testing and deterministic structural testing. They constructed input distributions either from static analysis for simple programs or a manual trial-and-error process for complex programs, which creates nonnegligible labor costs in the testing process.

Search-based Statistical Structural Testing (SBSST) is able to automate the distribution construction process. Similar to SBST, SBSST is also a dynamic process. As shown in Figure 4.1, input distribution(s) are sampled to produce test input set(s) that run(s) with the instrumented SUT to form a set of program traces in terms of edges. For instance, the trace $\{1, 0, 1, 0\}$ indicates that a test input triggers edge $e_1$ and $e_3$, but does not trigger edge $e_2$ and $e_4$. The trace set is then used to estimate each edge's triggering probability, and the minimum triggering probability among all edges is used as the fitness measure. According to the fitness

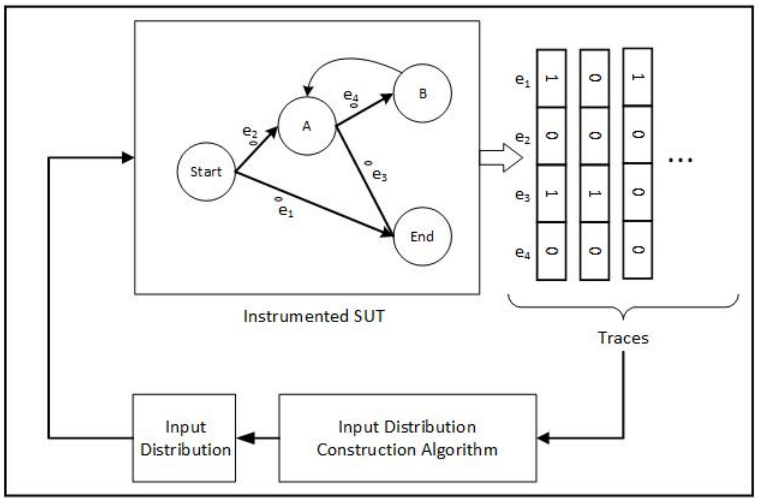


Figure 4.1: The testing environment

values, the input distribution construction algorithm produces new distributions by modifying the existing ones. This process continues until a sampled test input set satisfies a coverage criterion. Simon and Pual in [3] provide convincing empirical evidence that automated search can derive optimal input distributions in a reasonable timeframe. They modeled the input distribution as a Bayesian network and applied the traditional hill-climbing strategy as the input distribution construction algorithm.

In their experiments, the hill-climber performs well for SUT *tri* and *neichneu*. However, it performs poorly for *bestMove* whose BCEs' input sub-domain space is non-consecutive and small. The issue resides in the estimated triggering probabilities. The estimation brings noises to search, which significantly increases the computation time. The problem is especially evident when a BCE's input subdomain space is non-consecutive and small, and triggering the BCE is considered a rare event. The probability estimation of a rare event is usually inaccurate. We conducted a small experiment to show the problem: Our synthetic SUT has two inputs, with each consist of 30 elements. A cover element $C$ can be triggered by 4 non-consecutive test inputs, and the sample set used to estimate fitness has 90 test inputs. We use the hill-climber with a Tabu list to search for an input distribution that maximizes the triggering probability of $C$. The estimation uses the Wilson Score approach with continuity correction. Over the 5000 iterations, fitness swings around 0.01. The confidence band, which ranges from near 0 to an average around 0.15, could not provide useful information to guide the search direction.

This issue reminds us that the search algorithm for the SBSST problem should be effective in an uncertain environment. Compared with various EA algorithms, Ant Colony Optimization (ACO) can deal with arbitrarily large noise in a graceful manner [23]. Due to the pheromone accumulation effect, the estimated fitness for an input distribution is aggregated, and the noise is reduced in the long run. In this article, we present to use an extension of the ACO algorithm for SBSST. We model the input distribution as a sum of weighted one-dimensional uniform distributions. Each uniform distribution occupies a non-overlapped, consecutive input sub-domain space (a.k.a a bin). In the view of ACO, constructing an optimized input distribution is a resource allocation process. There are two resources: the size of an input domain space and the bins' probabilities. Assigning the right amount of input sub-domain size and the associated probability to each bin to maximize the triggering probabilities is the optimization objective. The size of an input domain space can sometimes be enormous. For instance, an integer's size is $2^{32}$. By using a discrete optimization strategy, enumerating the whole input domain space is not possible. Hence, we adopt Socha's $\text{ACO}_{\mathbb{R}}$ algorithm, which expands the discrete ACO to the continuous domain [6]. Since the resource amount always has an upper boundary and lower boundary, we modify Socha's algorithm to satisfy constraints and name it the $\text{CACO}_{\mathbb{R}}$ algorithm.

We performed experimental studies to demonstrate the search efficiency of the $\text{CACO}_{\mathbb{R}}$ algorithm. The comparison takes place between $\text{CACO}_{\mathbb{R}}$ and the hill-climbing algorithm, which to the best knowledge, is the only algorithm used in

SBSST. The experimental study also shows the fault-detecting ability of the optimized input distributions by two assessment methods. The first method is mutation testing, which uses a template to produce faulty source code versions. The fault-detecting ability is measured by the percentage of mutants killed by an input distribution's sampled tests. The second method focuses on the fault-detecting ability of real software bugs. We use the Siemens test suite as a benchmark [24]. Each SUT in the suite contains many copies of faulty versions with mistakes made by programmers.

In this chapter, we start with a detailed description of the proposed algorithm, followed by presents the experimental results to demonstrate the effectiveness of the algorithm.

## THE CACO$_{\mathbb{R}}$ APPROACH

We model the input distribution construction as a resource allocation process. The input sub-domain space and the weights applied to each uniform distribution component are represented as two types of resources. The ACO algorithm aims to allocate proper resources to each component in order to minimize a fitness function. Specifically, the two types of resources are:

- **Resource One: Bin's Interval**.The total amount of available resources for allocating the intervals of $k$ bins is the size of the input domain space $H - L + 1$. Let $\{\Delta_1, \Delta_2, \ldots, \Delta_k\}$ denotes the input sub-domains (resources) allocated to each bin separately. The boundary value $\delta_i$ where $1 \leq i \leq k-1$ equals

$$\delta_i = \sum_{k=1}^{i} \Delta_k + L \tag{4.1}$$

- **Resource Two: Weights** The total amount of available resources for allocating weight values is 1. For each weight component, the resource is the weight value itself which is denoted as $w_i$.

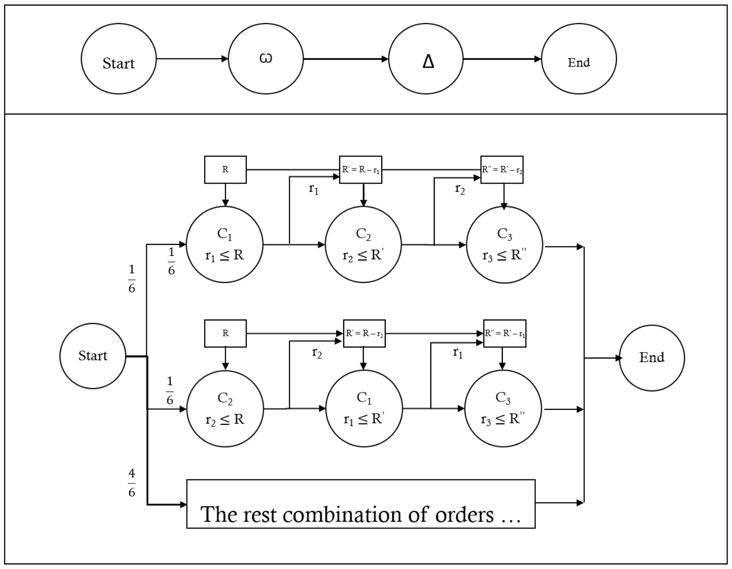


Figure 4.2: Solution Construction Mechanism

### 4.2.1 The Ant Dynamics

Figure 4.2 presents the any dynamics. There are two types of vertices, the weight $\omega$ and the bin's interval $\Delta$. An ant could select either one to proceed at first. For each resource type, an ant begins with randomly selecting a permutation order of vertices (for instance, $\{C_1, C_2, C_3\}$). Next, it moves to each vertice in the selected order and randomly assigns a resource value with pheromone-proportionate selection. The resource value is at most equal to the remaining resources ($R$). Pheromone amount represents the degree of preference that an ant would choose a resource amount for a vertice.

The above workflow could be implemented by the traditional discrete ACO algorithm. However, there are several issues. First, in the above model, the resource weights are quantized into countable intervals. An accurate solution requires the weight variable to be a continuous variable. Second, as the number of components increases, it becomes infeasible to enumerate all of the permutation orders. Third, the available resource amounts at each component subject to a constraint. These three issues lead us to continuous ACO, which generalizes the problem domain space from discrete to continuous.

### 4.2.2 CACO$_{\mathbb{R}}$ Implementation

There are several implementations of the ACO algorithm generalized to continuous domains [25; 26]. In this study, we adopt Socha's ACO$_{\mathbb{R}}$ algorithm [6]. The ACO$_{\mathbb{R}}$ algorithm belongs to a category of *estimation of density* algorithms which uses a probability density function($PDF$) to produce new populations.

#### Population and *PDF*

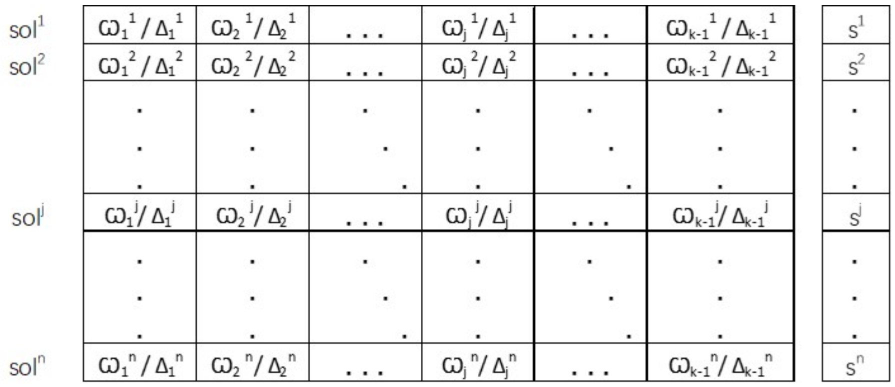
Figure 4.3 illustrates a solution pool with $n$ solutions for resource allocations that satisfy the constraints on the $w$ set and the $\Delta$ set. The constraints are formally defined as follows:

$$\begin{cases} \sum_{i=1}^{k} \Delta_i = H - L + 1;\ \Delta_i \geq 0\ \ \forall i \in \{1, \dots, k\} \\ \\ \sum_{i=1}^{k} w_i = 1;\ w_i \geq 0,\ \forall i \in \{1, \dots, k\} \end{cases} \tag{4.2}$$

The i-th column, represented as the i-th bin stores the valid values for weight $w_i$ and the input sub-domain size $\Delta_i$. The right-most column stores the scores for each solution in the pool. To simulate the two pheromone updating functions, Socha uses a probability model called the *Gaussian kernel PDF* on each column of the population pool. The probability model applied on the i-th column is defined as

$$G^i(r) = \sum_{j=1}^{n} s_j * \mathcal{N}(r_j^i, \sigma_j^i) \tag{4.3}$$

where $G^i(r)$ refers to the probability of the resource value $r$ for the i-th bin. In this context, resource refers to both the weights and the input domain size. $\mathcal{N}(r_j^i, \sigma_j^i)$ denotes a normal distribution centered at the resource value $r_j^i$ with a variance of



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $sol^1$ | $\omega_1^1/\Delta_1^1$ | $\omega_2^1/\Delta_2^1$ | ... | $\omega_j^1/\Delta_j^1$ | ... | $\omega_{k-1}^1/\Delta_{k-1}^1$ | $s^1$ |
| $sol^2$ | $\omega_1^2/\Delta_1^2$ | $\omega_2^2/\Delta_2^2$ | ... | $\omega_j^2/\Delta_j^2$ | ... | $\omega_{k-1}^2/\Delta_{k-1}^2$ | $s^2$ |
| | ⋮ | ⋮ | ⋱ | ⋮ | ⋱ | ⋮ | ⋮ |
| $sol^j$ | $\omega_1^j/\Delta_1^j$ | $\omega_2^j/\Delta_2^j$ | ... | $\omega_j^j/\Delta_j^j$ | ... | $\omega_{k-1}^j/\Delta_{k-1}^j$ | $s^j$ |
| | ⋮ | ⋮ | ⋱ | ⋮ | ⋱ | ⋮ | ⋮ |
| $sol^n$ | $\omega_1^n/\Delta_1^n$ | $\omega_2^n/\Delta_2^n$ | ... | $\omega_j^n/\Delta_j^n$ | ... | $\omega_{k-1}^n/\Delta_{k-1}^n$ | $s^n$ |

Figure 4.3: A representation of solution pool

$\sigma_j^i$. The score $s_j$ is determined by the equation

$$s_j = \frac{1}{qn\sqrt{2\pi}} e^{-\frac{(l-1)^2}{2q^2n^2}} \tag{4.4}$$

where $l$ denotes the ranking of the l-th best solution in the population and, $q$ is a preference parameter. A small value of $q$ leads to the selection of Gaussian components that are more biased towards the current best solution in the population, resulting in a fast convergence speed but the possibility of being trapped in a local optimum and vice versa. $\sigma_j^i$ denotes the averaged distance between the j-th solution and the other solutions in the i-th column. $\sigma_j^i$ is defined

$$\sigma_j^i = \rho \sum_{k=1}^{n} \frac{\sqrt{(r_k^i - r_j^i)^2}}{n-1} \tag{4.5}$$

where $\rho > 0$ is also a preference parameter. The higher the value of $\rho$, the lower the convergence speed, but the less possibility of being best trapped in the local optima.

### Solution Generation

To produce a new solution, three steps are performed. First, as in the discrete case, a permutation order of *Gaussian kernel PDFs* is randomly selected.

Second, for each selected $G^i$ in order, a Gaussian component is randomly selected with *score proportionate selection.* Specifically, the probability of selecting the i-th solution is

$$P_i = \frac{s_i}{\sum_{i=1}^{n} s_i}$$

Third, since the available resources at $G^i$ are

$$r_a = r_t - \sum_{k=1}^{i-1} r^k \tag{4.6}$$

where $r^k$ denotes the taken resource amount on the k-th order of the *Gaussian kernel PDF* sequence, sampling a new resource value at $G^i$ must satisfy the constraint that $r^i \leq r_a$. That is, the selected Gaussian component is truncated to the $[0, r_a]$ domain space. To sample a truncated Gaussian component, we adopt the accept-reject sampling (ARS) method [27].

The concept behind the ARS method is to use an alternative distribution $g(r)$ to sample a resource value $r_s$. This value is accepted as a sampled value from a truncated Gaussian component if

$$u \leq \frac{f(r_s)}{M * g(r_s)}$$

where $f(r_s)$ denotes the *PDF* of the selected Gaussain component, $u$ is a random variable following the $U(0, 1)$ distribution, and $M$ is a constant derived from

$$M = \sup_{r} \left( \frac{f(r)}{g(r)} \right) \tag{4.7}$$

If $f(r)$ is $z \sim \mathcal{N}(0, 1)$, a standard Gaussian *pdf*, a common pair for $g(r)$ is a shifted standard exponential distribution. The exponential distribution provides a lower bound constraint on $x$. However, the available resource $r_a$ is a higher-bound constraint. To coincide with the exponential distribution property, $r_a$ is mirrored from higher-bound to lower-bound by using Gaussian distribution's symmetrical property. After the sampling process, the actual sampled resource amount is derived from the result of the ARS process followed by another mirroring of the result. After the first mirroring process, the lower-bound constraint is $r^i \geq 2r^i_j - r_a$ where

$r_j^i$ represents the mean of the j-th Gaussian component in the i-th *Gaussian kernel PDF*. Then the truncated Gaussian distribution can be written as

$$r^i \sim \mathcal{N}(r_j^i, \sigma_j^i)\mathcal{I}(r^i \geq 2r_j^i - r_a) \tag{4.8}$$

$\mathcal{I}$ is an indication function that outputs 1 if $r^i$ satisfies the condition and otherwise outputs 0. It is also known that $\mathcal{N}(r_j^i, \sigma_j^i) = \sigma_j^i * z + r_j^i$ where $z \sim \mathcal{N}(0,1)$. The standardized version of the truncated Gaussian distribution is therefore

$$z_t \sim \mathcal{N}(0,1)\mathcal{I}(z \geq \frac{r_j^i - r_a}{\sigma_j^i}) \tag{4.9}$$

Thus, $f(r)$ is the standard normal distribution which *PDF* is $f(r) = e^{-\frac{1}{2}r^2}$. Since the lower-bound constraint is $\frac{r_j^i - r_a}{\sigma_j^i}$, the *PDF* of the exponential distribution $g(r)$ is setup to be

$$g(r) = \begin{cases} e^{-\frac{1}{2}(r - \frac{r_j^i - r_a}{\sigma_j^i})} & r \geq 0 \\ \\ 0 & r < 0 \end{cases} \tag{4.10}$$

Next, in order to find the optimal value $M$, we expand the equation $\frac{f(r)}{g(r)}$ and solve the following equation:

$$\frac{\mathrm{d}}{\mathrm{dr}}\left[e^{0.5r^2 + r - \frac{r_j^i - r_a}{\sigma_j^i}}\right] = 0 \tag{4.11}$$

Finally, $M$ is derived as

$$M = e^{\frac{1}{2} - \frac{r_j^i - r_a}{\sigma_j^i}} \tag{4.12}$$

**Algorithm 2** Procedure of Sampling Resources and the CACO$_{\mathbb{R}}$ algorithm

```
procedure SampleResAmount
    input: $l$ - a column index in the pool
            $r_a$ - available resources
            $S$ - solution pool

    output: $r^*$ - sampled resource amount

    $r_i^l = ScorePropotionateSelection(S^l)$
    $r_c = 2r_i^l - r_a$
    $\sigma_j^l = \rho \sum_{k=1}^{n} \frac{\sqrt{(r_k^l - r_j^l)^2}}{n-1}$
    $r_z = \frac{r_c - r_i^l}{\sigma_j^l}$
    $\eta(r) = \frac{f(r)}{Mg(r)} = e^{\frac{1}{2}r^2 + r - \frac{1}{2}}$

    $r^* = -1$
    while $true$ do
        $r_s = y + r_z,\ y \sim Exp(1.0)$
        if $r_s > 2r_i^l$ then
            continue
        end if
        $u \sim \mathcal{U}(0,1)$
        if $u \le \eta(r_s)$ then
            $r^* = r_s$
            break
        end if
    end while

    $r^* = 2r_i^l - r^*$
    return $r^*$
end procedure
procedure CACO
    input: $k$ - number of bins
            $c_h$ - fitness history length
            $c_s$ - stop condition constant

    output: $S^*$ - The 1st rank solution in the pool

    $H_f \leftarrow \emptyset$
    $S \leftarrow PopulationInitialization()$
    $FitnessEvaluation(S)$
    $SortAndUpdateScore(S)$

    while $g \le g_{max}$ do
        $g = g + 1$
        $S_t \leftarrow \emptyset$
        for $i = 1$ to $n_s$ do
            $L = PermOrder(k)$
            for $r$ in $\{\mathbf{w}, \Delta\}$ do
                $r_a = r_t$
                for $l$ in L do
                    $r_l^i = Sample(l, r_a, S)$
                    $r_a = r_a - r_l^i$
                    $S_t^i \leftarrow r_l^i$
                end for
            end for
        end for

        $S = S_t$
        $FitnessEvaluation(S)$
        $SortAndUpdateScore(S, I)$
        $S^* = S_{best}$
        $H_f \leftarrow f^*$
        $div = CalStdDiv(H_f, c_h)$
        if $div \le c_s$ then
            break
        end if
    end while
    return $S^*$
end procedure
```

Algorithm 4 illustrates the process of sampling an amount of resources for a selected component (column) in the pool. The process starts by calling the *ScorePropotionateSelection* function to randomly selects a mean value $r_i^l$ for the Gaussian component. Next, the higher-bound constraint $r_a$ is converted to the lower-bound $r_c$. The variance of the Gaussian component is then computed with Equation 10 and the constraint $r_c$ standardized to $r_z$. Next, the process enters a loop that terminates when an acceptable condition is satisfied. In the loop, a random value $r_s$ is first sampled from the $Exp(1.0)$ distribution and then $r_z$ is calculated. In order to prevent sampling of the negative resource value, $r_s$ must be smaller than $2r_i^l$. Finally, a random value $u$ is sampled from $\mathcal{U}(0,1)$. If $u \leq \eta(r_s)$, the sampled result $r_s$ is accepted. Once accepted, $r_s$ is converted to the real resource value $r^*$.

#### Fitness Evaluation

The fitness of an input distribution is represented by the distances to the optimal triggering probability $\frac{1}{|CC|}$. Formally,

$$\begin{aligned} f(\hat{\mathbf{tri}}) &= \textstyle\sum_{i=1}^{m} s(\hat{\mathbf{tri}}_i) \\ where \;\; s(x) &= \begin{cases} \frac{1}{|CC|} - x, & \frac{1}{|CC|} - x > 0 \\ \\ 0, & otherwise \end{cases} \end{aligned} \tag{4.13}$$

The fitness value equals zero when input distribution is optimal. To calculate the estimated triggering probabilities($\hat{\mathbf{tri}}_i$), we first estimate the triggering probabilities for each bin. Suppose $\hat{\mathbf{tri}}_{b_i}$ denotes the estimated triggering probabilities of the i-th bin, the overall estimated triggering probabilities are derived by the following formula:

$$\hat{\mathbf{tri}} = \sum_{i=1}^{k} w_i * \hat{\mathbf{tri}}_{b_i}$$

### 4.2.3 The Overall Process

The complete CACO$_{\mathbb{R}}$ algorithm is detailed in Algorithm 4. To begin with, the history fitness list $H_f$ is initialized to empty and the *PopulationInitialization* function randomly produces a pool of initial solutions. Then, the *FitnessEvaluation* function calculates the fitness of each solution in the pool and the *SortAndUpdateScore* function then ranks the population according to the fitness values. The algorithm then enters the main loop. In the main loop, the algorithm begins by producing a pool of new solutions and storing them in $S_t$. More specifically, in order to produce a new solution, the algorithm first randomly generates a permutation order of the column indices of the pool. Then, for each resource type ($\omega$ or $\Delta$), the *SampleResAmount* function takes the bins index $l$, the current available resource amount $r_a$ and the solution pool $S$ as inputs and outputs a sampled resource value. The returned value $r_l^i$ is assigned to the l-th component of the new solution $S_t^i$ and the available resource amount is reduced by the value of $r_l^i$. After producing a set of new solutions, the algorithm replaces the existing pool $S$ by $S_t$ and enters the fitness evaluation phase. Next, the score of each solution in the pool is updated once more with the *SortAndUpdateScore* function, alongside the history fitness list $H_f$.

The main loop has two termination conditions. The first condition is met when the algorithm reaches the maximum generation limit $g_{max}$. The second condition is met when the standard deviation ($div$) of the fitness stored in $H_f$ is less than $c_s$.

## EXPERIMENTS

The primary objective of our experimental study is to demonstrate the efficiency of the proposed search algorithm. To my best knowledge, hill-climbing algorithm is the only search algorithm being used [3]. Hence, we implemented the hill-climbing algorithm for comparison. The secondary objective is to show the fault-revealing ability of the input distributions produced by the search algorithm. We used a benchmark program called Siemens testing suite to show the ability to discover real faults. We also performed mutation testing to show the fault discover rate. All of the experiments run on Linux with Core i7 with 3.5 GHz, 8 logical processors and 16 GB memory.

### 4.3.1 Benchmark Programs

The detail information of the benchmark programs are listed in Table 4.1, in which benchmark programs are divided into two groups. The first group shown above the *triangle* function are collected from Siemens test suite. The Siemens test suite is dedicated to studying the fault detection capabilities of control-flow and data-flow coverage criteria[24]. In the suite, for each program under test, there are multiple faulty versions of the source code. Each faulty version contains an error. The second groups after *non_crossing_biased_decend* uses mutation testing. The program domain sizes range from 1000 to 26200. Specifically, *bestMove* determines the best move for the current player in a tic-tac-toe (noughts-and-crosses) game. *bestMove* has the largest domains size and complexity among all SUTs. Moreover, it has a small sub-domain space of only 4 non-consecutive inputs that exercises a branch cover element, which makes the search more difficult. *triangle* receives three integer numbers and decides the kind of triangle they represent: equilateral, isosceles, scalene, or no triangle. *gcd* computes the greatest common denominator of the two integer arguments. *calday* computes the day of the week given a date as three integer arguments. *bessel* computes the Bessel functions given an order n and real argument. These programs[1] are presented in [28]. *lednum*[2], which is originated from the WCET project [29] reads ten values as input and output half to LCD. In the table, CC refers to the cyclomatic complexity, LOC represents the line of code in the SUT.

### 4.3.2 Experiment One: Search Efficiency

To assess the efficiency and the effectiveness of the proposed solution, we run $\text{CACO}_{\mathbb{R}}$ and hill climbing algorithm(H.C) for 32 times separately over 27 benchmark program functions. The hill-climbing algorithm has two types of operations

[1]http://tracer.lcc.uma.es/problems/testing/index.html
[2]http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

to create a neighbor solution. The first operation alters a bin's size by shifting up/down the upper bound of the bin. The second operation alters the weight value assigned to the associated bin. In each iteration, a new input distribution is produced by first randomly selecting a bin, followed by randomly performing one of the two operations.

The search performance of CACO$_{\mathbb{R}}$ and Hill-climbing algorithm are assessed by the following evaluation metrics:

1. The average achieved fitness values(AvgFit) by a search algorithm
2. The average run time of a search algorithm. (C.T)

To measure the achieved fitness value, we let the two search algorithms run until either the target fitness value is reached or the fitness values in the last ten iterations do not change within a standard deviation of 0.01. If search algorithm A finishes before search algorithm B, we rerun A until B finishes and use the best fitness value of A in the previous runs. We repeat this process for 32 times and averaging the fitness and computation time as the experimental results. Since the fitness represented by the triggering probability is estimated from samples, it is normal to see noises from estimation. Hence, we broaden the stopping criterion from the optimal to the near-optimal input distribution with a range of fitness values $f = [0, 0.05]$.

The search results are shown in Table 4.2. Specifically, H.C and CACO$_{\mathbb{R}}$ cannot reach the target fitness value when testing *bessel*, *bestMove*, *calday* and *esc*(function names appended with #). However, CACO$_{\mathbb{R}}$'s fitness value is closer to the target than H.C, with sacrifices of C.T to converge to a better local optimum. The four SUTs have a common trait: Both of them carry BCEs with a small non-consecutive input sub-domain space, which leads to a rugged fitness landscape, causing search inefficiency. For this type of SUTs, we should consider the trade-off between C.T and fitness value. It might not be worth spending extra C.T to gain a small fitness improvement. For functions with names appended with *, CACO$_{\mathbb{R}}$ reached the target fitness value, whereas H.C did not. Although the computation time of H.C is shorter than CACO$_{\mathbb{R}}$ in some functions, CACO$_{\mathbb{R}}$ outperformed H.C, since H.C converged into a local optimum, whereas CACO$_{\mathbb{R}}$ converged into a global optimum. In the rest functions(not marked with # and *), both CACO$_{\mathbb{R}}$ and H.C reached the target fitness value. CACO$_{\mathbb{R}}$'s computation time is less than H.C except *find_nth* and *get_token*, where there is not much difference in C.T between the two search algorithms. CACO$_{\mathbb{R}}$ outperformed H.C.

Above all, the CACO$_{\mathbb{R}}$ algorithm produced the near-optimal input distributions for 72.41% of benchmark functions whereas Hill-climbing algorithm produced 48.28%. These observations provide the evidence that the CACO$_{\mathbb{R}}$ algorithm is more effective than the hill-climbing algorithm. To conclude with much higher confidence, we performed the rank-sum test on the fitness values for the 32 runs. The hypotheses are defined as follows:

- $H_0$: There is no significant difference on the fitness value of the best input distributions produced from the CACO$_{\mathbb{R}}$ or the hill climbing algorithm
- $H_a$: The fitness value of the best input distribution produced from the CACO$_{\mathbb{R}}$ algorithm is significantly smaller than the one produced from the Hill-climbing algorithm.

Except for the SUTs with optimal input distributions, the p-values (see Column $CACO_{\mathbb{R}} \xrightarrow{-} HC$ in Table 4.2 ) are far less than 0.05. The null hypothesis is rejected.

To see the difference of the fault-detecting ability between the biased input distribution and the uniform input distribution, it is also necessary to list the avgFit value for the uniform distribution and calculate the percentage of improvement on avgFit by using the CACO$_{\mathbb{R}}$ algorithm (see Table 4.2). The minimum improvement is 30.011% on *get_token* function in the program *printtokens*. The maximum improvement is 100% on 13 programs which avgFit values are 0. Furthermore, no benchmark program shows the negative improvements.

We also noticed that the $\text{CACO}_{\mathbb{R}}$ algorithm performs less effectiveness for the mutation testing group than the Siemens group. The major reason resides in the complexity of the functions. Except *gcd*, the rest functions have relative higher cyclomatic complexities and the larger amount of BCEs than functions in the Siemens suite. One of the potential solutions is to partition the BCEs into a set of groups. For each group, the $\text{CACO}_{\mathbb{R}}$ algorithm produces an input distribution.

Table 4.1: Benchmark Programs

| Function | Faulty Versions | BCE | CC | LOC | Program | Domain Size |
|---|---|---|---|---|---|---|
| get_token | v1,v2,v3,v4 | $\{8,\ldots,33\}$ | 17 | 50 | printtokens2 | 4.06E+03 |
| is_str_constant | v5, v10 | $\{94,\ldots,99\}$ | 4 | 16 | | |
| is_num_constant | v6 | $\{88,\ldots,93\}$ | 4 | 18 | | |
| is_token_end | v7,v8,v9 | $\{34,\ldots,47\}$ | 10 | 23 | | |
| get_token | v1,v2,v3,v5 | $\{8,\ldots,25\}$ | 23 | 77 | printtokens | 4.07E+03 |
| numeric_case | v7 | $\{13,\ldots,31\}$ | 4 | 27 | | |
| dodash | v1,v2,v5,v9,v10,v11,v32 | $\{10,\ldots,19\}$ | 11 | 35 | replace | 4.68E+03 |
| subline | v3,v4,v6,v13 | $\{52,\ldots,55\}$ | 6 | 22 | | |
| stclose | v8,v16 | $\{11,12\}$ | 2 | 16 | | |
| makepat | 15,28,29,30 | $\{24,\ldots,43\}$ | 17 | 56 | | |
| omatch | v14,v18,v24,v25,v26,v27,v31 | $\{56,\ldots,73\}$ | 17 | 58 | | |
| addstr | v17 | $\{0,1\}$ | 2 | 16 | | |
| getccl | v22 | $\{20,21\}$ | 2 | 20 | | |
| find_nth | v1,v6 | $\{4,\ldots,7\}$ | 4 | 13 | schedule | 2.42E+03 |
| unblock_process | v2 | $\{24,\ldots,27\}$ | 3 | 19 | | |
| upgrade_process_prio | v3,v4,v5,v7,v8 | $\{18,\ldots,23\}$ | 4 | 24 | | |
| InfoTbl | v1, v7, v13 | $\{12,\ldots,41\}$ | 17 | 85 | totInfo | 2.79E+03 |
| gser | v8, v15 | $\{2,\ldots,5\}$ | 4 | 15 | | |
| alt_sep_test | v3, v5, v9, v12, v13, v14 | $\{6,\ldots,13\}$ | 14 | 24 | tcas | 1.58E+03 |
| non_crossing_biased_climb | v1, v2, v4, v20 | $\{2,3\}$ | 6 | 16 | | |
| non_crossing_biased_decend | v6,v7,v8,v11,v15,v16,v17,v18,v19 | $\{4,5\}$ | 6 | 16 | | |
| triangle | 5001 mutants | $\{0,\ldots,27\}$ | 22 | 53 | triangle | 9.26E+03 |
| gcd | 351 mutants | $\{0,\ldots,5\}$ | 6 | 38 | gcd | 4.04E+04 |
| calday | 680 mutants | $\{0,\ldots,21\}$ | 12 | 72 | calday | 8.12E+06 |
| besselj | 3041 mutants | $\{0,\ldots,27\}$ | 17 | 245 | bessel | 4.04E+04 |
| lcdnum | 3001 mutants | $\{0,\ldots,31\}$ | 17 | 64 | lednum | 1.00E+03 |
| bestMove | 5001 mutants | $\{0,\ldots,41\}$ | 29 | 132 | bestMove | 2.62E+05 |

Table 4.2: Search Performance Part One

| Function | C.T(H.C) | C.T(CACO$_{\mathbb{R}}$) | AvgFit | | | P-value | |
|---|---|---|---|---|---|---|---|
| | | | CACO$_{\mathbb{R}}$ | HC | Uniform | CACO$_{\mathbb{R}}$->HC | CACO$_{\mathbb{R}}$->Uniform |
| bessel# | 831.60 | 1936.418 | 0.125 | 0.278 | 0.351 | 1.84486E-11 | 1.50993E-11 |
| bestMove# | 2393.46 | 2986.317 | 0.080 | 0.557 | 0.655 | 1.4323E-11 | 1.4323E-11 |
| calday# | 2407.38 | 2218.997 | 0.107 | 0.162 | 0.176 | 1.28409E-10 | 1.2603E-11 |
| gcd* | 406.32 | 278.954 | 0.011 | 0.143 | 0.153 | 8.08947E-12 | 8.08947E-12 |
| lcd_num* | 288.48 | 497.237 | 0.047 | 0.816 | 0.872 | 1.50993E-11 | 1.50993E-11 |
| triangle* | 741.78 | 596.644 | 0.050 | 0.421 | 0.582 | 3.23745E-12 | 2.60952E-12 |
| addstr | 2.76 | 2.191 | 0.000 | 0.000 | 0.001 | 9.99000E-01 | 7.27583E-05 |
| dodash | 3.952 | 1.413 | 0.000 | 0.017 | 0.051 | 3.18727E-08 | 1.18284E-12 |
| esc# | 280.5 | 495.755 | 0.102 | 0.208 | 0.283 | 1.43824E-07 | 1.40016E-11 |
| getccl* | 3.932 | 20.485 | 0.000 | 0.116 | 0.365 | 2.39420E-08 | 6.05890E-13 |
| makepat | 69.78 | 2.032 | 0.000 | 0.001 | 0.014 | 3.30837E-04 | 6.05890E-13 |
| omatch | 3.243 | 2.157 | 0.000 | 0.010 | 0.067 | 6.35576E-10 | 2.60952E-12 |
| stclose* | 353.28 | 45.148 | 0.000 | 0.374 | 0.496 | 4.56430E-11 | 2.05549E-12 |

Table 4.3: Search Performance Part Two

| Function | C.T(H.C) | C.T(CACO$_{\mathbb{R}}$) | AvgFit | | | P-value | |
|---|---|---|---|---|---|---|---|
| | | | CACO$_{\mathbb{R}}$ | HC | Uniform | CACO$_{\mathbb{R}}$->HC | CACO$_{\mathbb{R}}$->Uniform |
| subline | 3.54 | 1.599 | 0.000 | 0.000 | 0.001 | 9.99000E-01 | 1.07886E-02 |
| alt_sep_test | 56.04 | 2.265 | 0.000 | 0.003 | 0.036 | 3.30837E-04 | 6.05890E-13 |
| ncbc | 4.86 | 2.238 | 0.000 | 0.000 | 0.024 | 4.07615E-02 | 6.05890E-13 |
| ncbd | 31.08 | 2.435 | 0.000 | 0.001 | 0.044 | 4.07615E-02 | 6.05890E-13 |
| gcf | 38.28 | 2.936 | 0.000 | 0.000 | 0.011 | 9.99000E-01 | 8.28623E-12 |
| gser* | 419.46 | 386.408 | 0.003 | 0.237 | 0.258 | 3.23947E-12 | 3.23947E-12 |
| infotbl | 4.44 | 2.998 | 0.000 | 0.000 | 0.003 | 1.66855E-01 | 2.88601E-11 |
| get_token* | 477.12 | 113.40 | 0.051 | 0.061 | 0.075 | 2.42801E-03 | 1.50993E-11 |
| numeric_case* | 369.6 | 616.715 | 0.052 | 0.851 | 1.040 | 1.40202E-11 | 5.50986E-12 |
| get_token | 2.444 | 2.795 | 0.000 | 0.008 | 0.040 | 1.39400E-03 | 6.05890E-13 |
| is_num_constant* | 391.56 | 59.378 | 0.001 | 0.353 | 0.524 | 8.96896E-11 | 3.23947E-12 |
| is_str_constant* | 496.0 | 197.530 | 0.000 | 0.235 | 0.394 | 1.58011E-12 | 1.58011E-12 |
| is_token_end | 20.82 | 3.187 | 0.000 | 0.004 | 0.029 | 1.39400E-03 | 6.05890E-13 |
| find_nth | 3.54 | 3.043 | 0.000 | 0.000 | 0.004 | 9.99000E-01 | 6.52781E-08 |
| unblock_process | 11.4 | 3.159 | 0.000 | 0.000 | 0.013 | 9.99000E-01 | 6.05890E-13 |
| upgrade_process_prio* | 7.437 | 18.940 | 0.000 | 0.237 | 0.327 | 1.63417E-10 | 3.23947E-12 |

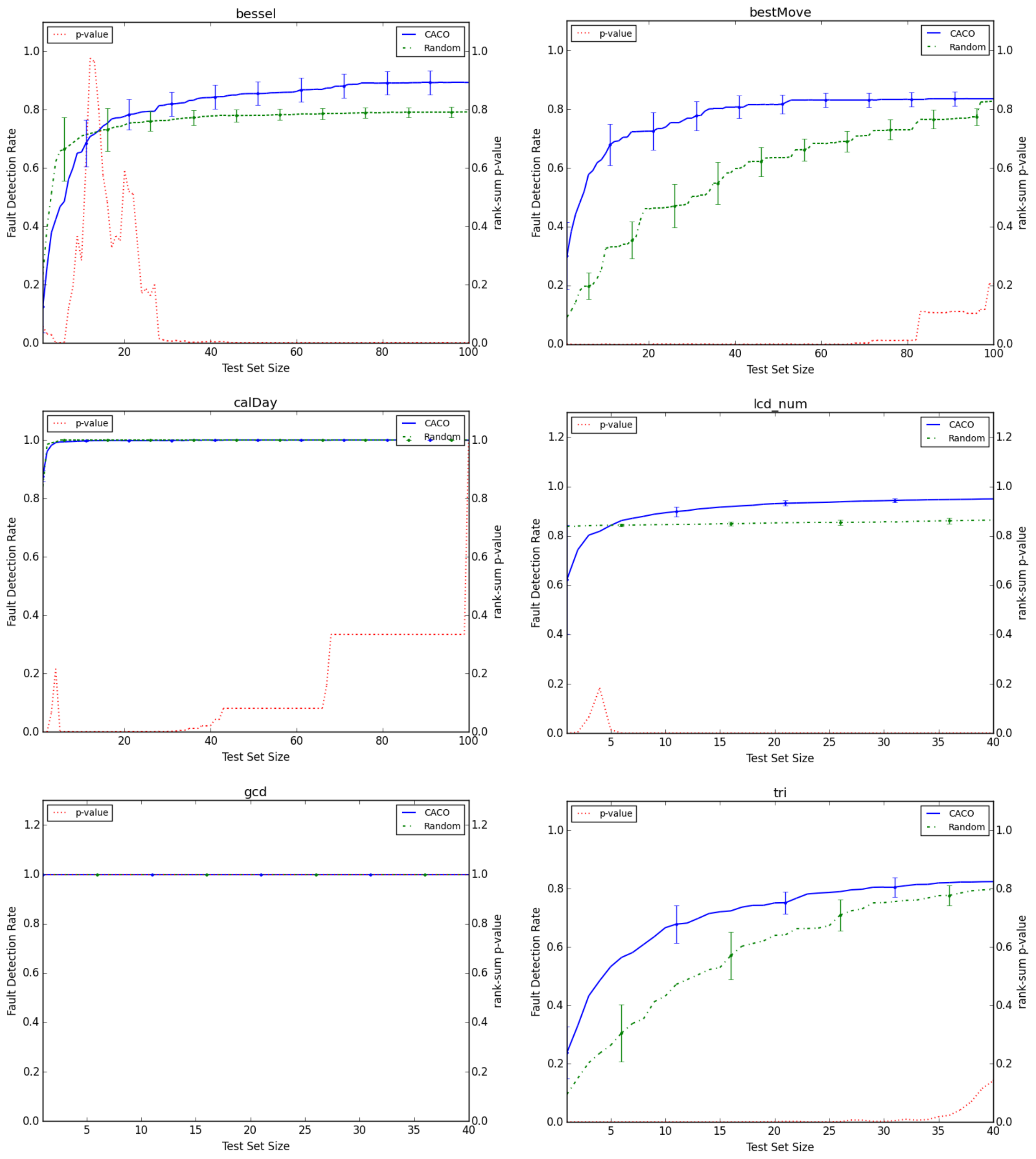


Figure 4.4: Fault Detection Ability Test on Mutation Testing Group

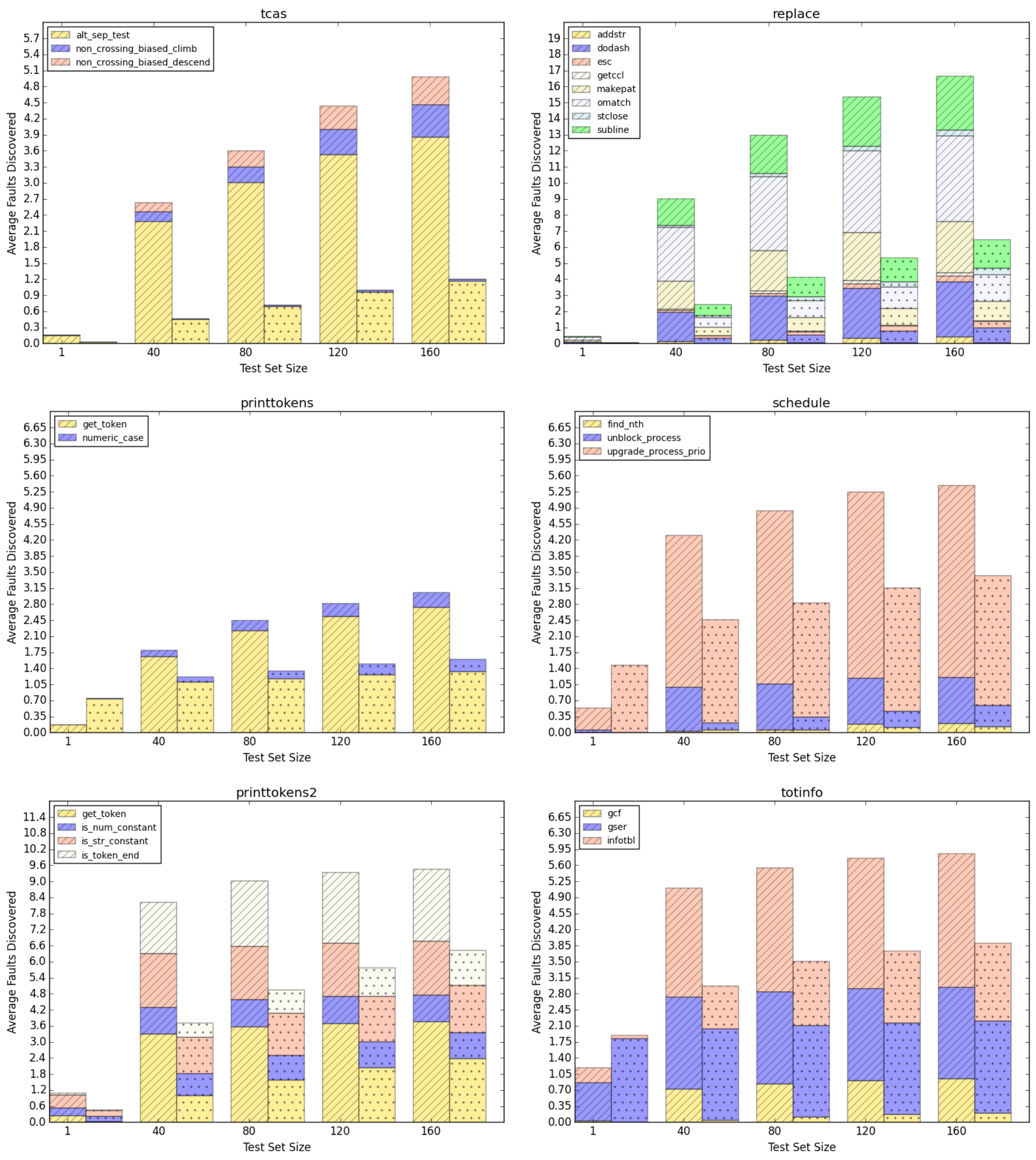


Figure 4.5: Fault Detection Ability Test On Siemens Test suite

### 4.3.3 Experiment Two: Fault-detecting Ability

To measure the fault-detecting ability, we collect multiple faulty versions of the original copy of each SUT function. For each one of the thirty two evolved input distributions, five test sets with each containing two hundred test inputs are sampled. Each test input runs both of the original and the faulty versions of the executable. If the outputs produced from the faulty versions are different from the original SUT's output, the faults are explained as "killed." A higher amount of faults found by a test set indicates a superior fault-detecting ability of the corresponding input distribution.

Two resources are used to collect the faulty versions of SUTs. First resource comes from a mutation testing tool, called Milu [30] to automatically generate a set of 350 to 5000 first-order and second-order mutants (faulty versions) for each SUT. Second resource comes from the Siemens test suite. The suite provides a set of benchmark programs and 7 to 41 faulty versions for each program.

The objective of the experiments on the fault-detecting ability is to demonstrate that an input distribution which satisfies or closes to the proposed criterion reveals an superior fault-detecting ability. The fault-detecting ability is expressed as the expected number of faults found at each test set size.

#### Mutation Testing Group

For each program in the mutation testing group, the mutant quantity produced by Milu is large. The faults are injected perhaps every possible location in the source code. In the mutant set, not every mutant is valid. For instance, the *calDay* function has 680 mutants. However, most of them trigger the segment fault exception. In this situation, although the output from mutants and the original copy are different, the "killed" mutant does not provide any useful information for the fault-detecting ability assessment. On the other hand, the live mutants possibly exist in the mutant set. These mutants cannot be "killed" by any test input. Hence, the fault-detecting rate hardly reaches 100%.

Theoretically, the biased input distribution cannot always outperform the uniform input distribution for all levels of test set size and vice versa [2]. Due to the randomness nature of mutant generation and the sum-up-to-one property on the probability distribution, some faults have a higher chance while other faults have lower chances being detected. Thus, at certain levels of test set size, the uniform input distribution should perform equally good as the biased input distribution. Hence, the comparison of the fault detecting ability via mutation testing requires knowing the interval of the test set size that $\text{CACO}_{\mathbb{R}}$ outperforms random testing. The test set sizes inside of the interval are so-called the effective test set sizes. The mutation testing results are shown in the Figure 4.4 in which, the left axis represents the averaged expected faults found per test (the fault detection rate) at each test set size over the 160 test sets. The right axis represents the rank-sum test results for the test sets that comes from $\text{CACO}_{\mathbb{R}}$ or random testing. If the p-value is less than 0.05, $\text{CACO}_{\mathbb{R}}$ is significantly better than random testing in the expected faults found per test.

Except for *CalDay* and *Gcd*, all the other programs show a clear effective test set size interval. For the *CalDay* program, executing most of the mutants raises the segment fault exception. Hence, the fault-detecting rate starts from a high level at the initial test set size and converges to 100% in the early test stages. For comparison, $\text{CACO}_{\mathbb{R}}$ performs almost the same as random testing with a slightly worse performance at the test set size around 4. For*Gcd*, all of the mutants can be "killed" by any test input. We believe that the *Gcd* program's structure is sensitive to its output. Then, whenever a modification is made on the program structure, the output is altered.

For *BestMove* and *Tri*, both of the effective test set size starts from 1. The upper bounds of the effective test set size are approximately 105 and 44 respectively. The p-values with test set sizes of more than 83 and 37 test inputs are

Table 4.4: Number of Tests for Full Fault Coverage

| Function | A.F | | p-Value |
|---|---|---|---|
| | CACO | Random | |
| addstr | 4.20E+02 | 7.88E+02 | 1.55E-04 |
| dodash | 1.00E+03 | 1.00E+03 | 1.00E+00 |
| esc | 6.51E+02 | 6.15E+02 | 5.77E-01 |
| getccl | 6.45E+02 | 8.77E+02 | 1.15E-02 |
| makepat | 4.06E+02 | 9.51E+02 | 3.95E-07 |
| omatch | 8.20E+02 | 1.00E+03 | 6.62E-04 |
| stclose | 8.31E+02 | 8.07E+02 | 6.98E-01 |
| subline | 2.12E+02 | 6.62E+02 | 1.58E-07 |
| alt_sep_test | 7.34E+02 | 1.00E+03 | 3.45E-07 |
| ncbc | 8.11E+02 | 1.00E+03 | 1.46E-04 |
| ncbd | 8.69E+02 | 1.00E+03 | 5.58E-03 |
| gcf | 6.54E+01 | 5.69E+02 | 6.41E-09 |
| gser | 2.87E+00 | 1.10E+00 | 9.89E-05 |
| infotbl | 6.48E+01 | 5.37E+02 | 1.50E-08 |
| get_token | 5.76E+02 | 9.91E+02 | 2.93E-07 |
| numeric_case | 4.77E+02 | 5.69E+02 | 5.04E-01 |
| get_token | 1.55E+02 | 6.80E+02 | 6.36E-08 |
| is_num_constant | 2.10E+00 | 2.76E+01 | 6.39E-05 |
| is_str_constant | 3.90E+00 | 4.38E+01 | 1.64E-03 |
| is_token_end | 1.09E+02 | 5.33E+02 | 1.92E-06 |
| find_nth | 8.23E+02 | 8.66E+02 | 7.17E-01 |
| unblock_process | 1.48E+01 | 2.90E+02 | 1.01E-09 |
| upgrade_process_prio | 3.27E+02 | 6.67E+02 | 6.66E-04 |

greater than 0.05. This implies that CACO$_{\mathbb{R}}$ does not significantly outperform the random testing with more than 83 or 37 test inputs for *BestMove* and *Tri* respectively. For *Besselj* and *lcd_num*, the effective test set sizes start from 18 and 4 respectively. The p-values are dramatically decreasing as the test set size cross the starting point. The upper bounds of the effective test set size, which are far from the lower bound cannot be suiteably displayed in the graph.

The above results demonstrate that biased input distributions provide superior fault-detecting ability than random testing at the majority of test set sizes. Furthermore, at the highest test set size shown in the graph, the fault-detecting rates of the biased input distributions are higher than 0.8. This observation implies that even though the fault-detecting rates from random testing might exceed biased input distributions with larger test set sizes (for *bestMove* or *tri*), random testing has a relatively small space of improvement on the fault-detecting rate. Hence, the above results provide evidence that, for mutation testing group, the biased input distributions show a superior fault-detecting ability.

### Siemens Test suite

In the Siemens test suite, each program has a few artificial faults that are publicly known erroneous mistakes. To assess the fault detection ability of an input distribution by using the provided faults, we use the following two assessment metrics.

1. The average number of faults discovered at each test set size. (A.T)
2. The average number of test inputs required to discover all faults. (A.F)

The A.T results are represented as a set of stacked bar charts in Figure 4.5. It can be seen that the summation of the average discovered faults for each test set size results from $\text{CACO}_{\mathbb{R}}$ (shown on the LHS bar) are always greater than random testing (shown on the RHS bar) except for the benchmarks *tcas*, *printtokens*, *schedule* and *totinfo* with test set size equals 1. We argue that the smaller test set size is not useful for evaluating the actual fault detection ability in a practical testing environment. Speaking in detail, with regards to *schedule*, 86% of comparisons for each function in every test set size shows that the average faults discovered by using the $\text{CACO}_{\mathbb{R}}$ approach are higher than the random testing method. With regards to other programs, the statistics are 0.69%, 0.995%, 0.79%, 0.998% and 0.667% for *printtokens*, *printtokens2*, *replace*, *tcas* and *tcas* respectively.

The A.F results are shown in Table 4.4. Only for the program *dodash*, both of the $\text{CACO}_{\mathbb{R}}$ and random testing methods cannot reach the full fault coverage within the 1000 (the maximum) tests. For other programs, *ncbd* requires the largest (869) test set size by using the $\text{CACO}_{\mathbb{R}}$ algorithm. To compare the A.F between the two methods with confidence, we performed the two-tail rank-sum test. The p-values (see Table 4.4) indicate that the $\text{CACO}_{\mathbb{R}}$ and random testing are not significantly different in the expected faults found for functions *dodash*, *esc*, *stclose*, *numeric_case* and *find_nth*. In other functions, random testing outperforms $\text{CACO}_{\mathbb{R}}$ in 61.63% for *gser*, whereas $\text{CACO}_{\mathbb{R}}$ outperforms random testing for the rest functions. The largest improvement from random testing is 94.89% on function *unblock_process*. The smallest improvement from random testing is 13.10% on function *ncbd*.

Among the overall (23) benchmark functions, 73.9% of them demonstrate that the fault detection ability of biased input distributions produced by the $\text{CACO}_{\mathbb{R}}$ algorithm is superior to random testing. 21.7% of the functions show that there is no statistical difference between the two methods. Lastly, 4.3% of the functions show that random testing outperforms $\text{CACO}_{\mathbb{R}}$. This results provide the evidence that, $\text{CACO}_{\mathbb{R}}$ algortihm is more efficient in detecting faults than random testing.

## SUMMARY

Statistical structural testing is a powerful tool for software testing. However, due to the complicated input distribution construction process, it is hard to apply it to real-world problems. Even with the help of automated search, under an uncertain environment, search is primarily inefficient. In this chapter, we introduced a novel search-based input distribution construction algorithm called constrained ant colony optimization on the real domain ($\text{CACO}_{\mathbb{R}}$). We model the input distribution construction process as a resource allocation process and uses the ant colony metaphor to optimize an input distribution. ACO can deal with arbitrarily large noise in a graceful manner due to the pheromone accumulation effect. The traditional discrete ACO is not suitable for SBSST since a vast input domain could explode computation resources. Hence, we adopt the $\text{ACO}_{\mathbb{R}}$ algorithm and enhance it to support constrained optimization for SBSST problems. The experimental study shows that CACO$_{\mathbb{R}}$ algorithm has a significant advantage over the hill-climbing algorithm in searching for optimal input distributions. More importantly, CACO$_{\mathbb{R}}$ can derive the target solution in a reasonable timeframe.

There are three areas for future works. First, the current SBSST is still limited to unit testing with considerable cyclomatic complexity. Coverage-driven fuzzing is a popular testing approach that has been used in the industry recently. Coverage-driven fuzzing treats the SUT as a black box, and the test inputs are randomly generated without the structural knowledge of the SUT. One research direction is to improve fuzz testing by using the SBSST approach. Second, we could continue to explore other search algorithms, which might perform better than CACO$_{\mathbb{R}}$ algorithm in a noisy environment. Third, the current SBSST only deals with primitive data types, whereas handle complex data types(e.g., string, class) is necessary for real-world applications.

# Chapter 5

# SEARCH-BASED STATISTICAL STRUCTURAL FUZZING

## INTRODUCTION

Coverage-guided fuzzing is one of most practical fuzzing strategies at present. Comparing to grammar-based fuzzers, it requires much lower computational overhead. In practise, the coverage-guided fuzzers such as AFL, libFuzzer, FairFuzz have shown the profound effectiveness in revealing bugs and security vulnerabilities for a wide range of real world software, including various system libraries, jpeg reader, webbrowers etc. Those fuzzers are also deployed in the cloud platform like Google's OSS-Fuzz platform, which continuously test open source application and found over 1000 bugs in 5 months. AFL essentially tries to evolve test cases with respect to branch coverage. The whole process is similar to the hill-climbing algorithm, where during each cycle, a test case(parent) is mutated to produce an offspring, and the offspring undergoes evaluation against the branch coverage. If the branch coverage is higher than the one received from the parent, this offspring becomes the parent in the next cycle. Same as AFL's test case enhancement process, the probability distribution can be trained on-the-fly via search-based methods. Poulding and Clark has demonstrated the efficacy and practicality use of hill-climbing algorithm to search for such optimal distribution. However, their bench-mark programs are relatively simple. To study the effectiveness of the search-based SST framework for uncovering the vulnerabilities, we applied this framework into AFL and conducted experiments with real-world binaries for comparison. Instead of hill-climbing algorithm, we develop a novel search algorithm called $\text{CACO}_{\mathbb{R}}$, dedicating to searching for optimal probability distribution, and compare the fault-detecting ability of our algorithm with the popular fuzzing technique AFL.

## OVERVIEW

This section briefly introduces the proposed search-based framework, followed by providing the definition of input probability distributions. Lastly, it explains the input distribution construction process via the ant system dynamics.

### 5.2.1 Input Probability Distribution

We build a set of discrete probability distributions over the input domain space. In AFL, a test case can be a binary file or text. For binary file, byte is the minimum data unit for memory access, whereas for text file, a character is represented by a byte value. Hence, we use the byte-level representation for the input domain space. For each byte, we assign probabilities to every possible outcome $x \in \{0, 255\}$. Formally, the probability distribution for the i-th byte is denoted as $P_{b_i}(x)$ where $x \in \{0, 255\}$. We also define a $\text{CACO}_{\mathbb{R}}$ specific parameter $\delta$ representing the minimum unit of probability amount can be assigned to an outcome. The $\text{CACO}_{\mathbb{R}}$ approach models the probability distribution optimization problem as a *resource allocation problem.* The resource refers to the total amount of probabilities values, which is 1.0.

### 5.2.2 Ant System Dynamics

Ant foraging phenomenon can be viewed as a positive feedback loop system. In the system, rational ants always choose the shortest path to a food source. The two key mechanisms involved are the *pheromone accumulation* process which describes the accumulation of pheromones on a path, and the *pheromone evaporation* process which describes the evaporation of pheromones over time. Suppose each ant emit the same amount of pheromones on the path, and the pheromones are evenly distributed. The shortest path to a food source would have the highest amount of pheromones. A rational ant favors the path that has more pheromones. Therefore, it is more likely to pick the shortest path. In the meantime, the pheromones on all the paths are constantly evaporating over the time. The final outcome is that only the shortest path contains pheromones, and all the ants have to travel along this path. We use the ant system dynamics to explain the resource allocation problem for constructing probability distributions. For each byte, there are 256 outcomes, each outcome requires a probability value. We could assign $p \in \{0, \delta, 2\delta, ..., 1\}$ to an outcome. Therefore, the total number of possible values for each outcome is $\frac{1}{\delta}$. Assume an ant system tries to select the best value for each outcome. Suppose an ant firstly selected $2\delta$ for outcome 0. The resources left for the remaining outcomes are $1 - 2\delta$. It can be seen that the values available to the next outcome depends on the previous resource assignment. This brings up two observations:

- The ant travel path is order-dependent(ie. $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$ is different from $0 \rightarrow 2 \rightarrow 3 \rightarrow 1$). However, the pheromones represents the favor level of a resource assignment is independent of paths.

- When an ant select resource amount, it is constrained to the available resources.

Based on the above key points, The proposed ant system works as follows. An ant always firstly randomly pick up a permutation order of the outcomes(e.g. $C_1 \rightarrow C_2 \rightarrow C_3$). For each outcome in turn, the ant will pick up a resource value according to the pheromone level left on the value. However, the value is also constrained on the available resources(e.g $R$, $R^{'}$, $R^{''}$). Once the resource allocation completes, we calculate the pheromone amount based on the fitness evaluation of the newly generated input distribution. Then, we update the pheromone on the assigned resource values. Also, we update pheromone on all resource values for

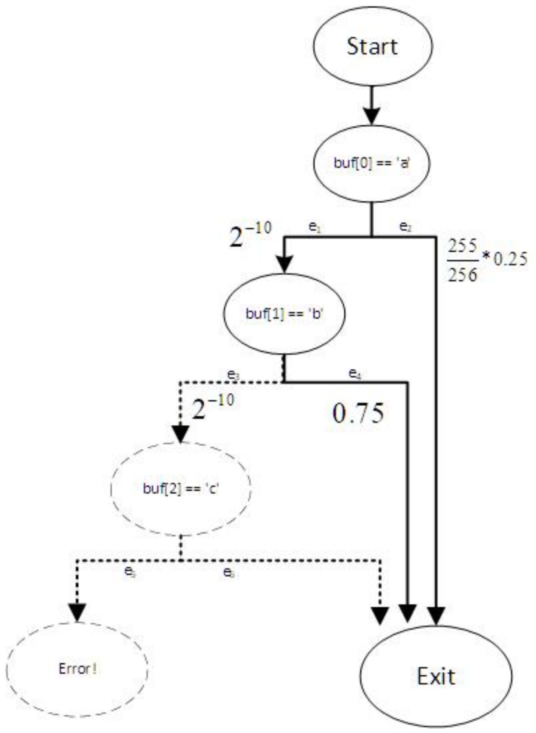


Figure 5.1: Example of nested conditions

simulating the pheromone evaporation process. If we continue this process for enough cycles, the ants would finally favor with the resource values that have the most pheromones.

## MOTIVATIONS

While AFL can produce a wide range of mutated inputs, the ability to discover new program paths is still questionable. We list the following code structures that AFL is hard to perform well.

### Deep nested conditions

In this code structure, with a random mutation strategy, the majority of generated input cannot reach the deeper condition. For instance, in Figure 5.1, suppose AFL has found an input that exercises buf[0] == 'a' and buf[1] == 'b' and buf[2] != 'c'. By randomly mutating a byte in one of the 4 bytes in buf, the produced test input has the probability of $2^{-10}$ to reach buf[2] == 'c', has the probability of 0.75 to remain in the current path, and has the probability of $0.25 * \frac{255}{256}$ to reach buf[0] != 'a'. It is noticed that the path buf[0] != 'a' occupied the majority of exercised paths in the previous runs.

To increase the chance to trigger the deeper condition, AFLfast deploys an *indirect* way that controls the mutation times. AFLfast notice that the high-density region of exercised paths are all in buf[0] != 'a', then its energy scheduler will reduce the mutation times on the path that exercise buf[0] != 'a' and increase the mutation times on the path that exercises buf[0] == 'a' and buf[1] == 'b'. Let n denote the mutation times. The probability of exercising at least one times of the path buf[0] == 'a' and buf[1] == 'b' and buf[2] == 'c' is $1 - (1 - 2^{-10})^n$. It is noticed that the chance becomes greater as n increases. However, choosing the value of n is still debatable. Effortless test's quantity is high if n is too large, whereas chances to discover the deeper condition are low if n is too small.

### Scanty input sub-domain space

It is common to see the code structure that holds a conditional statement with a scanty input sub-domain space. The code in Figure 5.1 shows such structure. Another example is the magic bytes, which refers to a block of arcane byte values used to designate a file type (e.g, jpeg). Generally, an application decides whether to continue execution by checking with the magic bytes. To test the application, the condition to check with magic bytes must be true. Random mutation strategy for this type of code structure is inefficient. In Figure 5.1, triggering the taken branch of any conditional statement requires an average of 1024 tests.

## METHODOLOGY

This section first provides an overview of the proposed method followed by giving a detailed description about each component.

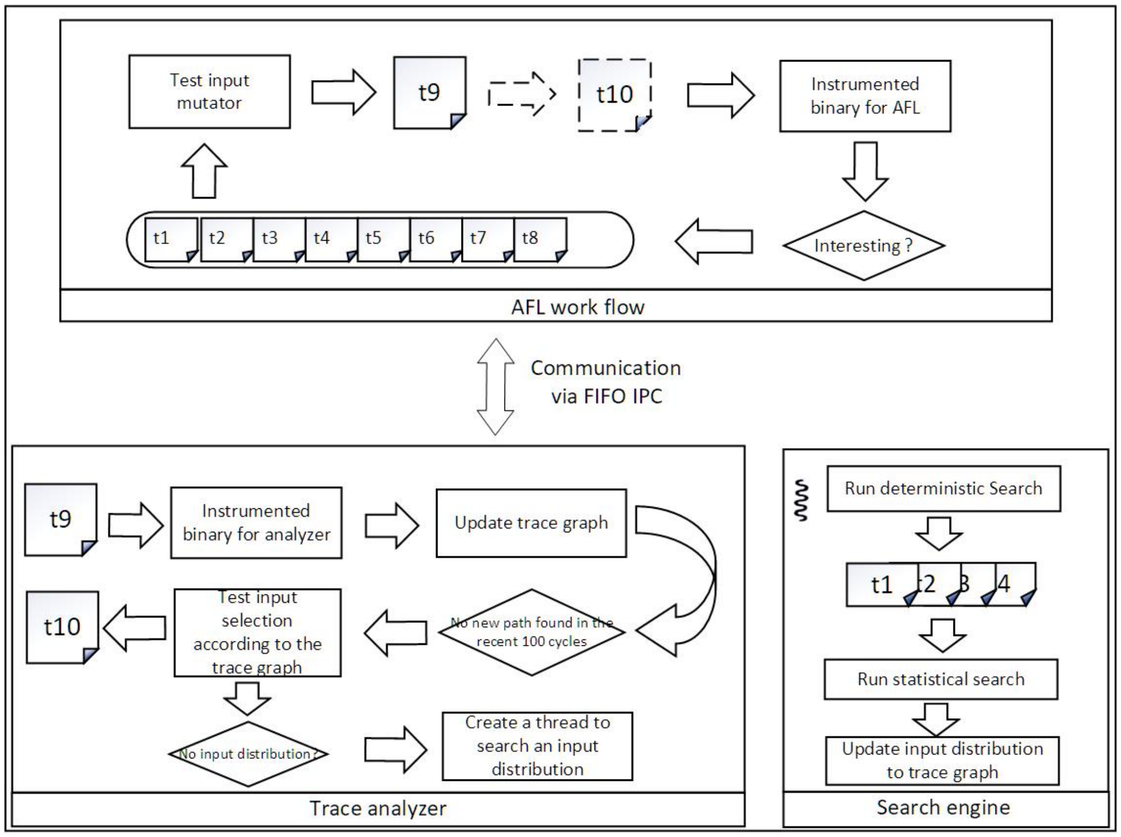

Test input mutator
t9
t10
Instrumented binary for AFL
Interesting ?
t1 t2 t3 t4 t5 t6 t7 t8
AFL work flow
Communication via FIFO IPC
t9
Instrumented binary for analyzer
Update trace graph
t10
Test input selection according to the trace graph
No new path found in the recent 100 cycles
No input distribution?
Create a thread to search an input distribution
Trace analyzer
Run deterministic Search
t1 t2 t3 t4
Run statistical search
Update input distribution to trace graph
Search engine

Figure 5.2: SBSFuzz's architecture

### 5.4.1 Overview

The SBSFuzz is different from the AFLfast in the way that it *actively* produces random test inputs that trigger a target path. The SBSFuzz's architecture is shown in Figure 5.2. SBSFuzz utilizes a modified AFL as the main fuzz engine. Before running the instrumented binary with a mutated test input (e.g, t9 in the graph), AFL sends a signal to the trace analyzer via FIFO, an interprocess communication protocol(IPC). The trace analyzer then runs its instrumentation version of the target binary with the test input ($t9$) to obtain a piece of *fine-grained path information*. The trace analyzer then uses the path information to update a trace graph. By examining the trace graph, the trace analyzer decides to produce a random test input if AFL is stuck in uncovering new paths in the last 100 cycles.

The test input is sent to AFL via FIFO and replaces the ready-to-run test input.

Nodes in a trace graph represent conditional statements. Each node is associated with an input probability distribution. The chance of a randomly selected test input that exercises an outgoing edge is proportionate to the *path depths* after the edge.

To randomly generate a test input, the trace analyzer follows two steps. First, it randomly selects a node in the graph with *input-sensitivity proportionate selection*. Second, it samples a test input from the node's input distribution.

If the input distribution is unavailable, the trace analyzer initiates a thread to run a comprehensive search to find the input distribution. The search begins with a deterministic strategy to discover individual test inputs that exercise all outgoing edges. Then, the statistical strategy leverages the discovered test inputs to construct input distributions.

### 5.4.2 Trace Infomration

The compile-time instrumentation allows recording the following trace information.

- *cmp_id*. The instrumentation assigns unique id stars from 1 to each cmp instruction sequentially.
- *edge_id*. The instrumentation assigns a unique and random 32-bit unsigned id to the beginning of each basic block.
- *source/destination operand values*. The instrumentation stores the source and destination operand values before executing the cmp instruction.
- *jump type*. The instrumentation stores the jump type that follows each cmp instruction.

Each information is stored in a Linux shared memory which is limit to 64MB per allocation in the Ubuntu kernel. The Typical shared memory size of the *cmp_id* trace for the LAVA-M benchmark is 16MB.

An example of the *cmp_id* trace information is shown in the followings: 1,2,5,10,12,13,17,20. Due to the spatial limitations, loop detail is deliberately ignored. The above *cmp_id* trace is equivalent to 1,2,5,10,12,13,10,12,13,17,20

### 5.4.3 Trace Graph

The trace analyzer maintains a trace graph $G(V, A)$ to record exercised program paths. In the graph, $V$ denotes the set of *cmp_id*, $A$ denotes the *edge_id* that connects between source and destination *cmp_id* pairs. Each *cmp_id* node is associated with a probability distribution to randomly generate test inputs. The trace analyzer keeps a counter for each exercised edge to record the testing progress. When no new program paths found, it randomly produce test inputs in accordance with the test input's ability to discover new paths. We measure the ability from two perspectives.

#### Cmp's input-sensivity

Let $m_k$ denote the number of test input exercises the taken edge of the k-th *cmp_id* node in the trace graph, $n_k$ denote the number of input exercises the non taken edge of the k-th *cmp_id* node in the trace graph. We borrow the idea from information entropy for cmp's input sensitivity $\mathcal{I}_k$, which is defined as follows:

$$\mathcal{I}_k = -(\frac{m_k}{m_k + n_k} * log_2 \frac{m_k}{m_k + n_k} + \frac{n_k}{m_k + n_k} * log_2 \frac{n_k}{m_k + n_k})$$

cmp node's input sensitivity measures the chances that test inputs generated in the previous runs can affect the execution flow of the cmp node. Note that if $\mathcal{I}$ equals 1, meaning 50% chances are observed on both outgoing edges, this node has low priority to be sampled. On the other end, if $\mathcal{I}$ equals 0, meaning all previous test inputs are observed on one edge. The cmp node possibly owns a scanty input subdomain, or it might be a condition that checks the loop counts. This type of node has a high priority to be investigated.

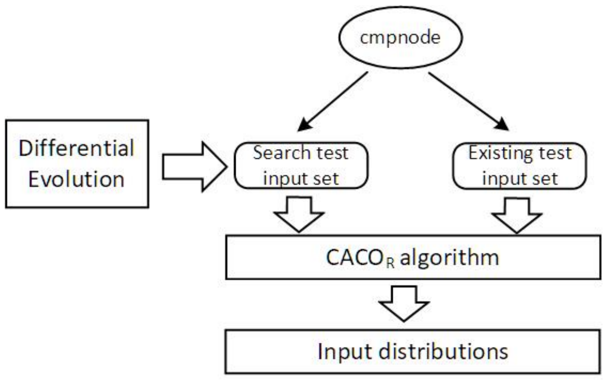


Figure 5.3: Architecture of the comprehensive search

The trace analyzer performs a *input-sensivity proportionate selection* to select a *cmp_id* node before producing a test input from the associated probability distribution. Specifically, the probability$p_k$ of selecting the k-th *cmp_id* node is

$$p = \frac{I_k}{\sum_i I_i}$$

### 5.4.4 Comprehensive Search

#### Path depth

We set up the probability distribution that triggers an outgoing edge with priority to the existing longest path after the edge. In the example of Figure 5.1, suppose that at the current stage, AFL found $e_2$ and $e_1, e_4$. The trace analyzer requires to set up a probability distribution for node "buf[0] == 'a'. Note that only the test inputs that execute $e_1$ provide testing benefits, since the observed path depth at present after $e_1$ is 1, whereas path depth at present after $e_2$ is 0. In this case, testing $e_1$ has the higher priority. Formally, let $d_t$ be the depth of the existing path depth after the taken edge, $d_n$ be the depth of the existing path depth after the non-taken edge. The expected triggering probability for the taken edge is $tri_t = \frac{d_t}{d_t+d_n}$ and the expected triggering probability for the non-taken edge is $tri_n = 1 - \frac{d_t}{d_t+d_n}$.

Figure 5.3 shows the overview of the search algorithm. In the trace graph, each *cmp_id* node holds repositories to store test inputs for both outgoing edges. If no test inputs are found in one repository, the trace analyzer starts a deterministic search via differential evolution. Then, $CACO_{\mathbb{R}}$ is performed to construct the input distribution. The differential evolution(DE) leverages the *branch distance* and the *cmp_id* level to measure a test input's fitness. By checking with the jump type, the DE selects a proper branch distance function as the fitness measure.

Table 5.1: Branch distance function for conditions

| **jump type** | **fitness** |
|---|---|
| *jeq* | f = abs(dest - src) |
| *jne* | f = -abs(dest - src) |
| *jae/ja/jge/jg* | f = dest - src |
| *jbe/jb/jle/jl* | f = src - dest |

Table 6.2 shows the list of the jump types with branch distance functions. SBSFuzz's testing strategy deploys a direct way that performs random walks over the existing trace graph. At each *cmp_id* node, we construct an input probability distribution. The test inputs sampled from the distribution guarantee to trigger the outgoing edge with a probability lower bound. Setting up the probability criterion helps to prioritize tests to deep paths. In the example of Figure 5.1, A example input distribution for node "buf[0] == 'a' is $P^0(x =' a') = 100\%$ $P^i(x \in S) = \frac{1}{|S|} S = 0, 1, ..., 127$

## EVALUATION

To evaluate SBSFuzz's bug-discovering ability, we compared the number of bugs found by SBSFuzz, AFL, and AFLFast with the LAVA-M benchmark and multiple CVEs. We conducted experiments under the platform with an AMD Ryzen Threadripper 1950X 16-Core Processor and 16 GB memory running 64-bit Ubuntu 18.04 LTS.

### 5.5.1 Fault-detecting Ability

#### LAVA-M

LAVA is a technique for producing common corpora by injecting a large number of realistic bugs into program source code. LAVA-M is a set of faulty binaries injected by LAVA. LAVA-M consists of four GNU *coreutils* programs: *uniq*, *base64*, *md5sum*, and *who*. Each program is injected with multiple well-crafted and realistic bugs. Each injected bug has a unique ID. Table 5.2 shows the SUT's statistics.

Table 5.2: SUT statistics

| | ELOC | Listed_bugs |
|---|---|---|
| *base64* | 104 | 44 |
| *md5sum* | 297 | 57 |
| *uniq* | 195 | 28 |
| *who* | 279 | 1443 |

Table 5.3: Bugs found by SBSFuzz, AFL and AFLFast

| Program | | AFL | | AFLFast | | SBSFuzz | |
|---|---|---|---|---|---|---|---|
| | | bugs found | total inputs | bugs found | total inputs | bugs found | total inputs |
| uniq | 28 | 9 | 22,072k | 12 | 16,854k | 27 | 12,984k |
| base64 | 44 | 0 | 29,341k | 10 | 22,005k | 43 | 14,778k |
| md5sum | 57 | 0 | 19,225k | 9 | 14,251k | 50 | 11,792k |
| who | 2136 | 0 | 10,611k | 22 | 8058k | 214 | 6,632k |

We compared the fault-discovery ability of SBSFuzz with AFL and AFLFast. We let each fuzzer run 5 hours on each binary in the LAVA-M test set and repeat five times to report the average performance. The search algorithm takes an input file size of less than 50kbytes. For differential evolution, the search pool consists of 50 input files. For $CACO_R$, the search pool consists of 20 input distributions. The sample-set size is set to 500. The search terminated when either fitness equals 0 or no fitness improvement during the last 100 runs. Table 5.3 compares the bugs found by SBSFuzz with AFL and AFLFast. SBSFuzz found more bugs in each binary

Table 5.4: Time to expose to vulnerabilities

| Vulnerability | nm | | | objdump | | | c++filt | | |
|---|---|---|---|---|---|---|---|---|---|
| | AFL | AFLFast | SBSFuzz | AFL | AFLFast | SBSFuzz | AFL | AFLFast | SBSFuzz |
| CVE-2016-2226 | 18.48 | 6.82 | 4.31 | 32.64 | 17.23 | 12.14 | 13.26 | 7.23 | 5.49 |
| CVE-2016-4487 | 1.57 | 0.5 | 0.63 | 18.92 | 10.23 | 6.53 | 3.73 | 2.14 | 1.53 |
| CVE-2016-4488 | 5.74 | 2.53 | 2.48 | 12.43 | 11.74 | 7.48 | 2.43 | 1.23 | 1.12 |
| CVE-2016-4489 | 10.13 | 6.25 | 3.84 | 18.29 | 7.32 | 8.21 | 7.24 | 4.31 | 3.26 |
| CVE-2016-4490 | 3.21 | 1.42 | 0.72 | 8.74 | 2.10 | 2.24 | 2.11 | 0.74 | 1.23 |
| CVE-2016-4491 | 19.12 | 9.14 | 10.13 | 29.21 | 19.32 | 15.43 | 7.43 | 2.13 | 2.04 |
| CVE-2016-4492 | 10.32 | 6.59 | 4.86 | 20.32 | 14.32 | 10.23 | 14.21 | 5.42 | 4.89 |
| CVE-2016-4493 | 3.28 | 1.24 | 1.14 | 7.24 | 4.31 | 2.71 | 8.43 | 3.15 | 2.84 |
| CVE-2016-6131 | 19.21 | 12.34 | 5.32 | 21.34 | 15.43 | 12.53 | 11.23 | 5.22 | 4.83 |
| Average | 10.12 | 5.20 | 3.71 | 18.79 | 11.33 | 8.61 | 7.79 | 3.51 | 3.03 |

in the data set. Particularly, SBSFuzz has found 214 bugs in *who*, nine times more than AFLFast, whereas AFL did not find any bug. SBSFuzz outperforms the other two fuzzers mainly due to its active searching strategy. LAVA injects bugs with the "magic bytes" format. The magic bytes are not copied directly but computed from input files. The high effectiveness of detecting bugs on LAVA-M demonstrated that SBSFuzz's input distribution setup allows the paths with a deeper depth to be executed with priority. Those paths are valuable to catch bugs.

**Real Vulnerabilities**

GNU Binutils is a non-trivial and widely used open-source Linux utility for evaluating fuzzer's performance. It consists of several famous tools, including *nm*, *c++filt*. We choose the same experimental settings as AFLFast, which leverages GNU Binutils v2.26 as the subject. AFLFast discovered several vulnerabilities, and nine of them are assigned with CVE numbers. To compare the performance of SBSFuzz with AFLFast and AFL, we run both fuzzers with utility *nm*, *c++filt*, and *objdump*. Each utility makes use of the *libiberty* library which exposes to the listed CVE vulnerabilities. We run each utility five times and compare the average time first to expose each listed vulnerability. As observed from Table 5.4, although for particular CVEs(e.g, CVE-2016-4493 on *nm*), SBSFuzz performs at the same level of fuzzing time. On average, SBSFuzz caught the vulnerabilities with fuzzing time 29% less than AFLFast for *nm* binary, 24% less than AFLFast for *objdump* and 13% less than AFLFast for *c++filt*.

## SUMMARY

The structural coverage-guided fuzz testing technique is getting more and more popular recently. The most popular tools AFL and AFLFast show a satisfactory bug-revealing ability. The principle behind the sense is that a test case that could detect more paths has a higher chance of discovering bugs. However, if a bug can only be triggered by a small subset of the test input sub-domain spaces, the bug is hard to be detected. To minimize this negative effect, we proposed to use a more tightened coverage criterion: statistical branch coverage, which is demonstrated to be more effective in the fault discovering ability. The test cases are sampled from a probability distribution in our approach, and we developed comprehensive search algorithms to evolve the distribution. Our experimental results showed that SBSFuzz is more efficient in discovering vulnerabilities than AFLFast and AFL.

# Chapter 6

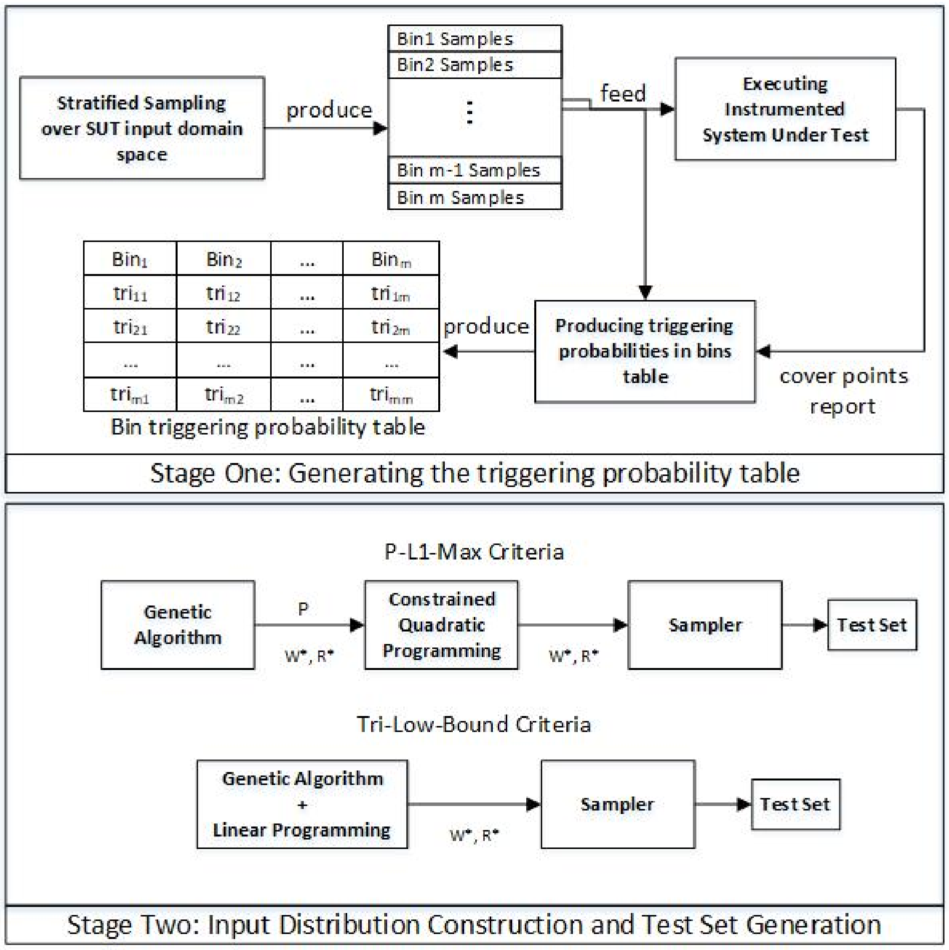
# OTHER SBST TECHNIQUES

In this chapter, we consider an input distribution construction process as a two-step process shown in Figure 6.1. We begin with sampling the entire input domain space by stratified sampling method, which takes samples in each consecutive and non-overlapped sub-input domain space (a.k.a, bins). Then we estimate the triggering probabilities in bin-wise by running the samples with a SUT and store the triggering probabilities in a *bin triggering probability table.* Recall that the input distribution model is the weighted sum of uniform distributions. Given the table, our target is to optimize the weight vector $\omega$ and the arrangement of bins for each uniform component. Suppose there are $k$ bins that store inputs. Each bin denoted as $b_i$ is an input set $S_i$ and is associated with a weight parameter $w_i$. $w_i$ ranges in $[0, 1]$, represents the probability of selecting the input set $S_i$. $\sum_{i=1}^{k} w_i = 1$. Each input $x$ can choose 1 out of the $k$ bins to be stored. An *arrangement* is the selection of bins over the entire input domain space $S$.

For simplicity, we assume the independence of each input variable. The input distribution model is a weighted sum of uniform distributions over a collection of bins in which the vectors of triggering probabilities associated with each bin are linearly independent. The input distribution formula is defined as follows:

$$P(x) = \sum_{i=0}^{k} w_i * U(x), x \in S_i$$

The triggering probabilities revealed by the weighted uniform distribution can be written in a matrix form $A$, where the columns represent the bins, the rows


Stratified Sampling over SUT input domain space
produce
Bin1 Samples
Bin2 Samples
Bin m-1 Samples
Bin m Samples
feed
Executing Instrumented System Under Test
cover points report
Producing triggering probabilities in bins table
produce
Bin triggering probability table
Stage One: Generating the triggering probability table
P-L1-Max Criteria
Genetic Algorithm
P
W*, R*
Constrained Quadratic Programming
Sampler
Test Set
Tri-Low-Bound Criteria
Genetic Algorithm + Linear Programming
Sampler
Test Set
Stage Two: Input Distribution Construction and Test Set Generation

| Bin$_1$ | Bin$_2$ | ... | Bin$_m$ |
|---|---|---|---|
| tri$_{11}$ | tri$_{12}$ | ... | tri$_{1m}$ |
| tri$_{21}$ | tri$_{22}$ | ... | tri$_{2m}$ |
| ... | ... | ... | ... |
| tri$_{m1}$ | tri$_{m2}$ | ... | tri$_{mm}$ |

Figure 6.1: work-flow of the input distribution construction process

represent the branch cover elements. The value of the i-th row and the j-th column denoted as $a_{ij}$ is the triggering probability of $c_i$ in $b_j$.

$$A = \begin{array}{c c} & \begin{array}{cccc} b_1 & b_2 & \dots & b_k \end{array} \\ \begin{array}{c} tri_1 \\ tri_2 \\ \vdots \\ tri_m \end{array} & \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1k} \\ a_{21} & a_{22} & \dots & a_{2k} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mk} \end{bmatrix} \end{array}$$

The parameters under control are the weight vector $\omega$ and arrangement of bins, which forms a matrix $A$. Optimizing the arrangement of bins is a combinatorial optimization problem, which we adopt the Genetic algorithm to search for the solution. Depending on the chosen optimization objective, tunning the weight vector requires different techniques. In this chapter, we introduce two objectives.

## CONSTRUCT THE $\mathcal{A}$ MATRIX

We construct the primitive matrix $A$ from samples to estimate the triggering probabilities. We adopt the Stratified Random Sampling approach. The benefit of using SRS is that it may produce a smaller error of estimation than a simple random sampling approach of the same sample size. On the other hand, it could also reduce the search complexity by decreasing the search dimension for the Genetic Algorithm.

**Domain Partitioning into Stratums** The entire input domain space is partitioned into $s$ equivalent size, non-overlapping, consecutive groups, denoted by $\delta$: $\{\delta_0, \ldots, \delta_{s-1}\}$. Suppose the input domain space is 1-dim, with $x \in \mathbb{Z}^+$. Let $d$ be the sub-input domain size, then $\delta_i$ represents sub-input domain space ranges within: $[i * d, i * (d + 1)]$.

**Estimating Triggering Probabilities in Statum** For each $\delta_i$, we randomly select equal number of inputs from its sub-input domain space and run against the SUT to retrieve the cover element information. Then the estimated triggering probabilities for $\delta_i$, denoted by $tr\hat{i}_{\delta_i}$: $[p_i^1, p_i^2, \ldots, p_i^m]$ can be derived by applying Equation 2.2.

**Estimating Triggering Probabilities in a Primitive Matrix** $A$ An arrangement of inputs in terms of $\delta$ set is the concatenation of the input sets or bins, in which each set $S_i$ is a mutually exclusive subset of $\delta$. Let $s_\delta^i$ be the subset of $\delta$

that belongs to the i-th bin $S_i$. Formally, an arrangement $\mathcal{C}$ is defined as follows:

$$\mathcal{C} = S_1 \parallel S_2 \parallel \ldots \parallel S_m$$

$$where:$$

$$\begin{cases} S_1 = s_\delta^1,\ s_\delta^1 \subseteq \delta \\ S_2 = s_\delta^2,\ s_\delta^2 \subseteq \delta \setminus s_\delta^1 \\ \quad\vdots \\ S_m = s_\delta^m,\ s_\delta^m \subseteq \delta \setminus \bigcup s_\delta^x,\ x \in \{1, \ldots, m-1\} \end{cases}$$

Then, the triggering probability for the i-th column of the primitive matrix $A$, denoted by $\hat{\alpha}_i$ is derived by averaging the estimated triggering probabilities of the elements in $s_\delta^i$. The formula is defined as follows:

$$\hat{\alpha}_i = \frac{1}{|s_\delta^i|} * \sum_{\mathcal{X} \in s_\delta^i} \hat{\mathbf{tri}}_{\mathcal{X}}$$

## SELF-ADAPTIVE SEARCH WITH L2-DISTANCE CRITERION

The first objective leverages the L2-distance between the estimated triggering probabilities $\hat{\mathbf{tri}}$ and the expected triggering probabilities $\mathbf{tri}$. Formally,

$$d(\mathbf{w}) = \sum_{i=1}^{m} (A * \mathbf{w_{[0:\,m-1]}} + \mathbf{b} - \mathbf{tri})^2$$

This equation is a quadratic form which has a unique global solution w* that minimizes the distance $d$. w* can be analytically derived as follows:

$$\begin{cases} \mathbf{w}^*_{\mathbf{[0:\,m-1]}} = (A^T A)^{-1} A^T (b - \bar{\mathbf{tri}}) \\ w_m^* = 1 - \sum w_i^* \end{cases} \tag{6.1}$$

Hence, for any arrangement of inputs, there is one and only one weight vector that provides the minimum distance. In other words, there is one optimal arrangement that minimizes the distance d(w*). However, a solution from the least square method might violate the inequality constraints on the weight vector. We could consider the least square problem as a Constrained Quadratic Optimization problem and use iterative methods, for instance, the interior method to solve the problem. However, since those methods are iterative, embedding them in GA causes a massive computation time. In GA, 5000 generations run with 100 individuals required to run the CQO algorithm $5 * 10^4$ times, which is not feasible.

Consequently, we reduce the search domain space from $\{\mathbf{w}, bins'arrangement\}$ to bins' arrangement only. Then, the optimization objective becomes searching for an optimal arrangement such that the L2-distance is minimized and the derived weight vector satisfies its constraints. To deal with invalid weight vectors, we add a penalty for an arrangement produced by GA if the associated weight vector is invalid.

### 6.2.1 Penalty Function

The penalty of $w_i^*$ denoted by $g_i(w_i^*)$ is defined in the following:

$$g_i(w_i^*) = [max(0, -1 * w_i^* - f_{min})]^b$$

Where $f_{min}$ represents the minimum feasible value of $w_i$. Since $\forall i,\ w_i \geq 0$, the minimum feasible value of $w_i$ is set to 0. $b$ is a control parameter, which reprensets the rate of penalty increment as $w_i$ turns negative. This equation shows that if $w_i$ is far away from 0 in the negative direction, the penalty of $w_i^*$ increases. If $w_i \geq 0$, there is no penalty.

The normalized penalty of $w_i^*$, denoted by $G_i(w_i^*)$ is defined as follow:

$$G_i(w_i^*) = \begin{cases} dp, & \frac{g_i(w_i^*)}{m} > 1.0 \\ \frac{g_i(w_i^*)}{m}, & otherwise \end{cases}$$

Where "dp" is a sizeable constant number, represents the "death penalty." If any arrangement experiences the death penalty on one or more $w_i^*$, It is most likely to be phased out in the evolvement. In this way, GA is concentrated on searching in the sub-domain space ranges within $[-m, m]$.

Finally, the penalty, denoted by $I_N(w^*)$ is defined as the average normalized penalty of $w^*$. Formally,

$$I_N(w^*) = \frac{1}{m} * \sum(G_i(w_i^*))$$

### 6.2.2 Distance to Optimal Solution

The distance to optimal solution incorporates the penalty function into consideration. Therefore, the L2-distance $d$ should be normalized in the range of $[0, 1]$. Let $d_N(w*)$ denotes the normalized L2-distance:

$$d_N(w^*) = \begin{cases} \frac{d(w*)-d(w*)_{min}}{d(w*)_{max}-d(w*)_{min}}, d(w*) < d(w*)_{max} \\ 1, otherwise \end{cases}$$

Where $d(w^*)_{min}$ represents the minimum value of $d(w^*)$, its value is setup to 0. If $w$ satisfies all of the constraints, $d(w^*)$ equals to 0. $d(w^*)_{max}$ represents the maximum value of $d(w^*)$, its value is manually set up to $m^2$. Same as the penalty function, $d(w^*)_{max}$ limits the search-range to $[0, m^2]$. An arrangement which $d(w^*)$ and $I_N(w^*)$ equal 0, the corresponding input distribution is optimal. Thus, the distance to optimal solution is defined as the L2-distance between the current solution $(d_N(w*), I_N(w*))$ and the optimal solution (0, 0).

$$f(w*) = \sqrt{(d_N(w*)^2 + (I_N(w^*))^2}$$

| Arrangement: A | | | |
|---|---|---|---|
| C.E | $S_1$ | $S_2$ | $S_3$ |
| Path #1 | 1 | 0 | 0 |
| Path #2 | 0 | 1 | 0 |
| Path #3 | 0 | 0 | 1 |

| Arrangement: B | | | |
|---|---|---|---|
| C.E | $S_1$ | $S_2$ | $S_3$ |
| Path #1 | 0.2 | 0.4 | 0.5 |
| Path #2 | 0.6 | 0.3 | 0.3 |
| Path #3 | 0.2 | 0.3 | 0.2 |

| Arrangement: C | | | |
|---|---|---|---|
| C.E | $S_1$ | $S_2$ | $S_3$ |
| Path #1 | 0.5 | 0.7 | 0.8 |
| Path #2 | 0.2 | 0.1 | 0.1 |
| Path #3 | 0.3 | 0.2 | 0.1 |

| Arrangement: D | | | |
|---|---|---|---|
| C.E | $S_1$ | $S_2$ | $S_3$ |
| Path #1 | 0.8 | 0.2 | 0.2 |
| Path #2 | 0.2 | 0.8 | 0.2 |
| Path #3 | 0.2 | 0.2 | 0.8 |

Table 6.1: Examples of Arrangements

| Fitness | | | | |
|---|---|---|---|---|
| Arr. | $d_N$ | $I_N$ | $f$ | Rank |
| A | 0 | 0 | 0 | 1 |
| B | $2 \times 10^{-3}$ | 0 | $2.1 \times 10^{-2}$ | 2 |
| C | 0 | dp | $\frac{dp}{m}$ | 4 |
| D | $4.5 \times 10^{-2}$ | $1.67 \times 10^{-1}$ | $5 \times 10^{-2}$ | 3 |

Table 6.2: Fitness And Rankings For Arrangements A,B,C,D

As an example, Table 6.1 shows the triggering probabilities from 4 arrangements for an SUT with 3 independent paths, and $\bar{\mathbf{tri}}$ is set to $[0.3, 0.6, 0.1]$. Table 6.2 shows $d_N$,$I_N$,$f$ and the ranking of the 4 corresponding arrangements. Arr.A is an ideal input distribution where $S_{c_i} = S_i$, its rank is 1. Arr.B has feasible weights, and the overall fitness is grater than Arr.D, it is ranked 2. Arr.C gives the optimal $d_N$, but its $w_2 = -10$. Therefore, its $I_N = dp$. Suppose $dp = 1$, the total fitness equals to 0.33, it is ranked 4.

### 6.2.3 Automated Adaptive Search

Searching for an arrangement that satisfies the weight vector constraints is difficult since the feasible weight vector domain space is quite small ([0, 1]). The weight vector may remain invalid when GA terminates. It is necessary to explore the solution space more strategically. Therefore, we develop a strategy to break down the overall optimization process into several levels.

- *Level-1:* The population contains no feasible solution. It usually occurs when solutions in the population are far away from the feasible solution region. Hence, GA's objective in Level one is to minimize the penalty of weight vectors.

- *Level-2:* The population contains a proportion $1 - r$ of infeasible solutions. $r$ is a constant value ranges within [0, 1]. Although feasible solutions have a higher fitness value than infeasible solutions, the infeasible solution is still valuable to the search algorithm. For instance, infeasible solution "$x$" might be more close to the optimal than the feasible solution "$x2$". Since the feasible solution does not give GA more information than the infeasible solution towards the optimal. Hence, GA's objective in Level-2 varies according to $r$. If $r$ is small, the search is guided towards the direction of minimizing the penalty. If $r$ becomes large, the search is directed towards minimizing the distance to optimal.

- *Level-3:* The population converges into the feasible solution region, the only objective of GA is to search for the minimum distance-to-optimal solution.

To apply the above strategy into search, we adopt the Self-Adjust Penalty Function(SAPF) method [31]. The SAPF is a generic fitness function, which adjusts penalties based on the distribution of the population. If the population has too many infeasible solutions, the GA emphasizes searching for feasible solutions. On the other hand, if the population has many feasible solutions, the GA emphasizes the distance to an optimal solution. The major advantage of the SAPF method is that it saves much work tunning the penalty coefficient.

Evaluating the fitness of an arrangement requires the following steps: First, with a given estimated triggering probability vectors $\hat{\mathbf{a}}_\mathbf{i}$, we form the matrix $A$ and vector $\mathbf{b}$. Second, by using the Least Square Method, we solve for the weights vector $\mathbf{w}^*$ and the distance to optimal solution $d(\mathbf{w}^*)$. Third, we calculate the overall fitness, the normalized distance to optimal, and the normalized penalty. Finally, we apply the SAPF method to fitness calculation. Specifically, let $\mathcal{F}$ be the set of feasible solutions in the population, the SAPF method defines the fitness function as follows:

$$\phi(\mathbf{w}) = dist(\mathbf{w}) + (1-r)X(\mathbf{w}) + rY(\mathbf{w})$$

Where:

$$dist(\mathbf{w}) = \begin{cases} I_N(\mathbf{w}), & if\ \mathcal{F} = \emptyset \\ \\ f(\mathbf{w}), & if\ \mathcal{F} \neq \emptyset \end{cases} \tag{6.2}$$

And:

$$X(\mathbf{w}) \begin{cases} 0, & if\ \mathcal{F} = \emptyset \\ \\ I_N(\mathbf{w}), & if\ \mathcal{F} \neq \emptyset \end{cases} \tag{6.3}$$

$$Y(\mathbf{w})\begin{cases} 0, & if\ x \in \mathcal{F} \\ \\ d_N(\mathbf{w}), & if\ x \notin \mathcal{F} \end{cases} \tag{6.4}$$

Where $r \in [0,1]$ represents the proportion of feasible solutions in the population. It is clear that if there is no feasible solution in the pool, the fitness only depends on the penalty. On the other hand, if there are feasible solutions in the pool, the proportion $r$ guides the search into two directions: 1. If $r$ closes to 1, $Y(\mathbf{w})$, represented as the cost-based penalty, takes the major effect for fitness evaluation. 2. If $r$ closes to 0, $X(\mathbf{w})$, represented as the constraint violation penalty, takes the major effect for fitness evaluation. The SAPF method suits the SST problem well. In the problem, the initial population might start from many infeasible solutions. GA mostly searches towards the direction to the feasible solution region. As iteration increases, the infeasible solution counts become smaller, and GA becomes more focused on minimizing the distance to optimal solutions.

### 6.2.4 The Complete Work-flow

**Algorithm 3** GA-LS Algorithm

```
1: procedure GALS($D$,$n_p$,$n_c$,$n_s$,$\bar{\mathbf{tri}}$)
2:     $s_0 \longleftarrow GenerateRandomSolution(n_p, n_c)$
3:     $triList \longleftarrow Sampling(D, n_s, n_c)$
4:     $g = 0$
5:     $\mathbf{w}List = \emptyset$
6:     while !SC do
7:         $fitnesses = FitnessEva(s_g, triList, \mathbf{w}List, \bar{\mathbf{tri}})$
8:         $parents_g = RouletteWheelSel(s_g, fitnesses)$
9:         $s_{g+1} = Reproduction(parents_g)$
10:        $g = g + 1$
11:    end while
12:    $s_{best} = s_g[fitnesses.Index(min(fitness))]$
13:    $\mathbf{w}_{best} = \mathbf{w}List[fitnesses.Index(min(fitness))]$
14:    if $s_{best}$.contains$(x < 0) = true$ then
15:        $\mathbf{w_{best}} = GoldfarbIdnaniSolver(s_{best})$
16:    end if
17:    return $(s_{best}, \mathbf{w}_{best})$
18: end procedure
19:
20: procedure FitnessEvaluation($s_g$,$triList$,$\mathbf{w}List$,$\bar{\mathbf{tri}}$)
21:    $fitnesses = \emptyset$
22:    for i=1 to $n_p$ do
23:        $A, \mathbf{b} = DeriveMatrixAandVectb(s_{g_i}, triList)$
24:        $\mathbf{v} = \mathbf{b} - \bar{\mathbf{tri}}$
25:        $d(w^*), \mathbf{w}^* = LeastSquareMethod(A, v)$
26:        $\mathbf{w}List_i = \mathbf{w}^*$
27:        $f, d_N, I_N = OverallFitnessEva(d(w^*), \mathbf{w}^*)$
28:        $\phi = AdjustFitness(f, d_N, I_N, \mathbf{w}^*)$
29:        $fitnesses_i = \phi$
30:    end for
31:    return$fitnesses$
32: end procedure
```

The algorithm starts by splitting the input domain space into $\Delta$ set. For each set, we take an equal number of n samples uniformly, followed by running them against the SUT to estimate the triggering probabilities. Next, GA is started to allocate elements from set $\Delta$ into set S. In each iteration, new arrangements are produced by the two-point crossover and the multi-point uniform mutation operators. after a new arrangement is produced, the least square method calculates the optimal probability assignments $w^*$ that minimizes the distance to optimal $d(w^*)$. An individual's fitness depends on the following factors: the normalized distance to optimal, the normalized infeasibility, and the feasible solution counts in the population. After calculating the fitness for each solution in the population, roulette wheel selection with elitism is performed to compose a new population. The following conditions terminate the search algorithm: First, the search algorithm finds the optimal solution. Second, the fitness value has not improved since the last 200 generations. Third, the generation reaches the pre-fined maximum. If the second or third conditions terminate the search algorithm, we take the best arrangement found in the last generation and run a constrained quadratic solver to find the near-optimal probability assignments.

## MULTI-OBJECTIVE SEARCH WITH NONPREEMPTIVE GOAL PROGRAMMING

Given a matrix $A$, the problem of maximizing $\mathbf{P_c}$ w.r.t $\mathbf{w}$ is formally defined as follows:

$$\begin{aligned}
&max && \mathbf{P_c} \\
&s.t && \textstyle\sum_{i=1}^{k} w_i = 1 \\
& && w_i \geq 0, \quad i \in \{1, ..., k\} \\
&where && P_{c_1} = a_{11}w_1 + a_{12}w_2 + \cdots + a_{1k}w_k \\
& && P_{c_2} = a_{21}w_1 + a_{22}w_2 + \cdots + a_{2k}w_k \\
& && \qquad\quad \vdots \\
& && P_{c_m} = a_{m1}w_1 + a_{m2}w_2 + \cdots + a_{mk}w_k
\end{aligned} \tag{6.5}$$

Since each row is a convex function, it is obvious to see that for any $P_{c_i} \in \mathbf{P_c}$, its maximum value:

$$\begin{aligned}
& && P^*_{c_i} = a_{ij} \\
&with && \\
& && w_j = 1.0 \\
&where && \\
& && j = Index(\max(\mathbf{a_i})),\ \mathbf{a_i} : \{a_{i1}, \ldots, a_{ik}\}
\end{aligned} \tag{6.6}$$

Therefore, each column vector in the matrix $A$ is an optimal solution in the *Pareto Optimal Set*. However, these are not the preferred solutions. For instance, if we choose any column vector as the final solution for a $3 \times 3$ identity matrix, 66.7% of cover elements have 0 chance of being triggered. If a bug is in the 66.7% input sub-domain space, the bug will never expose. To avoid such problems, we adopt the Nonpreemptive Goal Programming method.

### 6.3.1 Nonpreemptive Goal Programming

The Nonpreemptive Goal Programming method minimizes the deviations from established goals under a given set of constraints rather than maximizing the objective function directly as Linear Programming. The value of deviation variables represents the distance of the objective function's actual and target value. Each deviation variable is associated with a priority factor that represents the relative importance of each objective. NGP tries to minimize the sum of the products of each deviation variable and the associated priority factor such that each objective value can be maximized.

Specifically, let $\{\mathcal{S}_1, \ldots, \mathcal{S}_m\}$ be the set of deviations and $\{\mathcal{I}_1, \ldots, \mathcal{I}_m\}$ be the set of priority factors for each triggering probability respectively. The targeted value for each cover element is set to its expected triggering probability respectively. In this case, minimizing $\mathcal{S}_i$ is equivalent to increase the actual triggering probability so that the gap between $\mathbf{P}_{c_i}$ and the expected $\bar{tri}_i$ can be minimized.

Then, the original optimization problem can be written in the NGP form as follows:

$$\begin{aligned}
&min && \mathcal{I}_1\mathcal{S}_1 + \cdots + \mathcal{I}_m\mathcal{S}_m \\
\\
&subject\ to && \\
& && a_{11}w_1 + a_{12}w_2 + \cdots + a_{1k}w_k + \mathcal{S}_1 \geq \bar{tri}_1 \\
& && a_{21}w_1 + a_{22}w_2 + \cdots + a_{2k}w_k + \mathcal{S}_2 \geq \bar{tri}_2 \\
& && \qquad\vdots \\
& && a_{m1}w_1 + a_{m2}w_2 + \cdots + a_{mk}w_k + \mathcal{S}_m \geq \bar{tri}_m \\
\\
& && w_1 + \cdots + w_k = 1 \\
& && w_i \geq 0, \quad i \in \{1, ..., k\}
\end{aligned} \tag{4}$$

## 6.3.2 Priority Factor

In the SST problem, a solution in the *Pareto Optimal Set* is more preferred than another if its triggering probability low bound, derived from $\min(\mathbf{P}_c)$ is greater than the other one.

Suppose, given a matrix $A$, the maximum triggering probability for cover element $c_i$ is the smallest among all of the cover elements. To minimize the objective function defined in Equation 4, decreasing the value of deviation $\mathcal{S}_i$ doesn't have much influence on the function output. In other words, the triggering probability for $c_i$ is less important than other cover elements for the optimization objective. In often, the solution $tri_i$ after optimization is close to 0, and therefore, the result $\mathbf{P}_c$ is not a preferred solution. To overcome this problem, we set up a linear relation between the maximum triggering probability and the priority factor for each cover element such that a cover element with the maximum triggering probability $P^*_{c_i}$ below a threshold $\delta$ needs to be prioritized. Also, the more distance from $\delta$, the higher the priority value should be. Thus, we create a function of $\mathcal{I}$ in terms of $P^*_{c_i}$ and a pre-defined constant $\delta$. The function is formally defined as follows:

$$\mathcal{I}_i = \begin{cases} \frac{2}{\delta} * (\delta - P^*_{c_i}) + 1, \ P^*_{c_i} < \delta \\ \\ 1, otherwise \end{cases}$$

Where $\delta = \frac{1}{|\mathbf{c}|}$. When $P^*_{c_i} = 0$, the priority factor is at the maximum value, which is 3. If the maximum triggering probability reaches or beyond $\delta$, the priority factor is at the minimum 1.0. On the other hand, if the maximum triggering probability is below $\delta$, the priority value is linearly increased as the maximum triggering probability moves towards 0.

### 6.3.3 Example: Benefit of using priority factor

To illustrate the effect of the priority factor function, we give an example of $A_{ex}$ matrix, shown in the next and compare two sets of priority values, one with equal priority, and the other with values generated by the function.

$$A_{ex} = \begin{bmatrix} 0.6 & 0.2 \\ 0.0 & 0.27 \\ 0.30 & 0.01 \end{bmatrix}$$

In $A_{ex}$, $P^*_{c_2}$ and $P^*_{c_3}$ are below $\delta$ value. After apply Simplex Method with equal priority values, the solution $S_1$ gives the actual triggering probabilities $\{0.6, 0.0, 0.3\}$ with weights $\{1.0, 0.0\}$. On the other hand, with the use of the priority factor function, which sets the priority values to $\{1, 1.38, 1.2\}$, the solution $S_2$ gives the actual triggering probabilities $\{0.33, 0.18, 0.11\}$ with weights $\{0.33, 0.66\}$. $S_2$ is more preferred than $S_1$ since $0.11 > 0.0$.

### 6.3.4 Optimizing the $\mathcal{A}$ Matrix

This section first shows the benefit of using full column rank $A$ matrix to an SST problem. In what follows, we provide an algorithm that, given an input arrangement, the $A$ matrix can be pruned to be linearly independent.

#### Sample Diversity

In the SST problem, the diversity of sampled inputs can affect the quality of the test set. A higher diversity usually leads to a better fault detecting ability [**?** ]. However, it is often the case that probability low bound and diversity contradict each other. For instance, if only one input $x \in S$ triggers $c_i$, and if the probability low bound is 0.5, the input $x$ occupies 50% of spaces in the test set. Of course, nearly 50% of spaces are wasted.

In the process of searching for an optimal arrangement that maximizing the probability low bound, the loss of diversity is not inevitable. However, it is better to preserve diversity as much as possible. In our approach, we create a so-called *Bin Prune Process*, which re-arranges a given arrangement such that the corresponding matrix $A$, also named as the *primitive* matrix $A$ becomes a full column rank matrix $A^*$. And the linear independence property of column vectors provides a certain level of diversity.

## Benefit of Linearly Independence

A direct measure of diversity is the distribution variance. For the proposed weighted uniform input distribution model presented in Equation (2.1), the total variance equals to the following equations:

$$\begin{cases} \sigma^2 = \sigma_1^2 w_1^2 + \sigma_2^2 w_2^2 + \cdots + \sigma_m^2 w_m^2 \\ \sigma_i = \frac{k_i^2 - 1}{12} \end{cases} \tag{10}$$

Where $k_i$ denotes the size of the input set $S_i$.

Without loss of generality, for any arrangement, we split the set of column vectors $T$ into two sets. $L$ represents the linearly independent vectors. $N$ represents $T \setminus L$. And the Equation 10 is re-arranged according $L$ and $N$ as follows:

$$\sigma^2 = [\sigma_{L_1}^2 w_{L_1}^2 + \sigma_{L_2}^2 w_{L_2}^2 + \dots] + [\sigma_{N_1}^2 w_{N_1} + \sigma_{N_2}^2 w_{N_2} + \dots] \tag{11}$$

According to the Theorem of Linear Programming, for any basic feasible solution $\mathbf{w}$, $\mathbf{w}_N = 0$. Thus, the total variance only equals to the left portion of the Equation 11. If a matrix $A$ is not full column rank, at least one inputs reside in the bins that belong to the right portion of the Equation 11, the total variance is shrunken by these slipping inputs, since they have 0 probability being sampled. Hence, To have a full variance matrix $A$, we create the *Bin Prune Process.*

## Bin Prune Process

The *Bin Prune Process*, which is based on the Gram–Schmidt Process, which produces matrix $A^*$ from a given primitive matrix $A$. The Gram-Schmidt Process is used to generate an orthogonal set $U = \{u_1, \ldots, u_k\}$ for a finite, linearly independent set $S = \{v_1, \ldots, v_k\}$. It can also be adopted to determine whether a vector $v$ is a linear combination of the existing linearly independent set $S$. If it is, the vector $v$ can be decomposed by the orthogonal set $U$. If it is not, a new orthogonal basis vector $u_v$ is produced such that the vector $v$ can be decomposed by the union of $U$ and $u_v$

The overall process is mainly a recursion on the following equation, which decomposes vector $v$ into two orthogonal basis vectors $u_i$ and $o$.

$$o = v - \frac{\langle v, u_i \rangle}{\langle u_i, u_i \rangle} u_i \tag{4}$$

Suppose, we want to verify vector $v_{k+1}$. The process started from decomposing $v_{k+1}$ into vector $u_1$ and vector $o_1$. In the next iterations, it further decompose the remainder vector $o_{i-1}$ into $u_i$ and $o_i$. If vector $o_{i-1}$ is in parallel with $u_i$, the process stops. In this situation, $v$ is linearly dependent of $S$.

Suppose vector $v_{k+1}$ is a linear combination of $v_m, v_n \in S,\ n > m > 1$. Since $v_{k+1}$ is the linear combination of $v_m$ and $v_n$, $v_{k+1}$ is also the linear combination of $u_m$ and $u_n$. Since $u_m$ and $u_n$ are perpendicular to all the other vectors in the orthogonal set, the following equation is true:

$$\langle v_{k+1}, u_i \rangle = 0,\ \forall i \in \{1, \ldots, k\} \setminus \{m, n\} \tag{5}$$

**Algorithm 4** BinPruneProcess

```
procedure BinPruneProcess
    input: $\mathbf{a}$ - arrangement of inputs
           $A$ - primitive matrix $A$

    output: $A^*$ - pruned matrix $A$
            $a^+$ - pruned arrangement

    $\mathbf{a}^+, A^*, U \longleftarrow \mathbf{a}_1, A_1, A_1$

    for $i = 2$ to $n_b$ do
        $li, u = ProduceOrthBasisVector(A_i, U)$
        if $li = true$ then
            $v^*, u^*, a^* = A_i, u, \mathbf{a}_i$
        else
            $U = U \setminus U_{len-1}$
            $v^* = \frac{1}{2} * (A^*_{len-1} + A_i)$
            $u^* = ProduceOrthBasisVector(v, U)$
            $a^* = \mathbf{a}_i \cup \mathbf{a}^+_{len-1}$
        end if
        $\mathbf{a}^+, U, A^* \longleftarrow a^*, u^*, v^*$
    end for
    return $A^*, \mathbf{a}^+$
end procedure

procedure ProduceOrthBasisVector
    input: $v$ - a vector
           $U$ - orthogonal set

    output: $li$ - a boolen variable. Ture if $v$ is L.I. of $U$
            $u$ - orthogonal basis vector
    $v^t = v$
    $li = false$
    for $i = 1$ to $n_o$ do
        if $\langle |e_{v^t}|, |e_{u_i}| \rangle = 1.0$ then
            $li = true$
            break
        else
            $v^t = v^t - \frac{\langle v^t, u_i \rangle}{\langle u_i, u_i \rangle} u_i$
        end if
    end for
    $u = v^t$
    return $li, u$
end procedure
```

At the beginning, the first orthogonal basis vector $u_1$ and the vector $v_{k+1}$ is selected as inputs to Equation 4. According to Equation 5, the function output is $o = v_{k+1}$. At the following $i < m$ iterations, the function continuously outputs $v_{k+1}$. At iteration $m$, since $o_m \perp u_m$ and $u_n \perp u_m$, $o_m$ and $u_n$ must be in parallel. At the next $m < i < n$ iterations, according to Equation 5, the function output remains $o_m$. At iteration $n$, $o_{n-1} = o_m$. It is clear that $o_{n-1}$ and $u_n$ are in parallel. Hence, if a vector is a linear combination of $S$, the process must reach to an iteration $i$ such that $o_{i-1}$ and $u_i$ are in parallel.

If $v_{k+1}$ is linearly independent of $S$, $v_{k+1}$ is also linearly independent of $U$. In this case, all of the orthogonal basis vectors can be exercised.

The main steps to derive matrix $A^*$ from the primitive matrix $A$ are described in Algorithm 4. In the beginning, the first column vector $A_1$ is added to the orthogonal set $U$ and the matrix $A^*$. Also, the first bin $\mathbf{a_1}$ is added to the new arrangement $\mathbf{a}^+$. Then for each column vector $A_i$, $i \geq 2$, the algorithm performs the following process: First, it calls the function *ProduceOrthBasisVector*, which takes a column vector $A_i$ and the orthogonal set $U$ as inputs, and output a Boolean value $li$ to indicate whether $A_i$ is independent of $U$. It also outputs the remaining orthogonal basis vector $u$ if $A_i$ is independent of $U$. If $li$ is true, $\{A_i, u, \mathbf{a}_i\}$ is added to $\{A^*, U, \mathbf{a}^+\}$ respectively. If $li$ is false, the latest inserted vector $A^*_{len-1}$ is averaged over the column vector $A_i$. Then the orthogonal basis vector $U_{len-1}$ is updated by calling *ProduceOrthBasisVector*. On the bin side, the algorithm updates the latest inserted bin $\mathbf{a^+_{len-1}}$ by performing the union operation with the currently being exercised bin $\mathbf{a}_i$.

### 6.3.5 The GAGP algorithm

**Algorithm 5** GAGP Algorithm

```
procedure GAGP
    input: $S$ - input domain space
            $n_k$ - sample size
            $n_\delta$ - number of sub-input domain space
            $n_\mu$ - population size
            $\gamma_c$ - crossover rate
            $\gamma_m$ - mutation rate
            $\overline{\mathbf{tri}}$ - expected triggering probabilities

    output: $s_t^*$ - the best arrangement
            $\mathbf{w}^*$ - the optimal weights associated with $s_t^*$

    $s_0 \longleftarrow GenerateInitialSolutionPool(n_\mu)$
    $\widehat{\mathbf{tri}} \longleftarrow SRSampling(n_k, n_\delta)$
    $g = 0$
    $\mathbf{w}^* = \emptyset$
    while !SC do
        $\mathbf{f}, \mathbf{w}, s_g^+ = Evaluation(s_g, \overline{\mathbf{tri}}, \widehat{\mathbf{tri}})$
        $s_g^*, \mathbf{w}^* = BestIndividual(s_g^+, \mathbf{w})$
        $parents = Selection(s_g^+, \mathbf{f})$
        $newPool = Reproduction(parents, \gamma_c, \gamma_m)$
        $g{+}{+}$
        $s_g = newPool$
    end while
    return $(s_g^*, \mathbf{w}^*)$
end procedure

procedure Evaluation
    input: $s$ - solution pool
            $\widehat{\mathbf{tri}}$ - estimated triggering probabilities
            $\overline{\mathbf{tri}}$ - expected triggering probabilities

    output: $\mathbf{f}$ - individual fitness
            $\mathbf{w}$ - optimal weights
            $s^+$ - light weight individuals

    $\mathbf{f}, \mathbf{w}, \mathbf{s}^+ \longleftarrow \emptyset$
    for i=1 to $n_\mu$ do
        $A = ProduceAMatrix(\widehat{\mathbf{tri}}, s^i)$
        $A^*, s^{i+} = BinPruneProcess(A, s^i)$
        $\mathbf{P}_c^i, \mathbf{w}^i = GoalProgramming(A^*, s^{i+}, \overline{\mathbf{tri}})$
        $f_{lb}^i = ProbabilityLowBound(\mathbf{P}_c^i)$
        $\mathbf{f}, \mathbf{w}, \mathbf{s}^+ \longleftarrow f_{lb}^i, \mathbf{w}^i, s^{i+}$
    end for
    return $(\mathbf{f}, \mathbf{w}, s^+)$
end procedure
```

The GAGP algorithm to optimize an input distribution is presented in Algorithm 5. The process starts from randomly generating a pool of initial solutions $s_0$. Next, the *SRSampling* function performs the Stratified Random Sampling over the input domain space to estimate the triggering probabilities $\widehat{\mathbf{tri}}$ w.r.t $\delta$ set. Next, GA starts to evolve arrangements. In the *Selection* function, the *rolette-wheel-selection* method is applied. In the *Reproduction* function, the *two-point-crossover* operator and the *uniform-mutation* operator are performed with a pre-defined rate: $\gamma_c, \gamma_m$. Elitism is applied to GA to stabilize the search process. The evolution stops when either of the two conditions $SC$ are satisfied:

1. The number of iterations reaches to $\tau_{max}$
2. The maximum fitness in the pool doesn't vary within 5% standard deviation over the last 30 generations.

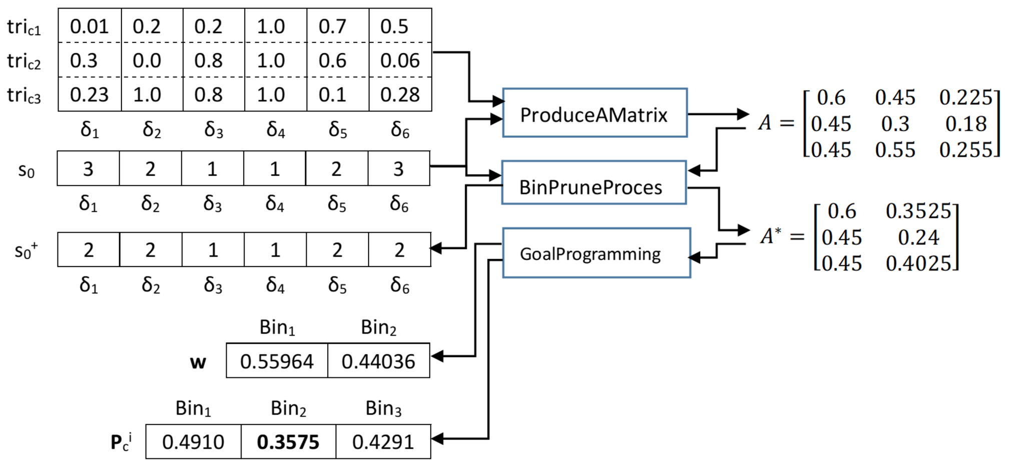
Specifically, the solution encoding and the *Evaluation* function is described as follows:

## Encoding

A chromosome, represented as an arrangement, is encoded as an array of integers. Each position of the array represents an element in the $\delta$ set. The length of the array equals the size of the $\delta$ set. Each integer represents a selection of bins.

## Evaluation

In the *Evaluation* function, an arrangement $s^i$ undergoes four processes. First, it is used to create the corresponding primitive $A$ matrix by the function *ProduceA-Matrix*. Second, it is refined by the *BinPruneProcess*. With the generated matrix $A^*$ and the refined arrangement $s^{i+}$, the function *GoalProgramming* produces the weights vector $\mathbf{w}$ and the estimated triggering probability set $\mathbf{P}^i_c$. At the end, the fitness of the refined arrangement is the probability low bound of the triggering probability set.



tri_c1 0.01 0.2 0.2 1.0 0.7 0.5
tri_c2 0.3 0.0 0.8 1.0 0.6 0.06
tri_c3 0.23 1.0 0.8 1.0 0.1 0.28
δ1 δ2 δ3 δ4 δ5 δ6
s0 3 2 1 1 2 3
s0+ 2 2 1 1 2 2
ProduceAMatrix
BinPruneProces
GoalProgramming
A = [0.6 0.45 0.225; 0.45 0.3 0.18; 0.45 0.55 0.255]
A* = [0.6 0.3525; 0.45 0.24; 0.45 0.4025]
w Bin1 0.55964 Bin2 0.44036
Pc^i Bin1 0.4910 Bin2 0.3575 Bin3 0.4291

Figure 6.2: An example work-flow of a solution under evaluation

### 6.3.6 Example: Bin Prune Process

The example SUT has 3 branch cover elements. Its input domain space is divided into 6 subsets. The overall data and the workflow is shown in Figure 6.2. A child solution $s_0$, which has three bins is sent to the function *ProduceAMatrix* to create its $A$ matrix. It is observed from Figure 6.2 that the column vector $A[3]$ is a linear combination of $A[1]$ and $A[2]$. Therefore, after the *BinPruneProcess*, the new solution $s_0^*$ whose positions hold the index of the third bin in the solution $s_0$ are replaced by the index of the second bin, and the matrix $A^*$ only has two columns. Finally, with the matrix $A^*$ as the input, the function *GoalProgramming* produces the optimal weight vector $\mathbf{w}$, the triggering probabilities $\mathbf{P}_c^i$. The fitness of the new solution $s_0^*$ is the minimum value in the vector $\mathbf{P}_c^i$, which is 0.3575.

# Chapter 7

# RELATED WORKS

## STATISTICAL STRUCTURAL TESTING

The traditional coverage-oriented test data generation has been widely studied for decades. In those studies, people believe that a test set that achieves a higher coverage provides a more thorough test indicating a stronger fault-detecting ability [32]. However, Tasiran pointed out that the through tests may not provide a high fault-detecting ability [33]. Also, even a coverage criterion subsumes another, the test data set that satisfies the first criterion does not necessarily prove the stronger fault-detecting ability. The lack of randomness using the traditional method is one of the reasons causing low fault-detecting rate. Since the coverage criteria require a fixed number of tests for each cover element, there is no chance that a particular cover element can be triggered multiple times than another. However, the chances are beneficial for detecting faults. In the early work, Duran and Ntafos [34] performed the cost-effective analysis for random testing. They showed that the random testing demonstrates a higher fault detecting ability over branch testing for some fault programs that have critical errors that can be discovered with a low failure rate. To combine the randomness and the traditional coverage adequacy into test data generation, Thevenod-Fosse created a new method, called Statistical Structural Testing (SST). In SST, test inputs are sampled from a probability distribution over the input domain space. The distribution guarantees that the sampled test inputs have probabilities at least greater than a pre-defined value to trigger each cover element (a.k.a, the triggering probability lower bound). They compared the fault-detecting power from the three approaches: deterministic, random, and SST by using mutation testing technique [35]. The experiment results demonstrate that the test set generated by SST is superior efficacy in detecting software fault. Constructing an optimal input distribution that satisfies the probabilistic coverage is not a trivial work. A tester needs to know the knowledge of the sub-input domain space associated with each cover element. Then he needs to assign proper probabilities to each sub-input domain space to create an optimal input distribution. However, as the computing power increases dramatically in recent years, the Search-Based Software Testing (SBST) framework has gained much attention. SBST refers to a software testing methodology that automates the test data generation process using intelligent search algorithms. It is often a dynamic testing process, meaning that the test set is refined during the SUT's run-time. A typical contribution made by Tracey, al.[36] used G.A to build the test set against the branch coverage criteria. Up to now, there are plenty of Meta-heuristic algorithms dedicated to generating test input set [37; 38]. The SBST framework for SST problems is firstly studied by Poulding and Clark. They modeled the input distribution as a Bayesian Network, with nodes represented as inputs, the values in each node defined as a collection of sub-input domain spaces. Their objective is to optimize the Bayesian network's parameters such that the sampled inputs achieve a probability lower-bound of triggering each branch. They used the hill-climbing as the search algorithm. Their experiment results demonstrated the practicality of applying the SBST framework for producing an optimal input distribution. However, their experiment results also show that efficiency is still a crucial issue. Based on their research, we analyze the problem that causes the low-efficiency issue and proposed the new criterion p-L1-Max.

## FUZZ TESTING TECHNIQUES

Fuzz testing refers to a testing strategy that randomizes test inputs in test data generation. How to randomize test inputs to make efficient test is a hot research area. In this chapter, we give a brief overview of the state-of-art fuzzers.

**IoT protocols fuzzing:** An IoT system can consist of a large number of devices. Those devices are all connected and communicated by IoT protocols. Detecting vulnerabilities in the protocols have a significant influence on web security and privacy. Bernhard in [39] combines automata learning and fuzz testing techniques for testing the MQTT protocols. SungJin Kim, et al. proposed a smart seed selection strategy to generate seeds efficiently for fuzzing IoT protocols [40]. Fw-fuzz [41] is proposed to detect vulnerabilities of network protocols on firmware. SPFuzz [42] proposed a hierarchical scheduling framework for fuzzing stateful network protocols.

**Kernel Fuzzing** Due to the complexity of an operating system, testing kernels is complex. Up to now, there exist several techniques for kernel fuzzing. NTFUZZ [43] is designed to fuzz Windows operating system, which can automatically infer system call types with static binary analysis. Hydra [44] is an extensible fuzzing framework that provides building blocks for file system fuzzing.

**Grammar-based fuzzing** is a technique to leverage SUT's specifications to generate test inputs efficiently. This method benefits the most for test inputs that have a complex structure. Soyeon Park, et al in [45] invented the aspect-preserving mutation to detect bugs in javascript programs. The G-EvoSuite [46] combines search-based testing with grammar-based fuzzing for detecting highly structured input data.

**Differential fuzzing** uses different programs of the same functionality as cross-referencing oracles, comparing their outputs across many inputs: any discrepancy in the programs' behavior on the same input is marked as a potential bug. Differential fuzzing has been successfully applied for detecting side-channel vulnerabilities[47], vulnerabilities in C compilers[48] and deep learning systems[49].

# Chapter 8

# CONCLUSION

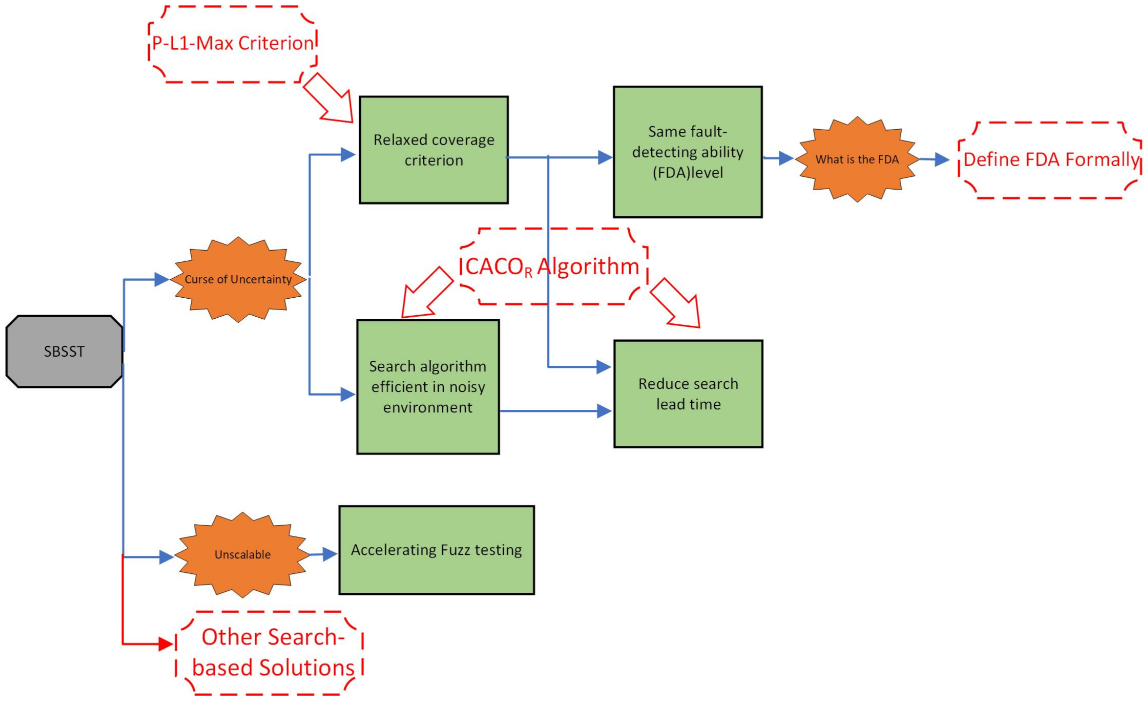

P-L1-Max Criterion
Relaxed coverage criterion
Same fault-detecting ability (FDA)level
What is the FDA
Define FDA Formally
CACO$_R$ Algorithm
Curse of Uncertainty
SBSST
Search algorithm efficient in noisy environment
Reduce search lead time
Unscalable
Accelerating Fuzz testing
Other Search-based Solutions

Figure 8.1: Research mindset

This dissertation addressed several weaknesses of the existing SBSST approach and proposed solutions to overcome the shortcomings. Figure 8.1 gives a mindset view of the research path. The SBSST has two key issues, the curse of uncertainty issue and the unsalable issue. To minimize the curse of uncertainty issue, we proposed two solutions. We proposed a relaxed coverage criterion called P-L1-Max, which provides input distribution the same level of fault-detecting ability while requires less search time. We also proposed the $CACO_R$ algorithm, which shows the ability to search in a noisy environment. Additionally, we formally define the fault-detecting ability with respect to input distributions. To let the BSST's testing method be applicable for real-world programs, we proposed a fuzzing tool called SBSFuzz.

The main contributions are concluded as follows:

**p-L1-Max** Based on the observation, a strong statistical structural coverage criterion results in a high impact of the noisy fitness estimation. Hence, we proposed the p-L1-Max criterion, which can significantly reduce the search time without the loss of substantial fault-detecting power.

**CACO**$_R$ We investigated search algorithms that resist noisy fitness estimation. Based on the existing Ant Colony Optimization, we developed a constrained ACO algorithm $CACO_R$ that is dedicated to the SST problem. Experimental studies demonstrate the excellent search performance of the $CACO_R$ algorithm and the high-grade fault-detecting ability of the input distributions produced by the algorithm.

**SBSFuzz** We developed a test input generator called SBSFuzz with the search-based SST technique for AFL. The test input generator works as complementary to mutation-based fuzzing when AFL is stuck in discovering new program paths. In the test input generator, we maintain a trace graph and a set of optimized probability distributions. To efficiently discovering new paths, we prioritize test inputs in terms of the exercised program traces. The input-sensitivity prioritization allows test inputs in the scanty input subdomain spaces to be sampled with priority. The path-depth prioritization allows test inputs that execute paths with deeper depth to be sampled with priority. The sampled test input is then fed back to AFL for testing.

**Other SBSFuzz Techniques** We proposed a new SBST workflow that leverages the $\mathcal{A}$ matrix to generate test inputs. We modeled the optimization problem either as a multi-objective problem or the least square problem. The noisy fitness estimation cannot influence the search algorithm in the new workflow. Hence, a future research direction is to apply this method to fuzz testing.

# BIBLIOGRAPHY

[1] R. S. Pressman, *Software Engineering (3rd Ed.): A Practitioner's Approach*. USA: McGraw-Hill, Inc., 1992.

[2] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE Trans. Softw. Eng.*, vol. 10, p. 438–444, July 1984.

[3] J. A. Clark and S. Poulding, "Efficient software verification: Statistical testing using automated search," *IEEE Transactions on Software Engineering*, vol. 36, pp. 763–777, nov 2010.

[4] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 679–696, 2018.

[5] V. Chew, "Point estimation of the parameter of the binomial distribution," *The American Statistician*, vol. 25, no. 5, pp. 47–50, 1971.

[6] K. Socha and M. Dorigo, "Ant colony optimization for continuous domains," *European Journal of Operational Research*, vol. 185, no. 3, pp. 1155–1173, 2008.

[7] P. Puschner and R. Nossal, "Testing the results of static worst-case execution-time analysis," in *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, pp. 134–143, 1998.

[8] J. Wegener, H. Sthamer, B. F. Jones, and D. E. Eyres, "Testing real-time systems using genetic algorithms," *Software Quality Journal*, vol. 6, p. 127–135, Oct. 1997.

[9] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and Software Technology*, vol. 43, no. 14, pp. 841–854, 2001.

[10] P. Thevenod-Fosse, H. Waeselynck, and Y. Crouzet, "An experimental study on software structural testing: deterministic versus random input generation," in *[1991] Digest of Papers. Fault-Tolerant Computing: The Twenty-First International Symposium*, pp. 410–417, 1991.

[11] S. Poulding and J. A. Clark, "Efficient software verification: Statistical testing using automated search," *IEEE Transactions on Software Engineering*, vol. 36, no. 6, pp. 763–777, 2010.

[12] S. Wallis, "Binomial confidence intervals and contingency tests: Mathematical fundamentals and the evaluation of alternative methods," *Journal of Quantitative Linguistics*, vol. 20, no. 3, pp. 178–208, 2013.

[13] Y. K. Shestopaloff, *Sums of exponential functions and their new fundamental properties, with applications to natural phenomena.* AKVY Press, 2008.

[14] Y. Jia and M. Harman, "Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language," in *Testing: Academic Industrial Conference - Practice and Research Techniques (taic part 2008)*, pp. 94–98, 2008.

[15] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[16] F. E. Allen, "Control flow analysis," in *Proceedings of a Symposium on Compiler Optimization*, (New York, NY, USA), p. 1–19, Association for Computing Machinery, 1970.

[17] "Triangle, http://tracer.lcc.uma.es/problems/testing/index.html."

[18] "Nichneu, http://tracer.lcc.uma.es/problems/testing/index.html."

[19] A. J. S. C. e. a. R. Veerasamy, H. Rajak, "Validation of qsar models - strategies and importance," *International Journal of Drug Design and Discovery*, vol. 3, pp. 511–519, 2011.

[20] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios, "Application of genetic algorithms to software testing," in *Proceedings of the 5th International Conference on Software Engineering and Applications*, pp. 625–636, 1992.

[21] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in *Proceedings of the 2011 11th International Conference on Quality Software*, QSIC '11, (USA), p. 31–40, IEEE Computer Society, 2011.

[22] P. Thevenod-Fosse, H. Waeselynck, and Y. Crouzet, "An experimental study on software structural testing: deterministic versus random input generation," in *[1991] Digest of Papers. Fault-Tolerant Computing: The Twenty-First International Symposium*, pp. 410–417, 1991.

[23] T. Friedrich, T. Kötzing, M. S. Krejca, and A. M. Sutton, "Robustness of ant colony optimization to noise," in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, (New York, NY, USA), p. 17–24, Association for Computing Machinery, 2015.

[24] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria," in *Proceedings of 16th International Conference on Software Engineering*, pp. 191–200, 1994.

[25] S. Tsutsui, "Ant colony optimization for continuous domains with aggregation pheromone metaphor," 01 2004.

[26] F. O. de Franca, G. P. Coelho, F. J. Von Zuben, and R. R. d. F. Attux, "Multivariate ant colony optimization in continuous search spaces," in *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, GECCO '08, (New York, NY, USA), p. 9–16, Association for Computing Machinery, 2008.

[27] L. Martino and J. Míguez, "Generalized rejection sampling schemes and applications in signal processing," *Signal Processing*, vol. 90, p. 2981–2995, Nov 2010.

[28] E. Alba and J. F. Chicano, "Software testing with evolutionary strategies," in *Rapid Integration of Software Engineering Techniques* (N. Guelfi and A. Savidis, eds.), (Berlin, Heidelberg), pp. 50–65, Springer Berlin Heidelberg, 2006.

[29] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The mälardalen wcet benchmarks - past, present and future," in *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, July 2010.

[30] Y. Jia and M. Harman, "Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language," *Proc. Int. Conf. Testing: Academic and Industrial Conf. Practice and Research Techniques*, 08 2008.

[31] B. Tessema and G. Yen, "A self adaptive penalty function based algorithm for

constrained optimization," in *2006 IEEE International Conference on Evolutionary Computation*, pp. 246–253, 2006.

[32] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, p. 366–427, Dec. 1997.

[33] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs," *IEEE Design Test of Computers*, vol. 18, no. 4, pp. 36–45, 2001.

[34] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 438–444, 1984.

[35] W. Howden, "Weak mutation testing and completeness of test sets," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, pp. 371–379, 1982.

[36] N. Tracey, J. Clark, K. Mander, and J. McDermid, "An automated framework for structural test-data generation," in *Proceedings 13th IEEE International Conference on Automated Software Engineering (Cat. No.98EX239)*, pp. 285–288, 1998.

[37] J. H. Andrews, T. Menzies, and F. C. Li, "Genetic algorithms for randomized unit testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 1, pp. 80–94, 2011.

[38] A. A. L. de Oliveira, C. G. Camilo-Junior, and A. M. R. Vincenzi, "A coevolutionary algorithm to automatic test case selection and mutant in mutation testing," in *2013 IEEE Congress on Evolutionary Computation*, pp. 829–836, 2013.

[39] B. K. Aichernig, E. Muškardin, and A. Pferscher, "Learning-based fuzzing of iot message brokers," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pp. 47–58, 2021.

[40] "Smart seed selection-based effective black box fuzzing for iiot protocol," *J Supercomput*, p. 10140–10154, 2020.

[41] Z. Gao, W. Dong, R. Chang, and Y. Wang, "Fw-fuzz: A code coverage-guided fuzzing framework for network protocols on firmware," *Concurrency and Computation: Practice and Experience*, vol. n/a, no. n/a.

[42] C. Song, B. Yu, X. Zhou, and Q. Yang, "Spfuzz: A hierarchical scheduling framework for stateful network protocol fuzzing," *IEEE Access*, vol. 7, pp. 18490–18499, 2019.

[43] J. Choi, K. Kim, D. Lee, and S. Cha, "Ntfuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis," in *2021 2021 IEEE Symposium on Security and Privacy (SP)*, (Los Alamitos, CA, USA), pp. 677–693, IEEE Computer Society, may 2021.

[44] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim, "Finding bugs in file systems with an extensible fuzzing framework," *ACM Trans. Storage*, vol. 16, May 2020.

[45] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, "Fuzzing javascript engines with aspect-preserving mutation," in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1629–1642, 2020.

[46] M. Olsthoorn, A. van Deursen, and A. Panichella, "Generating highly-structured input data by combining search-based testing and grammar-based fuzzing," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, (New York, NY, USA), p. 1224–1228, Association for Computing Machinery, 2020.

[47] S. Nilizadeh, Y. Noller, and C. S. Păsăreanu, "Diffuzz: Differential fuzzing for

side-channel analysis," in *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, p. 176–187, IEEE Press, 2019.

[48] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "Nezha: Efficient domain-independent differential testing," in *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 615–632, 2017.

[49] J. Guo, Y. Zhao, H. Song, and Y. Jiang, "Coverage guided differential adversarial testing of deep learning systems," *IEEE Transactions on Network Science and Engineering*, vol. 8, no. 2, pp. 933–942, 2021.