RBF Neural Network-based Fault Localization

Technical Report UTDCS-20-10

Department of Computer Science

The University of Texas at Dallas

July 2010

W. Eric Wong, Vidroha Debroy, **Bhavani Thuraisingham and Richard Golden**

Technical Report UTDCS-20-10

RBF Neural Network-based Fault Localization

W. Eric Wong, Vidroha Debroy, Bhavani Thuraisingham
Department of Computer Science
University of Texas at Dallas

Richard Golden
School of Behavioral and Brain Sciences
University of Texas at Dallas

July 2010

Department of Computer Science

University of Texas at Dallas

Abstract

An RBF (radial basis function) neural network-based fault localization technique is proposed in this paper to assist programmers in locating bugs effectively.. We employ a three-layer feed-forward artificial neural network with radial basis functions as its hidden unit activation functions and a linear function as its output layer activation function. This neural network is trained to learn the relationship between the statement coverage information of a test case and its corresponding execution result, success or failure. The trained network is then given as input, a set of *virtual* test cases, each covering a single statement, and the output of the network, for each virtual test case, is considered to be the *suspiciousness* of the corresponding covered statement. A statement with a higher suspiciousness has a higher likelihood of containing a bug, and thus, statements can be ranked in descending order of their suspiciousness. The ranking can then be examined one by one, starting from the top, until a bug is located. Six case studies on different programs (both small and large in size) were conducted, with each faulty version containing a distinct bug, and the results clearly show that our proposed technique is much more effective than Tarantula, another popular fault localization technique.

Keywords: fault localization, program debugging, RBF (radial basis function) neural network, suspiciousness of code, *EXAM* score, statement ranking

1. Introduction

Regardless of how much effort goes into developing a computer program[1], it will still contain bugs. In fact, the larger and more complex a program, the higher the likelihood of it containing bugs. But to remove bugs from a program, we must first be able to identify exactly where they are. Known as fault localization, this can be extremely tedious and time consuming, and is recognized to be one of the most expensive activities in program debugging [44]. This has sparked the development of several fault localization techniques, over the recent years, that aim to assist developers in finding bugs, thereby reducing the manual effort spent. In this paper we propose an RBF (radial basis function) neural network-based fault localization technique that is more effective at locating bugs, in that a relatively smaller amount of code needs to be examined to find bugs, than other state of the art contemporary techniques.

Neural network-based models have several advantages over other comparable models, such as their ability to learn. Given a sample data set, a neural network can learn rules from the data with or without supervision. Neural networks are also fault-tolerant by virtue of the fact that the information is distributed among the weights on the connections, and so a few faults in the network have relatively less impact on the model. In addition they have the capability to adapt their synaptic weights to changes in the surrounding environment. That is, a neural network trained to operate in a specific environment can be easily re-trained to deal with minor changes in the operating environmental conditions. Such qualities make neural networks popular among researchers, and therefore, neural networks have been successfully applied to many fields, such as pattern recognition [12], system identification [9], intelligent control [30], and software engineering areas including risk analysis [31], cost estimation [39], reliability estimation [38], and reusability characterization [6]. However, to the best of our knowledge, they have not been applied to help developers find bugs except for in our previous study [49], which uses a back-propagation (BP) neural network-based technique for fault localization. In this paper, we propose to use an RBF neural network-based fault localization technique because RBF networks have several advantages over BP networks, including a faster learning rate and a resistance to problems such as paralysis and local minima [24,42].

A typical RBF neural network has a three-layer feed-forward structure that can be trained to learn an input-output relationship based on a data set. In this paper, the input is the statement coverage of a test case which indicates how the program is executed by the test case, and the output is the result (success or failure) of the corresponding program execution. Once the network has been trained, the coverage of a *virtual* test case with only one statement covered[1] is used as an input to compute the *suspiciousness* of the corresponding statement in terms of its likelihood of containing

[1]In this paper, we use "programs" and "software" interchangeably. We also use "bugs" and "faults" interchangeably. In addition, "a statement is covered by a test case" and "a statement is executed by a test case" are used interchangeably.

bugs. The larger the value of the output, the more suspicious the statement seems. Statements can then be ranked in descending order of their suspiciousness, such that developers can examine the ranking of statements (starting from the top), one by one, until the first faulty statement (statement containing bug(s)) is identified. Good fault localization techniques should rank faulty statements towards the top, if not at the very top, of their rankings. An assumption that is typically made (not just by the proposed RBF technique, but by all such fault localization techniques) is that developers can correctly identify a faulty statement as faulty upon examination, and by the same token they will not identify a non-faulty statement as faulty. We systematically evaluate the RBF technique across several different sets of programs (*Siemens suite*, *Unix suite*, *space*, *grep*, *gzip*, *make* and *gcc*) each consisting of many faulty versions. Results show that our proposed technique is much more effective than techniques such as Tarantula [21].

The remainder of the paper is organized as follows. Section 2 provides an overview of RBF neural networks, followed by Section 3 which explains the proposed fault localization technique, as well as presents an example to demonstrate its application. Section 4 then reports on our case studies: we present details on our subject programs, describe how the data was collected, discuss our metric for evaluation, and present data to evaluate the effectiveness of the RBF technique with respect to Tarantula. Section 5 then overviews some related studies, and finally, our conclusions and a discussion on future work are presented in Section 6.

2. Background

2.1 Neural networks

Traditionally, the term "neural network" has been used to refer to a network of biological neurons. The modern definition of this term is an artificial construct whose behavior is based on that of a network of artificial neurons. These neurons are connected together with weighted connections following a certain structure. Each neuron has an activation function that describes the relationship between the input and the output of the neuron [15]. The data can be processed in parallel by different neurons and distributed on the weights of the connections between neurons. Different neural network models have been developed, including BP neural networks [15], RBF neural networks [17], self-organizing map (SOM) neural networks [18], and adaptive resonance theory (ART) neural networks [17]. A particularly important attribute of a neural network is that it can learn from experience. Such learning is normally accomplished through an adaptive process using a learning algorithm. These algorithms can be divided into two categories: *supervised* and *unsupervised* [42]. Each network learning algorithm has certain strengths and weaknesses in the areas of reliability, performance, and generality; however, none has a clear advantage over another.

In fault localization, the output of a given input can be defined as a binary value of 0 or 1, where 1 represents a program failure on this input and 0 represents a successful execution. With this definition, the expected output of each network input (test case coverage) is known because we know exactly whether the corresponding program execution fails or succeeds. Moreover, two similar inputs can produce different outputs because the program execution may fail on one input but succeed on another input. This makes unsupervised learning algorithms inappropriate for our study because those algorithms adjust network weights so that similar inputs produce similar outputs. Therefore, neural networks using supervised learning algorithms are better candidates for solving the fault localization problem. Although BP networks are widely used for supervised learning, RBF networks, whose output layer weights are trained in a supervised way, are even better in our case because they can learn much faster than BP networks and do not suffer from pathologies like paralysis and local minima problems as BP networks [24,42].

2.2 RBF neural networks

A radial basis function (RBF) is a real-valued function whose value depends only on the distance from its receptive field center $\boldsymbol{\mu}$ to the input $\mathbf{x}$. It is a strictly positive radially symmetric function, where the center has the unique maximum and the value drops off rapidly to zero away from the center. When the distance between $\mathbf{x}$ and $\boldsymbol{\mu}$ (denoted as $\|\mathbf{x}-\boldsymbol{\mu}\|$) is smaller than the receptive field width σ, the function has an appreciable value.
A typical RBF neural network has a three-layer feed-forward structure. The first layer is the input layer, which serves as an input distributor to the hidden layer by passing inputs to the hidden layer without changing their values. The

second layer is the hidden layer where all neurons simultaneously receive the n-dimensional real-valued input vector $\mathbf{x}$. Each neuron in this layer uses an RBF as the activation function. A commonly used RBF is the Gaussian basis function [17]

$$R_j(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x}-\boldsymbol{\mu}_j\|^2}{2\sigma_j^2}\right) \quad (1)$$

where $\boldsymbol{\mu}_j$ and σ_j are the mean (the center) and the standard deviation (the width) of the receptive field of the j^{th} hidden layer neuron, and $R_j(\mathbf{x})$ is the corresponding activation function. Usually the distance in Equation (1) is the Euclidean distance between $\mathbf{x}$ and $\boldsymbol{\mu}$, but in this paper we use a weighted bit-comparison-based dissimilarity. To make a distinction, hereafter we use $\|\mathbf{x}-\boldsymbol{\mu}\|$ to represent a *generic* distance, $\|\mathbf{x}-\boldsymbol{\mu}\|_E$ for the *Euclidean* distance, and $\|\mathbf{x}-\boldsymbol{\mu}\|_{WBC}$ (Equation (8) in Section 3.3) for the weighted bit-comparison-based dissimilarity. The third layer is the output layer. The output can be expressed as $\mathbf{y} = [y_1, y_2, \ldots, y_k]$ with y_i as the output of the i^{th} neuron given by

$$y_i = \sum_{j=1}^{h} w_{ji} R_j(\mathbf{x}) \quad \text{for } i = 1, 2, \ldots, k \quad (2)$$

where h is the number of neurons in the hidden layer and w_{ji} is the weight associated with the link connecting the j^{th} hidden layer neuron and the i^{th} output layer neuron.

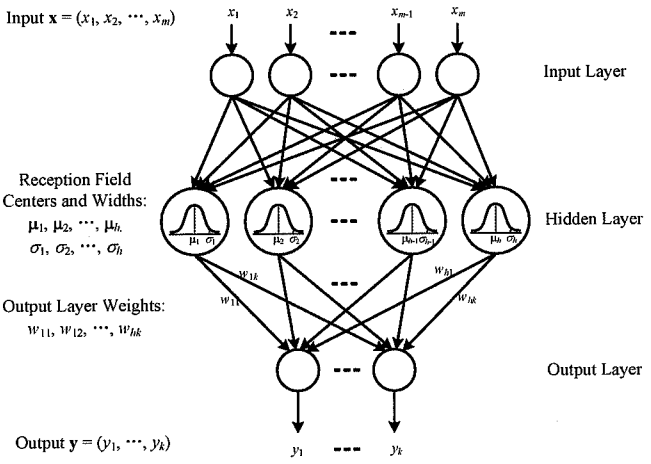


Figure 1. A sample three-layer RBF neural network

An RBF network implements a mapping from the m dimensional real-valued input space to the k dimensional real-valued output space with hidden layer space in between. The transformation from the input space to the hidden-layer space is nonlinear, whereas the transformation from the hidden-layer space to the output space is linear [18]. Figure 1 shows an RBF network with m neurons in the input layer, h neurons in the hidden layer, and k neurons in the output layer. The parameters that need to be trained are the centers ($\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_h$) and widths ($\sigma_1, \sigma_2, \ldots, \sigma_h$) of the receptive fields of hidden layer neurons, and the output layer weights. Many methods have been proposed to train these parameters [17]. Section 3.2 explains how they are trained in our study.

3. The Proposed RBF neural network-based Fault Localization Technique

We first explain the use of an RBF neural network to compute the suspiciousness of each statement in a program P for effective fault localization, and then introduce a two-stage training of the RBF network, including an algorithm for simultaneously determining both the number and the receptive field centers of hidden units. We also provide the formal definition of a weighted bit-comparison-based dissimilarity used by the RBF during the fault localization process, followed by an example to demonstrate the use of our proposed technique.

3.1 Fault localization using an RBF neural network

Suppose we have a program P with m statements. Suppose also that P is executed on n test cases. Let t_i be the i^{th} test case executed on P, $\mathbf{c}_{t_i}$ and r_{t_i} be the coverage vector and the execution result (success or failure) of t_i, respectively, and s_j be the j^{th} statement of P. The vector $\mathbf{c}_{t_i}$ provides us with information on how the program P is covered by test t_i. In this paper, such coverage is reported in terms of which statements[2] in P are executed by t_i. We have $\mathbf{c}_{t_i} = \left[(\mathbf{c}_{t_i})_1, (\mathbf{c}_{t_i})_2, \cdots, (\mathbf{c}_{t_i})_m\right]$ where

$$(\mathbf{c}_{t_i})_j = \begin{cases} 0, & \text{if statement } s_j \text{ is not covered by test } t_i \\ 1, & \text{if statement } s_j \text{ is covered by test } t_i \end{cases} \quad \text{for } 1 \le j \le m$$

The value of r_{t_i} depends on whether the program execution of t_i succeeds or fails. It has a value 1 if the execution fails and a value 0 if the execution succeeds.

We construct an RBF neural network with m input layer units, each of which corresponds to one element in a given $\mathbf{c}_{t_i}$ and one output layer unit, corresponding to r_{t_i} – the execution result of test t_i. In addition, there is a hidden layer between the input and output layers, and the number of hidden units can be determined by using the algorithm in Figure 4, which will be explained in Section 3.2. Each of these neurons uses the Gaussian basis function as the activation function. The receptive field center and width of each hidden layer neuron and the output layer weights are established by training the underlying network.

Once an RBF network is trained, it provides a good mapping between the input (in this case the coverage vector of a test case) and the output (the corresponding execution result). It can then be used to identify suspicious code of a given program in terms of its likelihood of containing bugs. To do so, we use a set of *virtual* test cases v_1, v_2, …, v_m whose coverage vectors are $\mathbf{c}_{v_1}, \mathbf{c}_{v_2}, \cdots, \mathbf{c}_{v_m}$, where

$$\begin{bmatrix} \mathbf{c}_{v_1} \\ \mathbf{c}_{v_2} \\ \vdots \\ \mathbf{c}_{v_m} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \tag{3}$$

Note that the execution of test v_j covers only one statement s_j. As reported in [2,47,48], if the execution of a test case fails, program bugs that are responsible for this failure are most likely to be contained in the corresponding execution slice: the statements executed by this failed test case[3]. Hence, if the execution of v_j fails, the probability that the bugs are contained in s_j is high. This suggests that during the fault localization, we should first examine the statements whose corresponding virtual test case fails. However, the execution results of these virtual tests can rarely be collected in the real world because it is very difficult, if not impossible, to construct such tests.[4] Nevertheless, when the coverage vector $\mathbf{c}_{v_j}$ of a virtual test case v_j is input to the trained neural network, its output $\hat{r}_{v_j}$ is the conditional expectation of whether the execution of v_j fails given $\mathbf{c}_{v_j}$. This implies that the larger the value of $\hat{r}_{v_j}$, the more likely that the execution of v_j fails. Together we have: the larger the value of $\hat{r}_{v_j}$, the more likely it is that s_j contains the bug. We can treat $\hat{r}_{v_j}$ as the suspiciousness of s_j in terms of its likelihood of containing a bug. The process of using the RBF neural network for fault localization is illustrated in Figures 2 and 3, and can be summarized as follows:

---

[2] In addition to statement coverage, without loss of generality, our technique can also be applied to other program components such as functions, blocks, decisions, c-uses and p-uses [50].

[3] In some situations, a test case may fail only because a previous test did not set up an appropriate execution environment. To account for this, we combine these test cases into a single failed test, with an execution slice consisting of the union of each test case's slice [47,48].

[4] In general, the virtual test cases are not "real" test cases and their coverage vectors are not used as training data for the RBF network.

1) Build up a modified RBF neural network with m input units and one output unit. Each unit in the hidden layer uses the Gaussian basis function as its activation function.
2) Determine the number of hidden units h, and the receptive field center and width of each hidden unit.
3) Use the generalized inverse (Moore-Penrose pseudo-inverse) to compute the optimal linear mapping from the hidden units to the output unit.
4) Use the coverage vectors $\mathbf{c}_{v_j}, 1 \le j \le m$ defined in Equation (3) as the inputs to the trained network to produce the outputs $\hat{r}_{v_j}, 1 \le j \le m$.
5) Assign $\hat{r}_{v_j}$ as the suspiciousness of j^{th} statement.

Now statements s_j, $1 \le j \le m$ can be ranked in descending order of their suspiciousness, and examined one by one from the top, until a fault is located. Hereafter, we refer to our proposed technique simply as 'RBF'. We take this opportunity to emphasize that the traditional RBF neural network has been modified to better fit our fault localization context. First, in Step 2, we develop an algorithm (Figure 4) to simultaneously determine the number of hidden neurons and their receptive field centers. Second, we define a weighted bit-comparison-based dissimilarity in order to estimate the distance between two coverage vectors, as opposed to using the traditional Euclidean distance. Further details are provided in Sections 3.2 and 3.3.

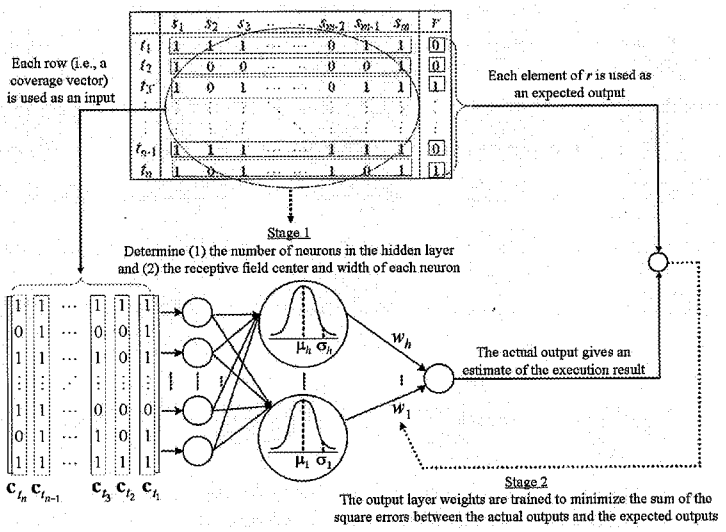


Figure 2. Train an RBF neural network using the coverage vectors and program execution results

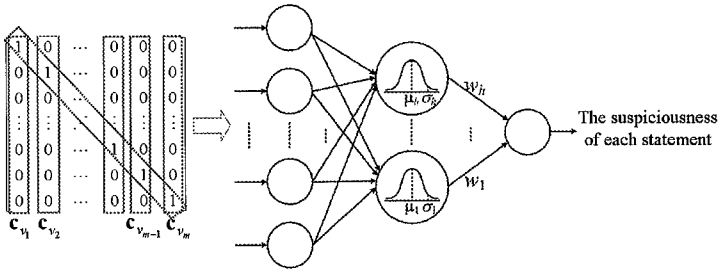


Figure 3. Compute the suspiciousness of each statement in P using virtual test cases

3.2 Training of the RBF neural network

In this section, we discuss the details of the training procedure as described in Section 3.1. The training of an RBF neural network can be divided into two stages [42]. First, the number of neurons in the hidden layer, the receptive field center μ_j and width σ_j of each hidden layer neuron should be assigned values. Second, the output layer weights have to be trained. Many methods have been proposed to determine the receptive field centers. Using standard k-means clustering, input data are assigned to k clusters, with the center of each cluster taken to be the receptive field center of a hidden layer neuron [11,17,26,40]. Unfortunately, this approach does not provide any guidance as to how

many clusters should be used; the number of clusters (and so, the number of receptive field centers) must be chosen arbitrarily. Another disadvantage is that *k*-means is very sensitive to the initial starting values. Its performance will significantly depend on the arbitrarily selected initial receptive field centers.

To overcome these problems, we developed an algorithm (as shown in Figure 4) to simultaneously estimate the number of hidden units and their receptive field centers. The inputs to this algorithm are the coverage vectors $\{\mathbf{c}_{t_1}, \mathbf{c}_{t_2}, \cdots, \mathbf{c}_{t_n}\}$ and a parameter β ($0 \le \beta < 1$) for controlling the number of field centers. The output is a set of receptive field centers $\{\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \cdots, \boldsymbol{\mu}_h\}$ which is a subset of the input vectors such that $\|\boldsymbol{\mu}_i - \boldsymbol{\mu}_j\|_{\text{WBC}} \ge \beta$ for any i and j, $i \ne j$, where $\|\ \|_{\text{WBC}}$ is the weighted bit-comparison-based dissimilarity defined in Section 3.3. Our algorithm not only assigns values to each receptive field center but also decides the number of hidden units because each such hidden unit contains exactly one center. The larger the value of β is, the fewer the number of neurons to be used in the hidden layer, which makes the training at the second stage much faster (as explained at the end of this section). However, if the number of hidden layer units is too small, then the mapping between the input and the output defined by the neural network loses its accuracy.

Once the receptive field centers have been found, we can use different heuristics to determine their widths in order to get a smooth interpolation. Park and Sandberg [32,33] show that an RBF neural network using a single global fixed value σ for all σ_j values has the capability of universal approximation. Moody and Darken [29] suggest that a good estimate of σ is the average over all distances between the center of each neuron and that of its nearest neighbor. In this paper, we use a similar heuristic to define the global width σ as

$$\sigma = \frac{1}{h}\sum_{j=1}^{h} \|\boldsymbol{\mu}_j - \boldsymbol{\mu}_j^*\|_{\text{WBC}} \tag{4}$$

where h is the number of hidden layer neurons and $\|\boldsymbol{\mu}_j - \boldsymbol{\mu}_j^*\|_{\text{WBC}}$ is the weighted bit-comparison-based dissimilarity between $\boldsymbol{\mu}_j$ and its nearest neighbor $\boldsymbol{\mu}_j^*$.

```
input:  C = {c_t1, c_t2, ..., c_tn} and β
output: O
1   begin
2       O ← ∅
3       for each c ∈ C {
4           Temp ← false
5           for each μ ∈ O {
6               if (||c−μ||_WBC < β) {
7                   Temp ← true
8                   break
9               }
10          }
11          if (Temp = = false)
12              O ← {c} ∪ O
13      }
14      output O
15  End
```

Figure 4. The algorithm to determine the receptive field centers

After the centers and widths of the receptive fields of the RBFs in the hidden layer are determined, the remaining parameters that need to be trained are the hidden-to-output layer weights (w_1, w_2, ..., w_h).[5] To do so, we first select a

[5]In our fault localization study, the RBF network is a single-output network which produces only one output for each input coverage vector. Hence, each output layer weight (say w_j) has only one subscript, rather than two subscripts as in Equation (2), showing the connection between the output layer neuron and the corresponding hidden layer neuron (j^{th} hidden neuron in this case).

training set composed of input coverage vectors $(\mathbf{c}_{t_1}, \mathbf{c}_{t_2}, \cdots, \mathbf{c}_{t_n})$ and the corresponding expected outputs $(r_{t_1}, r_{t_2}, \cdots, r_{t_n})$. For an input coverage vector $\mathbf{c}_{t_i}$, its actual output from the network $\hat{r}_{t_i}$ is computed as

$$\hat{r}_{t_i} = \sum_{j=1}^{h} w_j R_j(\mathbf{c}_{t_i}) \text{ where } R_j(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x}-\boldsymbol{\mu}_j\|_{\text{WBC}}^2}{2\sigma_j^2}\right) \text{ for } 1 \le j \le h \tag{5}$$

is the activation functions of the j^{th} hidden layer neuron. Thus, the output of the network is:

$$\hat{\mathbf{r}} = \mathbf{A}\mathbf{w} \tag{6}$$

$$\text{where } \mathbf{A} = \begin{bmatrix} R_1(\mathbf{c}_{t_1}) & R_2(\mathbf{c}_{t_1}) & \cdots & R_h(\mathbf{c}_{t_1}) \\ R_1(\mathbf{c}_{t_2}) & R_2(\mathbf{c}_{t_2}) & \cdots & R_h(\mathbf{c}_{t_2}) \\ \vdots & \vdots & \ddots & \vdots \\ R_1(\mathbf{c}_{t_n}) & R_2(\mathbf{c}_{t_n}) & \cdots & R_h(\mathbf{c}_{t_n}) \end{bmatrix},$$

$$\mathbf{w} = [w_1, w_2, \cdots, w_h]^T \text{ and } \hat{\mathbf{r}} = [\hat{r}_{t_1}, \hat{r}_{t_2}, \cdots, \hat{r}_{t_n}]^T$$

Also, let the expected output $\mathbf{r} = [r_{t_1}, r_{t_2}, \cdots, r_{t_n}]^T$ and the *prediction error* across the entire set of training data be defined as $\|\hat{\mathbf{r}} - \mathbf{r}\|_E^2$ (the sum of squared error between $\hat{\mathbf{r}}$ and $\mathbf{r}$). To find the optimal weights $\mathbf{w}^*$, we have to compute $\mathbf{w}^* = \underset{\mathbf{w}}{\text{argmin}} \|\hat{\mathbf{r}} - \mathbf{r}\|_E^2 = \underset{\mathbf{w}}{\text{argmin}} \|\mathbf{A}\mathbf{w} - \mathbf{r}\|_E^2$. To achieve this objective, we use the generalized inverse (Moore-Penrose pseudo-inverse) of $\mathbf{A}$ [34]:

$$\mathbf{w}^* = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{r} \tag{7}$$

The complexity of computing $\mathbf{w}^*$ depends on the size of $\mathbf{A}$ which is $n \times h$, where n is the number of test cases in the training set and h is the number of hidden units. For a fixed n, the smaller h is, the smaller the complexity. Therefore, an RBF network with a smaller number of hidden units can be trained faster than a network with more hidden units.

3.3 Definition of a weighted bit-comparison-based dissimilarity

From Equation (5), for a given test case t_i and its input coverage vector $\mathbf{c}_{t_i}$, the actual output $\hat{r}_{t_i}$ is a linear combination of the activation functions of all hidden layer neurons. Each R_j depends on the distance $\|\mathbf{x}-\boldsymbol{\mu}_j\|$ (referring to Equation (1)). In our case, $\mathbf{x}$ is the input coverage vector $\mathbf{c}_{t_i}$ and $\boldsymbol{\mu}_j$ is the receptive field center of the j^{th} hidden layer neuron. So, we have $\|\mathbf{x}-\boldsymbol{\mu}_j\| = \|\mathbf{c}_{t_i}-\boldsymbol{\mu}_j\|$. From the algorithm in Figure 4, we observe that the set of receptive centers is a subset of the coverage vectors. This implies each $\boldsymbol{\mu}_j$ by itself is also the coverage vector of a certain test case. As a result, the distance $\|\mathbf{x}-\boldsymbol{\mu}_j\|$ can also be viewed as the distance between two coverage vectors.

The most commonly used distance is the Euclidean distance. However, this distance is not suitable for our problem because it cannot represent the difference between coverage vectors accurately. For the purpose of explanation, let us use the following example. Suppose we have an RBF network trained by $\mathbf{c}_{t_1}$ = [0, 0, 1, 1, 0], $\mathbf{c}_{t_2}$ = [1, 0, 1, 1, 1] and their execution results r_{t_1} and r_{t_2}. Suppose also the trained network has two neurons in the hidden layer with $\boldsymbol{\mu}_1 = \mathbf{c}_{t_1}$ and $\boldsymbol{\mu}_2 = \mathbf{c}_{t_2}$. When we have $\mathbf{c}_{v_1}$ = [1, 0, 0, 0, 0] as the input to the trained neural network, the output $\hat{r}_{v_1} = w_1 R_1(\mathbf{c}_{v_1}) + w_2 R_2(\mathbf{c}_{v_1}) = w_1 \exp\left(-\frac{\|\mathbf{c}_{v_1}-\mathbf{c}_{t_1}\|^2}{2\sigma_1^2}\right) + w_2 \exp\left(-\frac{\|\mathbf{c}_{v_1}-\mathbf{c}_{t_2}\|^2}{2\sigma_2^2}\right)$, where $\sigma_1 = \sigma_2 = \sigma$. Since the first statement is covered by t_2 and v_1, but not t_1, we should have $R_1(\mathbf{c}_{v_1}) \neq R_2(\mathbf{c}_{v_1})$, which implies $\|\mathbf{c}_{v_1}-\mathbf{c}_{t_1}\| \neq \|\mathbf{c}_{v_1}-\mathbf{c}_{t_2}\|$. More precisely, v_1 is more similar to t_2 than to t_1, which means the output of the hidden unit with $\mathbf{c}_{t_2}$ as its center should contribute more

to the network output. That is, we should have $R_1(\mathbf{c}_{v_1}) < R_2(\mathbf{c}_{v_1})$ and therefore $\|\mathbf{c}_{v_1} - \mathbf{c}_{t_1}\| > \|\mathbf{c}_{v_1} - \mathbf{c}_{t_2}\|$. However, the Euclidean distance between $\mathbf{c}_{v_1}$ and $\mathbf{c}_{t_1}$ is the same as that between $\mathbf{c}_{v_1}$ and $\mathbf{c}_{t_2}$. To overcome this problem, we use a weighted bit-comparison-based dissimilarity defined as

$$\|\mathbf{c}_{t_i} - \boldsymbol{\mu}_j\|_{\text{WBC}} = \sqrt{1 - \cos\theta_{\mathbf{c}_{t_i}, \boldsymbol{\mu}_j}} \tag{8}$$

where $\cos\theta_{\mathbf{c}_{t_i}, \boldsymbol{\mu}_j} = \dfrac{\sum_{k=1}^{m} (\mathbf{c}_{t_i})_k (\boldsymbol{\mu}_j)_k}{\sqrt{\sum_{k=1}^{m} [(\mathbf{c}_{t_i})_k]^2} \times \sqrt{\sum_{k=1}^{m} [(\boldsymbol{\mu}_j)_k]^2}}$, $(\mathbf{c}_{t_i})_k$ and $(\boldsymbol{\mu}_j)_k$ are the k^{th} elements of $\mathbf{c}_{t_i}$ and $\boldsymbol{\mu}_j$, respectively. The dissimilarity measure between two binary vectors in Equation (8) is more desirable since it effectively takes into account the number of bits that are both 1 in two coverage vectors (those statements covered by both vectors). In the above example, if we replace the Euclidean distance by the weighted bit-comparison-based dissimilarity, then we have $\|\mathbf{c}_{v_1} - \mathbf{c}_{t_1}\|_{\text{WBC}} = 1$ which is greater than $\|\mathbf{c}_{v_1} - \mathbf{c}_{t_2}\|_{\text{WBC}} = \sqrt{1/2}$.

3.4 An Illustrative Example of the Proposed Technique

Let us suppose we have a program with ten statements s_j, $1 \le j \le 10$ and that a total of seven test cases have been executed on the program. Table 1 gives the coverage vector and the execution result of each test.

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	r
t_1	1	1	1	1	0	1	0	0	1	1	0
t_2	1	0	0	1	1	0	1	0	0	1	0
t_3	1	1	1	0	0	1	0	0	1	1	0
t_4	1	0	1	0	0	1	1	0	1	1	0
t_5	1	1	1	0	1	0	0	0	1	0	0
t_6	1	1	1	1	0	0	0	1	1	1	1
t_7	1	0	1	1	1	1	1	1	0	1	1

t_1 is a successful test

t_6 is a failed test

s_1 is executed by t_1 s_6 is not executed by t_2

Table 1: The coverage data and execution results used in the example

$$\begin{bmatrix} \mathbf{c}_{v_1} \\ \mathbf{c}_{v_2} \\ \mathbf{c}_{v_3} \\ \mathbf{c}_{v_4} \\ \mathbf{c}_{v_5} \\ \mathbf{c}_{v_6} \\ \mathbf{c}_{v_7} \\ \mathbf{c}_{v_8} \\ \mathbf{c}_{v_9} \\ \mathbf{c}_{v_{10}} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Part (a): Input coverage vectors

$\hat{r}_{v_1}$	0.0384	$\hat{r}_{v_6}$	0.0179
$\hat{r}_{v_2}$	0.0481	$\hat{r}_{v_7}$	0.0157
$\hat{r}_{v_3}$	0.1246	$\hat{r}_{v_8}$	0.2900
$\hat{r}_{v_4}$	0.0768	$\hat{r}_{v_9}$	0.0066
$\hat{r}_{v_5}$	0.0173	$\hat{r}_{v_{10}}$	0.0782

Part (b): Outputs of the trained network are the suspiciousness of the corresponding statements

Figure 5. Inputs and outputs/statement suspiciousness based on the example

We follow the steps listed in Section 3.1. An RBF neural network with ten input units and one output unit is constructed. Using the algorithm in Figure 4 with $\beta = 0.1$, we find that each coverage vector also serves as the receptive field center of a hidden unit. This implies there are seven units in the hidden layer. The field width σ computed by using Equation (4) is 0.395. The output layer weights are trained by the data in Table 1. We have $\mathbf{w} = [w_1, w_2, w_3, w_4, w_5, w_6, w_7]^T = [-1.326, -0.665, 0.391, -0.378, -0.308, 1.531, 1.381]^T$. We use the coverage vectors of

the virtual test cases in Part (a) of Figure 5 as the inputs to the trained network. The outputs are shown in Part (b) and they correspond to the suspiciousness of their respective statements. Ranking the statements in descending order of their suspiciousness, we have: s_8, s_3, s_{10}, s_4, s_2, s_1, s_6, s_5, s_7, s_9.

Let us also take this opportunity to discuss Tarantula [21] which is a benchmark fault localization technique, to which RBF shall subsequently be compared. Tarantula follows the intuition that entities in a program that are primarily executed by failed test cases are more likely to be faulty than those executed by passed (successful) test cases. Suspiciousness is assigned to each statement s according to the formula:

$$susp(s) = \frac{\frac{failed(s)}{totalfailed}}{\frac{passed(s)}{totalpassed} + \frac{failed(s)}{totalfailed}} \quad (9)$$

Where *failed*(s) and *passed* (s) are the number of failed and successful test cases that execute statement s, respectively. Also, the quantities *totalfailed* and *totalpassed* correspond to the total number of failed and successful test cases respectively. Using the information presented in Table 1, we make some observations regarding the suspiciousness assigned to statements by Tarantula versus that assigned to statements by RBF (as per part (b) of Figure 5). Firstly, Tarantula is unable to distinguish between the statements s_5 and s_7 in terms of assigned suspiciousness (both are assigned the value 0.556), which is not the case as far as RBF is concerned. However, what is of more significance is that Tarantula cannot distinguish statement s_3 from statements s_5 and s_7 either, and they are all assigned the same suspiciousness. Intuitively this seems strange, as regardless of how many successful test cases may execute a statement, statement s_3 is executed by both failing test cases whereas statements s_5 and s_7 are only executed by one failing test case a piece; and therefore, statement s_3 should be more suspicious than s_5 or s_7. This is in keeping with the intuition in [45] where (in the context of the computation of statement suspiciousness) the contribution from all of the successful test cases is always considered to be less than that of the failed test cases. Interestingly enough, RBF assigns s_3 a suspiciousness that is higher than both s_5 and s_7. This illustrates that not just are techniques such as Tarantula and RBF very different by construction, but also the statement rankings that are produced might vary greatly, and consequently, so might the fault localization effectiveness of the techniques.

4. Case Studies

In this section, we report our case studies using the RBF technique for effectively locating program bugs. We also compare the effectiveness (formally defined in Section 4.3) of our technique with Tarantula [21] which has been shown to be more effective than other fault localization techniques such as set union, set intersection, nearest neighbor [35], and cause transitions techniques [10].

4.1 Subject programs

To show the effectiveness of our RBF network-based fault localization technique, we conduct case studies on six suites of programs: Siemens suite, Unix suite, space, make, grep and gzip, all of which are written in C.

The seven programs in the Siemens suite have been used in the testing and fault localization related studies [10,19,21,35]. The correct versions, 132 faulty versions of the programs and all the test cases are downloaded from the web site http://www-static.cc.gatech.edu/aristotle/Tools/subjects. Three faulty versions are excluded in our study: version 9 of "schedule2" because there is no failed test; versions 4 and 6 of "print_tokens" because the faults are in the header files instead of the C files. In a previous study [21] also utilizing the Siemens Suite, ten faulty versions were excluded. Among those excluded faults, versions 27 and 32 of "replace" and versions 5, 6, and 9 of "schedule" were not used because the tool used in [21] (gcc with gcov) does not dump its coverage before the program crashes. The use of a revised version of χSuds [50], which can collect runtime trace correctly even with a segmentation fault, allows us to circumvent this problem and therefore make use of those faulty versions. Also, we use all the test cases downloaded, and therefore the size of the test set used numbers slightly higher than that reported in [21]. As a pleasant side-effect, version 10 of "print_tokens" and version 32 of "replace" could also have been used in our study even though they had to be excluded in [21] because no test cases failed on those versions. The summary of each

program including the name and a brief description, the number of faulty versions, LOC (the size of the program before any non-executable code is removed), number of executable statements and number of test cases is presented in Table 2. Similar to [21], multiple-line statements are combined as one source code line so that they will be counted only as one executable statement. The same approach is also applied to other programs.

Program	Description	Number of faulty versions	LOC	Number of executable statements	Number of test cases
print_tokens	Lexical analyzer	7	565	175	4130
print_tokens2	Lexical analyzer	10	510	178	4115
replace	Pattern replacement	32	563	216	5542
schedule	Priority scheduler	9	412	121	2650
schedule2	Priority scheduler	10	307	112	2710
tcas	Altitude separation	41	173	55	1608
tot_info	Information measure	23	406	113	1052

Table 2. Summary of the Siemens suite

The Unix suite consists of ten Unix utility programs. Since these programs have been so thoroughly used, they can be a reliable basis for evaluating the behavior of fault-injected programs derived from them. These faulty versions are created by using mutation-based fault injection, which has been shown in a recent study [3] as an appropriate approach for simulating real faults in software testing research. Table 3 gives a summary of this suite. More descriptions of the test case generation, fault set, and erroneous program preparation can be found in [46].

Program	Description	Number of faulty versions	LOC	Number of executable statements	Number of test cases
Cal	Print a calendar for a specified year or month	20	202	88	162
Checkeq	Report missing or unbalanced delimiters and .EQ/.EN pairs	20	102	57	166
Col	Filter reverse paper motions from nroff output for display on a terminal	30	308	165	156
Comm	Select or reject lines common to two sorted files	12	167	76	186
Crypt	Encrypt and decrypt a file using a user supplied password	14	134	77	156
Look	Find words in the system dictionary or lines in a sorted list	14	170	70	193
Sort	Sort and merge files	21	913	448	997
Spline	Interpolate smooth curves based on given data	13	338	126	700
Tr	Translate characters	11	137	81	870
Uniq	Report or remove adjacent duplicate lines	17	143	71	431

Table 3. Summary of the Unix suite

The space program was developed at the European Space Agency. It provides a language-oriented user interface that allows the user to describe the configuration of an array of antennas by using a high level language. Following the same convention as described in [21], if we combine multi-line statements as one source code line and count it only as one executable statement, we have 3657 executable statements. The correct version, the 38 faulty versions, and a suite of 13585 test cases used in this study were downloaded from [20]. Three faulty versions were not used in our study because no test cases can reveal the faults in these versions.

The grep program searches for a pattern in a file. The source code of version 2.2 was also downloaded from [20], together with a suite of 470 test cases and 18 bugs of which four bugs could be detected in our environment. Two additional bugs injected by Liu et al. [27] were also used. The authors in [27] argue that although faults are manually injected, they do mimic realistic logic errors. We followed a similar approach to inject 13 additional bugs. With the addition of these faults, altogether, there are 19 faulty versions.

The gzip program reduces the size of named files using the Lempel-Ziv coding. We downloaded version 1.1.2 with 16 seeded bugs and 217 test cases from [20]. Six test cases were discarded because they could not be executed in our

environment; nine bugs were also excluded since none of the remaining 211 test cases reported failures on any of them. We also followed a similar approach to inject 21 additional bugs. In total, 28 faulty versions were used.

The make program is a software utility that manages the building of executables and other products from source code. Version 3.76.1 of make was downloaded from [20], together with 793 test cases and 19 faulty versions of the program. Of these, 15 faulty versions were excluded as they contained bugs which could not be detected by any of the downloaded test cases in our environment. Using a similar fault injection approach to that described in [27], we generated an additional 27 bugs for a total of 31 usable faulty versions.

Table 4 gives a summary of the space, gzip, grep and make programs. A list of all the additional bugs for grep, gzip and make is available upon request. As explained in [3,27], they are similar to real bugs.

Program	Number of faulty versions	LOC	No. of executable statements	No. of test cases
space	38	9126	3657	13585
gzip (version 1.1.2)	28	6573	1670	211
grep (version 2.2)	19	12653	3306	470
make (version 3.76.1)	31	20014	5318	793

Table 4. Summary of space, gzip, grep and make.

The six suites of programs vary dramatically in both their sizes and functionalities, which makes our results even more convincing and representative.

4.2 Data collection

For the Siemens suite, Unix suite and space, all executions were on a PC with a 2.13GHz Intel Core 2 Duo CPU and 8GB physical memory. The operating system was SunOS 5.10 (Solaris 10) and the compiler used was gcc 3.4.3. For grep, gzip and make, the executions were on a Sun-Fire-280R machine with SunOS 5.10 as the operating system and gcc 3.4.4 as the compiler. Each faulty version was executed against all its corresponding available test cases. The statement coverage with respect to each test case was measured by using a revised version of χSuds [50], which could collect runtime trace correctly even if the execution crashed due to a segmentation fault. The success or failure of an execution was determined by comparing the outputs of the faulty version and the correct version of a program.

Our focus is to help programmers find a starting point to fix a bug rather than provide the complete set of code that has to be corrected with respect to each bug. Therefore, although a bug may span multiple statements, which may not be contiguous, or even multiple functions, the fault localization stops when the first statement containing the bug is reached. Note that in no way does this mean the proposed RBF technique is limited to faults that are only across a single line. We also assume perfect bug detection, that is, a bug in a statement will be detected by a programmer if the statement is examined. If such perfect bug detection does not hold, then the number of statements that need to be examined in order to find the bug may increase. The same concern applies to all the fault localization techniques discussed in this paper. In addition, we assume the cost of examining each statement for locating the bug is fixed.

In Section 4.4, the results of our RBF technique are compared with those of Tarantula in order to evaluate the relative effectiveness of each technique. For a fair comparison, we compute the effectiveness of Tarantula using our test data and their ranking mechanism. Note that statistics such as fault revealing behavior and statement coverage of each test can vary under different compilers, operating systems, and hardware platforms. Also, the ability of the coverage measurement tool (revised version of χSuds in our experiments versus gcc with gcov in theirs) to properly handle segmentation faults has an impact on the use of certain faulty versions. However, such variance is not expected to be very large in nature, and does not detract from the validity of our experiments or their accompanying results.

4.3 Criteria to Evaluate Fault Localization Effectiveness

In previous studies, Renieris et al. [35] assign a score to every faulty version of each subject program, which is defined as the percentage of the program that need not be examined to find a faulty statement in the program or a

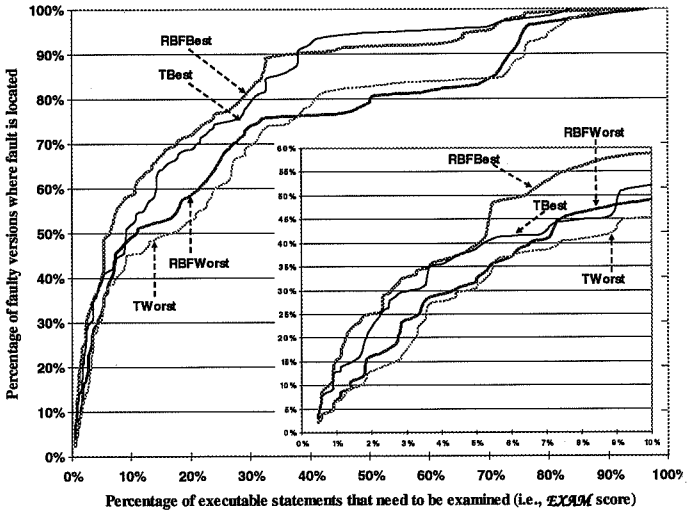
faulty node in the corresponding program dependence graph. This score or effectiveness measure is later adopted by Cleve and Zeller in [10], and is defined as $1-\frac{|N|}{|PDG|}$ where N is the set of all nodes examined and PDG is the set of all nodes in the program dependence graph. Instead of the program dependence graph, Tarantula directly uses the program's source code, and therefore, in order to make their effectiveness computations comparable to those of the program dependence graph, Jones et al. [21] consider only executable statements to compute their score. They omit from consideration source code such as blank lines, comments, function and variable declarations. They also combine multi-line statements into one source code line so that they are only counted once. The comparison now becomes a fair one – only statements that can be represented in the program dependence graph are considered [21]. Since the RBF fault localizer also operates directly on the program source, we also follow this same strategy and consider only executable statements in all of our experiments. However, while the authors of [21] define their score to be the percentage of code that need not be examined in order to find a fault, we feel it is more straightforward to present the percentage of code that *has to be examined* in order to find the fault. This modified score is hereafter referred to as $\mathcal{EXAM}$ and is defined as the percentage of executable statements that have to be examined until the first statement containing the bug is reached. We note that the two scores are equivalent and it is easy to derive one from the other. Thus, the effectiveness of various fault localization techniques can be measured and compared on the basis of the $\mathcal{EXAM}$ score. We also draw comparisons based on the number of faulty versions where one technique might perform better, worse or equal to another; as well as based on the total number of statements that need to be examined to locate bugs in all the faulty versions of a program under study, by either technique. Thus, our evaluation is conducted on the basis of several criteria.

Prior to presenting the results of our case studies, there is one more aspect of fault localization effectiveness that remains to be discussed. It is not necessary that the suspiciousness assigned to a statement by a fault localization technique be unique. Thus, the same suspiciousness value may be assigned to multiple statements, thereby yielding two different types of effectiveness: the "best" and the "worst." The "best" effectiveness assumes that the faulty statement is the first to be examined among all the statements of the same suspiciousness. Supposing there are ten statements of the same suspiciousness of which one is faulty, the "best" effectiveness is achieved if the faulty statement is the first to be examined of these ten statements. Similarly, the "worst" effectiveness occurs if the faulty statement is the last to be examined of these ten statements. Hereafter, we refer to the effectiveness of the RBF technique under the best and the worst cases as RBFBest and RBFWorst. Similarly, we have TBest and TWorst as the best and the worst effectiveness of the Tarantula technique respectively. Data corresponding to both the best and worst effectiveness, according to each of the evaluation criteria discussed above, is provided for either technique.

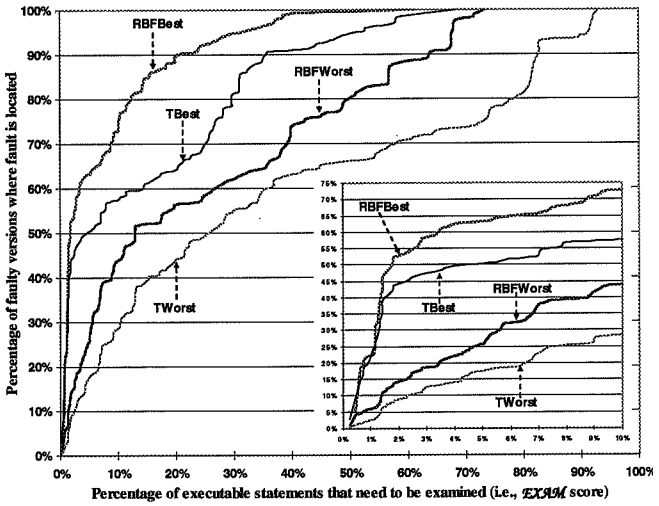
4.4 Comparison of RBF with Tarantula

Figure 6 gives the effectiveness of the RBF and Tarantula techniques for all the programs used in our studies. The curves labeled RBFBest (in red) and RBFWorst (in blue) are for the best and the worst effectiveness of the RBF technique, and those labeled TBest (in black) and TWorst (in green) are for the best and the worst effectiveness of the Tarantula technique. Since these curves are displayed in different colors, they are best viewed in color. For a given x value (percentage of executable statements examined), its corresponding y value is the percentage of the faulty versions whose $\mathcal{EXAM}$ score is less than or equal to x.

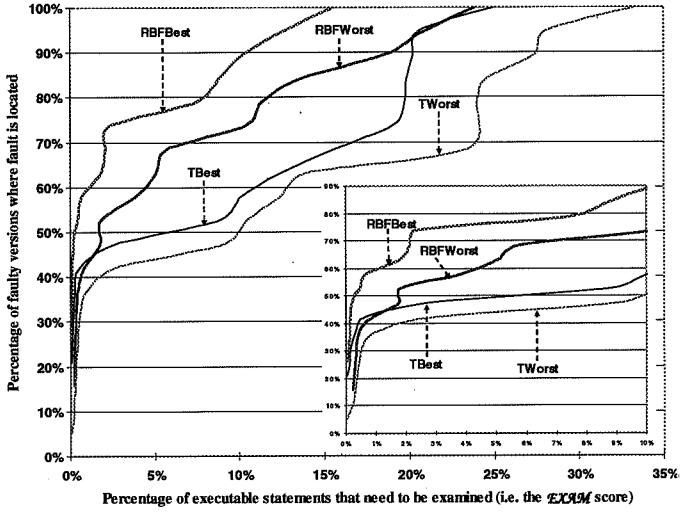
From the graphs we find that in RBFBest generally performs better than TBest, and that RBFWorst performs better than TWorst. For example, by examining less than 1% of the code RBFBest can locate 19 (14.73%) of the 129 faults in the Siemens suite whereas TBest can only locate 16 (12.40%). Similar observations can also be made of RBFWorst. Furthermore, the fact that RBF is better than Tarantula, becaomes especially evident for the Unix suite, grep, space and make where this observation holds for any $\mathcal{EXAM}$ score. Another significant point is that in many cases even RBFWorst is more effective than TBest. For example, with less than 1.67% of the statements being examined, RBFWorst can guide the programmers to find bugs in 29 (82.86%) faulty versions of the space program, whereas TBest can only do this for 24 (68.57%) faulty versions.



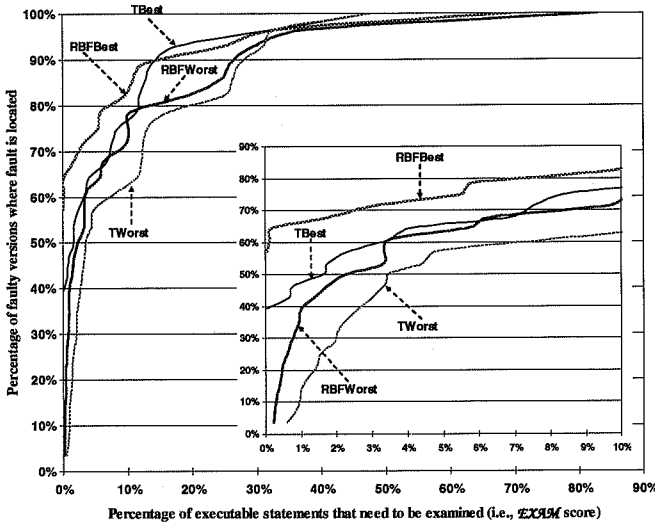
Part (a). Effectiveness comparison for the Siemens suite



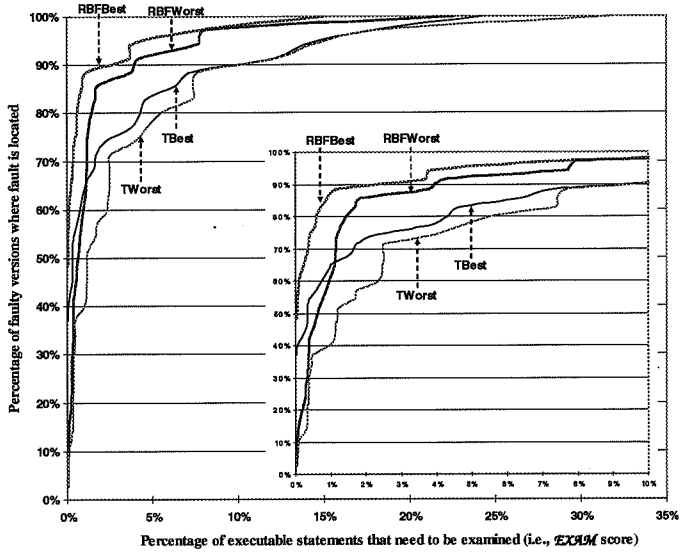
Part (b). Effectiveness comparison for the Unix suite



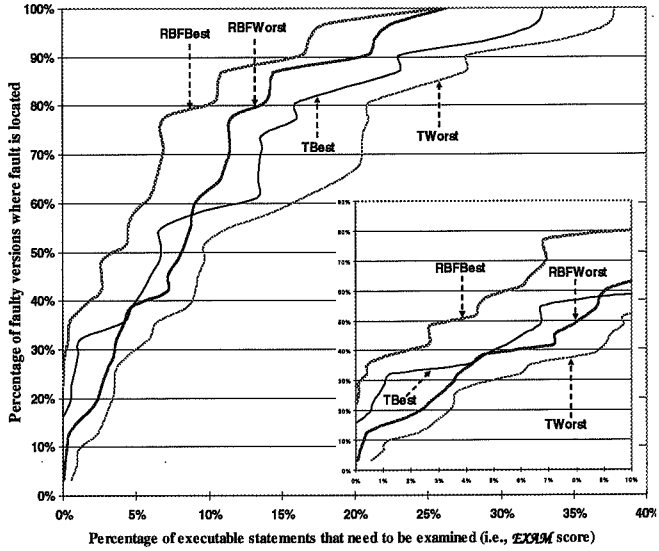
Part (c). Effectiveness comparison for the grep program



Part (d). Effectiveness comparison for the gzip program



Part (e). Effectiveness comparison for the space program



Part (f). Effectiveness comparison for the make program

Figure 6. Effectiveness comparison between the RBF technique and Tarantula

Data is also provided in Table 5 to show the pairwise comparison between the effectiveness of the RBF technique and Tarantula (as discussed in Section 4.3) to decide for how many faulty versions one technique outperforms another, equals another, and underperforms another. As an example, for the make program RBFBest is more effective (i.e., examining fewer statements before the first faulty statement containing the bug is identified) than TBest for 23 of the 31 faulty versions, as effective (i.e., examining the same number of statements) for 5 faulty versions, and less effective (i.e., examining more statements) for 3 faulty versions. The table also lists the comparison between RBFWorst and TWorst. Even when RBFWorst is compared with TBest, still the former is at least as effective as the latter for a good number of the faulty versions.

		RBFBest versus TBest	RBFWorst versus TWorst	RBFWorst versus TBest
Siemens	More effective	68	67	39
	Same effectiveness	25	25	11
	Less effective	36	37	79
Unix	More effective	87	125	48
	Same effectiveness	69	30	11
	Less effective	16	17	113
grep	More effective	14	15	9
	Same effectiveness	3	2	1
	Less effective	2	2	9
gzip	More effective	12	17	5
	Same effectiveness	11	6	1
	Less effective	5	5	22
space	More effective	18	21	16
	Same effectiveness	12	9	1
	Less effective	5	5	18
make	More effective	23	28	16
	Same effectiveness	5	0	0
	Less effective	3	3	15

Table 5. Pairwise comparison between RBF and Tarantula

Table 6 presents the effectiveness comparison in terms of the total number of statements that need to be examined to locate all the bugs. For each program, RBFBest requires the examination of fewer statements than TBest. The same applies to the comparison between RBFWorst and TWorst. For example, the ratio of the number of statements examined by RBFBest to the number of statements examined by TBest for all 35 faulty versions of space is 34.49%; and the ratio between RBFWorst and TWorst is 47.45%. This also implies RBFBest examines 65.51% fewer statements than TBest, and RBFWorst examines 52.55% fewer statements than TWorst. Moreover, for three of the six suites (grep, space and make) even RBFWorst examines fewer statements than TBest. Note that there may not be any subset/superset relationship between the statements examined by RBF and Tarantula, because as per the discussion in Section 3.4, the rankings produced by RBF and Tarantula can be very different from one another. Thus, when we say RBF requires the examination of only a fraction (percentage) of the statements that Tarantula requires, this fraction is based purely on the number of statements, and not on the sets of statements, examined.

	RBFBest	TBest	RBFBest/TBest	RBFWorst	TWorst	RBFWorst/TWorst
Siemens	2114	2453	86.18%	2980	3311	90.00%
Unix	1302	3364	38.70%	4758	7629	62.37%
Grep	2075	5793	35.82%	3964	7812	50.74%
Gzip	2966	3110	95.37%	4743	5032	94.26%
Space	1337	3876	34.49%	2417	5094	47.45%
Make	9188	16890	54.40%	14590	23468	62.17%

Table 6. Total number of statements examined by RBF and Tarantula

Based on the data collected from our studies on the Siemens suite, Unix suite, space, grep, gzip and make programs, we observe that not only is RBFBest more effective than TBest, and RBFWorst is more effective than TWorst, but also RBFWorst is more effective than TBest in many cases. This clearly indicates the RBF technique is more effective in fault localization, because less code needs to be examined to locate faults by using RBF, than Tarantula.

5. Related Studies

Over the recent years several studies have been performed, and several techniques proposed, in the area of software fault localization. In this section we overview a representative set of techniques, and since it is impossible to do justice to each technique using brief descriptions, we direct interested readers to the accompanying references for additional details.

Renieris and Reiss [35] propose a nearest neighbor debugging technique that contrasts a failed test with another successful test which is most similar to the failed one in terms of the "distance" between them. The execution of a test is represented as a sequence of basic blocks that are sorted by their execution times. If a bug is in the difference set between the failed execution and its most similar successful execution, it is located. For a bug that is not contained in the difference set, the technique continues by first constructing a program dependence graph and then including and checking adjacent un-checked nodes in the graph step by step until the bug is located. The set union and set intersection techniques are also reported in [35]. The former computes the set difference between the "program spectra" of a failed test and the union spectra of a set of successful tests. It focuses on the source code that is executed by the failed test but not by any of the successful tests. The latter is based on the set difference between the intersection spectra of successful tests and the spectra of the failed test. It focuses on statements that are executed by every successful test but not by the failed test case.

In [10], Cleve and Zeller report a program state-based debugging technique, cause transition, to identify the locations and times where a cause of failure changes from one variable to another. This is an extension of their earlier work with delta debugging [51,52]. An algorithm named *cts* is proposed to quickly locate cause transitions in a program execution. A potential problem of the cause transition technique is that its cost is relatively high; there may exist thousands of states in a program execution, and delta debugging at each matching point requires additional test runs to narrow down the causes. As per [21] Tarantula reported performs better than techniques such as set union, set intersection, and cause transitions.

Liblit et al. propose a statistical debugging technique (Liblit05) that can isolate bugs in the programs with instrumented predicates at particular points [25]. Feedback reports are generated by these instrumented predicates. For each predicate P, the algorithm first computes *Failure*(P), the probability that P being true implies failure, and *Context*(P), the probability that the execution of P implies failure. Predicates that have *Failure*(P) – *Context*(P) ≤ 0 are discarded. The remaining predicates are prioritized based on their "*importance*" scores which give an indication of the relationship between predicates and program bugs. Predicates with a higher score should be examined first to help programmers find bugs. Once a bug is found and fixed, the feedback reports related to it are removed. This process continues to find other bugs until all the feedback reports are removed or all the predicates are examined.

As extension (and improvement) to Liblit05, Liu et al. propose the SOBER technique to rank suspicious predicates [27]. First, $\pi(P)$ which is the probability that predicate P is evaluated to be true in each run is computed as $\pi(P) = \frac{n(t)}{n(t)+n(f)}$, where $n(t)$ is the number of times P is evaluated to be true in a specific run and $n(f)$ is the number of times P is evaluated as false. If the distribution of $\pi(P)$ in failed runs is significantly different from that of $\pi(P)$ in successful runs, then P may be fault-relevant and this *relevance* is quantified using a ranking score. All instrumented predicates can then be ranked in order of their scores and examined in order of their fault-relevance.

Zhang et al. [53] present a technique such that for a given failed test, their technique requires multiple executions against that test. In each execution, the outcome of one predicate is switched, and this process continues until the program produces the correct output as a result of the switch; this predicate is a critical predicate. Bidirectional dynamic slices of such critical predicates are then computed to help programmers locate the bugs. There are also

many slicing-based studies which can be further classified as static slicing-based [28,43], dynamic slicing-based [1,23], and execution slicing-based [2,47,48] fault localization techniques.

In addition to the above, there are other fault localization techniques based on program spectrum (e.g., [13,16,36]), program state (e.g., [14,41]), slicing (e.g., [22,37,54]), machine learning (e.g., [4,5,7]), data mining (e.g., [8]), etc.

6. Conclusions and Future Work

An RBF (radial basis function) neural network-based fault localization technique is presented in this paper. The network is trained on coverage information for each test case paired with its execution result, either success or failure, and the network so trained is then given as input a set of virtual test cases, each of which covers a single statement. The output of the network is considered to be the suspiciousness the statement corresponding to the virtual test. Statements with a higher suspiciousness should be examined first as they are more likely to contain program bugs. Empirical data based on the Siemens suite, the Unix suite, space, grep, gzip and make programs (i.e., both small and large-sized programs) indicates that RBF is significantly more effective in fault localization than Tarantula (another popular fault localization technique). Studies that target a wider range of application domains are currently in progress to further validate the general effectiveness of our fault localization technique. We also intend to extend our studies using other machine learning algorithms (e.g., support vector machines, decision trees, logistic regression, etc.) and observe the potential variation of the performance, if any. Additionally, we are currently working with our industry partners to evaluate the RBF fault localization technique on ongoing, and large-scale, software development.

References:

1. H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Debugging with dynamic slicing and backtracking," *Software: Practice & Experience*, 23(6):589-616, June, 1996.
2. H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," in *Proceedings of the 6th International Symposium on Software Reliability Engineering*, pp. 143-15, Toulouse, France, October, 1995.
3. J.H. Andrews, L.C. Briand and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of the 27th International Conference on Software Engineering*, pp. 402-411, St. Louis, Missouri, USA, May, 2005.
4. L. C. Ascari, L. Y. Araki, A. R. T. Pozo, and S. R. Vergilio, "Exploring machine learning techniques for fault localization," in Proceedings of the 10th Latin American Test Workshop, pp. 1-6, Buzios, Brazil, March 2009
5. L. C. Briand, Y. Labiche, and X. Liu, "Using Machine Learning to Support Debugging with Tarantula", in Proceedings of the 18th IEEE International Symposium on Software Reliability, pp. 137-146, Trollhattan, Sweden, November 2007
6. G. Boetticher and D. Eichmann, "A neural network paradigm for characterizing reusable software," in *Proceedings of the 1st Australian Conference on Software Metrics*, pp. 41-54, Sydney, Australia, November, 1993.
7. Y. Brun and M. D. Ernst, "Finding latent code errors via machine learning over program executions," in Proceedings of the 26th International Conference on Software Engineering, pp. 480- 490, Edinburgh, UK, May 2004
8. P. Cellier, S. Ducasse, S. Ferre, and O. Ridoux, "Formal concept analysis enhances fault localization in software", in Proceedings of the 4th International Conference on Formal Concept Analysis, pp. 273-288, Montréal, Canada, February 2008
9. S. R. Chu, R. Shoureshi, and M. Tenorio, "Neural networks for system identification," *IEEE Control Systems Magazine*, 10(3):31-35, April, 1990.
10. H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th International Conference on Software Engineering*, pp. 342-351, St. Louis, Missouri, USA, May, 2005.
11. J. Dang, Y. Wang, and S. Zhao, "Face recognition based on radial basis function neural networks using subtractive clustering algorithm," in *Proceedings of the 6th World Congress on Intelligent Control and Automation*, pp. 10294-10297, Dalian, China, June, 2006.
12. K. Fukushima, "A neural network for visual pattern recognition," *Computer*, 21(3):65-75, March 1998.
13. L. Guo, A. Roychoudhury, and T. Wang, "Accurately choosing execution runs for software fault localization," in Proceedings of the 15th International Conference on Compiler Construction, pp. 80-95, Vienna, Austria, March 2006
14. N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating faulty code using failure-inducing chops," in Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 263-272, Long Beach, California, USA, November 2005
15. M. T. Hagan, H. B. Demuth, and M. Beale, *Neural network design*, PWS Publishing, 1995.
16. M. J. Harrold, G.. Rothermel, K. Sayre, R. Wu, and L. Yi, "An empirical investigation of the relationship between spectra differences and regression faults," Journal of Software Testing, Verification and Reliability, 10(3):171-194, September 2000
17. M. H. Hassoun, *Fundamentals of Artificial Neural Networks*, MIT Press, 1995.
18. S. Haykin, *Neural Networks: A Comprehensive Foundation (2nd Edition)*, Prentice Hall, 1999.
19. 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 the 16th International Conference on Software Engineering*, pp. 191-200, Sorrento, Italy, May, 1994.
20. http://sir.unl.edu/portal/index.html
21. J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM Conference on Automated Software Engineering*, pp. 273-282, Long Beach, California, USA, December, 2005.

22. B. Korel, "PELAS – Program error-locating assistant system," IEEE Transactions on Software Engineering, 14(9):1253-1260, September 1988
23. B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, 29(3):155-163, October, 1988.
24. C. C. Lee, P. C. Chung, J. R. Tsai, and C. I. Chang, "Robust radial basis function neural networks," *IEEE Transactions on Systems, Man, and Cybernetics: Part B Cybernetics*, 29(6):674-685, December, 1999.
25. B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 15-26, Chicago, Illinois, USA, June, 2005.
26. G. F. Lin and L. H. Chen, "Time series forecasting by combining the radial basis function network and the self-organizing map," *Hydrological Processes*, 19(10):1925-1937, June, 2005.
27. C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: a hypothesis testing-based approach," *IEEE Transactions on Software Engineering*, 32(10):831-848, October, 2006.
28. J. R. Lyle and M. Weiser, "Automatic program bug location by program slicing," in *Proceedings of the Second International. Conference on Computer and Applications*, pp. 877-883, Beijing, China, June, 1987.
29. J. Moody and C. J. Darken, "Learning with localized receptive fields," in *Proceedings of Connectionist Models Summer School*, pp. 133-142, 1988.
30. K. S. Narendra and S. Mukhopadhyay, "Intelligent control using neural networks," *IEEE Control System Magazine*, 12(2):11-18, April, 1992.
31. D. E. Neumann, "An enhanced neural network technique for software risk analysis," *IEEE Transactions on Software Engineering*, 28(9):904-912, September, 2002.
32. J. Park and I. W. Sandberg, "Universal approximation using radial-basis-function Networks," *Neural Computation*, 3(2), 1991.
33. J. Park and I. W. Sandberg, "Approximation and radial-basis-function networks," *Neural Computation*, 5(2):305-316, March, 1993.
34. R. Penrose, "A generalized inverse for matrices," *Proceedings of the Cambridge Philosophical Society*, 51: 406-413, July, 1955.
35. M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Proceedings of the 18th International Conference on Automated Software Engineering*, pp. 30-39, Montreal, Canada, October, 2003.
36. T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the Year 2000 problem," in Proceedings of the 6th European Software Engineering Conference, pp. 432-449, Zurich, Switzerland, September, 1997
37. C. D. Sterling and R. A. Olsson, "Automated bug isolation via program chipping," in Proceedings of the 6th International Symposium on Automated Analysis-Driven Debugging, pp. 23-32, Monterey, California, USA, September 2005
38. Y. S. Su and C. Y. Huang, "Neural-network-based approaches for software reliability estimation using dynamic weighted combinational models," *Journal of Systems and Software*, 80(4):606-615, April, 2007.
39. N. Tadayon, "Neural network approach for software cost estimation," in *Proceedings of the International. Conference on Information Technology: Coding and Computing*, pp. 815- 818, Las Vegas, Nevada, USA, April, 2005.
40. C. Wan and P.B. Harrington, "Self-configuring radial basis function neural networks for chemical pattern recognition," *Journal of Chemical Information and Modeling*, 39(6):1049-1056, November, 1999.
41. T. Wang and A. Roychoudhury, "Automated path generation for software fault localization," in Proceedings of 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 347-351, Long Beach, California, USA, November 2005
42. P. D. Wasserman, *Advanced Methods in Neural Computing*, Van Nostrand Reinhold, 1993.
43. M. Weiser, "Programmers use slices when debugging," *Communications of the ACM*, 25(7):446-452, July, 1982.
44. I. Vessey, "Expertise in debugging computer programs," in International Journal of Man-Machine Studies: A Process Analysis, 23(5):459-494, 1985
45. W. E. Wong, V. Debroy and B. Choi, "A family of code coverage-based heuristics for effective fault localization," Journal of Systems and Software, 83(2): 188-208, February 2010
46. W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of Test Set Minimization on Fault Detection Effectiveness," *Software-Practice and Experience*, 28(4):347-369, April 1998
47. W. E. Wong and Y. Qi, "Effective program debugging based on execution slices and inter-block data dependency," *Journal of Systems and Software*, 79(7):891-903, July, 2006.
48. W. E. Wong, T. Sugeta, Y. Qi, and J. C. Maldonado, "Smart debugging software architectural design in SDL," *Journal of Systems and Software*, 76(1):15-28, April, 2005.
49. W. E. Wong, Y. Qi, Y. Shi and J. Dong "BP neural network-based effective fault localization," *International Journal of Software Engineering and Knowledge Engineering* (to appear). An earlier version appeared in the *Proceedings of the 19th International Conference on Software Engineering and Knowledge Engineering*, pp. 374-379, Boston, Massachusetts, USA, July, 2007.
50. χSuds User's Manual, Telcordia Technologies, 1998.
51. A. Zeller, "Isolating cause-effect chains from computer programs," in *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pp. 1-10, Charleston, South Carolina, USA, November, 2002.
52. A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, 28(2):183-200, February, 2002.
53. X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proceedings of the 28th International Conference on Software Engineering*, pp. 272-281, Shanghai, China, May, 2006.
54. X. Zhang, H. He, N. Gupta, and R. Gupta, "Experimental evaluation of using dynamic slices for fault location," in Proceedings of the 6th International Symposium on Automated Analysis-driven Debugging, pp. 33-42, Monterey, California, USA, September 2005