Learning Universal Probabilistic Models for Fault Localization

Min Feng Rajiv Gupta
University of California at Riverside, CSE Department, Riverside CA, 92521
{mfeng, gupta}@cs.ucr.edu

Abstract

Recently there has been significant interest in employing probabilistic techniques for fault localization. Using dynamic dependence information for multiple passing runs, learning techniques are used to construct a probabilistic graph model for a given program. Then, given a failing run, the probabilistic model is used to rank the executed statements according to the likelihood of them being faulty. In this paper we present a novel probabilistic approach in which universal probabilistic models are learned to characterize the behaviors of various instruction types used by **all programs**. The universal probabilistic model for an instruction type is in form of a probability distribution that represents how errors in the input (operand) values are propagated as errors in the output (result) of a given instruction type. Once these models have been constructed, they can be used in the analysis of **any program** as follows. Given a set of runs for any program, including at least one passing and one failing run, a Bayesian network called the *Error Flow Graph* (EFG) is then constructed from the dynamic dependence graphs of the program runs and the universal probabilistic models. Standard inference algorithms are employed to compute the probability of each executed statement being faulty. We also present optimizations to reduce the runtime cost of inference using the EFG. Our experiments demonstrate that our approach is highly effective in fault localization even when very few passing runs are available. It also performs well in the presence of multiple faults.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Experimentation, Verification

1. Introduction

Software often contains many faults since developing software is a human-intensive activity. Software debugging is the process of locating and correcting faulty program statements. Unfortunately, debugging can be a difficult task. The point of program failure is typically different from the faulty statement responsible for the failure. Thus, identifying the possible fault locations occupies most time of the debugging phase. Techniques developed to help automate the fault localization process to assist the developer include delta debugging [8, 28], dynamic program slicing [10, 16, 30], state altering techniques [13, 25, 29], and probabilistic techniques [5, 15, 17–19, 23].

Probabilistic techniques have proven to be effective in prioritizing the possible faulty statements with relatively low time cost. Most existing probabilistic techniques [5, 7, 17, 18] work as follows: (1) two statistical profiles (e.g., path or predicate) are created — one for the passing runs and the other for the failing runs; (2) the two profiles are compared by using probabilistic methods; and (3) faulty statement is located based on the differences between the two profiles. Following this procedure, most existing probabilistic techniques require many distinct passing runs and failing runs to make the statistical profiles accurate. However, since test suites are not often available, the number of program runs available for debugging is often limited in practice. This paper targets at using only a few passing runs and failing runs for fault localization.

We present a novel probabilistic method which differs from the previous approaches in one significant way. While the previous techniques employ the learning process to each individual program, our approach uses learning *once* and then applies its results in analyzing *all* programs. This approach is enabled by the following key observation: on a given platform, all programs are constructed from the same basic instruction types; thus learning techniques can be applied *once* to develop *universal probabilistic models* for the behavior of these instruction types and then these models can be used in the analysis of program runs of *any program*. In our probabilistic method, each universal model can be seen as a probabilistic error transfer function, describing how an instruction type propagates errors. The universal models are learned and fixed before debugging. Given a few runs of a faulty program, we construct a graph that describes how errors are propagated in the faulty program. By applying probabilistic inference techniques in the graph, we calculate the fault probability for each statement in the program. When the universal probabilistic models are used during fault localization for a specific program, we simply require at least one passing and one failing run. To the best of our knowledge, this is the first paper that applies probabilistic inference models for fault localization.

We have implemented our technique and evaluated it using the Siemen's suite [12]. Experimental results indicate that our approach to learning and inference is very effective. Our method using 5 passing and 5 failing runs works better than Sober [18] using all passing and failing runs. The following observations show the effectiveness and versatility of our approach:

- (Single Fault) While the effectiveness of fault localization increases with the number of available passing runs, our technique is highly effective even when only a *small number of passing runs* are available. It also effectively exploits availability of failing runs. Thus, our approach is both effective in exploiting positive evidence from passing runs and negative evidence from failing runs.

PASTE'10, June 5–6, 2010, Toronto, Ontario, Canada.

- (Multiple Faults) When multiple failing runs corresponding to different faults are analyzed simultaneously, as long as enough passing runs are available, our technique assigns a *high rank to at least one of the faulty statements*. Thus, multiple failing runs can be exploited simultaneously by our technique – if they correspond to the same fault, effectiveness of fault localization is improved while if they correspond to different faults, at least one of the faults is ranked high.

The remainder of the paper is organized as follows. In section 2 we provide background information on Bayesian Networks and illustrate their relevance for fault location. Section 3 presents the fault localization framework in detail. Section 4 describes the implementation details and section 5 presents the results of experimentation. Section 6 discusses related work and the conclusions are given in section 7.

2. Bayesian Network and Faults

Our probabilistic graph model is based upon a Bayesian Network (BN) derived from the Dynamic Dependence Graphs (DDGs) of program runs. In this section we provide a brief introduction to Bayesian Network and illustrate its suitability for fault localization.

A *Bayesian network*, also called a belief network, is a probabilistic graph model that represents a set of variables and their probabilistic independencies. It has been widely used in the machine learning community to represent the probabilistic relationship between cause and consequence.

Definition 1. The core of the *Bayesian network* representation is a directed acyclic graph, denoted as $BN(U, N, E, Pr)$, where U is a domain of random variables $x_1, \ldots, x_n$, N is a set of nodes each of which corresponds to a variable x_i in U, E is a set of edges which correspond to direct influence of one node on another, and Pr is a set of local distributions. If there is an edge from node x_i to another node x_j, x_i is called a parent of x_j, and x_j is a child of x_i. The set of parent nodes of a node x_i is denoted by $Pa(x_i)$. Each local distribution in Pr represents the probabilistic relationship between a node and its parents and can be written as $P(x_i|Pa(x_i))$. The joint distribution of all the variables is the product of the local distributions:

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i|Pa(x_i)).$$

If the value of a variable is observed, then the variable is said to be an evidence. A Bayesian network can be used to answer probabilistic queries about any variable based on a set of evidences. According to the Bayesian rule, the probability of variable x_i on the condition of evidence x_j can be written as follows:

$$P(x_i|x_j) = \frac{P(x_i, x_j)}{P(x_j)} = \frac{\sum_{U-x_i-x_j} P(x_1, \ldots, x_n)}{\sum_{U-x_j} P(x_1, \ldots, x_n)}.$$

This process of computing the posterior distribution of variables from given *evidence* is called *probabilistic inference*. Many proven Bayesian network inference algorithms have been published in the machine learning community. In this paper, we use two popular inference algorithms: the clustering algorithm [11] and the EPIS sampling algorithm [27].

Now let us consider a simple example that illustrates the potential of using Bayesian probabilistic model to estimate the probability of a statement being faulty. Fig. 1(a) shows a piece of code that contains only one branch statement and one assignment statement. Fig. 1(b) shows the Bayesian network of two program runs of the sample code. Both program runs go through the statement 2. In the Bayesian network, the nodes "S1" and "S2" denote the correctness

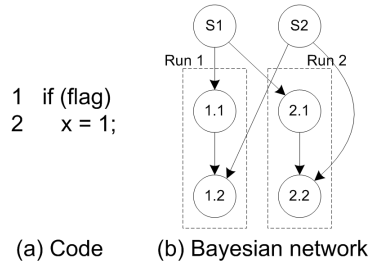


Figure 1. Example.

of statement 1 and 2 and the node "$i.j$" denotes the correctness of the execution instance of statement j in the i-th program run. According to the figure, the correctness of instance 1.1 and 2.1 only depends on statement 1 while the correctness of instance 1.2 and 2.2 depends on not only statement 2 but also the previous instances. For simplicity, we ignore the data dependence on variable *flag*. Our goal is to find the faulty statement based on the correctness of variable *x* in these two program runs, assuming that *x* is an output. We can calculate the probability of statement 1 being faulty by using the following equation:

$$\begin{aligned} & P(S_1|V_1^2, V_2^2) \\ \propto \; & P(S_1, V_1^2, V_2^2) \\ = \; & \sum_{S_2, V_1^1, V_2^1} P(S_1, S_2, V_1^1, V_2^1, V_1^2, V_2^2) \\ = \; & \sum_{S_2, V_1^1, V_2^1} \{P(S_1)P(S_2)P(V_1^1|S_1)P(V_2^1|S_1) \\ & P(V_1^2|V_1^1, S_2)P(V_2^2|V_2^1, S_2)\} \end{aligned}$$

where S_i stands for the correctness of statement "i", V_i^j stands for the correctness of instance "$i.j$", $P(S_1|V_1^2, V_2^2)$ stands for the probability of statement "1" being correct/faulty by given the correctness of instance 1.2 and 2.2, and all the probabilistic distributions above are learned in advance. We can also use the same way to calculate the fault probability of statement 2. To further explain the rationale behind our fault localization method, let us assume:

$$\begin{aligned} & P(S_1 = f) = P(S_2 = f) = 0.1, \\ & P(V_i^1 = w|S_1 = f) = 0.5, \; and \\ & P(V_i^2 = w|V_i^1 = w) = P(V_i^2 = w|S_2 = f) = 1.0. \end{aligned}$$

where *w* and *f* mean *wrong* and *faulty*, and i is 1 or 2. If we observe the outputs of both execution instance 1.2 and 2.2 are wrong, then we get the following probabilities:

$$\begin{aligned} P(S_1 = f|V_1^2 = w, V_2^2 = w) \quad & \propto 0.0325 \\ P(S_2 = f|V_1^2 = w, V_2^2 = w) \quad & \propto 0.1 \end{aligned}$$

Thus, the results show the statement 2 is more likely to be faulty in this case. If only execution instance 1.2 is wrong in the two program runs, then the probabilities will be:

$$\begin{aligned} P(S_1 = f|V_1^2 = w, V_2^2 = c) \quad & \propto 0.0225 \\ P(S_2 = f|V_1^2 = w, V_2^2 = c) \quad & \propto 0.0 \end{aligned}$$

In this case, it is impossible for statement 2 to be faulty because if statement 2 is faulty then both instances have to be wrong. Thus, statement 1 will be the only possible faulty statement. The example shows that what our method does is essentially to find the statement whose fault maximizes the probability of the observation. The example illustrates how Bayesian model can help us locate faulty

statements. It is a good way to model the error propagation in a program. Let us now consider our framework for fault localization.

3. Fault Localization Framework

The main purpose of our approach is to prioritize the statements according to their probabilities of being faulty. A probabilistic graph model, called the *Error Flow Graph* (EFG), which is based on a Bayesian network that formalizes the propagation of error in program runs, is used to estimate the fault probability of each statement. This framework consists of three main steps: (*Learning*) in this step the parameters of probability distribution of each instruction type are learned; (*EFG Construction*) given a set of runs for a program, we build the Dynamic Dependence Graphs (DDGs) for the runs and convert them into an EFG; and (*Fault Localization*) from the EFG this step estimates the fault probability of each statement. The rest of this section describes these steps in detail and illustrates them using a sample program.

3.1 Universal Probabilistic Models

Our probabilistic model consists of a set of probability distributions, one distribution for each instruction type, such that it represents the probability with which errors in an instruction's input values propagates to its output. The output of an instruction depends upon its operands (i.e., dependences) as well as the statement itself. Our model takes as input the correctness of these factors and predicts the correctness of the output.

Often probabilistic models use a conditional probability table to represent the local probability distribution for an instruction in a given program. However, these tables are not suitable for our model because we need to represent the probabilistic distribution corresponding to an instruction type used by all programs. The number of (input) dependences of an instruction type varies from program to program and thus building a single conditional probability table to represent the distribution is not possible. Therefore instead of using a conditional probability table, we design a special function to represent the probabilistic distribution for each instruction type. First, we define the following function to stand for the probability of the output being wrong given the correctness of inputs of an instruction type:

$$f(n_c, n_w) = \lambda(1 - e^{-\alpha \frac{n_w}{n_c + n_w}})$$

where n_c is the number of correct parents, n_w is the number of wrong or faulty parents, and λ and α are parameters to be *learned for each instruction type*. Since the function can be used in any situation regardless of the number of (input) dependences of an instruction, the learned probabilistic distribution is applicable in all programs. The function is based upon the idea that the probability of the output being wrong increases with the percentage of incorrect values in the inputs. When all the parents (including statement nodes) are correct, its value is 0, i.e., the instance must be correct. When all the parents are faulty or wrong, the function reaches its peak value. Then the probabilistic distribution of each instruction type is defined as follows:

$$P(x_i|Pa(x_i)) = \begin{cases} f(n_c, n_w) & x_i = \text{wrong} \\ 1 - f(n_c, n_w) & x_i = \text{correct} \end{cases}$$

The learning of the probabilistic model for a given instruction type consists of two steps: generating the *training data* for the instruction type; and using the training data to derive an approximation of the *parameters* (λ, α). Let us discuss these steps in greater detail.

Collecting Training Data. Given an instruction type, the training data is generated by repeatedly applying the following process. Inputs are generated randomly for an instruction type and the corresponding correct output is calculated. Next, rules are used to perform changes to the inputs and/or the instruction, the instruction is reevaluated, and the correctness information of the operands, instruction, and the output is recorded. This process is applied repeatedly and information is recorded. The collected data essentially characterizes the behavior of an instruction type under different types of errors.

Type	Example Instructions	Replacement Rules
logical	AND, OR	operator & operand
compare	CMP, SCASB	operand
branch	JA, JB, JNE	operator & operand
arithmetic	ADD, SUB	operator & operand
move	MOV, STOSB	operand
shift	SAL, SHL, ROL	operator & operand
other	CLC, CLI, PUSH	operand

Table 1. Seven instruction types and their rules.

Next we discuss the various instruction types and the rules that are used in performing changes during the generation of training data. Our work is performed in context of the Intel x86 instruction set and we divide the instructions into seven instruction types shown in Table 1. These types are formed based upon the type of operations the instructions do which include: logical operations, comparison operations, branches that test conditions, arithmetic operations, move operations, shift operations, and others. Table 1 also shows the *replacement rules* that are applicable to each instruction type. Before we discuss which rules can be applied to which instruction type, we discuss the two types of rules used:

- *Operand replacement*. We generate random values to substitute all or a subset of inputs. The new values should be in the same domain as the original inputs. This rule simulates the situation that the input values are wrong. The generated data from such a replacement allows us to infer how probable it is for the output to be wrong if the inputs are partially or fully wrong.
- *Operator replacement*. We use another instruction of the same type to substitute the current instruction. The new instruction has the same number of inputs as the original one. This rule simulates operator mutation. The generated data allows us to infer how probable it is for the output to be wrong if the instruction (and the statement it corresponds to) itself is faulty.

Both types of rules are not applicable to all instruction types as shown in Table 1. In particular, the *operator replacement* rule is not applicable to three instruction types: *move*, *compare*, and *other*. There are a number of reasons for this. In case of *move* instructions no operator is involved — the compiler generates them to simply alter the location of a value or make a copy of a value. In case of most instructions in type *other*, there is no adequate replacement for an instruction as the other instructions have different number and/or type of inputs. In other cases, operations are generated by the compiler to clear flags etc. Finally, in the case of *compare* due to its semantics for x86 there is no possible replacement. The compare operation performs all possible relational operations and puts the results in bits of the *flag* register. Conditional branch instructions test the bit for the appropriate relation operator; thus, operator replacement is applied to conditional branch instructions.

Finding Parameters. After generating the training data, we can learn the model by adjusting the parameter of the function to approximate the training data. Alg. 1 learns the parameters of the function $f(n_c, n_w)$ from a set of training data. Each training data is a triple (n_c^i, n_w^i, d^i), where d^i is the correctness of an instance node given n_c^i correct parents and n_w^i wrong parents. Alg. 1 enumerates

Algorithm 1: Learn parameters.

Input: a set of training data (n_c^i, n_w^i, d^i)
Output: λ and α
begin
  for *each possible* n_c **do**
    for *each possible* n_w **do**
      $num_c[n_c, n_w] \leftarrow 0$;
      $num_w[n_c, n_w] \leftarrow 0$;
    end
  end
  for *each* (n_c^i, n_w^i, d^i) **do**
    if d^i = *correct* **then**
      $num_c[n_c, n_w] \leftarrow num_c[n_c, n_w] + 1$;
    else
      $num_w[n_c, n_w] \leftarrow num_w[n_c, n_w] + 1$;
    end
  end
  for *each possible pair* (λ, α) **do**
    $\delta \leftarrow \sum_{n_c, n_w} (f(n_c, n_w) - \frac{num_w[n_c, n_w]}{num_c[n_c, n_w] + num_w[n_c, n_w]})^2$;
    if $\delta < lowest$ **then**
      record (λ, α);
      $lowest \leftarrow \delta$;
    end
  end
end

all possible parameters for which we keep 4 digits after the decimal point and chooses the one which minimizes the deviation between training data and learned function.

3.2 Error Flow Graph

In this section, we describe the Dynamic Dependence Graph (DDG) and its use in constructing the Error Flow Graph (EFG). The DDG is a directed acyclic graph representing dependences between different execution instances of statements in a program run.

Definition 2. The DDG of a program run, $DDG(N, E)$, consists of a set of nodes N and a set of directed edges E where: each node $n_i \in N$ represents the i^{th} execution instance of statement n in the program; and each edge $m_j \rightarrow n_i \in E$ corresponds to a dynamic data or control dependence of the i^{th} execution instance of statement n on the j^{th} execution instance of statement m.

Fig. 2(b) shows an execution of the sample code (shown in Fig. 2(a)) that follows the path corresponding to the false evaluation of the predicate at node 2. The value shown to the right of each statement is the value computed by the statement during the runtime. The DDG of this run is shown in Fig. 2(c) — the solid (dotted) edges are data (control) dependence edges.

Next, we present the definition of an EFG and the algorithm for constructing it from the DDG. The nodes in a DDG only represent the instances of the statements and thus mining a DDG does not naturally result in the probability of a statement being faulty. Therefore, to infer the fault probability of a statement, we introduce the EFG which not only represents the statement instances but also the statements themselves.

Definition 3. An Error Flow Graph (EFG) for a program run is a 4-tuple (S, I, E, Pr) where S is a set of statement nodes which represent the statements in a program, I is a set of instance nodes which correspond to the statement instances in a program run, E is a set of edges which correspond to direct dependence of one node on another, and Pr is a set of probability distributions which stand for the probability of one node given the value of its parents. Each statement has a unique node in the EFG.

An EFG is a Bayesian network, representing the conditional independencies between statements and instances. In the EFG, each statement node takes two values, correct and faulty, which indicate whether the statement is faulty. Each instance node also takes two values, correct or wrong, which indicates whether the value produced by the statement instance is correct or corrupted. The edges between instance nodes means that the correctness of one instance affects its children, while the edges from statement nodes to instance nodes means that the correctness of an instance also depends on its corresponding statement nodes.

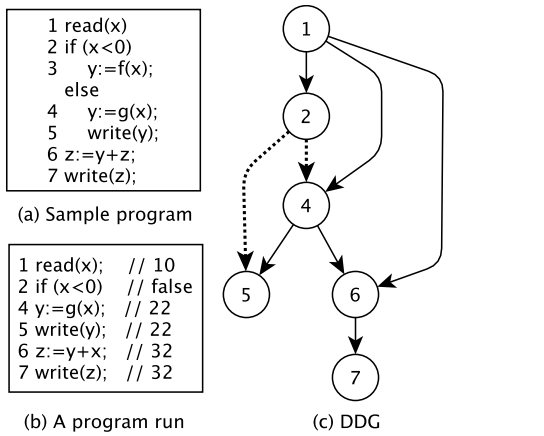


Figure 2. Example program.

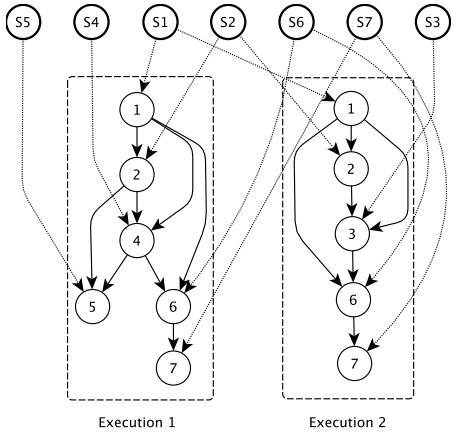


Figure 3. EFG of two program runs.

Fig. 3 shows the EFG of two runs of the sample program shown in Fig. 2. One run takes the path corresponding to the true evaluation of the predicate at node 2 (as shown in Fig. 2(b)) while the other run follows the path corresponding to the false evaluation. The EFG contains all the nodes from the DDGs of both runs and has a distinct statement node for each statement. The statement nodes are marked by the bold circles. Each instance node has incoming edges from its parents in the DDG and the corresponding statement node, which forms a probabilistic distribution indicating that the correctness of the value produced by the statement instance depends on the input values and the statement itself. For example in execution 1, node 2 has incoming edges from node 1 and node S2, which forms a distribution $P(V_2|V_1, S_2)$ that indicates the correctness of the value produced by node 2 is probabilistically determined by the correctness of the value produced by node 1 and the correctness of the statement 2. The EFG describes how the errors propagates in the program run. Inference in the EFG is like tracing back from the leaf nodes (where we observe the correctness of the outputs) along the error flows to find the statement nodes

that have the highest probability of causing the errors. Intuitively, by using multiple runs, we can get better estimation since more evidence is observed.

Optimizations. Usually, there are thousands of instances in a program run. Correspondingly, the constructed EFG contains thousands of nodes and edges. Inference in such a large network can be time consuming. To speed up the inference process, we propose two optimization techniques.

First, in practice, we do not need to estimate the probability for all the statements, since many statements are unrelated to the evidence. A dynamic slice of a statement instance x_i, denoted by $DS(x_i)$, is the subgraph of the EFG which includes node x_i as well as all other nodes and edges from which x_i is reachable. This is because the faulty statements are always in the dynamic slices of the evidences. In other words, all the statements outside the dynamic slices are removed from the EFG.

Second, most nodes in an EFG are built for library code. We assume that the library code is correct and the fault is in the user code. Therefore, we do not perform inference on the corresponding nodes. However, we cannot simply remove the library nodes from the EFG because it will break the dependences in the original EFG. Instead, we compress the consecutive library nodes on a dependence path into one node. This greatly reduces the number of nodes in an EFG while keeping all the dependences between non-library nodes in the original EFG.

3.3 Fault Localization

An EFG is a Bayesian network which models the error propagation in a set of program runs. Therefore, we can apply an inference technique on the EFG to find the root cause of the error. This process is made up of the following steps. First, we initialize the evidences associated with the EFG. An evidence is the correctness of a particular instance node. Most evidences can be obtained by inspecting the output of a program run. If a statement instance outputs a wrong value, its corresponding node is set to be wrong as an evidence in the EFG, and vice versa. If a programmer has additional knowledge about the program, he/she can also create additional evidences. Next we can estimate the probability of each statement being faulty, i.e., we compute the results of the following equation:

$$P(S_i = \text{faulty} | \text{evidence}).$$

After estimating the probabilities of all statements as above, in the final step we rank the statements in descending order of their probabilities.

In our implementation we considered two popular inference algorithms to estimate these probabilities: clustering algorithm [11] and EPIS algorithm [27]. The clustering algorithm is widely used for exact inference in Bayesian networks. Although this greedy algorithm works well in practice, it may incur high time overhead for very complicated networks. Therefore, we also use the state-of-art EPIS algorithm which is a sampling based algorithm used for approximate inference.

4. Experimental Evaluation

4.1 Experimental Setup

We implemented a fault localization framework which takes the C source code of the programs and uses *gcc* generated Intel x86 binaries. Then we used the *Diablo* [1] binary rewriting framework to construct the static control flow graph from the Intel x86 binaries. After that, it uses a *Valgrind*[2]-based dynamic tracing tool developed by Zhang and Gupta [31] to execute the binaries and build the dynamic data and control dependence graphs. The DDGs are then converted into an EFG. The evidences in the EFG are initialized by comparing the actual outputs with the standard outputs. Finally, our implementation calls *SMILE* (Structural Modeling, Inference, and Learning Engine) library [3] to estimate the fault probability of each statement and outputs the sorted statement list.

The Siemens programs [12] are used for our experiments. It is a set of programs commonly used to measure the effectiveness of fault-localization techniques. Table 2 shows the characteristics of the seven Siemens programs. All Siemens faulty versions contain seeded faults. Most faults are related to computation, including operator and operand mutations, missing and extraneous code, and constant value mutation. Most faulty versions have only one faulty statement, but some are seeded with several faults in different statements. We excluded a few faulty versions due to the following two reasons: (1) they did not produce any failing runs from the provided test cases; (2) one of faulty versions from printtokens loops indefinitely far past the end of a string.

Program	LOC	# Ver.	# test cases
tcas	138	41	1608
replace	516	31	5542
printtokens	402	7	4130
printtokens2	483	9	4115
schedule	299	9	2650
schedule2	297	9	2710
totinfo	346	23	1052

Table 2. The Siemens benchmark programs. From left to right: program name, # lines of code, # faulty versions and # test cases.

In our experiments, we also compare the fault localization effectiveness of our method with Tarantula [15] and Sober [18]. This is because they have proved to be quite effective on the Siemens suite programs [15]. We also compare the results of exact Bayesian network inference (the clustering algorithm [11]) and approximate Bayesian network inference (EPIS algorithm [27]).

In our experiments, we rank only those program statements that are in the backward slice of the wrong inputs. To evaluate the techniques, we assign a score to each ranked set of statements that is the percentage of program statements executed by failing runs in the test suite that *need to be examined* if statements are examined in rank order. Suppose that for a ranked list of statements S, the actual faulty statement occurs at rank r and there are a total of $|S|$ statements exercised by failing runs. Then the score of ranked statement list S is defined as follows:

$$score(S) = \frac{r}{|S|} \times 100\%.$$

A lower score is preferable because it means that more of the statements executed by failing runs are ignored before the faulty statement is found.

In the experiments, our goal is to determine the effectiveness of our method when only a few runs are available. Therefore, every time we ranked the executed statements for a faulty version, we just used a small portion of the test cases. We ranked the statements of each faulty version one hundred times. Each time, we randomly selected a few test cases for statement ranking. Thus, each faulty version has a hundred ranked sets of statements. The percentages shown in the experimental results are the percentages with respect to all ranked sets of statements, rather than the number of faulty versions.

There are a few special considerations that we make in our experiments for certain faults. First, macro definitions and variable declarations will not appear in DDG traces. Therefore, any fault in those places will not be exposed by doing inference in the EFG. However, we can find that the faulty probability of the statements related to those declarations is very high. We consider these kinds

of faults to be examined if we examine the statement where the faulty macro or variable is used. Second, faults that involve omitted statements will mean that we cannot actually examine the missing code. However, we can examine statements that are adjacent to the location where the code is missing.

4.2 Learning Parameters

We learned the parameters for each type of instruction in Intel x86 instruction set from randomly generated data sets using the method presented in Section 3.1. For each type of instruction, we generated the training data. Table 3 shows the learned parameter values and learning time for each type of instruction. As we can see, it took only a few hours to learn the probabilistic models.

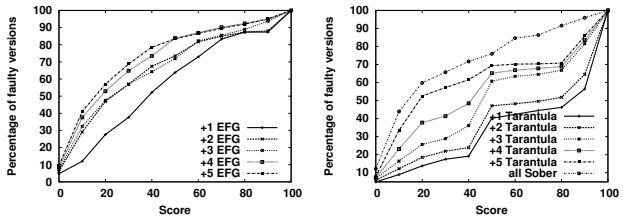
	λ	α	Time (min)
logical	0.5978	16.4777	31
compare	0.5617	17.5729	30
branch	1.0451	3.1431	22
arithmetic	1.0000	16.9250	37
move	1.0000	30.0000	30
shift	1.0000	19.6437	28
other	1.0000	30.0000	19

Table 3. Parameters for probability functions and learning time.

These parameters show that the "logical" and "compare" type of instructions will probably produce correct outputs even though their inputs are partially wrong while "move" and "other" type of instructions will definitely produce wrong outputs if any input is wrong. In other words, an error will probably be propagated via "move" and "other" type of instructions while it may be canceled out via "logical" and "compare" type of instructions.

4.3 Single Fault

Usually, a software can contain multiple faults. Different failing runs may be caused by different faults. Therefore, a conservative way for fault localization is to use only one failing run at a time and analyze it with the help of multiple passing runs. The goal of our first study is to determine the effectiveness of our method using a single failing run for a single-fault program. We excluded the faulty versions that are seeded with multiple faults in this study. Fig. 4 shows the cumulative percentage of all ranked sets of statements in each score range computed by our method and Tarantula using different numbers of passing runs (denoted as +1 to +5). In the graph, the x-axis represents the upper bound of each score range, and the y-axis represents the percentage of ranked sets of statements achieving a score lower than or equal to that upper bound.



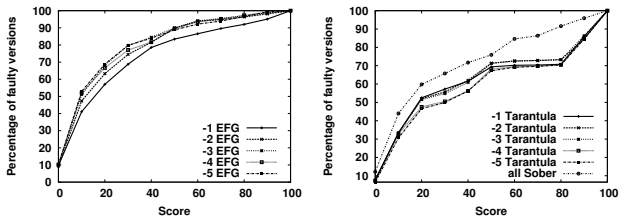
(a) EFG (b) Tarantula & Sober

Figure 4. Single fault & Single failing run.

The data shows that the overall our method performs better than the Tarantula approach when only a few passing runs are available. When using five passing runs, about 41% of the ranked sets of statements had a score of 10% or lower with our method, whereas the same was true for only about 33% using Tarantula. Both techniques achieve better results as more passing runs are available. *However, the main conclusion of this study is that our approach does not require a large test suite as a few runs is sufficient for effective fault location.*

Fig. 4 also shows the results of Sober by using all the test cases in Siemens suite. Although Sober uses more test cases than our method does, our method still outperforms it. Although Sober is excellent at localizing the faults on predicate statements, it does not deal very well with the non-predicate faulty statements.

We also examined the effectiveness of our method using multiple failing runs for the same fault. We excluded the faulty versions that are seeded with multiple faults. The number of passing runs used in this study is five. Fig. 5 shows the results with our method and Tarantula on different numbers of failing runs (denoted as -1 to -5).



(a) EFG (b) Tarantula & Sober

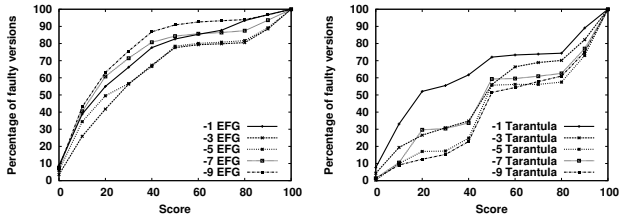
Figure 5. Single fault & Multiple failing runs.

By using multiple failing runs for the same fault, our method still performs much better. About 10% of the ranked sets of statements had a score of 1% or lower with our method, while Tarantula only achieved 6% in the same score range. Similarly, almost 53% of ranked sets of statements had a score of 10% or lower using our method, whereas the same was true for about 30% using Tarantula. Even though our method was able to uniquely identify the faulty statement in 29 cases, only 13 cases yielded scores of 1% or less. This is because in the other 16 cases, the number of statements in the backward slice of the wrong inputs was few enough that even a rank of 1 would lead to a score larger than 1%. Moreover, our method achieved better results using more failing runs, while Tarantula became worse if more failing runs were used. This is because when more failing runs are used, it is more likely that some failing runs will go through a few statements that are never touched by any passing run and are not faulty. According to the Tarantula suspiciousness formula, these statements have the highest suspiciousness. Therefore, the rank of the actual faulty statement drops when more failing runs are applied. *The main conclusion of this study is that our technique exploits both positive and negative evidence effectively.*

4.4 Multiple Faults

In this experiment, we measure the ability of our method when dealing with the programs containing multiple faults. The benchmarks used in this study contain: (1) faulty versions that are seeded with multiple faults; and (2) combinations of two faulty versions with single faults. We only combined two single-fault faulty versions if their version numbers are consecutive. In this study, we calculated the score for a ranked set of statements by using the faulty statement with higher rank. For example, suppose there are two faulty statements in a faulty version. The rank of one statement is 3 and the other's rank is 16. The final score of the ranked set of statements is equal to 3 divided by total number of executed statements. This is reasonable since our aim is to find at least one faulty statement in the faulty version no matter what faulty statement it is. Fig. 6 shows the experimental results with our method and Taran-

tula using five passing runs and varying number of failing runs (-1 to -9).



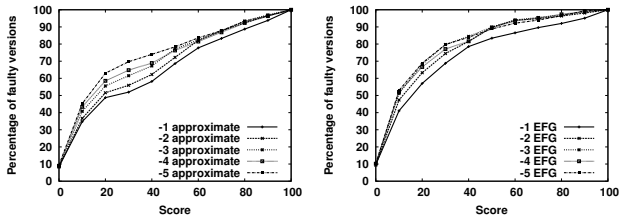
(a) EFG (b) Tarantula

Figure 6. Multi-fault programs.

The data shows that our method performs much better than Tarantula when dealing with multiple faults. About 43% of ranked sets of statements had a score of 10% or lower while the same was true for only about 9% using Tarantula. When less than 10% of the code must be examined, our technique is approximately 5 times more effective than Tarantula. As the number of analyzed failing runs increases, both methods achieved worse results first and then became better and better. But to get good results on multi-fault benchmarks, our method needs more program runs than that required for single-fault programs. *The main conclusion of this study is that our technique is resilient as it functions effectively even if the failing runs exercise different faults.*

4.5 Approximate vs. Exact Inference

The above experimental results are computed by exact Bayesian network inference (the clustering algorithm). Since the theoretical time complexity of exact inference is exponential to the size of DDG, it may be slow for large commercial software which is much larger than Siemens programs, even though the greedy strategy in the clustering algorithm works very well in practice. Thus, we determine the effectiveness of our method with approximate inference in this study. Fig. 7 shows a cumulative graph view of the percentage of ranked sets of statements in each score range with approximate inference as well as exact inference when using five passing runs and varying the number of failing runs.



(a) Approximate (b) Exact

Figure 7. Approximate vs. Exact inference.

The data shows that the results using approximate inference are only slightly worse than that using exact inference. In fact these results are still better than the results of Tarantula. When using the approximate inference, about 45% of ranked sets of statements had a score of 10% or lower and about 70% had a score of 50% or lower. When less than 10% of the code must be examined, the approximate inference technique is worse by about 15% than using the exact inference technique, but still almost 1.5 times more effective than Tarantula. Therefore, for larger software, we can use approximate inference to prioritize the statements at lower cost but still achieve reasonably good results.

Program	DDG	Exact Infer.	Approx. Infer.	Tarantula	Sober
tcas	5.89	0.52	0.49	0.254	6.73
replace	7.76	35.73	23.22	0.459	37.97
printtokens	7.65	5.26	4.13	0.966	34.29
printtokens2	7.82	5.73	4.15	0.964	35.53
schedule	20.25	103.01	59.45	0.235	31.28
schedule2	26.52	109.92	70.37	0.247	33.98
totinfo	19.45	67.31	32.95	0.216	12.61

Table 4. Efficiency of techniques in seconds.

To compare the efficiencies of approximate and exact inferencing algorithms we measured their execution times — this study was conducted on a Dell PowerEdge 1900 server with two Intel Xeon quad-core processors at 3.00 GHz, 16 GB of RAM. Table 4 summarizes the results of the study. The columns show the programs, the average time taken to build DDGs for 5 passing and 5 failing runs, the average computation time with exact inference, the average computation time with approximate inference, the average time for Tarantula, and the average time for Sober (using all test cases). All the timings are in seconds. First we see that the time taken by our approach is reasonable. Second we can see that for larger traces, such as those of schedule and schedule2, using approximate inference can save more than 40% time over using exact inference. Finally, as we can see, the time for Tarantula is significantly smaller. However, given the superiority of scores produced by our approach, we believe that the added cost is well justified. Our approach in the long run saves time as it will reduce the time the developer spends on debugging.

5. Related Work

The most closely related works to our paper are those which perform fault localization using probabilistic program behavior models. The work of Liu et al. [18, 19] applies probabilistic models to analyze the behavior of predicates in passing and failing runs (i.e., they consider control dependences). Their method builds a distribution for the outcomes of each predicate in both passing runs and failing runs and locates the fault by comparing the distributions in passing runs with those in failing runs. They suggest if the behavior of a predicate in a failing run is significantly different from that in passing run, it is probably relevant to the failure. Probabilistic program dependence graph (*PPDG*) [5] compares the dependence behavior in passing runs with that of a failing run. It extends the previous work to model not only control dependences but also data dependences. As we already mentioned, while these works learn the distributions for the dependences of each statement in a program, our approach builds a general model for each instruction type and then uses it for all programs. Moreover, our approach uses the standard Bayesian network inference techniques to predict the faulty location.

Many other statistical techniques are proposed to identify bugs in programs. Liblit et al. [17] proposed a method which uses sampling to collect data during program execution and identifies predicates which are relevant to bugs. *DieHard* [6] and *Archipelago* [20] are runtime techniques that randomizes the space allocated in a heap at least twice as large as required to prevent heap corruption and memory errors. *Exterminator* [21] is a system that automatically derives *runtime patches* to fix heap-based memory errors. Jiang and Su [14] proposed an approach to automatically generate a faulty control flow path by clustering correlated predicates. The faulty flow path can help users but it cannot provide the exact location of faulty statements. Chilimbi et al. [7] presented a statistical debugging tool called HOLMES that used path profiles instead of predicate profiles to isolate bugs.

Delta debugging [8, 28] is a debugging framework which locates the fault statements by analyzing the difference between a failing and a passing run. The approach simplifies the failing test case to a minimal test case that induces the failure and isolates the difference between a passing and a failing test case. *Delta debugging* can also isolate the cause-effect chains, which is a set of variables and values relevant to the failure [28]. The points where new relevant variables become the failure causes can be further identified to precisely locate the faulty code [8]. Groce [9] proposed a novel approach which aims to explain errors based on distance metrics for program executions. The work of Renieres and Reiss [23] uses the *nearest neighbor* metric to search for a passing run which resembles a failing run and identifies the part of code which is responsible for the failure.

Several works focus on fault localization by altering the internal states of a program run. *Predicate switching* [29] identifies the root cause of the failure by altering the control flow at runtime. [25] presented a similar approach for altering branch outcomes in a failing run to produce a passing run. The work of Jeffrey et al. [13], called *Value Replacement*, generalizes the previous work to altering the states at any point of a program run. Qin et al. [22] presented a tool called Rx that makes the program survive from a software failure by changing its external environment instead of the internal states.

Program slicing [24, 26] identifies a subset of statements which influence the variable at some point of a program. Dynamic slicing [4, 16] has been shown to be effective for debugging. Recent research has focused on minimizing the dynamic slice sizes by intersecting multiple slices [10]. Confidence-based analysis [30] was proposed to further prioritize the statements according to their likelihood of being faulty.

6. Conclusions

We presented a probabilistic approach for fault localization to assist in debugging. Learning was employed to construct universal probabilistic models that characterize the behaviors of various instruction types used by *all programs*. Using these models a Bayesian network called EFG is constructed based on a set of runs for *a program*. Standard inference algorithms are employed to compute the probability of each executed statement being faulty. The learning process above does not require the availability of large test suites. Our experiments show that for the Siemens benchmarks, our method is highly effective in ranking statements using a *small number of runs* — therefore, it does not require a large test suites for the program being considered. Besides, it is effective in exploiting both *failing and passing run* information (i.e., positive and negative evidence) and works well for *multiple faults*. Finally, the learning required to construct probabilistic models is carried out in a few hours for the Intel x86 instruction set.

References

[1] http://www.elis.ugent.be/diablo/.

[2] http://valgrind.org/.

[3] http://genie.sis.pitt.edu/.

[4] H. Agrawal and J. Horgan. Dynamic program slicing. In *PLDI*, pages 246–256, 1990.

[5] G. K. Baah, A. Podgurski, and M. J. Harrold. The probabilistic program dependence graph and its application to fault diagnosis. In *ISSTA*, pages 189–199, 2008.

[6] E. D. Berger and B. G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *PLDI*, pages 158–168, 2006.

[7] T. M. Chilimbi, B. Liblit, K. K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *ICSE*, pages 34–44, 2009.

[8] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, pages 342–351, 2005.

[9] A. D. Groce. *Error explanation and fault localization with distance metrics*. PhD thesis, 2005.

[10] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *ASE*, pages 263–272, 2005.

[11] C. Huang and A. Darwiche. Inference in belief networks: a procedural guide. *IJAR*, 15:225–263, 1996.

[12] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow and controlflow-based test adequacy criteria. pages 191–200, 1994.

[13] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In *ISSTA*, pages 167–178, 2008.

[14] L. Jiang and Z. Su. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *ASE*, pages 184–193, 2007.

[15] J. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*, pages 273–282, 2005.

[16] B. Korel and J. Laski. Dynamic program slicing. 29(3):155–163, 1988.

[17] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug isolation. In *PLDI*, pages 15–26, 2005.

[18] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: statistical model-based bug localization. *SIGSOFT Softw. Eng. Notes*, 30(5):286–295, 2005.

[19] C. Liu, X. Yan, and J. Han. Mining control flow abnormality for logic error isolation. In *SDM*, pages 106–115, April 2006.

[20] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn. Archipelago: trading address space for reliability and security. In *ASPLOS*, pages 115–124, 2008.

[21] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: automatically correcting memory errors with high probability. In *PLDI*, pages 1–11, 2007.

[22] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies - a safe method to survive software failures. In *SOSP*, pages 235–248, 2005.

[23] M. Renieres and S. Reiss. Fault localization with nearest neighbor queries. In *ASE*, pages 30–39, 2003.

[24] F. Tip. A survey of program slicing techniques. 3(3):121–189, 1995.

[25] T. Wang and A. Roychoudhury. Automatded path generation for software fault localization. In *ASE*, pages 347–351, 2005.

[26] M. Weiser. Program slicing. 10(4):352–357, 1984.

[27] C. Yuan and M. Druzdzel. An importance sampling algorithm based on evidence pre-propagation. In *UAI*, pages 624–631, 2003.

[28] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE*, pages 1–10, 2002.

[29] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *ICSE*, pages 272–281, 2006.

[30] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *PLDI*, pages 169–180, 2006.

[31] X. Zhang and R. Gupta. Whole execution traces. In *MICRO*, pages 105–116, 2004.