Capturing Propagation of Infected Program States*

Zhenyu Zhang
The University of Hong Kong
Pokfulam, Hong Kong
zyzhang@cs.hku.hk

W. K. Chan†
City University of Hong Kong
Tat Chee Avenue, Hong Kong
wkchan@cs.cityu.edu.hk

T. H. Tse
The University of Hong Kong
Pokfulam, Hong Kong
thtse@cs.hku.hk

Bo Jiang
The University of Hong Kong
Pokfulam, Hong Kong
bjiang@cs.hku.hk

Xinming Wang
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
rubin@cse.ust.hk

ABSTRACT
Coverage-based fault-localization techniques find the fault-related positions in programs by comparing the execution statistics of passed executions and failed executions. They assess the fault suspiciousness of individual program entities and rank the statements in descending order of their suspiciousness scores to help identify faults in programs. However, many such techniques focus on assessing the suspiciousness of individual program entities but ignore the propagation of infected program states among them. In this paper, we use edge profiles to represent passed executions and failed executions, contrast them to model how each basic block contributes to failures by abstractly propagating infected program states to its adjacent basic blocks through control flow edges. We assess the suspiciousness of the infected program states propagated through each edge, associate basic blocks with edges via such propagation of infected program states, calculate suspiciousness scores for each basic block, and finally synthesize a ranked list of statements to facilitate the identification of program faults. We conduct a controlled experiment to compare the effectiveness of existing representative techniques with ours using standard benchmarks. The results are promising.

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

General Terms: Experimentation, Verification

Keywords
Fault localization, edge profile, basic block, control flow edge

1. INTRODUCTION
Coverage-based fault-localization (CBFL) techniques [2][8][15][21][23][24] have been proposed to support software debugging. They usually contrast the program spectra information [14] (such as execution statistics) between passed executions and failed executions to compute the fault suspiciousness [24] of individual program entities (such as statements [15], branches [21], and predicates [18]), and construct a list of program entities in descending order of their fault suspiciousness. Developers may then follow the suggested list to locate faults. Empirical studies [2][15][18][19] show that CBFL techniques can be effective in guiding programmers to examine code and locate faults.

During program execution, a fault in a program statement may infect a program state, and yet the execution may further propagate the *infected program states* [9][22] a long way before it may finally manifest failures [23]. Moreover, even if every failed execution may execute a particular statement, this statement is not necessarily the root cause of the failure (that is, the fault that directly leads to the failure) [9].

Suppose, for instance that a particular statement *S* on a branch *B* always sets up a null pointer variable. Suppose further that this pointer variable will not be used in any execution to invoke any function, until another faraway (in the sense of control dependence [5] or data dependence) statement *S*′ on a branch *B*′ has been reached, which will crash the program. If *S* is also exercised in many other executions that do not show any failure, *S* or its directly connected branches cannot effectively be pinpointed as suspicious. In this scenario, existing CBFL techniques such as *Tarantula* [15] or *SBI* [24] will rank *S*′ as more suspicious than *S*. Indeed, in the above scenario, exercising *B*′ that determines the execution of *S*′ always leads to a failure [24]. Thus, the branch technique proposed in [24], for example, will rank *B*′ as more suspicious than *B*, which in fact is directly connected to the first statement *S*. The use of data flow analysis may reveal the usage of the null pointer and help evaluate the suspiciousness of *S*, *S*′, *B*, and *B*′. However, data flow profiling is expensive [14][21].

A way out is to abstract a concrete program state as a control flow branch, and abstract the propagation of fault suspiciousness of these concrete program states by a “transfer function” of the fault suspiciousness of one branch or statement to other branches or statements. On one hand, existing techniques work at the individual program entity level and assess the fault suspiciousness of program entities separately. On the other hand, the transfer of fault suspiciousness of one program entity to another will change the fault suspiciousness of the latter. In the presence of loops, finding a stable propagation is non-trivial. Moreover, even if a stable propagation can be found, a direct implementation of such a propagation-based technique may indicate that the technique requires many rounds of iterations, which unfortunately are computationally expensive. We propose a steady and efficient propagation-based CBFL technique in this paper.

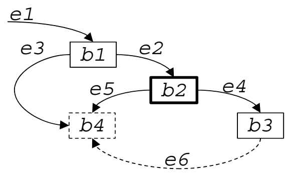
* This research is supported in part by the General Research Fund of the Research Grants Council of Hong Kong (project nos. 111107 and 716507).

† All correspondence should be addressed to Dr. W. K. Chan at Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Kowloon Tong, Hong Kong. Tel: (+852) 2788 9684. Fax: (+852) 2788 8614. Email: wkchan@cs.cityu.edu.hk.

susp.: suspiciousness of a statement/edge in relation to a fault; rank: ranking of a statement/edge in the generated list

Statement		**Basic block**	**Test case**							***Tarantula* [24]**		***SBI* [24]**		***Jaccard* [1]**		***Branch* [21]**		***CP* (this paper)**	
			t1	*t2*	*t3*	*t4*	*t5*	*t6*	*t7*	**susp.**	**rank**	**susp.**	**rank**	**susp.**	**rank**	**susp.**	**rank**	**susp.**	**rank**
if (block_queue) {	*s1*	*b1*	●	●	●	●	●	●	●	0.50	9	0.29	9	0.29	9	0.71	9	0.11	9
count = block_queue–>mem_count + 1;	*s2*		●		●		●	●		0.71	**7**	0.50	**7**	0.50	**7**	0.71	**9**	1.11	**4**
n = (int) (count*ratio);	*s3*	*b2* **(fault)**	●		●		●	●		0.71	7	0.50	7	0.50	7	0.71	9	1.11	4
proc = find_nth(block_queue, n);	*s4*		●		●		●	●		0.71	7	0.50	7	0.50	7	0.71	9	1.11	4
if (proc) {	*s5*		●		●		●	●		0.71	7	0.50	7	0.50	7	0.71	9	1.11	4
block_queue= del_ele(block_queue,proc);	*s6*	*b3* **(block whose execution leads to failure)**			●					1.00	3	1.00	3	0.50	7	0.71	9	1.00	7
prio = proc–>priority;	*s7*				●					1.00	3	1.00	3	0.50	7	0.71	9	1.00	7
prio_queue[prio] = append_ele(prio_queue[prio], proc);	*s8*				●					1.00	3	1.00	3	0.50	7	0.71	9	1.00	7
} } *// next basic block*	*s9*	*b4*	●	●	●	●	●	●	●	0.50	9	0.29	9	0.29	9	0.71	9	0.11	9

Control Flow Graph (CFG) for the code excerpt above:



(*We add a dummy block* b4 *containing statement s9, and an edge* e6 *to make a complete CFG.*)

Edge	*t1*	*t2*	*t3*	*t4*	*t5*	*t6*	*t7*	***Tarantula* susp.**	**rank**	***SBI* susp.**	**rank**	***Jaccard* susp.**	**rank**	***Branch* susp.**	**rank**	***CP* susp.**	**rank**
e1	●	●	●	●	●	●	●							0.53		0.00	
e2	●		●		●	●								0.71		0.43	
e3		●		●			●							0.00		–1.00	
e4			●											0.71		1.00	
e5	●				●	●								0.41		0.11	
e6			●											N/A		1.00	
Pass/fail status	P	P	F	P	P	F	P										
% of code examined according to the ranking of statements								**78%**		**78%**		**78%**		**100%**		**44%**	
Ranking order of basic blocks								b3 before b2				b2 and b3 have the same rank				b2 before b3	

Figure 1: Faulty version v2 of program schedule and fault localization comparison.

We first revisit a couple of terms for ease of subsequent discussions. A *failed execution* is a program execution that reveals a failure (such as producing an incorrect output or causing the program to crash). On the contrary, a *passed execution* is a program execution that shows no failure.

We abstract a given program as a control flow graph (CFG), sample a program execution as an edge profile [3], which indicates which edges of the CFG have been traversed during the execution, and quantifies every change of program state over an edge with the number of executions of the edge. We then compute a pair of mean edge profiles: the *mean pass profile* for all sampled passed executions, and the *mean failed profile* for all sampled failed executions. They capture abstractly the central tendency of the program state in a passed execution and that in a failed execution, respectively. For each edge, we contrast such a state abstraction in the mean pass profile with that in the mean failed profile to assess the fault suspiciousness of the edge. In our model, to backtrack how much every basic block [3] contributes to the observed program failures, we set up a system of linear algebraic equations to express the propagation of the suspiciousness scores of a basic block to its predecessor block(s) via directly connected control flow edges — for each edge, we split a fraction of the suspiciousness score to be propagated to a predecessor basic block. Our model always constructs homogeneous equations and ensures that the number of equations is the same as the number of variables. Such a constructed equation set satisfies a necessary condition of being solvable by standard mathematics techniques such as Gaussian elimination [30]. By solving the set of equations, our technique obtains the suspiciousness score for each basic block. We finally rank the basic blocks in descending order of their suspiciousness scores, and assign a rank for each statement. We conduct controlled experiments to compare the effectiveness of existing representative techniques with ours on four real-life medium-sized programs. The empirical results show that our technique can be more effective than peer techniques.

The main contribution of this work is twofold: (i) To the best of our knowledge, the work is the first that integrates the propagation of program states to CBFL techniques. (ii) We use four real-life medium-sized programs flex, grep, gzip, and sed to conduct experiments on our technique and compare them with five other techniques, namely *Tarantula*, *SBI*, *Jaccard*, *CBI*, and *SOBER*. The empirical results show that our technique is promising.

The rest of this paper is organized as follows: Section 2 shows a motivating example. Section 3 presents our technique. Section 4 presents further discussion on our technique, followed by an experimental evaluation in Section 5 and a literature review in Section 6. Section 7 concludes the paper.

2. MOTIVATING EXAMPLE

This section shows an example on how the modeling of propagation of infected program states via control flow edges helps locate faults effectively.

Figure 1 shows a code excerpt from the faulty version v2 of the program schedule (from SIR [10]). The code excerpt manages a process queue. It first calculates the index of a target process, and then moves the target process among priority queues. We seed an extra “+1” operation fault into statement *s2* in Figure 1. It may cause the program to select an incorrect operation for subsequent processing, which will lead to a failure.

This code excerpt contains two "if" statements (`s1` and `s5`), which divide the code excerpt into three basic blocks [5] (namely, `b1`, `b2`, and `b3`). The basic block `b1` contains only one statement `s1`. The result of evaluating "block_queue" in `s1` determines whether the statements `s2` to `s8` are skipped. The basic blocks `b2` contains statements `s2`, `s3`, `s4`, and `s5`. The basic block `b3` contains statements `s6`, `s7`, and `s8`. We also depict the code excerpt as a control flow graph (CFG) in Figure 1. In this CFG, each rectangular box represents a basic block and each arrow represents a control flow edge that connects two basic blocks. For example, `e2` indicates that the decision in `s1` has been evaluated to be true in an execution, so it transits from `b1` to `b2`. Since the fault lies in `b2`, we use a weighted border to highlight the rectangular box `b2`. Note that we add a dummy block `b4` (as a dashed rectangular box) and an edge `e6` (as a dashed arrow) to make this CFG more comprehensible.

We examine the program logic, and observe that many failures are caused by the execution of `b2` followed by `b3`, rather than merely executing `b2` without executing `b3`. Even if `b2` is executed and results in some infected program state, skipping `b3` will not alter the priority queue, and thus the effect of the fault at `s2` is less likely to be observed through subsequent program execution. On the other hand, executing `b2` followed by `b3` means that an infected program state (such as incorrect values for the variables *count*, *n*, or *proc*) at `b2` is successfully propagated to `b3` through the edge `e4`. Since previous studies suggest the comparison of execution statistics to assess the suspiciousness scores of statements, they will be more likely to result in a wrong decision — `b3` will appear to be more suspicious than `b2`. The following serves as an illustration.

In this example, we use seven test cases (dubbed `t1` to `t7`). Their statement and edge execution details are shown in Figure 1. A cell with the "●" notation indicates that the corresponding statement is exercised (or an edge is traversed) in the corresponding test execution. For instance, let us take the first test case `t1` (a successful one, referred to as "passed"). During its execution, the basic blocks `b1`, `b2`, and `b4` are exercised; moreover, the control flow edges `e1`, `e2`, and `e5` are traversed. Other test cases can be interpreted similarly. The passed/fail status of each test case is shown in the "Pass/Fail status" row. We apply *Tarantula* [24], *Jaccard* [1], and *SBI*[1] [24] to compute the suspiciousness score of every statement, and rank statements in descending order of their scores. Presuming that the programmer may check each statement according to their ranks until reaching the fault [15][24], we thus compute the effort of code examination to locate this fault [15][24]. We show their effectiveness in the "susp." and "rank" columns, and the row "% of code examined according to the ranking of statements" of Figure 1. We observe that `b3`, rather than `b2`, is deemed to be the most suspicious basic block if we apply *Tarantula* or *SBI*. When applying *Jaccard*, `b2` and `b3` are equally deemed to be the most suspicious basic blocks. As a result, the fault *cannot* be effectively located by *any* of these techniques. To locate the fault, each examined technique needs to examine 78% of the code.

Intuitively, the execution of `b3` may lead to a failure, and yet it is not the fault. On the other hand, `b2` contains the fault, but its execution does not lead to a failure as often as `b3`. Since existing techniques find the suspicious program entities that correlate to failures, they give higher ranks to those statements (such as `b3`) whose executions frequently lead to failures, but give lower ranks to those statements (such as `b2`) whose executions less often lead to failures. If we always separately assess the fault suspiciousness of individual statements (such as `b2` and `b3`) and ignore their relations, this problem may hardly be solved.

Since executing an edge can be regarded as executing both the two basic blocks connected by the edge, do edge-oriented techniques somehow capture the relationships among statements and perform effectively on this example? We follow [21] to adopt *br*, an edge-oriented technique (which we will refer to as *Branch* in this paper) to work on this example. *Branch* assesses the suspiciousness scores of control flow edges (say, `e1` and `e2`), and then associate their suspiciousness scores with statements that are directly connected (in sense of incoming edges or outgoing edges). *Branch* first ranks `e2` and `e4` as the most suspicious edges (both having a suspiciousness score of 0.71). In a program execution, traversing `e2` means to enter the true branch of `s1` followed by executing `b2`, and traversing `e4` means having executed `b2` and will continue to execute `b3`. We observe that executing `b2` generates infected program states (on variables *count*, *n*, and *proc*), which propagate to `b3` through `e4`. We further observe that these two highly ranked edges precisely pinpoint the fault location. When associating edges to statements, the rules in *Branch* only propagate the edge suspiciousness to those statements within the same block as the conditional statement for the edge. However, a fault may be several blocks away from the edge and the loop construct may even feedback a faulty program state. For this example, *Branch* assigns identical suspiciousness scores to all statements and they cannot be distinguished from one another. As a result, 100% code examination effort is needed to locate the fault. Since the core of *Branch* is built on top of *Ochiai* [1], we have iteratively replaced this core part of *Branch* by *Tarantula*, *Jaccard*, and *SBI*. However, the fault-localization effectiveness results are still unsatisfactory (100% code to be examined). In practice, the propagation of infected program states may take a long way, such as a sequence of edges, before it may finally result in failures. We need a means to transfer over the edges information about infected program states and failures.

3. OUR MODEL

3.1 Problem Settings

Let P be a faulty program, $T = \{t_1, t_2, \ldots, t_u\}$ be a set of test cases associated with passed executions, and $T' = \{t'_1, t'_2, \ldots, t'_v\}$ be a set of test cases that are associated with failed executions. In the motivating example, for instance, P is version v2 of schedule, T = {`t1`, `t2`, `t4`, `t5`, `t7`}, and T' = {`t3`, `t6`}. Similar to existing work (such as [15]), the technique assesses the fault suspiciousness of each statement of P and can also produce a list of statements of P in descending order of the suspiciousness scores.

3.2 Preliminaries

We use $G(P) = \langle B, E\rangle$ to denote the control flow graph (CFG) [3][5] of the program P, where $B = \{b_1, b_2, \ldots, b_n\}$ is the set of basic blocks (*blocks* for short) of P, and $E = \{e_1, e_2, \ldots, e_m\}$ is the set of control flow edges (*edges* for short) of P, in which each edge e_i goes from one block to another (possibly the same block) in B. Thus, we sometimes write e_i as edg(b_{i1}, b_{i2}) to denote an edge going from b_{i1} to b_{i2}; this edge e_i is called the *incoming* edge of b_{i2}

[1] In this paper, we use the term *SBI* to denote Yu et al.'s approach [24] of applying Liblit et al.'s work *CBI* [18] at statement level. At the same time, we still keep the term *CBI* when referring to the original technique in [18].

and the *outgoing* edge of b_{i1}. The block b_{i2} is a *successor block* of b_{i1}, and the block b_{i1} is a *predecessor block* of b_{i2}. We further use the notation edg(*, b_j) and edg(b_j, *) to represent the set of incoming edges and the set of outgoing edges of b_j, respectively. In Figure 1, for instance, edges `e4` and `e5` are the outgoing edges of block `b2`, and block `b3` is a successor block of `b2` with respect to edge `e4`; edg(`b3`, *) = {`e6`} is the set of outgoing edges of block `b3`, and edg(*, `b3`) = {`e4`} is the set of incoming edges of block `b3`.

An edge is said to have been *covered* by a program execution if it is traversed at least once. For every program execution of *P*, whether an edge is covered in *E* can be represented by an edge profile. Suppose t_k is a test case. We denote the edge profile for t_k by $P(t_k) = \langle \theta(e_1, t_k), \theta(e_2, t_k), \ldots, \theta(e_m, t_k)\rangle$, in which $\theta(e_i, t_k)$ means whether the edge e_i is covered in the corresponding program execution of t_k. In particular, $\theta(e_i, t_k) = 1$ if e_i is covered by the execution, whereas $\theta(e_i, t_k) = 0$ if e_i is not covered. Take the motivating example in Figure 1 for illustration. The edge profile for test case `t1` is $P(\texttt{t1}) = \langle \theta(\texttt{e1}, \texttt{t1}), \theta(\texttt{e2}, \texttt{t1}), \theta(\texttt{e3}, \texttt{t1}), \theta(\texttt{e4}, \texttt{t1}), \theta(\texttt{e5}, \texttt{t1}), \theta(\texttt{e6}, \texttt{t1})\rangle = \langle 1, 1, 0, 0, 1, 0\rangle$.

3.3 *CP:* Our Fault-Localization Model

We introduce our fault-localization model in this section.

A program execution passing through an edge indicates that the related program states have propagated via the edge. Therefore, we abstractly model a program state in a program execution as whether the edge has been covered by the program execution, and contrast the edge profiles in passed executions to those of failed executions to ***C***apture the suspicious ***P***ropagation of program states abstractly. We name our model as ***CP***.

In Section 3.3.1, *CP* first uses equation (1) to calculate the *mean edge profile* for the corresponding edge profiles of all passed executions, and another mean edge profile for those of all failed executions. Such mean edge profiles represent the central tendencies of the program states in the passed executions and failed executions, respectively. *CP* contrasts the two mean edge profiles to assess the suspiciousness of every edge. The formula to compute the suspiciousness score is given in equation (2) and explained with the aid of equation (3).

During a program execution, a block may propagate program states to adjacent blocks via edges connecting to that block. In Section 3.3.2, we use equation (4) to calculate the ratio of the propagation via each edge, and use such a ratio to determine the fraction of the suspiciousness score of a block propagating to a predecessor block via that edge. We use backward propagation to align with the idea of backtracking from a failure to the root cause. For each block, *CP* uses a linear algebraic equation to express its suspiciousness score by summing up such fractions of suspiciousness scores of successor blocks of the given block. Such an equation is constructed using equation (5) or (6), depending on whether the block is a normal or exit block. By solving the set of equations (by Gaussian elimination), we obtain the suspiciousness score for each block involved.

As presented in Section 3.3.3, *CP* ranks all blocks in descending order of their suspiciousness scores, then assigns a rank to each statement, and produces a ranked list of statements by sorting them in descending order of their suspiciousness scores.

3.3.1 Calculating the Edge Suspiciousness Score

In Section 2, we have shown that edges can provide useful correlation information for failures. However, the size of *T* may be very different from that of *T'*. To compare the sets of edge profiles for *T* with those for *T'*, we propose to compare their arithmetic means (that is, central tendencies).

If an edge is never traversed in any execution, it is irrelevant to the observed failures. There is no need to compute the propagation of suspicious program states through that edge. We thus exclude them from our calculation model in Sections 3.3.1 and 3.3.2.

In our model, we use the notation $P^{\surd} = \langle \theta^{\surd}(e_1), \theta^{\surd}(e_2), \ldots, \theta^{\surd}(e_m)\rangle$ to denote the mean edge profile for *T*, and $P^{\times} = \langle \theta^{\times}(e_1), \theta^{\times}(e_2), \ldots, \theta^{\times}(e_m)\rangle$ for *T'*, where $\theta^{\surd}(e_i)$ and $\theta^{\times}(e_i)$ for $i = 1$ to m are calculated by:

$$\theta^{\surd}(e_i) = \frac{1}{u}\sum_{t_k \in T}[\theta(e_i, t_k)]\,;\ \ \theta^{\times}(e_i) = \frac{1}{v}\sum_{t'_k \in T'}[\theta(e_i, t'_k)]. \quad (1)$$

Note that the variables *u* and *v* in equation (1) represent the total numbers of passed and failed test cases, respectively. Intuitively, $\theta^{\surd}(e_i)$ and $\theta^{\times}(e_i)$ stand for *the probabilities of an edge being exercised in a passed execution and failed execution, respectively, over the given test set*. For example, $\theta^{\times}(\texttt{e4}) = (\theta(\texttt{e4}, \texttt{t3}) + \theta(\texttt{e4}, \texttt{t6})) / 2 = (1 + 0) / 2 = 0.5$ and, similarly, $\theta^{\surd}(\texttt{e4}) = 0$.

Edge suspiciousness calculation. We calculate the suspiciousness score of any given edge e_i using the equation

$$\theta^{\Delta}(e_i) = \frac{\theta^{\times}(e_i) - \theta^{\surd}(e_i)}{\theta^{\times}(e_i) + \theta^{\surd}(e_i)}, \quad (2)$$

which contrasts the difference between the two mean edge profiles. Intuitively, $\theta^{\Delta}(e_i)$ models the (normalized) difference between the probability of e_i being traversed in an average passed execution and the probability of an average failed execution. When $\theta^{\Delta}(e_i)$ is positive, it reflects that the probability of edge e_i being covered in $P^{\times}$ is larger than that in $P^{\surd}$. Since such an edge is more frequently exercised in failed executions than in passed executions, it may be more likely to be related to a fault. When $\theta^{\Delta}(e_i) = 0$, the edge e_i has identical probabilities of being covered in $P^{\times}$ and $P^{\surd}$. Such an edge is deemed to be less likely to be related to a fault than an edge having a positive suspiciousness score. When $\theta^{\Delta}(e_i)$ is negative, it means that e_i is less frequently executed in $P^{\times}$ than in $P^{\surd}$.

In short, for an edge e_i, the higher the values of $\theta^{\Delta}(e_i)$, the more suspicious the edge e_i is deemed to be, and the more suspicious the propagation of program states via e_i is deemed to be.

To understand why equation (2) is useful for ranking edges according to their suspiciousness, let $Prob(e_i)$ denote the (unknown) probability that the propagation of infected program states via e_i will cause a failure. The proof in [28] shows that

$$Prob(e_i) = \frac{v \cdot \theta^{\times}(e_i)}{v \cdot \theta^{\times}(e_i) + u \cdot \theta^{\surd}(e_i)} \quad (3)$$

is the best estimate for this probability. It is also proved in [28] that, no matter whether equation (2) or (3) is chosen to estimate the fault-relevance of edges, sorting edges in descending order of the results always generates the same edge sequence (except tie cases). In other words, we can also determine the order of the suspiciousness of the edges through equation (3), or determine the probability that an edge causes a failure by using equation (2).

We will adopt equation (2) rather than equation (3) because the value range of equation (2), which is from −1 to 1 and symmetric with respect to 0, favors follow-up computation. For example, equation (3) always generates positive values. However, summing up positive operands always generates an operation result that is greater than each operand, and hence there may not be a solution for the constructed equation set (see Section 3.3.2).

Take the motivating example as an illustration. *Prob*(`e1`) = (2 × 1.00) / (2 × 1.00 + 5 × 1.00) = 0.29. Similarly, *Prob*(`e2`) = 0.50, *Prob*(`e3`) = 0.00, *Prob*(`e4`) = 1.00, *Prob*(`e5`) = 0.33, and *Prob*(`e6`) = 1.00. Furthermore, θ^{Δ}(`e1`) = (1.00 – 1.00) / (1.00 + 1.00) = 0.00. Similarly, θ^{Δ}(`e2`) = 0.43, θ^{Δ}(`e3`) = −1.00, θ^{Δ}(`e4`) = 1.00, $\theta^{\Delta}(e_5)$ = 0.11, and θ^{Δ}(`e6`) = 1.00. By sorting `e1` to `e6` in descending order of their values using equation (2), we obtain ⟨{`e4`, `e6`}, `e2`, `e5`, `e1`, `e3`⟩, where `e4` and `e6` form a tie case. Based on the two most suspicious edges (`e2` and `e6`), we can focus our attention to trace from `b2` to `b3` via `e4`, and then to `b4` via `e6`. However, one still does not know how to rank the fault suspiciousness of the blocks (other than examining all three blocks at the same time). In the next section, we will show how we use the concept of propagation to determine the suspiciousness score of each block.

3.3.2 Solving Block Suspiciousness Score

By contrasting the mean edge profiles of the passed executions and the failed executions, we have computed the suspiciousness of edges in the last section. In this section, we further associate edges with blocks, and assess fault suspiciousness score of every block. To ease our reference, we use the notation $BR(b_j)$ to represent the suspiciousness score of a block b_j.

Let us first discuss how a program execution transits from one block to another. After executing a block b_j, the execution may transfer control to one of its successor blocks. Suppose b_k is a successor block of b_j. The program states of b_j may be propagated to b_k via the edge edg(b_j, b_k). Rather than expensively tracking the dynamic program state prorogation from b_j to b_k, we approximate the expected number of infected program states of b_j *observed at* b_k as the fraction of the suspiciousness score of b_j from that of b_k. This strategy aligns with our understanding that we can only observe failures from outputs rather than from inputs.

Constructing an equation set. To determine the fraction mentioned above, we compute the sum of suspiciousness scores of the incoming edges of b_k, and compute the ratio of propagation via the edge edg(b_j, b_k) as the ratio of the suspiciousness score of this particular edg(b_j, b_k) over the total suspiciousness score for all edges. The formula to determine this ratio is:

$$W(b_j, b_k) = \frac{\theta^{\Delta}(\text{edg}(b_j, b_k))}{\sum_{\forall\ \text{edg}(*, b_k)} [\theta^{\Delta}(\text{edg}(*, b_k))]} \qquad (4)$$

$W(b_j, b_k)$ models the portion of the contribution of edg(b_j, b_k) with respect to the total contribution by all incoming edges of b_k. Intuitively, it represents the ratio of the observed suspicious program states at b_k that may be prorogated from b_j.

The fraction of the suspiciousness score that b_k contributes to b_j is, therefore, the product of this ratio and the suspiciousness score of b_k (that is, $BR(b_k) \times W(b_j, b_k)$).

In general, a block b_j may have any number of successor blocks. Hence, we sum up such a fraction from every successor block of b_j to give $BR(b_j)$, the suspiciousness score of b_j, thus:

$$BR(b_j) = \sum_{\forall\ \text{edg}(b_j, b_k)} [BR(b_k) \cdot W(b_j, b_k)] \qquad (5)$$

Let us take the motivating example for illustration: `b2` has two outgoing edges connecting to two successor blocks `b3` and `b4`, respectively. Its suspiciousness score *BR*(`b2`) is, therefore, calculated as *BR*(`b3`) · *W*(`b2`, `b3`) + *BR*(`b4`) · *W*(`b2`, `b4`). The propagation rate *W*(`b2`, `b4`) is calculated as (`b2`, `b4`) = θ^{Δ}(`e5`) / (θ^{Δ}(`e3`) + θ^{Δ}(`e5`) + θ^{Δ}(`e6`)) = 0.11 / (−1.00 + 0.11 + 1.00) = 1. The propagation rate *W*(`b2`, `b3`) is calculated as θ^{Δ}(`e4`) / θ^{Δ}(`e4`) = 1.00 / 1.00 = 1.

Handling exception cases. We normally calculate the suspiciousness score of a block via its successor blocks. Let us consider an exception case where a block may have no successor. For a block containing, say, a return, break, or exit() function call, or for a block that crashes, the program execution may leave the block, or cease any further branch transitions after executing the block. In our model, if a block is found not to transit the control to other successor blocks in the same CFG (as in the case of a return statement or callback function), we call it an *exit* block. Since exit blocks have no successor block, we do not apply equation (5) to calculate its suspiciousness score. Instead, we use the equation

$$BR(b_j) = \sum_{\forall\ \text{edg}(*, b_j)} [\theta^{\Delta}(\text{edg}(*, b_j))], \qquad (6)$$

which sums the suspiciousness scores of all the incoming edges to calculate the suspiciousness score of the exit block. Consider the motivating example again. There is no successor for b_4 in the CFG in Figure 1. Hence, *BR*(`b4`) = θ^{Δ}(`e3`) + θ^{Δ}(`e5`) + θ^{Δ}(`e6`) = −1.00 + 0.11 + 1.00 = 0.11. (We should add that there are alternative ways to model the suspiciousness score for an exit block, such as using the formulas for block-level CBFL techniques.)

We have constructed an equation of $BR(b_j)$ for every block b_j (including exit blocks). In other words, we have set up an equation set containing *n* homogenous equations (one for each block) and *n* variables as the left-hand side of each equation (one for each suspiciousness score of that block). In such a case, the equation set satisfies a necessary condition of being solvable by existing efficient algorithms for solving equation sets (such as Gaussian elimination [30], which we also adopt in the experiment in Section 5). In the motivating example, we can set up an equation set {*BR*(`b4`) = 0.11, *BR*(`b3`) = 9 × *BR*(`b4`), *BR*(`b2`) = *BR*(`b4`) + *BR*(`b3`), *BR*(`b1`) = −9 × *BR*(`b4`) + *BR*(`b2`)}. We can solve it to give *BR*(`b4`) = 0.11, *BR*(`b3`) = 1.00, *BR*(`b2`) = 1.11, and *BR*(`b1`) = 0.11.

3.3.3 Synthesizing a Ranked List of Statements

After obtaining the suspiciousness score for every block, we further group those statements not in any block (such as global assignment statements) into a new block, and give it a lower suspiciousness score than that of any other ranked block. We also merge those blocks having identical suspiciousness scores, and produce a ranked list of blocks in descending order of their suspiciousness scores. All the non-executable statements and statements that are not exercised by any given executions are consolidated into one block, which is appended to the end of the ranked list and given the lowest suspiciousness score. Finally, one may assign ranks for statements. Following previous work [15], the rank of a statement is assigned as the sum of total number of statements in its belonging block and the total number of statements in the blocks prior to its belonging block. The *CP* column of Figure 1 shows the ranks of statements by our method, which only needs 44% code examination effort to locate a fault.

4. DISCUSSIONS

In previous work, a tie-break strategy (see [21][24]) is further employed to optimize the baseline fault-localization techniques. They give a better ranking list when one follows the ranking list to locate faults. However, they do not modify the computed suspicious scores of the program entities by those techniques. Our technique can also be optimized in exactly the same way.

In our technique, we capture the propagation of infected program states via CFG. In fact, other flow-graph representations of program executions, such as program dependency graphs [3] or data flow graphs, may be employed to replace CFG. We do not iteratively show how to adapt each of them in our technique.

The space and time complexity of our technique is analyzed as follows: With the same problem settings (u passed executions, v failed executions, n blocks, and m edges), the space complexity is mainly determined by the space needed to maintain the mean edge profiles for the passed executions and the failed executions, and the suspiciousness scores for edges, which are $O(um)$, $O(vm)$, and $O(m)$, respectively. Therefore the space complexity of our technique is $O(um + vm)$. The time complexity is determined by the time used to solve the set of equations. If Gaussian elimination is adopted, the time complexity of our technique will be $O(n^3)$.

5. EXPERIMENTAL EVALUATION

In this section, we conduct a controlled experiment to evaluate the effectiveness of our technique.

5.1 Setup of Experiment

5.1.1 Subject Programs

We use four UNIX programs, namely, flex, grep, gzip, and sed, as subject programs. They are real-life medium-sized programs, and have been adopted to evaluate other CBFL techniques (as in [14][23][26]). We downloaded the programs (including all versions and associated test suites) from SIR [10] on January 10, 2008. Each subject program has multiple (sequentially labeled) versions. Table 1 shows the real-life program versions, numbers of lines of executable statements (LOC), numbers of applicable faulty versions, and the sizes of the test pools. Take the program flex as an example. The real-life versions include flex-2.4.7 to flex-2.5.4, and have 8571 to 10124 lines of executable statements. 21 single-fault versions are used in the experiment. All these faulty versions share a test suite that consists of 567 test cases. Following [12], we apply the whole test suite as inputs to individual subject programs.

Following the documentation of SIR [10] and previous experiments [12][15][18][19], we exclude any single-fault version whose faults cannot be revealed by any test case. This is because both our technique and the peer techniques used in the experiment [1][18][19][24] require the existence of failed executions. Moreover, we also exclude any single-fault version that fails for more than 20% of the test cases [10][12]. Besides, as Jones et al. have done in [15], we exclude those faulty versions that are not supported by our experimental environment and instrumentation tool (we use the Sun Studio C++ compiler and gcov to collect edge profile information on program versions). All the remaining 110 single-fault versions are used in the controlled experiment (see Table 1).

5.1.2 Peer Techniques

In our experiment, we select five representative peer techniques to compare with our technique. *Tarantula* [24] and *Jaccard* [1] are two statement-level techniques. They are often chosen as alternatives for comparison in other evaluations of fault-localization techniques. *CBI* [18] and *SOBER* [19] are predicate-level techniques. Since they make use of predicates (such as branch decisions) to locate suspicious program positions, which are related to the edge concept in our technique, we decide to compare these techniques with ours. Note that *CBI* originally proposed to use random sampling to collect predicate statistics to reduce overhead. In our evaluation on *CBI*, we sample all the predicates (as in [19]) via gcov. In Yu et al.'s work [24], *CBI* is modified to become a statement-level technique (*SBI* [24]), and we also include *SBI* for comparison with our technique. Note that a tie-breaking strategy is included in *Tarantula* as stated in [24]. *CP* uses no tie-breaking strategy in our experiment.

Table 1: Statistics of Subject Programs

	Real-life versions	Program Description	LOC	No. of single-fault versions	No. of test cases
flex	2.4.7–2.5.4	lexical parser	8571–10124	21	567
grep	2.2–2.4.2	text processor	8053–9089	17	809
gzip	1.1.2–1.3	compressor	4081–5159	55	217
sed	1.18–3.02	text processor	4756–9289	17	370

5.1.3 Effectiveness Metrics

Each of *Tarantula*, *SBI*, *Jaccard*, and *CP* produces a ranked list of all statements. For every technique, we check all the statements in ascending order of their ranks in the ordered list, until a faulty statement is found. The percentage of statements examined (with respect to all statements) is returned as the effectiveness of that technique. This metric is also used in previous studies [14][24]. We note also that statements having the same rank are examined as a group.

CBI and *SOBER* generate ranked lists of predicates. To the best of our knowledge, the metric T-score [20] is used to evaluate their effectiveness in previous studies [19]. T-score uses a program dependence graph to calculate the distance among statements. Starting from some top elements in a ranked list of predicates, T-score conducts breadth-first search among the statements to locate a fault. The search terminates when it encounters any faulty statement, and the percentage of statements examined (with respect to all statements) is returned as the effectiveness of that technique [20]. Since it is reported in [19] that the "top-5 T-score" strategy gives the highest performance for *CBI* and *SOBER*, we follow suit to choose the top-5 predicates and report the top-5 T-score results as their effectiveness in our experiment.

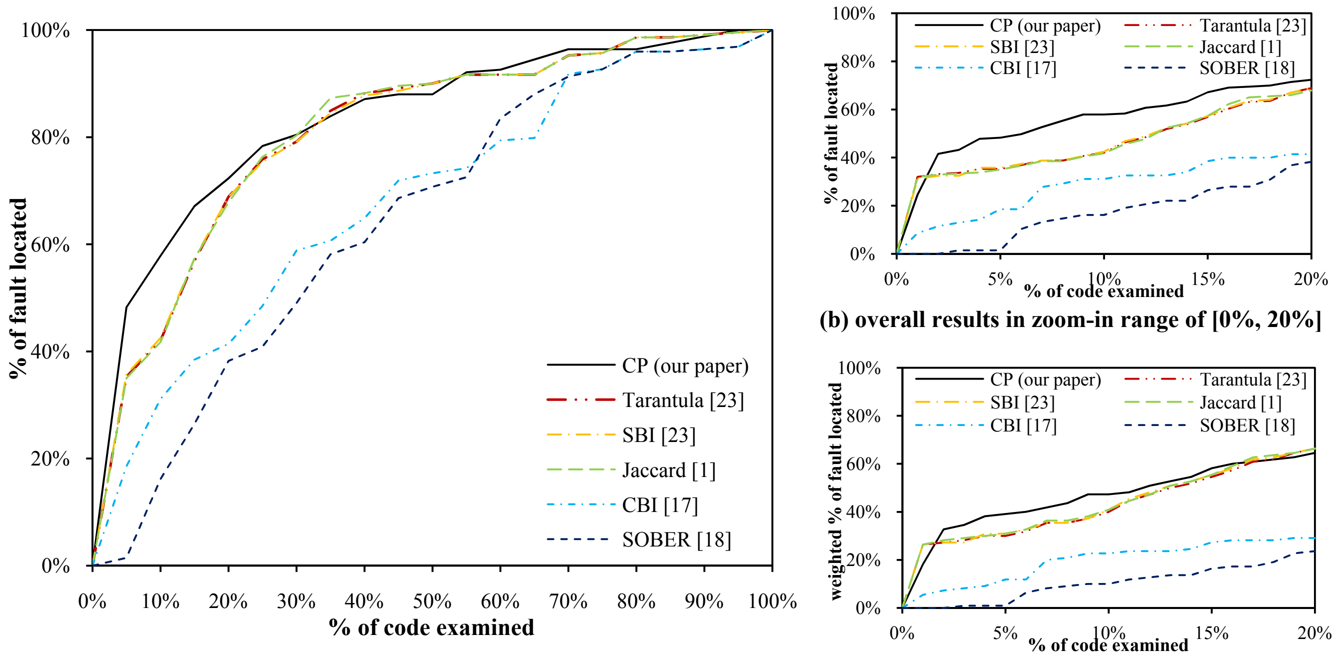
If a fault is in a non-executable statement (such as the case of a code omission fault), dynamic execution information cannot help locate the fault directly. To reflect the effectiveness of a technique, we follow previous studies (such as [14]) to mark the directly infected statement *or* an adjacent executable statement as the fault position, and apply the above metrics.

5.1.4 Experiment Environment and Issues

The experiment is carried out in a Dell PowerEdge 1950 server (4-core Xeon 5355 2.66GHz processors, 8GB physical memory, and 400GB hard disk) serving a Solaris UNIX with the kernel version Generic_120012-14. Our framework is compiled using Sun C++ 5.8.

When applying our technique, an exceptional case is that the denominator in equation (4) may be zero. For every occurrence of a zero denominator in the experiment, the tool automatically replaces it by a small constant. 10^{-10} is chosen as the constant, which is less than any intermediate computing result by many degrees of magnitude. We have varied this constant from to 10^{-11} to 10^{-9} and compared the effectiveness results of *CP*, and confirmed that the results are the same.

In the experiment, the time needed to generate a ranked list for one faulty version is always less than 1 second. The mean time spent for one faulty version is about 0.455 seconds.



(a) overall results in full range of [0%, 100%]

(b) overall results in zoom-in range of [0%, 20%]

(c) weighted overall results in range of [0%, 20%]

Figure 2: Overall effectiveness comparison

5.2 Data Analysis

In this section, we compare our technique with *Tarantula*, *SBI*, *Jaccard*, *CBI*, and *SOBER*, and report their effectiveness on the 110 single-fault program versions. In the following subsections, the data related to "*Tarantula*", "*SBI*", "*Jaccard*", "*CBI*", and "*SOBER*" are worked out using the techniques described in the papers [24], [24], [1], [18], and [19], respectively. The data related to "*CP*" are worked out using our technique. For every plot in Figure 2 and Figure 3, we use the same set of *x*-axis labels and legends.

5.2.1 Overall Results

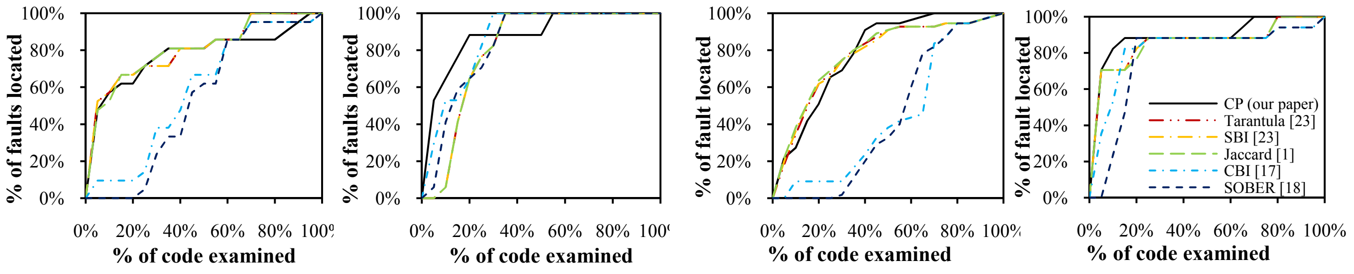
To evaluate the overall effectiveness of a technique, we first take the average of the effectiveness results on the four subject programs. The results are shown in Figure 2. In the plot in Figure 2(a), the *x*-axis means the percentage of code that needs to be examined in order to locate the fault (according to the effectiveness metrics). We also refer to it as the *code examination effort* in this paper. The *y*-axis means the mean percentage of faults located. Take the curve for *CP* in Figure 2(a) for illustration. On average, *CP* can locate 48.24% of all faults by examining up to 5% of the code in each faulty version. The curves of *Tarantula*, *Jaccard*, *CBI*, *SBI*, and *SOBER* can be interpreted similarly. Note that the effectiveness of *Tarantula*, *SBI*, and *Jaccard* are very close, and hence their curves in Figure 2 and Figure 3 almost completely overlap.

Figure 2(a) gives the overall effectiveness of *CP*, *Tarantula*, *SBI*, *Jaccard*, *CBI*, and *SOBER*. Each of the six curves starts at the point (0%, 0%) and finally reaches the point (100%, 100%). Obviously, it reflects the fact that no fault can be located when examining 0% of the code, while all the faults can be located when examining 100%. We observe from the figure that *CP* can locate more faults than *CBI* and *SOBER* in the range from 1% to 99% of the code affordable to be examined. Moreover, the figure also shows that *CP* can locate more faults than *Tarantula*, *SBI*, and *Jaccard* almost in the entire range of the first one third (from 2% to 33%) of the code examination effort.

When comparing the mean effectiveness, although one cannot meaningfully conclude the results from outliner segments (such as those data points beyond three standard deviations), previous studies such as [19] once reported the results on the first 20% code examination range. Therefore, we further zoom in (to the range of [0%, 20%]) as shown in Figure 2(b).

The figure shows that, if only 1% of the code is affordable to be examined, *Tarantula*, *SBI*, and *Jaccard* can locate 31.99% of all faults, *CP* can locate 24.50%, *CBI* can locate 8.54%, while *SOBER* cannot locate any fault. If 2% of the code is affordable to be examined, encouragingly, *CP* not only catches up with *Tarantula*, *SBI*, and *Jaccard*, but also exceeds them a lot. For example, *CP*, *Tarantula*, *SBI*, *Jaccard*, *CBI*, and *SOBER* locate 41.55%, 33.18%, 32.45%, 32.90%, 11.48%, and 0.00% of the faults in all faulty versions, respectively. In the remaining range (from 2% to 20%) in Figure 2(b), *CP* always locates more faults than the peer techniques. For example, when examining 8%, 9%, and 10% of the code, *CP* locates 55.31%, 57.86%, and 57.86% of the faults, respectively; *Tarantula* locates 38.75%, 40.67%, and 42.03% of the faults; *SBI* locates 38.75%, 40.67%, and 42.49%; and *Jaccard* locates 38.46%, 40.39%, and 41.75%. In summary, by examining up to 20% of the code, *CP* can be more effective than the peer techniques.

In previous studies, a weighted average method has also been used [8]. For example, Chilimbi et al. [8] uses the total number of faults located in all programs as the *y*-axis (in the sense of Figure 2(b)), rather than the average percentage of faults located. To enable reader to compare previously published results with ours, we follow [8] to present such a plot as Figure 2(c). From this figure, we observe that if 2% to 16% of the code is examined, *CP* performs better than the other five techniques. However, *Tarantula*, *SBI*, and *Jaccard* catch up with *CP* gradually. The range (21% to 99%) is not shown in this paper owing to space limit, and yet



(a) flex **(b) grep** **(c) gzip** **(d) sed**

Figure 3: Effectiveness on individual programs.

we do observe that the effectiveness of *CP*, *Tarantula*, *SBI*, and *Jaccard* are very similar. More detailed statistical comparisons can be found in Section 5.2.3.

Overall, the experiment shows that *CP* can be effective. At the same time, it also shows that *CP* can be further improved.

5.2.2 Results on Individual Subject Programs

We further compare the effectiveness of *CP* against the peer techniques on each subject program. Figure 3 shows the corresponding results on the programs flex, grep, gzip, and sed, respectively. Take the curve for *SBI* in Figure 3(a) for illustration. Like Figure 2(a), the *x*-axis means the percentage of code examined, and the *y*-axis means the percentage of faults located by *SBI* within the given code examination effort (specified by the respective value on the *x*-axis). The curves for *CP*, *Tarantula*, *Jaccard*, *CBI*, and *SOBER* can be interpreted similarly.

The four plots in Figure 3 give the overall effectiveness of *CP*, *Tarantula*, *SBI*, *Jaccard*, *CBI*, and *SOBER* on each subject program. If 5% of the code has been examined, *CP* can locate faults in 47.61%, 52.94%, 21.81%, and 70.58% of the faulty versions of the programs flex, grep, gzip, and sed, respectively. On the other hand, *Tarantula* can locate 52.38%, 0.00%, 18.18%, and 70.58% of the faults, respectively; *SBI* can locate 52.38%, 0.00%, 20.00%, and 70.58%; *Jaccard* can locate 52.38%, 0.00%, 21.81%, and 70.58%; *CBI* can locate 9.52%, 29.41%, 0.00%, and 35.29%; and *SOBER* can locate 0.00%, 5.88%, 0.00%, and 0.00%. The other points on the curves can be interpreted similarly.

Similarly to Section 5.2.1, let us discuss the first 20% of the code examination range. For flex and gzip, we observe that *CP* performs better than *CBI* or *SOBER*, and performs comparably with *Tarantula*, *SBI* and *Jaccard*. For grep and sed, *CP* locates more faults than *Tarantula*, *SBI*, *Jaccard*, *CBI*, and *SOBER* within the first 20% code examination range. In summary, *CP* performs outstandingly in this range.

5.2.3 Statistics Analysis on Individual Faulty Versions

In this section, we further use popular statistics metrics to compare different techniques. Table 2 lists out the minimum (min), maximum (max), medium, mean, and standard derivation (stdev) of the effectiveness of these techniques, on the 110 single-fault versions. The effectiveness of each technique is evaluated using the same metric as in the previous section; therefore, the smaller the magnitude, the better is the effectiveness. We observe that in each row, *CP* gives the best (smallest) value among the six techniques, which further strengthens our belief that *CP* can be effective on locating faults.

Table 2: Statistics of effectiveness

	CP **(this paper)**	***Tarantula*** **[24]**	***SBI*** **[24]**	***Jaccard*** **[1]**	***CBI*** **[18]**	***SOBER*** **[19]**
min	**0.01%**	0.01%	0.01%	0.01%	0.26%	2.97%
max	**93.55%**	97.50%	97.50%	97.50%	100.00%	100.00%
medium	**11.67%**	12.86%	12.48%	12.48%	40.74%	42.84%
mean	**17.98%**	19.63%	19.74%	19.26%	40.55%	42.96%
stdev	**20.92%**	22.47%	22.63%	22.39%	27.89%	23.98%

To further find the relative merits on individual versions, we compute the difference in effectiveness between *CP* and each peer technique, and the results are shown in Table 3. Take the cell in column "*CP*−*Tarantula*" and row "< −5%" as an example It shows that, for 42 (38.18%) of the 110 faulty versions, the code examination effort of using *CP* to locate a fault is less than that of *Tarantula* by more than 5%. Similarly, for the row "> 5%", only 27 (24.54%) of the 110 versions, the code examination effort of *CP* is greater than that of *Tarantula* by more than 5%. For 41 (37.27%) of the faulty versions, the effectiveness between *CP* and *Tarantula* cannot be distinguished at the 5% significance level.

We therefore deem that, at the 5% significance level, the probability of *CP* performing better than *Tarantula* on these subject programs is higher than that of *Tarantula* performing better than *CP*. We further vary the significance level from 5% to 1% and 10% to produce the complete table. The experimental result shows that the probability of *CP* performing better than its peer technique is consistently higher than that for the other way round.

5.2.4 Discussions of Multi-Fault Programs

In this section, we use a real-life multi-fault program to validate the effectiveness of *CP*. Our objective here is to study *CP* rather than comparing *CP* with peer techniques.

Version v3 of flex has the largest number of feasible faults (9 in total) and flex is the largest subject program in the entire experiment. Therefore, we enable all the nine faults of this version to simulate a 9-fault program. Part of the code excerpt is shown in Figure 4. After enabling all nine feasible faults, we execute the test pool in the 9-fault program. It results in 136 failed executions and 431 passed executions.

We apply *CP* to this 9-fault program, and locate the first fault in line 3369 after examining 0.11% of the code. This fault is on an incorrect logical operator. By analyzing the faulty Boolean ex-

Table 3: Statistics of differences in effectiveness

	Difference (percentage difference)				
	CP*−*Tarantula	***CP*−*SBI***	***CP*−*Jaccard***	***CP*−*CBI***	***CP*−*SOBER***
< −1%	47 (42.72%)	48 (43.63%)	46 (41.81%)	86 (78.18%)	95 (86.36%)
−1% to 1%	19 (17.27%)	18 (16.36%)	19 (17.27%)	6 (5.45%)	2 (1.81%)
> 1%	44 (40.00%)	44 (40.00%)	45 (40.90%)	18 (16.36%)	13 (11.81%)
< −5%	42 (38.18%)	41 (37.27%)	40 (36.36%)	80 (72.72%)	91 (82.72%)
−5% to 5%	41 (37.27%)	43 (39.09%)	42 (38.18%)	16 (14.54%)	7 (6.36%)
> 5%	27 (24.54%)	26 (23.63%)	28 (25.45%)	14 (12.72%)	12 (10.90%)
< −10%	31 (28.18%)	31 (28.18%)	30 (27.27%)	71 (64.54%)	82 (74.54%)
−10% to 10%	60 (54.54%)	60 (54.54%)	60 (54.54%)	28 (25.45%)	18 (16.36%)
> 10%	19 (17.27%)	19 (17.27%)	20 (18.18%)	11 (10.00%)	10 (9.09%)

```
620     do_yywrap = …; // Fault F_AA_4
        …
985     if ( ! do_yywrap )
        …
3369    if ( ( need_backing_up && ! nultrans ) … ) // Fault F_AA_3
        …
        static yyconst short int yy_chk[2775] = { …
11825     836, 836, 599, … // Fault F_AA_2 …
        }
        …
12193   while ( yy_chk […] != … )
```

Figure 4: Excerpts from multi-fault program.

pression, we find that the fault is enabled only if the decision of the Boolean expression is true. As such, this edge (namely, the true decision made in line 3369) rightly reveals failures due to the fault, and *CP* locates this fault effectively. We simulate the fixing of this fault by reverting the statement to the original version. We rerun all the test cases, and find that failed executions have been reduced to 123. We re-apply *CP* and locate the second fault in line 620 after examining 1.21% of the code. The fault is an incorrect assignment of the variable yy_chk. In this version, the first statement that uses the variable yy_chk is in line 985; it is the root cause of failures. We manually examine the corresponding CFG between the block (dubbed b_a) containing the statement in line 620 and the block (dubbed b_b) containing the statement in line 985. There are many blocks and edges. We observe that, since none of them uses or redefines yy_chk, the infected program state of b_a has successfully been propagated to b_b along the edges. Finally, even though the statement that outputs the failure is far away from the fault position, *CP* successfully locates the fault. According to previous studies [11], both of these two faults frequently occur in C programs. *CP* seems to be effective in locating certain popular faults, although more experiments are required to confirm this conjecture.

For space reason, we do not describe the remaining faults in detail. The next six faults are located in lines 1030, 1361, 1549, 3398, 2835, and 11058, respectively. The code examination efforts for locating them are 1.12%, 8.50%, 7.25%, 21.19%, 13.82%, and 88.2%, respectively. The last fault, which results in 6 failures among 567 test cases, is found in line 12193. It is an incorrect static variable definition. Since this fault is seeded in a global definition statement and the compiler tool gcov fails to log its execution, we mark its directly affected statement (say, line 12193) as the fault position. However, *CP* needs to examine 93.55% of the code to locate this fault. We scrutinize the case and find it to be a coincidence. For 7 out of 567 test cases that do not reveal failures, this branch statement is never covered. For the 6 test cases that reveal failures and the remaining 560 passed test cases, both the true branch and the false branch are covered. For more than 90% of the cases, the number of times that each branch is covered is very close to each other (with less than 5% difference). It is hard for *CP* to distinguish these two edges. We view that this practical scenario provides a hint for further improving *CP*, even though the current experiment shows that *CP* is promising.

5.3 Threats to Validity

We used gcov to implement our tool in which coverage profiling is completely conducted. The generation of the equation set by the tool is relatively straightforward. The equations are solved using a standard Gaussian elimination implementation. We have implemented the peer techniques ourselves and checked that the implemented algorithms adhere strictly to those published in the literature. We have also conducted trial runs on toy programs with limited test cases to assure the implementations of *CP* and other peer techniques.

Currently, we follow [19] to use T-score when evaluating *CBI* and *SOBER*. Some researchers have reported limitations in T-score (see [9], for example). A P-score has been proposed in [27] and may be considered for future studies.

CP, *Tarantula*, *SBI*, and *Jaccard* produce ranked lists of statements, while *CBI* and *SOBER* generate ranked list of predicates. Consequently, the experiment has used two effectiveness metrics to report the results of different techniques. It is unsuitable to compare *CP* on a par with *CBI* and *SOBER*. In this connection, the comparison and the discussion of *CP* in relation to *CBI* and *SOBER* should be interpreted carefully.

We use flex, grep, gzip, and sed as well as their associated test suites to evaluate our technique. These programs are real-life programs with realistic sizes, and they have been used in previous studies [14][23][26]. It is certainly desirable to evaluate *CP* further on other real-life subject programs and scenarios.

6. RELATED WORK

Comparing program executions of a faulty program over different test cases and considering program states are frequently used fault-localization strategies. *Delta debugging* [9] isolates failure-causing inputs, produces cause effect chains and locates the root causes of failures. It considers a program execution (of a failure-causing test case) as a sequence of program states. Each state induces the next state, until a failure is reached. Since delta debugging is not a CBFL technique [23], we do not include it in our detail study. *Predicate switching* [26] is another technique to locate a fault by checking the execution states. It switches a predicate's decision at execution time to alter the original control flow of a failure-causing test case, aiming at locating a predicate such that a switch of the decision will produce correct output. Their latest version [14] works on the value set of all variables and the result looks promising.

Tarantula [15] first calculates the fraction of failed (passed) executions that executes the statement over all failed (passed) executions. It uses the former fraction over the sum of the two fractions as an indicator to represent the suspiciousness of individual statements. *CBI* [18] estimates the chance of a predicate being evaluated to be true in all executions and the chance of the predicate being evaluated to be true in only failed executions. It then calculates the increase from the former to the latter, and uses such an increase as the measure of the suspiciousness score of the predicate. *SOBER* [19] introduces the concept of evaluation bias to express the probability that a predicate is evaluated to be true in an execution. By collecting such evaluation biases of a statement in all failed executions and those in all passed executions, *SOBER* compares the two distributions of evaluation biases, and accordingly estimate how much the predicate is suspicious. Jones et al. [16] further use *Tarantula* to explore how to cluster test cases to facilitate multiple developers to debug a faulty program in parallel. Baudry et al. [6] observe that some groups of statements (known collectively as a dynamic basic block) are always executed by the same set of test cases. To optimize *Tarantula*, they use a bacteriologic approach to find out a subset of original test set that aims to maximize the number of dynamic basic blocks. Liblit et al. [2] further adapt *CBI* to handle compound Boolean expressions. Chilimbi et al. [8], in their work *Holmes*, conduct statistical fault localization using paths instead of predicates. Zhang et al. [29] empirically show that short-circuit rules in the evaluation of Boolean expressions may significantly affect the effectiveness of predicate-based techniques. In Yu et al.'s work [24], *CBI* has been adapted to the statement level.

Edge profiles have been developed for years. Bond et al. [7] propose a hybrid instrumentation and sampling approach for continuous path and edge profiling. It has been used in fault localization. Santelices et al. [21] investigate the effectiveness of using different program entities (statements, edges, and DU-pairs) to locate faults. They show that the integrated results of using different program entities may be better than the use of any single kind of program entity. Slicing is also a means of locating faults. Gupta et al. [13] propose to narrow down slices using a forward dynamic slicing approach. Zhang et al. [26] integrate forward and backward dynamic slicing approaches for debugging.

7. CONCLUSION

Fault localization is a process to find the faults in failed programs. Existing coverage-based fault-localization approaches use the statistics of test case executions to serve this purpose. They focus on individual program entities, generate a ranked list of their suspiciousness, but ignore the structural relationships among statements.

In this paper, we assess the suspiciousness scores of edges, and set up a set of linear algebraic equations over the suspiciousness scores of basic blocks and statements, which abstractly models the propagation of suspicious program states through control flow edges in a back-tracking manner. Such equation sets can be efficiently solved by standard mathematical techniques such as Gaussian elimination. The empirical results on comparing existing techniques with ours show that our technique can be effective.

We have further conducted a case study on a multi-fault program to examine the effectiveness of our technique, and find cases that inspire future work. In order to further enhance the effectiveness of our approach, another prospective is to extend our edge profile technique to cover path profiles or data flow profiles as well.

8. REFERENCES

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of TAICPART-MUTATION 2007*, pages 89–98. IEEE Computer Society Press, Los Alamitos, CA, 2007.

[2] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit. Statistical debugging using compound Boolean predicates. In *Proceedings of ISSTA 2007*, pages 5–15. ACM Press, New York, NY, 2007.

[3] G. K. Baah, A. Podgurski, and M. J. Harrold. The probabilistic program dependence graph and its application to fault diagnosis. In *Proceedings ISSTA 2008*, pages 189–200. ACM Press, New York, NY, 2008.

[4] T. Ball and S. Horwitz. Slicing programs with arbitrary control-flow. In *Proceedings of AADEBUG '93* volume 749 of Lecture Notes in Computer Science, pages 206–222. Springer, London, UK, 1993.

[5] T. Ball, P. Mataga, and M. Sagiv. Edge profiling versus path profiling: the showdown. In *Proceedings of POPL '98*, pages 134–148. ACM Press, New York, NY, 1998.

[6] B. Baudry, F. Fleurey, and Y. Le Traon. Improving test suites for efficient fault localization. In *Proceedings of ICSE 2006*, pages 82–91. ACM Press, New York, NY, 2006.

[7] M. D. Bond and K. S. McKinley. Continuous path and edge profiling. In *Proceedings of MICRO 2005*, pages 130–140. IEEE Computer Society Press, Los Alamitos, CA, 2005.

[8] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, and K. Vaswani. Holmes: effective Statistical Debugging via Efficient Path Profiling. In *Proceedings of ICSE 2009*. IEEE Computer Society Press, Los Alamitos, CA, 2009.

[9] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of ICSE 2005*, pages 342–351. ACM Press, New York, NY, 2005.

[10] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. *Empirical Software Engineering*, 10 (4): 405–435, 2005.

[11] J. A. Durães and H. S. Madeira. Emulation of software faults: a field data study and a practical approach. *IEEE Transactions on Software Engineering*, 32 (11): 849–867, 2006.

[12] S. G. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Control*, 12 (3): 185–210, 2004.

[13] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *Proceedings ASE 2005*, pages 263–272. ACM Press, New York, NY, 2005.

[14] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In *Proceedings of ISSTA 2008*, pages 167–178. ACM Press, New York, NY, 2008.

[15] J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Proceedings of ASE 2005*, pages 273–282. ACM Press, New York, NY, 2005.

[16] J. A. Jones, M. J. Harrold, and J. F. Bowring. Debugging in parallel. In *Proceedings of ISSTA 2007*. ACM Press, New York, NY, pages 16–26, 2007.

[17] A. J. Ko and B. A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of ICSE 2008*, pages 301–310. ACM Press, New York, NY, 2008.

[18] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of PLDI 2005*, pages 15–26. ACM Press, New York, NY, 2005.

[19] C. Liu, L. Fei, X. Yan, S. P. Midkiff, and J. Han. Statistical debugging: a hypothesis testing-based approach. *IEEE Transactions on Software Engineering*, 32 (10): 831–848, 2006.

[20] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of ASE 2003*, pages 30–39. IEEE Computer Society Press, Los Alamitos, CA, 2003.

[21] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *Proceedings of ICSE 2009*. IEEE Computer Society Press, Los Alamitos, CA, 2009.

[22] J. M. Voas. PIE: a dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18 (8): 717–727, 1992.

[23] X. Wang, S. C. Cheung, W. K. Chan, and Z. Zhang. Taming coincidental correctness: refine code coverage with context pattern to improve fault localization. In *Proceedings of ICSE 2009*, pages 45–55. IEEE Computer Society Press, Los Alamitos, CA, 2009.

[24] Y. Yu, J. A. Jones, and M. J. Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *Proceedings of ICSE 2008*, pages 201–210. ACM Press, New York, NY, 2008.

[25] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of SIGSOFT 2002 / FSE-10*, pages 1–10. ACM Press, New York, NY, 2002.

[26] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *Proceedings of ICSE 2006*, pages 272–281. ACM Press, New York, NY, 2006.

[27] Z. Zhang, W. K. Chan, T. H. Tse, P. Hu, and X. Wang. Is non-parametric hypothesis testing model robust for statistical fault localization? *Information and Software Technology*, to appear.

[28] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang. Capturing propagation of infected program states. Technical Report TR-2009-14. Department of Computer Science, The University of Hong Kong, Hong Kong, 2009.

[29] Z. Zhang, B. Jiang, W. K. Chan, and T. H. Tse. Debugging through evaluation sequences: a controlled experimental study. In *Proceedings of COMPSAC 2008*, pages 128–135. IEEE Computer Society Press, Los Alamitos, CA, 2008.

[30] D. G. Zill and M. R. Cullen. *Advanced Engineering Mathematics*. Jones and Bartlett Publishers, Sudbury, MA, 2006.