Available online at www.ijpe-online.com

International Journal of Performability Engineering

vol. 16, no. 8, August 2020, pp. 1289-1298
DOI: 10.23940/ijpe.20.08.p16.12891298

Spectrum-based Security Bug Localization by Analyzing Error Propagation

Mengyu Ji*, Song Huang*, and Zhanwei Hui*

Command and Control Engineering Institute, Army Engineering University, Nanjing, 211101, China

Abstract

Software security bug is one of the key threats to the security of software systems. Isolating security bugs that may be potential security bugs is important. We formalize a program error propagation based model (PEP), which used to be applied to locate integer bug and our contribution are as follows: We formulate a theory model based on the mechanism of how the security bug triggers the program error propagation and propose a security bug localization approach by applying spectrum-based fault-localization (SFL) technique, a novel method to locate software fault to alleviate false negative and false positive problem. Our experimental results show that：1)Our model is more effective than present ones to locate nearly 97% integer bug and buffer overflow which are the main security bugs by examining 50% codes on average; 2) Compared with the traditional techniques, SFL can find 100% of integer bugs and buffer overflow so it is a promising, technology roadmap to reduce false negative and false positive for locating security bugs.

Keywords: accuracy; security bug; fault localization; error state propagation

(Submitted on May 22, 2020; Revised on June 24 2020; Accepted on July 21, 2020)

1. Introduction

Software security is one of the most important software quality properties. In the era of internet, developers have to pay more attention to the security issues in software development process [1-5]. Once the security bug is discovered by the attacker, it can be used to access or destroy the system without authorization, which threads the security of the computer system. Therefore, using security bug localization technology to find possible security bugs in software is an important guarantee to maintain the security of software systems. Locating these security bugs has been considered as a time-consuming but important process. Many traditional techniques, such as [6] and [7], are proposed to detect security bugs based on the features learned by prior knowledge, so false negative and false positive problem still exists.

In our previous paper [8], we proposed a innovative model, IntRank, to estimate every suspiciousness score of the statements. In this paper, we formulate IntRank into a precise error propagation (PEP) model based on Total Probability Theorem and PageRank [9], which is a link analysis technique to find popular Web pages. In our experiment, we focus on two main security related bugs involving integer bug and buffer overflow via applying mutant analysis [10]. Since SFL can locate the bug without any priori-knowledge, we evaluate existing various SFL models including ours. The result shows that ours is more effective than other models in locating security related bug which are injected into the programs. Meanwhile, almost all the SFL can reduce false negative and false positive when compared with the traditional security bug detection tools. In this paper, the terms 'software' and 'program' are used interchangeably. Also 'fault' and 'bug' are used interchangeably and defined based on [11].

The remainder of this article is organized in the following manner: we describe the research background in Section 2 and our approach in Section 3. In Section 4, we set up an experiment evaluation. Finally, conclusions and future work are presented in Sections 5 and 6, respectively.

* Corresponding author.
E-mail address: 729803720@qq.com, hs0317@163.com, hzw_1983821@163.com

2. Background

2.1. Security Bug Detection Technology

In the past years, many security bug detection approaches have been proposed. Security testing can be broadly categorized into static and dynamic based on complexity of program and type of integer bugs to be found [12]. Static testing is applied without running the software, and dynamic testing is performed by running the program.

Static analysis tool: ITS4 [7] and Flawfinder [13] are tools based on lexical analysis and maintains a database to read the database at runtime and compare with the source codes. SPLINT [14-15] is the expansion of LCLINT tool for detecting buffer overflows and other security threats. It employs several lightweight static analyses. Ebriman N. Ceesay [16] and other static analysis tools run on the Cqual to track distrust data, and use data flow sensitive analysis to generate an alarm when untrusted data is used. To avoid security bugs, some programming protection mechanisms such as compiler extensions and security C++ classes are widely used.

Dynamic analysis tool: Dynamic testing analysis varies in techniques; most popular are fuzz testing [17], concolic testing [18], and search-based testing [19]. Fuzz testing can be broadly divided into black box and white box fuzz testing depending on weather gaining knowledge of internals of the program. Concolic execution tools such as Symbolic JPF [20] test as many paths as possible; hence, they try to avoid untested paths that may lead to security bugs have been proposed to be very effective in security bug detection.

Many existing techniques are effective in security bug detection. Static analysis techniques may suffer from generation of false positive and false negative, because it does not run the program and only focus on the formal rules that define the features of security bugs. Existing dynamic techniques alway generate test data from every possible path to trigger the security bugs not to find the root cause fault; false positive and false negative also exist.

2.2. Overview of SFL Techniques

Various spectrum-based fault-localization (SFL) techniques [21-25] have been proposed to locate bugs in the program. The program spectrum information [26], including passed test coverages and failed test coverages, is used to compute suspiciousness [23] of individual program entities (Maybe statements [27], branches [28], and predicates [29]. The definition of program spectruis is $< s_1(E_1, t), s_2(E_2, t), \cdots, s_i(E_i, t), \cdots, s_n(E_n, t), r >$. $s_i(E_i, t) = 1$ denotes test t covers E_i and $s_i(E_i, t) = 0$ denotes test t does not cover E_i. E_i is the entity of P (such as statement, define-use pair, branch and so on). $r = 1$ means the test passed; $r = 0$ means test failed.

There are many popular statement spectrum level techniques such as Tarantula [30] and Ochiai [31]; empirical study shows that Ochiai is generally considered more effective than Tarantula. However, the above techniques consider individual program entities independent of each other; the performance is poor in some cases. Zhao et al. [32] state that using only individual coverage information may not reveal all the execution paths. Kai Yu [33] proposed model LOUPE, which uses multiple spectra-specific models including branch coverage and Du-pair coverage.

In recent years, researchers have proposed many fault-localization models by analyzing error propagation. Zhenyu Zhang [34] proposed CP model to estimate suspiciousness score of every statement by analyzing error propagation using the control edge in control flow graph (CFG). In [35], the propagation of failure states is analyzed by using variable reaching-definition analysis and dependence analysis. [36] proposed a page-rank algorithm based on fault localization by using the fault influence network. In fault influence network, the node is Java method.

The result of SFL is a ranked statement by the suspiciousness score that provides a clue to find out the root cause fault. In this paper, we apply SFL to help programmers locate security bugs.

2.3. PageRank Algorithm and Total Probability Theorem

In this part, we give the introduction of PageRank algorithm and total probability theorem.

PageRank: PageRank assumes that "highly linked pages should be regarded as more important than pages being seldom linked" [9]. Let p be a page and $M(p)$ be the set of pages p has links to. $N(q)$ is the set of pages linked to p. It uses $P(p)$ to denote the ranking of a page p. So, a link to a page is a expression of the importance of that page. $P(p)$ can be expressed as:

$$P(p) = (1-n) + n \sum_{q \in M(p)} P(q) \times \frac{1}{|N(q)|} \tag{1}$$

$|N(q)|$ is the number of pages in the set $N(q)$. The argument n is a damping factor introduced to simulate the probability that a user continues to browse pages using the links.

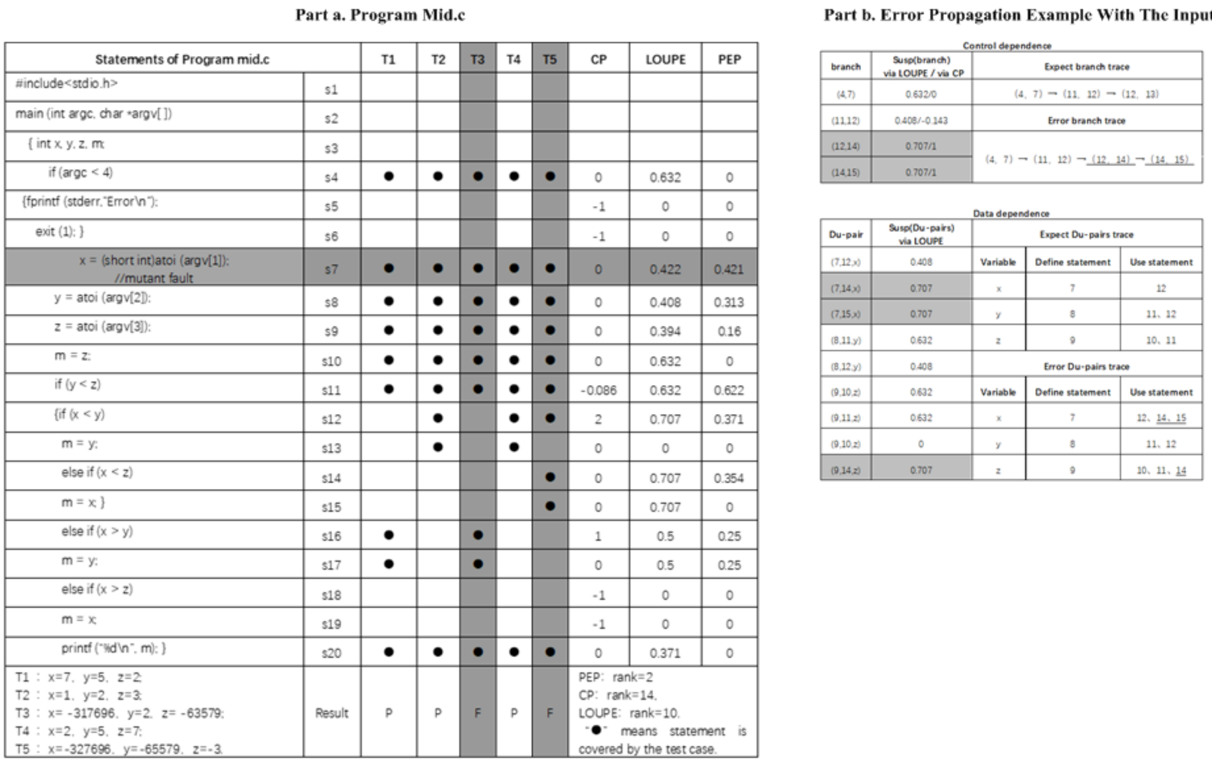
Total probability theorem [37]: Suppose $B_1, \cdots, B_n$ are mutually disjoint events on a probability space (Ω, F, P) such that $\Omega = \cup_{i=1} B_i$ and $P(B_i) > 0$ for each $i \in [1, n]$. Then for any event A,

$$P(A) = \sum_{i=1}^{n} P(A|B_i)P(B_i) \tag{2}$$

3. Our Approach

3.1. Motivating Example

In this part, we review the motivating example [8, 38] and explain why present models, CP and LOUPE, sometime perform poorly (As shown in Figure 1). The motivating example program is from [22]. We found that in Figure 1, the suspiciousness score of faulty *statement*(s_7) is not the maximum value in neither CP or LOUPE.



Part a. Program Mid.c

Statements of Program mid.c		T1	T2	T3	T4	T5	CP	LOUPE	PEP
#include<stdio.h>	s1								
main (int argc, char *argv[ ])	s2								
{ int x, y, z, m;	s3								
if (argc < 4)	s4	●	●	●	●	●	0	0.632	0
{fprintf (stderr,"Error\n");	s5						-1	0	0
exit (1); }	s6						-1	0	0
x = (short int)atoi (argv[1]); //mutant fault	s7	●	●	●	●	●	0	0.422	0.421
y = atoi (argv[2]);	s8	●	●	●	●	●	0	0.408	0.313
z = atoi (argv[3]);	s9	●	●	●	●	●	0	0.394	0.16
m = z;	s10	●	●	●	●	●	0	0.632	0
if (y < z)	s11	●	●	●	●	●	-0.086	0.632	0.622
{if (x < y)	s12		●		●	●	2	0.707	0.371
m = y;	s13		●		●		0	0	0
else if (x < z)	s14					●	0	0.707	0.354
m = x; }	s15					●	0	0.707	0
else if (x > y)	s16	●		●			1	0.5	0.25
m = y;	s17	●		●			0	0.5	0.25
else if (x > z)	s18						-1	0	0
m = x;	s19						-1	0	0
printf ("%d\n", m); }	s20	●	●	●	●	●	0	0.371	0
T1 : x=7, y=5, z=2; T2 : x=1, y=2, z=3; T3 : x= -317696, y=2, z= -63579; T4 : x=2, y=5, z=7; T5 : x=-327696, y=-65579, z=-3.	Result	P	P	F	P	F	PEP: rank=2 CP: rank=14, LOUPE: rank=10. "●" means statement is covered by the test case.		

Part b. Error Propagation Example With The Input

Control dependence

branch	Susp(branch) via LOUPE / via CP	Expect branch trace
(4,7)	0.632/0	(4, 7) → (11, 12) → (12, 13)
(11,12)	0.408/-0.143	**Error branch trace**
(12,14)	0.707/1	(4, 7) → (11, 12) → <u>(12, 14)</u> → <u>(14, 15)</u>
(14,15)	0.707/1	

Data dependence

Du-pair	Susp(Du-pairs) via LOUPE	Expect Du-pairs trace		
(7,12,x)	0.408	**Variable**	**Define statement**	**Use statement**
(7,14,x)	0.707	x	7	12
(7,15,x)	0.707	y	8	11、12
(8,11,y)	0.632	z	9	10、11
(8,12,y)	0.408	**Error Du-pairs trace**		
(9,10,z)	0.632	**Variable**	**Define statement**	**Use statement**
(9,11,z)	0.632	x	7	12、<u>14、15</u>
(9,10,z)	0	y	8	11、12
(9,14,z)	0.707	z	9	10、11、<u>14</u>

Figure 1. Motivating Program Mid.c [8, 38]

The main reasons why CP [34] and LOUPE are not precise in this example are as follows.

(1) CP model is only based on control dependence situation, so it cannot separate statement in the block. That means the suspiciousness scores calculated by CP of statements in one block are same. As shown in Table 2, statements from line 7 to 10 belong to the same bock; the scores are the same, '0'. The maximum score is assigned to the statement in line 12, as shown in Figure 1 Part a; the branch trace starts to deviate from the expected when using CP model. On the other hand, it ignores data dependence situation. Dataflow propagation is another important type of program state propagation.

(2) LOUPE model is based on the hypothesis that when the fault is activated, the statement has the control or data

dependence relation with the fault one immediately leads to the error branch trace or the Define-use trace. The suspiciousness score of the non-predicate statement is related to the mean score of the define-use pair which it belongs to. And the suspiciousness score of the branch-predicate statement is affected by the suspiciousness score of its branch. In Figure 1 Part b, the suspiciousness score of the define-use pairs (9, 14, *z*) and (7, 14, *x*) in line14 (s_{14}) are both 0.707. According to LOUPE, $susp_d(s_{14})$ = (0.707 + 0.707)/2 = 0.707. On the other hand, the suspiciousness score of the branch (14, 15) is 0.707. s_{14} only belongs to branch (14, 15); its suspiciousness score is calculated as $suspc(s_{14})$ = 0.707 - 0 = 0.707. So, $susp(s_{14})$ = max($suspc(s_{14})$ and $suspd(s_{14})$) = 0.707. s_{14} has the maximum suspiciousness score but it is not the faulty statement. In LOUPE, the faulty statement s_7 does not have the maximum suspiciousness score because $susp_d(s_7)$ = (0.707 + 0.707 + 0.707 + 0.408 + 0 + 0)/6= 0.422 = $susp(s_7) < susp(s_{14})$. s_7 is not the predicate, so we can make $susp(s_7) = susp_d(s_7)$. The faulty statement s_7 belongs to define-use pairs (7, 18, *x*) and (7, 19, *x*), of which suspiciousness scores are both 0. These two suspiciousness scores affect the suspiciousness score of s_7.

In the program execution, the faulty statement infects the program state (value or control flow), and the state is propagated to the failure position with the program execution. In Figure 1 Part b, T5: x = -327696, y = -65579, and z = -3; this is the failed test. When variable x is assigned by the function atoi() return value in program entrance, the data type of the return value is forced to change from int to short int. So *x* = -327696 is out of the range of short int and Truncation Error is activated. The error value is used by the following statements. When the predicate statement uses the error value, the control flow is also the error value. At last, the program failure occurs. As shown in Figure 1, when the fail test is T5, the error and actual branch trace is (4, 7)→(11, 12)→(12, 14)→(14, 15) which differs from the expect branch trace (4, 7)→(11, 12)→(12, 13) (always unknown in practice). s_{12} is a fault relevant statement , and s_{14} and s_{15} are failure relevant statements.

Above all, we make hypothesize that: 1) The precise error propagation model includes at least two situations: control flow and data flow; 2) Suspiciousness scores of the two statements in the same data dependence relation (control dependence) may have some statistical relation; 3) During a program execution, a fault relevant program entity may propagate program states to adjacent entities, which are more related to a failure using dependence edges connecting to that entity; 4) The event that a program statement *A* is the fault relevant can cause the event that statement *B* is failure relevant if (*A*, *B*) is a dependence edge during a program execution.

3.2. Details of PEP Model

Based on the above intuitive hypotheses, PEP has 4 processes. First is getting the edge profile, second is calculating the ratio of propagation using the edge, third is solving the statements suspiciousness score by constructing an equation set based on PageRank Equation (1), and fourth is producing the report.

We give the details of the model in Figure 2, and some preliminaries that may be involved next in *Preliminaries* table in Figure 2.

(1) The infected program states the propagation probability calculation. There are various kinds of similarity coefficient formulas to calculate the suspiciousness score of program entity so choosing an effective similarity coefficient is important.

In CP model [34], the edges suspiciousness score of e_i is $\Delta s_i(e_i)$:

$$\Delta s_i(e_i) = \frac{\overline{s_i'(e_i)} - \overline{s_i(e_i)}}{\overline{s_i(e_i)} + \overline{s_i'(e_i)}}, \overline{s_i'(e_i)} = \frac{|failed(e_i)|}{|T_f|}, \overline{s_i(e_i)} = \frac{|passed(e_i)|}{|T_p|}$$

$|failed(e_i)|$ is the sum frequency of the failed tests cover ei . $|T_f|$ is the sum of failed tests and $|passed(e_i)|$ is the sum frequency of the passed tests cover ei . $|T_p|$ is the sum of passed tests. We chose Ochiai to calculate $susp(e_i)$.

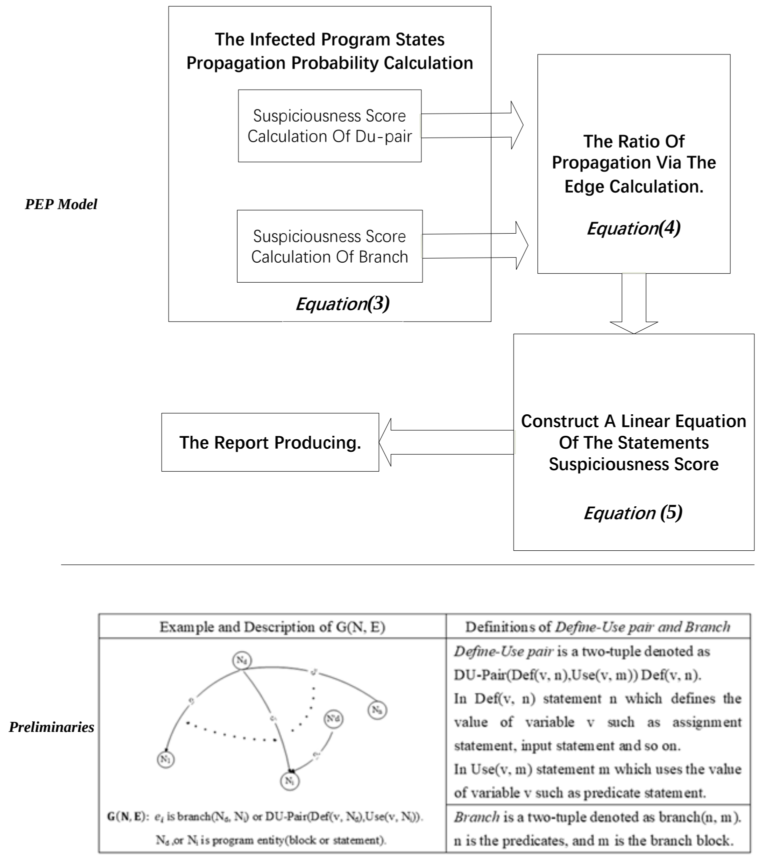
$$susp(e_i) = \frac{\overline{s_i'(e_i)}}{\sqrt{(\overline{s_i'(e_i)} + \overline{s_i(e_i)})|T_f|}}, (\overline{s_i'(e_i)} + \overline{s_i(e_i)})|T_f| \neq 0 \tag{3}$$

According to [39], since $\Delta s_i(e_i)$ can be denoted by Tarantula using increasing function $\Delta s_i(e_i)$= 2 $susp_{Tarantula}$ - 1, $\Delta s_i(e_i)$ is equivalent with Tarantula in sorting suspiciousness score. Ochiai is more effective than Tarantula, so Ochiai is more effective than $\Delta s_i(e_i)$.

Proof: We change the form of $\Delta s_i(e_i)$ as follows:

$$\Delta s_i(e_i)+1=\frac{\overline{s_i'(e_i)}-\overline{s_i(e_i)}}{\overline{s_i(e_i)}+\overline{s_i'(e_i)}}+\frac{\overline{s_i'(e_i)}+\overline{s_i(e_i)}}{\overline{s_i(e_i)}+\overline{s_i'(e_i)}}=2\times\frac{\overline{s_i'(e_i)}}{\overline{s_i(e_i)}+\overline{s_i'(e_i)}}$$

$$=2\times\frac{\frac{|failed(e_i)|}{|T_f|}}{\frac{|passed(e_i)|}{|T_p|}+\frac{|failed(e_i)|}{|T_f|}}=2\times susp_{Tarantula}$$


PEP Model
The Infected Program States Propagation Probability Calculation
Suspiciousness Score Calculation Of Du-pair
Suspiciousness Score Calculation Of Branch
Equation(3)
The Ratio Of Propagation Via The Edge Calculation.
Equation(4)
Construct A Linear Equation Of The Statements Suspiciousness Score
Equation (5)
The Report Producing.
Preliminaries
N_d
N_1
N_i
$N'd$
N_0

Example and Description of G(N, E)	Definitions of *Define-Use pair and Branch*
	Define-Use pair is a two-tuple denoted as DU-Pair(Def(v, n),Use(v, m)) Def(v, n). In Def(v, n) statement n which defines the value of variable v such as assignment statement, input statement and so on. In Use(v, m) statement m which uses the value of variable v such as predicate statement.
G(N, E): e_i is branch(N_d, N_i) or DU-Pair(Def(v, N_d),Use(v, N_i)). N_d ,or N_i is program entity(block or statement).	*Branch* is a two-tuple denoted as branch(n, m). n is the predicates, and m is the branch block.

Figure 2. Overview Of Model PEP

(2) The ratio of propagation using the edge calculation is shown. For a given N_i in $G(N, E)$, error state can be transferred to N_i using its every possible incoming edge as shown in Figure 2. The ratio of propagation using the edge is the probability of the error state is propagated using $e_i(N_d, N_i)$ among all the N_i's incoming edges. We get the following expression:

$$Weight\big(e_i(N_d,N_i)\big)=\frac{susp(e_i(N_d,N_i))}{\sum_{\forall e_i(*,N_i)} susp(e_i(*,N_i))} \tag{4}$$

When e_i is *branch*(n_i, n_j),

$$Weight(n_i,n_j)=\frac{susp_{Ochiai}(branch(n_i,n_j))}{\sum_{\forall branch(*,n_j)} susp_{Ochiai}(branch(*,n_j))}$$

When e_i is *DU-Pair*(*Def*(v, n_i), *Use*(v, n_j)),

$$weight(v,n_i,n_j)=\frac{susp_{Ochiai}(DU-Pair(Def(v,n_i),Use(v,n_j)))}{\sum_{\forall DU-Pair(*,Use(*,n_j))} susp_{Ochiai}(DU-Pair(*,Use(*,n_j)))}$$

As illustrated in the motivating example, $susp_{Ochiai}(DU-Pair(y,8,11)) = susp_{Ochiai}(DU-Pair(z,9,11))$= 0.632, so $Weight(y,8,11) = \frac{susp_{Ochiai}(DU-Pair(y,8,11))}{\sum_{\forall DU-Pair(*,Use(*,11))} susp_{Ochiai}(DU-Pair(*,Use(*,11)))}$ = 0.632/(0.632 + 0.632) = 0.5

(3) The statements suspiciousness score calculation based on Total Probability Theorem and PageRank. The node n_i in $G(N, E)$ (see Figure 2) may have its successor nodes $\{n_j, j = 1, 2, \cdots, n\}$. $\{susp(n_j), j = 1, 2, \cdots, n\}$ can be considered as mutually disjointed events on a probability space. Using Equation (2), let $P(A) = susp(n_i)$, and $P(B_i)$ can also be denoted as $susp(n_j)$. $P(A|B_i)$ can be denoted as $Weight(e_i(n_i, n_j))$. When a program statement n_i is executed, there are two main scenarios: Firstly, its successor n_j is executed as follows (1):

$$susp_s^{(1)}(n_i) = \sum Weight(e_i(n_i,n_j)) \times susp_s(n_j), j = 1,2,\cdots$$

Secondly, it halts and has no successor statement. In this scenario, we use $\sum_{\forall e(*,n_i)} susp_e$ to estimate that $susp_s^{(2)}(n_i)$ and $e(*,n_i)$ are the incoming edge of n_i. $\mathrm{susp}_s^{(2)}(n_i)$ is the probability of n_i fault in scenario 2.

We integrate the above two scenarios into one expression as follows. For n_i, $|e_{in}|$ is the number of its incoming edges and $|e_{out}|$ is the number of its outgoing edges. The more incoming edges, the more likely the second scenario could occur. The more outgoing edges, the more probable the first scenario could also occur. Using Equation(1), let $n = \frac{|e_{out}|}{|e_{in}|+|e_{out}|}$, $P(q) \times \frac{1}{|N(q)|}$is denoted as $\mathrm{Weight}(e_i(n_i,n_j)) \times \mathrm{susp}_s(n_j)$; we get Equation(5).

$$susp_s(n_i) = (1 - \frac{|e_{out}|}{|e_{in}| + |e_{out}|})susp_s^{(2)}(n_i) + \frac{|e_{out}|}{|e_{in}| + |e_{out}|}\sum Weight(e_i(n_i,n_j)) \times susp_s(n_j) \quad (5)$$

At last, we construct an equation set in which $susp_s(n_i)$ is the unknown number by (4). We solve the equation set by Gaussian elimination and get $susp_b(n_i)$ and $susp_{du}(n_i)$ for every n_i in P.

(4) The report is generated based on the following steps. 1) Models integrating. For every n_i in Program, we compare $susp_b(n_i)$ with $susp_{du}(n_i)$ and get $susp(n_i) = \max(susp_b(n_i), susp_{du}(n_i))$. $susp_b(n_i)$ is the suspiciousness score in control dependence scenario and $susp_{du}(n_i)$ is the one in data dependence scenario; 2) Exception handling. When n_i does not belong to any branch, $susp_b(n_i) = 0$. When it does not belong to any Du-pair, $susp_{du}(n_i) = 0$; 3) Suspiciousness score sorting. We sort the suspiciousness scores of every statement in descending order by rank. The statement with the lowest rank is on the top of the report and checked first by programmer. When the suspiciousness scores of statements in P are the same value, the rank of the last one is assigned to all these statements. In motivating example, the rank of faulty statement s_7 is 2 when applying PEP model. Meanwhile the rank of s_7 is 14 calculated by CP and 10 by LOUPE. So PEP is superior to CP and LOUPE in the fault localization accuracy. That means the programmer only checked 2 lines of codes to find the fault, rather than 14 by CP and 10 by LOUPE.

When the coincidental correctness is covered by a test, the test result is still passed, which is against the hypothesis of PEP. PEP also cannot locate the fault introduced by the missing code.

4. Empirical Evaluation

In this section, we conduct a controlled experiment to evaluate the effectiveness of our technique. Research questions addressed in this experiment are as follows:

RQ1: How is the accuracy of our model and existing models for security bug related fault localization?

RQ2: Compared with present security bug detection techniques, does SFL techniques (including our model) perform with lower false positive and false negative?

4.1. Experimental Setup

(1) Subject programs (shown in Table 1). Subject programs TCAS and schedule are from Siemens Test-suite, which is most frequently used for subject in previous works obtained from SIR [40]. TCAS is an aircraft collision, avoidance system and schedule is priority scheduler. They are both kernel of mission critical software. The branch and Du-pair coverage information

is produced by gcc with WET [41].

Table 1. Subject program

Program name	Num of versions	Loc	Size of tests
tcas	19	173	85
schedule	8	410	226

(2) Security bug and Test data. Injected into subject programs, we construct 24 versions of faulty programs. In TCAS, variable 'Own_Tracked_Alt' and variable 'Other_Tracked_Alt' respectively represent the altitudes of aircraft and the altitudes of detected flights. The two variables can accept malicious data. We select one or more statements and then change the variable types of 'Own_Tracked_Alt' and 'Other_Tracked_Alt' in these statements from 'int' type to 'short int' type in order to simulate truncation error(a kind of integer bug). v1-v6 are single fault versions and v7-v19 are multi-fault versions. We add 32767 to the fourth and the sixth parameter in test data obtained from 'tcas/testplans.alt/testplans-bigcov/suite.271' to generate malicious data. In 8 faulty versions of schedule, we inject a buffer overflow bug by changing the constant with large number or modify the data type of function malloc() return value and so on. v20-v25 are single fault and v26-v27are multi-fault versions. Test data is from 'schedule/testplans.alt/testplans-bigcov/suite.1000'.

4.2. Evaluation Metric

The accuracy of fault location model can be expressed as the number of codes that must be checked before the programmer finds the faulty statement by the order of the rank in the report list. The fewer statements the programmer checks, the more accurate the model. In this paper, accuracy is defined as follows [39]:

$$q = \frac{rank}{total}$$

'rank' is the rank in the report list, and 'total' is the total number of executable statements. The accuracy of security bug detection can be denoted as false negative and false positive. False negative:

$$FN = 1 - \frac{D}{E}$$

'D' is the quantity of faults which are detected as security bug in the report. *'E'* is the total quantity of faults in the program. False positive: $FP = 1 - \frac{D}{P}$. *'D'* is the quantity of faults which are detected as security bug in the report. *'P'* is the total quantity of faults in the report. The report of fault localization is a priority sequence, but security bug detection reports no priority. We define $FP = (rank - 1)/total$ for single-fault versions and $FP = (rank(n) - n)/total$ for multi-fault versions. Suppose n faults in a multi-fault program, $rank(n)$ is the quantity of codes that must be examined when the last fault is found and $(rank(n) - n)$ statements which are false positive are not faulty in the report.

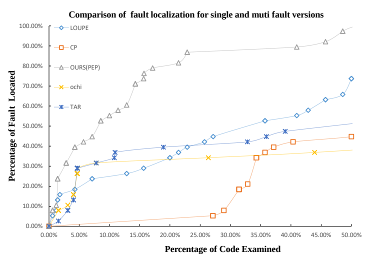
We define the $P_{FN=0}$ as the average FN of different techniques. $P_{FN=0} = |versions(FN = 0)|/|versions(all)|$. $|versions(FN = 0)|$ is the quantitiy of versions in which $FN = 0$. $|versions(all)|$ is the total quantities of versions.

4.3. Results and Analysis

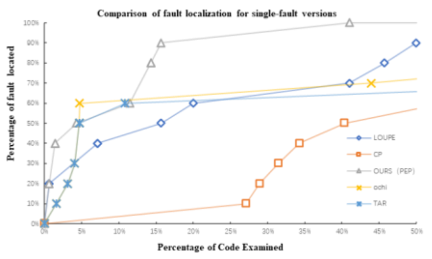
RQ1: For TCAS (v1-v19) and schedule (v20-v27), our model performs better than existing models. Illustrated in Figure 3-A, 3-B, 3-C, the horizontal axis represents the higher bound of each score range, which indicates the percentage of code that need to be examined when the fault is found. The vertical axis represents the percentage of located defects. When programmer examines 50% codes, nearly 97% faults can be located using our model (Figure 3-A). Other models perform more poorly than ours (73.6% faults located by using LOUPE, 44.7% faults located by using CP, 47.3% faults located by using Tarantula, and 36.8% faults located by using Ochiai) when 50% codes examined. Figure 3-B and Figure 3-C illustrate single-fault and multi-fault versions respectively. Our model can locate 100% single fault by examining only 50% codes for single versions in this experiment, which is more effective than existing models. By examining 50% codes， 90% faults located by LOUP abd less than 60% faults located by CP. For multi-fault versions, by examining only 50% codes, more than 96% faults, which is much more than the rest existing models, can be located by our model.

RQ2: Two typical security bug detection tools are selected as research objects. Splint is a static analysis tool that scans the source code. Valgrind-memcheck is a dynamic detection tool that uses the dynamic tracking taint propagation technology to monitor untrusted data, if untrusted data used in memory allocation function is caused by empty pointer use, memory leakage and so on. The tool also reports bugs (Figure 4). The detection results of the two tools are as follows. Splint can detect truncation error and sign error but only integer overflow (underflow). Valgrind-memcheck can find where the security bug occurs in malloc(), but only finds the function where the security bug is located, as illustrated in Figure 3-a, 3-b, 3-c.

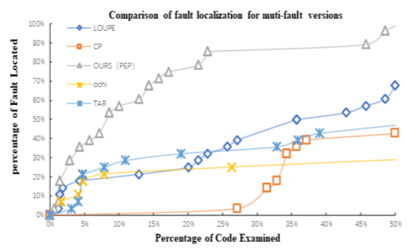
Part 1 Comparison of Fault localization



A

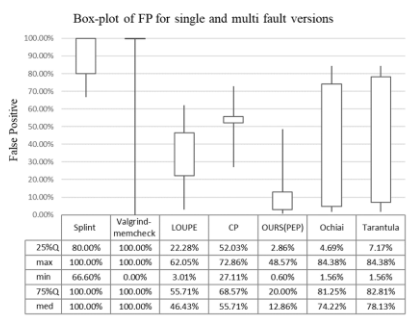


B



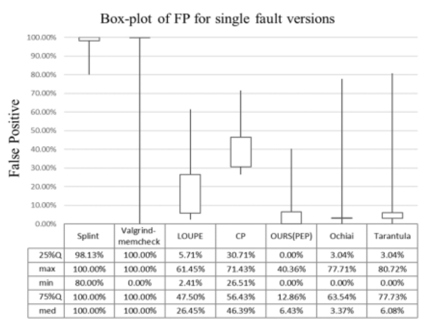
C

Part 2 Comparison of False Positive



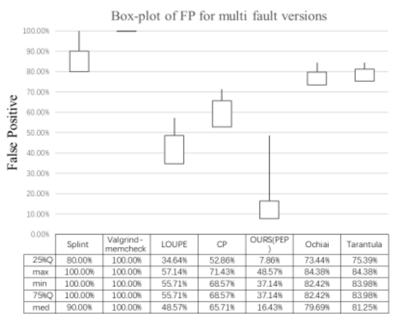
	Splint	Valgrind-memcheck	LOUPE	CP	OURS(PEP)	Ochiai	Tarantula
25%Q	80.00%	100.00%	22.28%	52.03%	2.86%	4.69%	7.17%
max	100.00%	100.00%	62.05%	72.86%	48.57%	84.38%	84.38%
min	66.60%	0.00%	3.01%	27.11%	0.60%	1.56%	1.56%
75%Q	100.00%	100.00%	55.71%	68.57%	20.00%	81.25%	82.81%
med	100.00%	100.00%	46.43%	55.71%	12.86%	74.22%	78.13%

a



	Splint	Valgrind-memcheck	LOUPE	CP	OURS(PEP)	Ochiai	Tarantula
25%Q	98.13%	100.00%	5.71%	30.71%	0.00%	3.04%	3.04%
max	100.00%	100.00%	61.45%	71.43%	40.36%	77.71%	80.72%
min	80.00%	0.00%	2.41%	26.51%	0.00%	0.00%	0.00%
75%Q	100.00%	100.00%	47.50%	56.43%	12.86%	63.54%	77.73%
med	100.00%	100.00%	26.45%	46.39%	6.43%	3.37%	6.08%

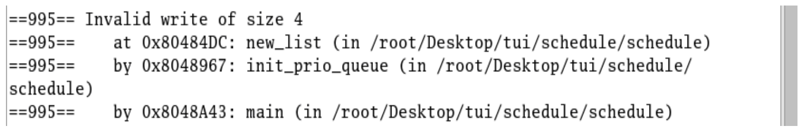
b



	Splint	Valgrind-memcheck	LOUPE	CP	OURS(PEP)	Ochiai	Tarantula
25%Q	80.00%	100.00%	34.64%	52.86%	7.86%	73.44%	75.39%
max	100.00%	100.00%	57.14%	71.43%	48.57%	84.38%	84.38%
min	100.00%	100.00%	55.71%	68.57%	37.14%	82.42%	83.98%
75%Q	100.00%	100.00%	55.71%	68.57%	37.14%	82.42%	83.98%
med	90.00%	100.00%	48.57%	65.71%	16.43%	79.69%	81.25%

c

Figure 3. Experiment result



```
==995== Invalid write of size 4
==995==    at 0x80484DC: new_list (in /root/Desktop/tui/schedule/schedule)
==995==    by 0x8048967: init_prio_queue (in /root/Desktop/tui/schedule/
schedule)
==995==    by 0x8048A43: main (in /root/Desktop/tui/schedule/schedule)
```

Figure 4. Report of Valgrind-memcheck

General speaking, SFL is more effective than existing tools (Splint [14] and Valgrind-memcheck [42]). Our model is the best of SFL. As illustrated in Figure 3-a, FP range by our model is from 0.6% to 48.57%, which is better than Splint (66.6%-100%) and Valgrind-memcheck (0%-100%). Max FP of Splint and Valgrind-memcheck are both 100%. Max FP is less than 100% in all SFL. Median is 100% for Splint and Valgrind-memcheck, which is much more than 12.86% by our model. For both single-fault and multi-fault versions in Figure 3-b and Figure 3-c, FP of our model has better statistically stability and is at a lower level. As shown in Table 2, all SFL can find 100% the injected bugs in 100% versions. But in 25.9% versions, Splint can find all the inject *faults*(*FN* = 0). In only 11.1% versions, Valgrind-memcheck detects all the inject *faults*(*FN* = 0)

Table 2. Comparison of $P_{FN=0}$

Versions	Splint	Valgrind-memcheck	Ours(PEP)	LOUPE	CP	Ochiai	Tarantula
v1-v27	25.9%	11.1%	100%	100%	100%	100%	100%

4.4. Threats to Validity

The overhead cost of monitoring dynamic data dependencies and control dependencies is too much in this paper. But in order to detect security bugs, the cost is affordable. The categories of security bugs that we study are integer bug and buffer overflow. Many other security bugs need to be studied by SFL.

5. Conclusion and Future Work

We apply SFL techniques to security bug detection for source level and find that security bugs can be located effectively by SFL. For the integer bug and buffer overflow, our model is more accurate than existing models and performs better than existing security bug detection with low *FP* and *FN*. But for the normal bugs from Siemens Test-suite, our model is more effective in multi-fault versions. As studied in [43], an elaborate information model has no strong correlation with localization effectiveness. PEP is proposed for the fault which can trigger the error state propagation. If the error state causes the failure when fault is active, models using individual coverage information just like LOUPE are more effective. All present SFL have lower *FP* and *FN* than Splint and Valgrind-memcheck. In the future, an effective test generation for fault localization [44] need to be discussed. Test oracle can be alleviated by metamorphic testing. In metamorphic testing [45], two or more test execution must be run and how effectiveness is affected by mechanism should be studied. The application for the binary level security bug detection by SFL needs to be discussed in the future.

Acknowledgements

This work was supported in part by National Key R&D Program of China (No.2018YFB1403400), Natural Science Foundation of China (No: 61702544) and Natural Science Foundation of Jiangsu Province, China (No: BK 20160769, BK20141072).

References

1. B. Potter and G. McGraw, “Software Security Testing, Security and Privacy Magazine,” *IEEE*, Vol. 2, No. 5, pp. 81-85, 2004
2. C. Wysopal, L. Nelson, D. D. Zovi, and E. Dustin, “The Art of Software Security Testing,” Symatec Press
3. N. Davis, W. Humphrey, S. T. Redwine, G. Zibulski, and G. McGrew, “Processes for Producing Security Software,” *IEEE Security and Privacy*, Vol. 2, No. 3, pp. 18-25, 2004
4. R. C. Sercord, “Secure Coding in Cand C++,” Addison Wesley and Person Education Asia, 2006
5. W. E. Wong, X. Li, and P. A. Laplante, “Be More Familiar with Our Enemies and Pave the Way Forward: A Review of the Roles Bugs Played in Software Failures,” *Journal of Systems and Software*, Vol. 133, pp. 68-94, 2017
6. Y. M. Xia, J. Luo, and M. X. Zhang, “Security Security bug Detection Study based on Static Analysis,” *Computer Science*, Vol. 33, No. 10, pp. 279-283, 2006
7. T. L. Wang, T. Wei, and Z. Q. Lin, “IntScope: Automatically Detecting Integer Overflow Security Bug in x86 Binary using Symbolic Execution,” in *Proceedings of 2009 Network and Distributed System Security Symposium*, San Diego: ISOC, 2009
8. M. Y. Ji, S. Huang, and Z. W. Hui, “Research on Spectra-based Integer Bug Localization,” *Chinese Journal of Computers*, Vol. 35, No. 10, October 2012
9. L. Page, S. Brin, R. Motwani, and T. Winograd, “The PageRank Citation Ranking: Bringing Order to the Web,” Stanford InfoLab Publication, Stanford University, Palo Alto, CA, (Available at http://dbpubs.stanford.edu/pub/1999-66, 1999)
10. F. Zeng, L. Mao, Z. Chen, et al., “Mutation-based Testing of Integer Overflow Vulnerabilities,” in *Proceedings of the 8th International Conference on Wireless Communications*, *Networking and Mobile Computing*, pp. 24-26, Beijing, China, 2009
11. A. Avizienis, J. C. Laprie, B. Randell, et al., “Basic Concepts and Taxonomy of Dependable and Secure Computing,” *IEEE Transactions on Dependable and Secure Computing*, Vol. 1, No. 1, March 2004
12. F. Anwer, M. Nazir, and K. Mustafa, “Security Testing,” Trends in Software Testing, Springer Science+Business Media Singapore, 2017
13. D. A. Wheeler, “Flawfinder,” (http://www.dwheeler.com/flawfinder/(2001))
14. Splint, a static analysis tool, (http://splint.org/)
15. X. C. Lu, G. Li, K. Lu, and Y. Zhang, “High-Trusted-Software-Oriented Automatic Testing for Integer Overflow Bugs,” *Journal of Software*, Vol. 21, No. 2, pp. 179-193, February 2010
16. E. N. Ceesay, J. N. Zhou, M. Gertz, et al., “Using Type Qualifiers to Analyze Untrusted Integers and Detecting Security Flaws in C Programs,” in *Proceedings of DIMVA 2006*, LNCS 4064, pp. 1-16, 2006
17. R. McNally, K. Yiu, D. Grove, and D. Gerhardy, “Fuzzing: The State of the Art,” 2012
18. C. S. Păsăreanu, N. Rungta, and W. Visser, “Symbolic Execution with Mixed Concrete-Symbolic Solving,” in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pp. 34-44, ACM, 2011
19. A. Avancini and M. Ceccato, “Comparison and Integration of Genetic Algorithms and Dynamic Symbolic Execution for Security Testing of Cross-Site Scripting Vulnerabilities,” *Information of Software Technology*, Vol. 55, No. 12, pp. 2209-2222, 2013
20. C. S. Psreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, et al., “Combining Unit-Level Symbolic Execution and System-Level Concrete Execution for Testing Nasa Software,” in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pp. 15-26, ACM, 2008
21. R. Abreu, A. Gonz_Alez, P. Zoeteweij, and A. J. van Gemund, “Automatic Software Fault Localization using Generic Program Invariants,” in *Proceedings of ACM Symposium on Applied Computing*, pp. 712-717, Ceara, Brazil, March 2008
22. H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, “Fault Localization using Execution Slices and Dataflow Tests,” in *Proceedings of the 6th International Symposium on Software Reliability Engineering*, pp. 143-151, Toulouse, France, October 1995
23. Z. A. Al-Khanjari, M. R. Woodward, H. A. Ramadhan, and N. S. Kutti, “The Efficiency of Critical Slicing in Fault Localization,”

Software Quality Journal, Vol. 13, No. 2, pp. 129-153, June 2005
24. W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The DStar Method for Effective Software Fault Localization," *IEEE Transactions on Reliability*, Vol. 63, No. 1, pp. 290-308, 2014
25. W. E. Wong, V. Debroy, and B. Choi, "A Family of Code Coverage-based Heuristics for Effective Fault Localization," *Journal of Systems and Software*, Vol. 83, No. 2, pp. 188-208, 2010
26. R. Abreu, P. Zoeteweij, and A. J. van Gemund, "Spectrum-based Multiple Fault Localization," in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, pp. 88-99, Auckland, CA, USA, November 2009
27. R. Abreu, B. Hofer, A. Perez, and F. Wotawa, "Using Constraints to Diagnose Faulty Spreadsheets," *Software Quality Journal*, Vol. 23, pp. 297-322, May 2014
28. H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault Localization using Execution Slices and Dataflow Tests," in *Proceedings of the 6th International Symposium on Software Reliability Engineering*, pp. 143-151, Toulouse, France, October 1995
29. H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Debugging with Dynamic Slicing and Backtracking," *Software–Practice Experience*, Vol. 23, No. 6, pp. 589-616, June 1993
30. J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization for Fault Localization," in *Proceedings of the 23rd International Conference on Software Engineering*, pp. 71-75, Ontario, BC, Canada, May 2001
31. R. Abreu, P. Zoeteweij, and A. J. van Gemund, "An Evaluation of Similarity Coefficients for Software Fault Localization," in *Proceedings of Pacific RIM International Symposium on Dependable Computing*, pp. 39-46, Riverside, CA, USA, December 2006
32. H. He, J. Ren, R. Zhao, and H. He, "Enhancing Spectrum-based Fault Localization using Fault Influence Propagation," *IEEE Access*, Vol. 8, 2020
33. K. Yu, X. Meng, L. Q. Gao, et al., "Locating Faults using Multiple Spectra-Specific Models," in *Proceedings of SAC'11*, TaiChung, Taiwan, March 21-25, 2011
34. Z. Zhang, W. K. Chan, T. H. Tse, et al., "Capturing Propagation of Infected Program States," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (*ESEC 7/FSE-17*), pp. 43-52, ACM Press, New York, NY, 2009
35. T. T. Wang, K. C. Wang, X. H. Su, and L. Zhang, "Invariant based Fault Localization by Analyzing Error Propagation," *Future Generation Computer Systems*, Vol. 94, pp. 549-563, May 2019
36. W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A Survey on Software Fault Localization," *IEEE Transactions on Software Engineering*, Vol. 42, No. 8, August 2016
37. L. Hong, "Law of Total Probability and Bayes' Theorem in Riesz Spaces," Mathematics Subject Classification 2010
38. mid.c, a program, (http://www.static.cc.gatech.edu/aristotle/Tools/Aristotle/samples/mid.c)
39. R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the Accuracy of Spectrum-based Fault Localization," in *Proceedings of Testing*: *Academic and Industrial Conference Practice and Research Techniqu*, pp. 89-98, IEEE Computer Society Press, Los Alamitos, CA, 2007
40. Siemens test suite, a test data set, (http://sir.unl.edu/portal/index.html)
41. WET, an advanced infrastructure, (http://wet.cs.ucr.edu/index.html)
42. Valgrind-memcheck, a dynamic analysis tool, (http://www.valgrind.org/)
43. Y. Lei, X. G. Mao, M. Zhang, J. Ren, and Y. Jiang, "Toward Understanding Information Models of Fault Localization: Elaborate is Not Always Better," in *Proceedings of IEEE 41st Annual Computer Software and Applications Conference*, 2017
44. S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed Test Generation for Effective Fault Localization," in Proceedings of IEEE *International Symposium Software Testing Analysis*, pp. 49-60, Trento, Italy, July 2010
45. S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortes, "A Survey on Metamorphic Testing," *IEEE Transactions on Software Engineering*, Vol. 42, No. 9, September2016

Mengyu Ji is a Master Degree graduated of Army Engineering University. His research interests include software testing and software fault localization.

Song Huang is a professor of Army Engineering University. His research interests include software testing and software engineering.

Zhanwei Hui is a post-doctoral of Army Engineering University. His research interests include software testing and software engineering.