IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received March 6, 2019, accepted March 12, 2019, date of publication March 18, 2019, date of current version April 9, 2019.

*Digital Object Identifier 10.1109/ACCESS.2019.2905842*

# Identify Silent Data Corruption Vulnerable Instructions Using SVM

**NA YANG AND YUN WANG**
School of Computer Science and Engineering, Southeast University, Nanjing 210000, China
Key Laboratory of Computer Network and Information Integration, Ministry of Education, Nanjing 210000, China

Corresponding author: Yun Wang (ywang_cse@seu.edu.cn)

This work was supported in part by the National Hi-Tech Project, China, under Grant 61320605, and in part by the Collaborative Innovation Center of Novel Software Technology and Industrialization.

**ABSTRACT** Silent data corruption (SDC) is the most insidious and harmful result type of soft error. Identify program vulnerable instructions (PVIns) that are likely to cause SDCs is extremely significant on selective software-based protection techniques. However, current identification techniques require tremendous fault injections or have non-negligible differences in performance among different programs as well as different program inputs. This paper proposes PVInsiden to reduce the cost of fault injection and improve the adaptability for programs and program inputs. Machine learning is used to learn a classifier which predicts whether an instruction is a PVIns. Partial fault injection is applied to generate a training dataset, reducing the cost of fault injection. The feature engineering, including selecting features and transforming the selected features into quantifiable representations is explored. Furthermore, the framework of learning the classifier is given. The experimental results show that PVInsiden only uses 35% fault injections to identify 85% PVIns with 80% precision, reducing the cost of fault injection efficiently. PVInsiden also shows high performance of precision, recall, and f0.5-score for different programs as well as different program inputs.

**INDEX TERMS** Silent data corruption, SVM, soft error, single event upset, fault injection.

## I. INTRODUCTION

A single event upset (SEU) is a change of state caused by one single ionizing particle striking a sensitive node in a micro-electronic device [1], [2]. Soft error is caused by a SEU, growing with Moore‘s Law. It is classified into four categories, i.e. benign, crash, hang and silent data corruption (SDC). Benign means that errors are masked and programs output right results. Crash and hang are two explicit errors, making programs produce indications such as stop execution or non-stop running. Different from the other three types, SDC has no explicit error phenomena but incorrect program outputs. It is the most serious and non-ignorable error type [3], [4].

In recent years, various techniques have been proposed to detect SDCs, for instance, duplication-based techniques and assertion-based techniques. The basic idea of duplication-based techniques is to duplicate original instructions with different registers or memories. If a mismatch between the original instruction and the duplicated instruction occurs at a certain synchronization point, an error is detected. Assertion-based detection techniques take an invariant assertion as a detector. Once the assertion is not satisfied, an error is detected [1]. Program variables or instructions have different contributions to SDCs, facilitating the study of selective detection and protection strategies [5]. In order to gain a high SDC detection rate with a lower protection cost, instructions that are likely to cause SDCs need to be identified and protected. In the rest of this paper, the instructions whose SDC vulnerabilities exceed SDC-vulnerability-threshold are called ‘‘Program Vulnerable Instructions (PVIns)’’. Although several approaches have been proposed to get the SDC vulnerability of the instruction and identify PVIns [6], [7], the following challenges still remain to hinder these approaches into practice.

(i) The high cost of fault injection.

The basic approach of determining whether an instruction is PVIns is full fault injection approach. Faults are injected to every bit of an operand of every dynamic instruction. Due to that programs may consist of a large number of dynamic instructions, making full fault injection approach take too much time to practice [8], [9]. For example, table 1 shows

The associate editor coordinating the review of this manuscript and approving it for publication was Francesco Tedesco.


2169-3536 © 2019 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

statistics of print_tokens.c and replace.c in Siemons suit and WTransform.c (wavelet transform) of full fault injection approach for an operand of 32 bits. Although the programs only have several hundreds of code lines, the number of injections may reach 200000 times larger than the size of source code.

**TABLE 1. Statistics of programs under full fault injection approach.**

| Name | print_tokens | replace | WTransform |
|---|---|---|---|
| Lines of code | 556 | 564 | 1000 |
| Number of static instructions | 1000 | 1300 | 6000 |
| Number of dynamic instructions | 1176 | 4061 | $7.6 \times 10^6$ |
| Number of injections | 37632 | 129952 | $2.432 \times 10^8$ |

(ii) The complex process of generating relational data.

Machine-learning-based approaches are used to get the vulnerability of the instruction or the program [7], [10]. Classification, regression and propagation analysis are used by SDCTune to get the SDC vulnerability of the instruction [7]. In order to get the relational data used by classification and regression, a large number of fault injections have to be conducted on similar instructions of other programs in advance. This makes SDCTune rather complex. What's more, SDCTune is launched when a new end instruction type is tackled.

(iii) The non-negligible difference in performance among different programs as well as different program inputs.

Fault injection is not necessary for dynamic modeling to compute the SDC vulnerability of an instruction. While dynamic modeling suffers from unstable performance for programs. According to our experimental observations, ePVF [6], which is a typical method of dynamic modeling, behaves dramatically in precision and f0.5-score from 60% to 80% for different program inputs and also different programs. The difference in performance is non-negligible.

In this paper, PVInsiden is proposed to identify PVIns with lower cost of fault injection and better applicability for programs. The process of PVInsiden is simple and light-weighted. Our main contributions are summarized as follows:

(i) A novel way is put forward to determine the minimum number of fault injections. A SVM classifier is trained and applied. Partial fault injection is applied to generate a training dataset. Experimental results show that given a program, 20% fault injections is sufficient to learn a classifier if the SDC vulnerability of registers and memories is accurate. In general, PVInsiden uses 35% fault injections at most to identify PVIns with 80% precision, 85% recall and 81% f0.5-score on average.

(ii) Instruction features that impact the SDC vulnerability of an instruction are selected and quantified. The dataset used to learn a classifier is generated according to the target program itself. Thus, the dataset is easy to build.

(iii) Experiments are conducted with various programs and program inputs to evaluate PVInsiden. The result shows that PVInsiden outperforms ePVF in program applicability.

## II. RELATED WORK

There is a bunch of literature on identifying the SDC vulnerability of the instruction or the program by fault injection, dynamic modeling or machine learning.

A fault injection approach simulates single event upsets by altering bits in registers or memories at runtime. It makes a program run in an error state till the program stops executing or generates anomalous symptoms. In order to reduce the cost of fault injection, various efforts have been made to refine the process. SDCInfer employs heuristics to infer SDC-vulnerable instructions based on instruction traces [11]. More specifically, it adjusts fault injection plans dynamically to reduce the cost of fault injection. SDCInfer can determine whether an instruction has the probability to lead to SDCs, but the probability is not quantified. CriticalFault eliminates wasteful derated fault injections that result in masks [12]. It reduces the injection space by pruning, whereas only 29% fault injections can be avoided.

PVF is a modeling approach that eliminates micro-architectural dependency from architectural vulnerability to get the vulnerability of the program [13]. It requires no fault injection, whereas it is poor in predicting the SDC vulnerability as it does not distinguish between SDCs and crashes. ePVF is an enhancement of PVF that eliminates crash-causing bits from ACE bits to get a tighter estimate of the SDC vulnerability of the instruction [6]. It is still poor as it only considers segment faults of crash faults. Further, the performance results may vary regarding different programs as well as different inputs for a specific program. This is because segment faults have different frequencies in crashes.

Gokcen et al. employ machine learning to model scientific application behaviors at a large scale based on experiments at a small scale [8]. The research work of [8] aims to understand the fault vulnerability of the program rather than that of the instruction. SDCTune uses classification and regression to predict the SDC vulnerability of store instructions and comparison instructions [7]. The SDC vulnerability of other instructions is obtained by propagation analysis, whereas it is program-specific in some circumstances. Vishnu *et al.* [10] use support vector machine to predict whether a multi-bit permanent or transient main memory fault will result in an error. Unfortunately, SDCs are not distinguished from other error types.

## III. PROBLEM DEFINITION

We assume that only single bit flip within register files or memories is considered in this paper. We also adopt an assumption that at most one fault occurs during a program's execution, which is assumed in other research studies, such as [6] and [7]. Faults in the opcode are not considered in this paper as they always cause illegal opcode exceptions [11]. The SDC vulnerability is defined as the probability of a SDC when a fault occurs.

The problem of identifying PVIns of a program is a binary classification problem. More specifically, for an instruction, if its SDC vulnerability exceeds specified threshold, it is considered as a PVIns. Otherwise it is not. Thus, the problem is defined as a binary classification problem on a collection of dataset. There are some classification models available in machine learning, such as decision-tree classification, neural networks classification [17] and support vector machine [18]. Among those models, support vector machine (SVM) is an outstanding model in soft error field. The SVM is selected in this paper.

Given a training dataset of $n$ points of the form $(\vec{x_1}, y_1), \cdots, (\vec{x_n}, y_n)$, where $\vec{x_i}$ is a multidimensional real vector and $y_i$ are either 1 or -1, each representing the class of the point $\vec{x_i}$. A hyperplane is defined as the set of points $\vec{x}$ satisfying $\vec{w}.\vec{x} - b = 0$, where $\vec{w}$ denotes the normal vector to the hyperplane, and $b$ is the offset from the origin. The core of SVM is finding the best hyperplane that separates the data with the maximum margin.

In the case of linearly separable data, the SVM select two parallel hyperplanes that separate the two classes of the data, so that the distance between them is as large as possible. The best hyperplane is the hyperplane that lies halfway between them. Geometrically, the distance between these two hyperplanes is $\frac{2}{\|\vec{w}\|}$ [19]. As each data point must lie on the correct side of the margin, the best hyperplane must satisfy $y_i(\vec{w}.\vec{x_i} - b) - 1 \geq 0$. Formally, the best hyperplane is defined by the primal optimization problem: $\min_{\vec{w},b} \|\vec{w}\|$, subject to $y_i(\vec{w}.\vec{x_i} - b) - 1 \geq 0$. This optimization problem is solved by lagrange duality. In the case of linearly inseparable data, the SVM map the input data into a high dimensional feature space by a kernel function to make the data separable. After that, a finest separating hyperplane is constructed in the dimensional feature space. Besides, the SVM uses a penalty term that determines the importance of a mismatch classification to weak the impact of outliers.

In order to find a solution to the problem of identifying PVIns by SVM, several important questions should be answered first: (i) How to get a dataset? What size of the dataset is proper? (ii) What are the important features that should be selected to learn a classifier model? And how to transform the selected features into quantifiable representations? (iii) How to train a classification model? These questions are addressed in the upcoming sections.

## IV. PARTIAL FAULT INJECTION

### A. OBSERVATIONS

Full fault injection is usually used to get instruction vulnerability in a program. The cost of full fault injection is dramatic. Partial fault injection is a natural choice. In order to make sure to what extent fault injection should conduct, we explore the relationship between partial fault injection and the performance of a classifier in this section. The metrics we are interested in are precision, recall and f-score, which are widely used to evaluate the performance of a classifier. In particular, f0.5-score is used as f-score. A classifier plays an important role in characterizing features of instruction vulnerability.

A training dataset is collected from a target program, whose PVIns is required to be identified. A classifier is used to compute the SDC vulnerability of the remaining instructions of the target program and to identify PVIns.

Denote the ratio of the number of training samples to the number of total instructions of a target program as $\theta$. Fig. 1 presents the relationship between the performance of a classifier and $\theta$ with print_tokens2.c in Siemens suite as a program example. We observe that precision experiences a great rise when $\theta$ goes from 5% to 20%. Then, it basically only has a slight increase. f0.5-score shares a similar changing trend as precision. Recall keeps the increase nearly all the time.

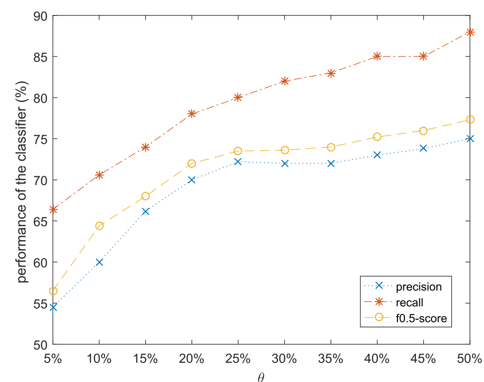



**FIGURE 1.** The relationship of the performance of the classifier and $\theta$.

Fig. 1 tells us that the performance of a classifier does not always go up along with the increasing number of training samples. More than a certain number of training samples will not improve the performance of a classifier obviously.

This observation indicates that the contribution of different number of training samples to the performance of a classifier is different. Partial fault injection is feasible in order to satisfy performance requirements with a lower cost of fault injection. Fault injection is applied to partial instructions of a target program to get their ground truth labels and a training dataset is generated. Furthermore, we try to find the minimum $\theta$, denoted as $\theta_m$.

### B. THE PROCESS OF GETTING $\theta_M$

To get $\theta_m$, a classifier based on SVM is trained iteratively on programs. The processing includes three steps. The first step is to build a sampling set. A small part of instructions are selected as the training set. The remaining instructions are considered as the testing set. We refer to it as the sampling stage. The next step is to train a classifier. The training set is fed into a SVM model to learn a classifier. This is the training stage. In the following verifying step, the testing set is utilized to assess the performance of the classifier. If the performance of the classifier changes obviously, the process goes back to

**TABLE 2.** Notations.

| Symbol | Description |
|---|---|
| $p$ | The program used to get $\theta_m$. |
| $t$ | The threshold of the SDC vulnerability of $p$. |
| $size(p)$ | The number of instructions of $p$. |
| $ins(p)$ | The instruction set of $p$, $ins(p) = \{ins_1, \cdots, ins_i, \cdots, ins_{size(p)}\}$. |
| $sdcvul\,(ins_i)$ | The SDC vulnerability of $ins_i$. |
| $label\,(ins_i)$ | The ground truth label of $ins_i$. |
| $p_label\,(ins_i)$ | The predictive label of $ins_i$. |
| $a_0$ | The percentage of instructions sampled in the first iteration. |
| $a_k$ | The percentage of instructions sampled in the current iteration. |
| $d$ | The difference in percentage of the sampled instructions in a consecutive sampling iteration. |
| $samp_0$ | Flag of whether it is the first time to sampling. |
| $trainflag$ | Flag of whether launch the next iteration. |
| $dataset$ | Set of samples. |
| $trainset$ | Set of the instructions used as training samples. |
| $testset$ | Set of the instructions used as testing samples. |
| $changed$ | Flag of whether the performance of the classifier changes obviously. |
| $PVInsset(p)$ | Set of PVIns of $p$. |
| $unPVInsset(p)$ | Set of unPVIns of $p$. |

the sampling stage to launch the next iteration. Otherwise, $\theta_m$ is found. This is the verifying stage.

Every stage of getting $\theta_m$ is described in detail and the corresponding pseudo-code is presented in Algorithm 1.

#### 1) SAMPLING STAGE

The instruction set is obtained by assembling and disassembling. Full fault injection is conducted on instructions to get their ground truth labels. Next, the samples are generated and put to the dataset (Line2-11). To find $\theta_m$, 10% instructions are initially selected as the training samples. If more samples are needed in the next iteration, an additional 5% samples are selected into the training set (Line 12, Line 16-26).

#### 2) TRAINING STAGE

Feature values are transformed to the range of 0 and 1 by scaling (Line 28). Then, the training set is used to train a classifier (Line 29).

#### 3) VERIFYING STAGE

At this stage, the performance of the classifier learned in this iteration is computed. It is compared with that of the classifier learned in the last iteration. If the performance changes obviously, the process goes back to the sampling stage to improve the performance of the classifier by adding additional training samples (Line 30-34). Otherwise, the process ends and $\theta_m$ is obtained.

**Algorithm 1** Get $\theta_m$

```
Input: p, t
Output: θm
 1: Get ins(p)
 2: for i = 0 → size(p) do
 3:     calculate sdcvul (ins_i) by full fault injection
 4:     if sdcvul (ins_i) > t then
 5:         label (ins_i) = 1
 6:     else
 7:         label (ins_i) = 0
 8:     end if
 9:     generate a sample s, denoting ins_i
10:     add s to dataset
11: end for
12: a0 = 10%, d = 5%, θm = 0
13: trainflag = true, sample0 = true
14: trainset = ∅
15: while trainflag do
16:     if sample0 then
17:         ak = a0
18:         sample0 = false
19:     else
20:         ak = d
21:     end if
22:     θm = θm + ak
23:     for i = 0 → ak ∗ size(p) do
24:         select s from {dataset}-{trainset}
25:         add s to trainset
26:     end for
27:     testset={dataset}-{trainset}
28:     transform trainset and testset by scaling
29:     train a SVM classifier using trainset
30:     if changed then
31:         trainflag = true
32:     else
33:         trainflag = false
34:     end if
35: end while
```

## V. FEATURE ENGINEERING

Previous studies manifest that critical instructions in a program are likely to lead to SDCs after their operands are corrupted by single event upsets. In this section, how to define and identify key instruction features is addressed.

### A. INSTRUCTION FEATURES

The critical program points of data flow and control flow propagations are connector instructions and branch instructions [1]. Connector instructions characterize the operation process of interactive data, which includes returned value, parameters and global variables [11]. Corrupted data in connector instructions is likely to lead to SDCs. Thus, connector instructions are key instructions this paper concerns. If an instruction is a connector instruction, it is labeled as *connins*.

With regard to the quantifiable representation of *connins*, if an instruction includes interactive data, the value of *connins* of the instruction is set to 1. Otherwise, the value is set to 0. For example, SET AL is a connector instruction which sets a returned value before a function call. Therefore, its *connins* value is set to 1.

Comparison instructions impact bits of flag registers, which determine the consequent jump performed by a branch instruction [1]. Altering bits of an operand of a comparison instruction may incur a control flow error and a SDC. Thus, comparison instructions are key instructions this paper concerns. If an instruction is a comparison instruction, it is labeled as *cmpins*.

Regarding the quantifiable representation of *cmpins*, if an instruction is a comparison instruction, the value of *cmpins* is set to 1. Otherwise, the value is set to 0. For example, in Fig. 2, TEST AL, BL (Line 1) is a comparison instruction that before JNE ab (Line 2). Therefore, the *compins* value of TEST AL, BL is set to 1.

```
1:    a1:  test   al,bl
2:    a3:  jne    ab
3:    a5:  mov    BYTE PTR [ebp-0x1],0x40
4:    a9:  jmp    f0
5:    ab:  mov    eax,DWORD PTR [ebp+0xc]
6:    ......
7:    b6:  mov    DWORD PTR [ecx],edx
8:    ......
9:    f0:  movzx  eax,BYTE PTR [ebp-0x1]
10:   f3:  sub    esp, 0x38
11:   f6:  mov    eax, ecx
```

**FIGURE 2.** An example of instruction codes.

In summary, two types of instructions, namely connector instructions and comparison instructions, are critical instructions.

### B. REGISTER-RELATED FEATURES

Registers have different SDC vulnerabilities. For example, the probability that the result type is SDC after injecting a fault to EAX is higher than that of ESP, indicating that EAX has a higher SDC vulnerability than ESP. The research work in [14] demonstrates that 80% fault injections on ESP and EBP lead to crashes rather than SDCs, owning to the fact that ESP is used as a stack pointer and EBP is used as a stack frame base pointer. This means that both EBP and ESP have a lower SDC vulnerability. Thus, the total SDC vulnerabilities of registers used by an instruction is labeled as an feature. It is represented by *regvulofins*.

The quantifiable representation of *regvulofins* is described by taking an instruction called *ins* as an example. The *regvulofins* value of *ins* is the sum of SDC vulnerabilities of registers it uses. Therefore, SDC vulnerability of every register used by *ins* needs to be acquired.

Denote the accurate SDC vulnerability of a register called *reg* as *evuln*(*reg*). To get *evuln*(*reg*), full fault injection experiments have to be conducted on instructions which use *reg*. This incurs a large number of fault injections. To reduce the cost of fault injection, partial fault injection approach is applied to calculate the SDC vulnerability of *reg*. Only a certain number of instructions that use *reg* are injected faults. The SDC vulnerability of *reg* is calculated by (1), where *countsdc*(*reg*) is the number of injected faults that lead to SDCs, and *suminject*(*reg*) is the number of total injections. As soon as the SDC vulnerability of the register is obtained, the *regvulofins* value of *ins* is denoted by (2), where *REG* is the set of the register used by *ins*, *size*(*REG*) stands for the size of *REG*, and $r_i$ is the $i$-th element of set *REG*.

$$vuln(reg) = \frac{countsdc(reg)}{suminject(reg)}. \tag{1}$$

$$regvulofins(ins) = \sum_{i=1}^{size(REG)} vuln(r_i). \tag{2}$$

Partial fault injection brings a difference between *vuln*(*reg*) and *evuln*(*reg*). Revisit EAX in Fig. 2 to give an illustration. In full fault injection, faults are injected to EAX in Line 1, Line 5, Line 9 and Line 11 respectively to get *evuln*(*eax*). In partial fault injection, *regpct*(*eax*) is defined as the ratio of the number of instructions that are selected for fault injection to the number of instructions using EAX. Suppose that *regpct*(*eax*) is 50% and instructions in Line 5 and Line 11 are selected. Table 3 shows the statistics of partial fault injection related to EAX with 32 bits.

**TABLE 3.** A comparison of full fault injection and partial fault injection.

| | full fault injection | | | | partial fault injection | |
|---|---|---|---|---|---|---|
| Line number of instructions | 1 | 5 | 9 | 11 | 5 | 11 |
| Injection times of the instruction | 32 | 32 | 32 | 32 | 32 | 32 |
| Type=SDC | 10 | 9 | 10 | 10 | 9 | 10 |
| Sum of injections | 128 | | | | 64 | |
| vuln(eax) | 39/128 ≈ 0.3046 | | | | 19/64 ≈ 0.297 | |
| regerror(eax) | 0 | | | | 2.6% | |

Notice that the number of fault injections decreases greatly from 128 to 64. Further, there is a difference between *evuln*(*eax*) and *vuln*(*eax*). The difference is calculated by (3). In order to explore a better trade-off between the number of fault injections and the accuracy of *vuln*(*reg*), experimental analysis is conducted in section VI-B to obtain the optimal *regpct* for registers.

$$regerror(reg) = \frac{|vuln(reg) - evul(reg)|}{evul(reg)}. \tag{3}$$

Restrepo-Calle *et al.* [15], [16] point out that the lifetime of a register is related to reliability. They gain a better trade-off between performance, code size, and fault coverage by duplicating registers with long lifetime. Thus, the total lifetime of registers used by an instruction is taken as another feature. It is denoted by *reglifeofins*.

The lifetime of every register in *ins* needs to be specified in order to quantify *reglifeofins*. The lifetime of a register is the sum of living intervals during one program execution. A living interval starts with a write operand and ends with the last read operand which precedes the next write operand

or the end of the program execution [15], [16]. With living intervals, the lifetime of a register is expressed by (4), where $l_j$ is the $j$-th living interval of *reg* and $livecount(reg)$ is the number of living intervals of *reg*. The *reglifeofins* value of *ins* is expressed by (5).

$$lifetime(reg) = \sum_{j=1}^{livecount(reg)} l_j. \quad (4)$$

$$reglifeofins(ins) = \sum_{i=1}^{size(REG)} lifetime(r_i). \quad (5)$$

### C. MEMORY-RELATED FEATURES

Memory cells are also suffered by single event upsets. Similar to register-related features, the total SDC vulnerabilities of memory cells used by an instruction, which is denoted as *memvulofins* and the total lifetime of memory cells used by an instruction, which is denoted as *memlifetimeofins*, are two features to characterize memory. The quantifiable representations of the two instruction features are similar to those of register-related features.

Memory cells store returned values, parameters and local variables. Since returned values and parameters have already been considered in connector instructions [1], only memory cells that associate with local variables are considered here.

### D. OTHER FEATURES

In addition to the aforementioned selected features, the number of operands and the execution times of an instruction are also selected as features.

### E. CLASSIFIER TRAINING

With the obtained $\theta_m$ and feature engineering, the process of training a classifier for a target program is described in Algorithm 2. $P$ denotes a target program and $T$ is the threshold of the SDC vulnerability of $P$.

Firstly, feature values are acquired (Line 2-4). Then, $\theta_m$ instructions are selected and full fault injection is applied on them to get their labels. Labels are combined with the corresponding features to generate a training set (Line 5-15). Next, a SVM classifier is trained by the training set (Line 16). Finally, the trained classifier is used to classify the remaining instructions (Line 17-24).

**Algorithm 2** Process of Training a Classifier for a Target Program

**Input:** $P, T, \theta_m$
**Output:** $PVInsset(P)$

```
1:  Get ins(P)
2:  for i = 0 → size(P) do
3:      Obtain the feature values of ins_i
4:  end for
5:  for i = 0 → θ_m * size(P) do
6:      select ins_i from ins(P)
7:      calculate sdcvul (ins_i) by full fault injection
8:      if sdcvul (ins_i) > t then
9:          label (ins_i) = 1, add ins_i to PVInsset
10:     else
11:         label (ins_i) = 0, add ins_i to unPVInsset
12:     end if
13:     generate a sample s, denoting ins_i
14:     add s to trainset
15: end for
16: training a classifier by trainset
17: for ins_j not in trainset do
18:     get p_label (ins_j) by the classifier
19:     if p_label (ins_j) = 1 then
20:         add ins_j to PVInsset
21:     else
22:         add ins_j to unPVInsset
23:     end if
24: end for
```

## VI. EXPERIMENTAL ANALYSIS

To make sure the feasibility of our proposed approach, experimental analysis is conducted, including evaluating $\theta$ to find $\theta_m$, performance of trained classifiers and adaptability of PVInsiden. What's more, the cost of fault injection of PVInsiden is presented.

### A. EXPERIMENTAL SETUP

Pin is a binary instrumentation framework for IA-32 and x86-64 instruction set. It has been widely used in soft error field [20]. Pin is employed to carry out fault injection experiments. Our experiments are performed on Dell Workstation with i7 processor running Ubuntu10.04. Programs considered here are replace.c (performs string matching and replacement), print_tokens.c (performs lexical analysis) and tot_info.c (computes statistics over input data) in Siemens suite.

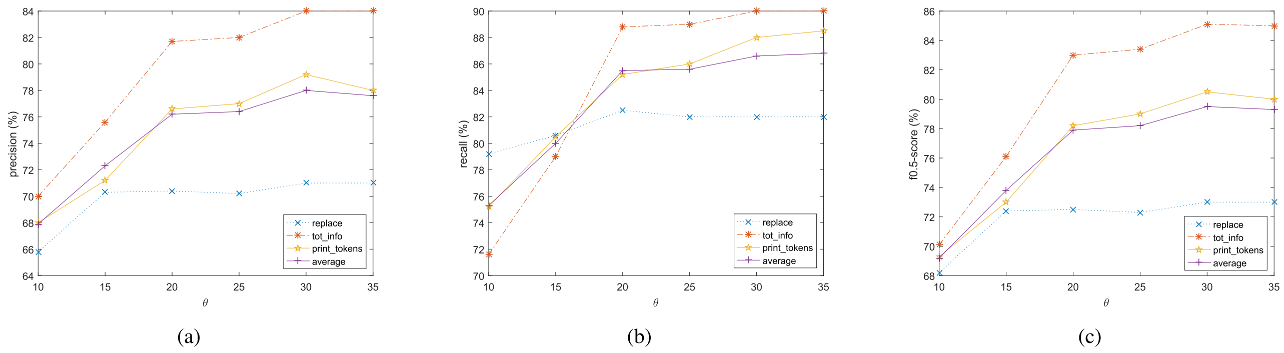
### B. EXPERIMENTAL EVALUATION

(i) The Influence of $\theta$

The accurate SDC vulnerability of registers and memories is used. Fig. 3 shows precision, recall and f0.5-score of the classifier. In Fig. 3(a), precision experiences a great rise when $\theta$ goes from 10% to 20% for all programs. After then, it does not increase obviously. When $\theta$ is 20%, the averaged precision is about 76%. In Fig. 3(b) and Fig. 3(c), recall and f0.5-score share a similar changing trend as precision. The averaged recall and f0.5-score up to 85% and 78% respectively when $\theta$ is 20%.

These curves illustrate that $\theta_m$ is 20% if the SDC vulnerability of registers and memories is accurate. This means that 20% fault injections is sufficient to generate a training dataset. With the training dataset, a SVM classifier is trained to classify the remaining instructions, with the averaged precision, recall and f0.5-score are 76%, 85%, and 78% respectively.

(ii) The Relationship Between $regerror(reg)$ and *regpct*, and Between $memerror(mem)$ and *mempct*



(a) (b) (c)

**FIGURE 3. The performance of the classifier learned under different $\theta$. (a) precision. (b) recall. (c) f0.5-score.**

The cost of PVInsiden includes the cost of generating a training dataset and the cost of getting the SDC vulnerability of registers and memories. In section V-B, to reduce the cost of the latter, we calculate the SDC vulnerability of registers and memories by partial fault injection. Partial fault injection results in $regerror(reg)$ and $memerror(mem)$, impacting the quality of feature values and the performance of the classifier. In order to gain a better trade-off between the cost of the latter and the performance of the classifier, we explore the relationship between $regerror(reg)$ and $regpct$ to find the optimal $regpct$. The same exploration is also conducted on memory cells.

Fig.4(a) depicts the results of registers. It takes EAX, EDX and ESP in replace.c as examples. In Fig. 4(a), $regerror(reg)$ decreases while $regpct$ increases. When $regpct$ is 10%, the maximum $regerror(reg)$ is less than 35%. Besides, $regerror(reg)$ is less than 25% as $regpct$ is 20%. Other programs show a similar changing trend, steering us to set $regpct$ to 10% or 20% to get the SDC vulnerability of registers with a small error at a lower cost.

Fig. 4(b) shows the results of memory cells. Addstrmem, omatchmem, omatchmem1 and omatchmem2 mark four different memory cells in replace.c. $memerror(mem)$ generally decreases as the increase of $mempct$. With the similar analysis to registers, $mempct$ is set to 30% or 40%.

To find the optimal $mempct$ and $regpct$, experiments are conducted on different $regpct$ and $mempct$ with $\theta$ of 20%. Fig. 5 gives the performance of classifiers learned in different combinations of $regpct$ and $mempct$. In Fig. 5, the performance of the classifiers in four scenarios are basically consistent for the three programs. The results only have a slight difference with the results posted in Fig.3. Take the precision of replace.c as an example. In Fig. 5(a), precision ranges from 70% to 70.3% across four scenarios. In Fig. 3(a), the precision of replace.c is 70.4% as $\theta$ is 20%.

Further thought on this phenomenon discovers that the SDC vulnerability ranks of registers in four scenarios are the same. They are also the same as that in full fault injection. Besides, memories perform a similar experimental results to registers.

Table. 4 shows a comparison of the ranks obtained under full fault injection ($regpct = 100\%, mempct = 100\%$) and

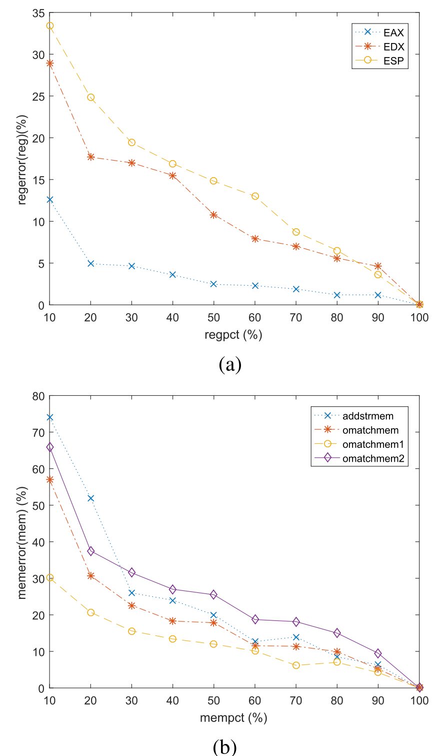



**FIGURE 4. (a) $regerror(reg)$ under different $regpct$. (b) $memerror(mem)$ under different $mempct$.**

scenario one ($regpct = 10\%, mempct = 30\%$). Although differences exist in the SDC vulnerability of registers and memories, the ranks are the same.

Based on this study, the conclusion is that setting $regpct$ to 10% and $mempct$ to 30% is enough. It is worth noting that under the circumstance, the averaged precision, recall and f0.5-score of the trained classifier are 75%, 81% and 76% respectively across the programs.

(iii) The Performance of PVInsiden

PVInsiden gets a training dataset from a target program by fault injection. PVIns identified by PVInsiden includes the PVIns identified by a classifier as well as the PVIns identified by fault injection. Taking these two results into consideration, the performance of PVInsiden is presented on

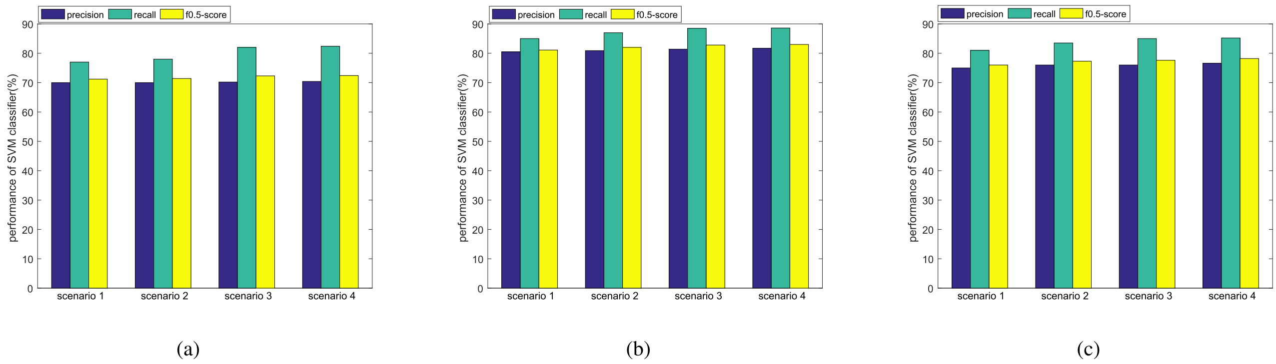



**FIGURE 5.** The performance of the classifier under different *regpct* and *mempct* (scenario 1: *regpct* = 10% and *mempct* = 30%. scenario 2: *regpct* = 10% and *mempct* = 40%. scenario 3: *regpct* = 20% and *mempct* = 30%. scenario 4: *regpct* = 20% and *mempct* = 40%). (a) replace. (b) tot_info. (c) print_tokens .

**TABLE 4.** The SDC vulnerability and rank of registers and memory cells in replace.c.

| register or memory | regpct=mempct=100% | | regpct=10%, mempct=30% | |
|---|---|---|---|---|
| | SDC vulnerability | rank | SDC vulnerability | rank |
| eax | 0.114 | 1 | 0.108 | 1 |
| edx | 0.108 | 2 | 0.089 | 2 |
| esp | 0.006 | 3 | 0.005 | 3 |
| addstrmem | 0.606 | 1 | 0.732 | 1 |
| omatchmem | 0.036 | 3 | 0.043 | 3 |
| omatchmem1 | 0.021 | 4 | 0.024 | 4 |
| omatchmem2 | 0.426 | 2 | 0.562 | 2 |

Fig. 6(a). Fig. 6(a) points out the minimum precision, recall and f0.5-score, which are 76%, 81%, and 77% respectively. On average, the three evaluation metrics are 80%, 85%, and 81% across the programs.

PVInsiden is compared with ePVF. The ePVF results are shown in Fig. 6(b). Its averaged precision, recall and f0.5-score are 66%, 100%, and 71% respectively. This means that PVInsiden outperforms ePVF in identifying PVIns.

(iv) The Adaptability of PVInsiden

Multiple program inputs are considered to evaluate the adaptability of PVInsiden. The results are shown in Fig. 7. Given a target program, the difference of precision under different inputs is slight, with the maximum of 7.2% (tot_info (input 2) and tot_info (input 3)). The maximum difference of recall and f0.5-score are 6% (tot_info (input 2) and tot_info (input 3)) and 4.9% (replace (input 2) and replace (input 3)) respectively.

In contrast, in Fig. 7(b), ePVF shows a larger difference. For print_tokens.c, though its recall is 100% for every program input, the maximum difference of precision and f0.5-score are 20.6% (input 2 and input 3) and 18% respectively (input 2 and input 3). This is due to the fact that there is a large difference in segment faults frequencies as shown in table 5.

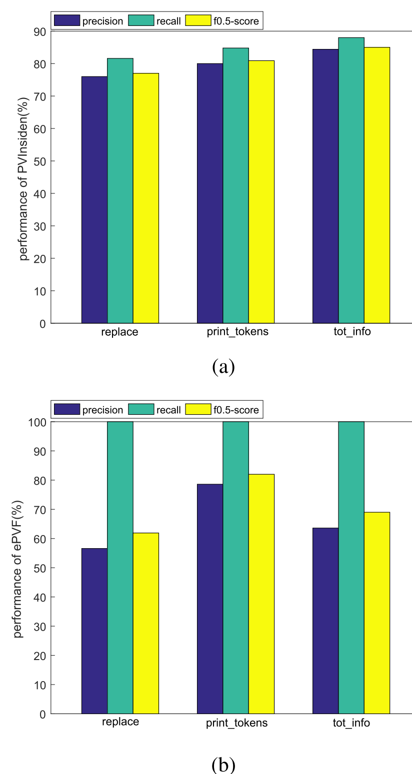



**FIGURE 6.** (a) The performance of PVInsiden. (b) The performance of ePVF.

The adaptability is also evaluated on different programs. In Fig. 6(a), the maximum difference of averaged precision, recall and f0.5-score among programs is 8.4% (replace and tot_info), 7% (replace and tot_info), and 8% (replace and tot_info) respectively. In Fig. 7(b), they are 22% (replace and print_tokens), 0%(replace and print_tokens) and 20% (replace and print_tokens).

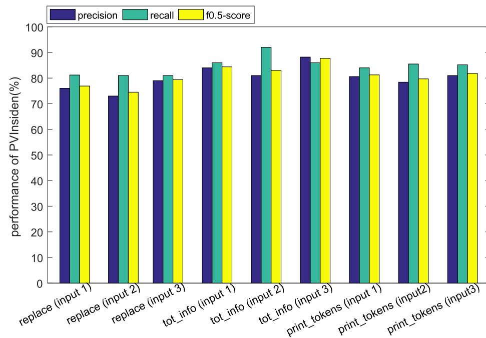
By the comparison, PVInsiden is more adaptable than ePVF for different program inputs and different programs.

(v) The Cost of PVInsiden

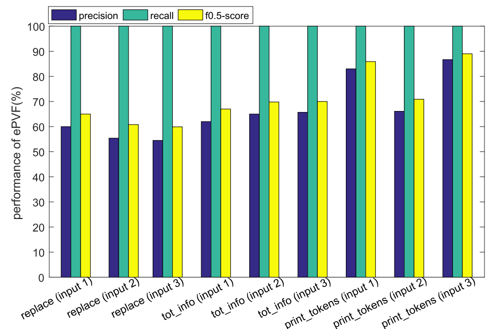
The cost of PVInsiden includes the cost of fault injection and the cost of a classifier. Full fault injection approach is

**TABLE 5.** Relative crash frequency analysis for programs.

| Name | Segment Faults | Abort Faults | Other Faults |
|---|---|---|---|
| replace (input 1) | 38.9% | 13.5% | 47.6% |
| replace (input 2) | 59.5% | 22.7% | 17.8% |
| replace (input 3) | 30.7% | 12% | 57.3% |
| print_tokens (input 1) | 96.8% | 1.6% | 1.6% |
| print_tokens (input 2) | 55.7% | 2% | 42.3% |
| print_tokens (input 3) | 96.9% | 2.9% | 0.2% |




(a)




(b)

**FIGURE 7.** (a) The performance of PVInsiden under different inputs. (b) The performance of ePVF under different inputs.

applied to programs and the statistic results are shown in table 6. In table 6, *sum* is the number of injections in full fault injection approach, and $n_{reg}$ stands for the number of injections related to registers, and $n_{mem}$ is the number of injections related to memories. *cost* is the ratio of cost of fault injection of PVInsiden and cost of full fault injection approach, expressed by (6). Table. 6 shows that the averaged *cost* is about 35.5% when $\theta_m$, *regpct*, and *mempct* are 20%, 10%, and 30% respectively.

The cost of a classifier includes the time of extracting feature values and training a classifier. It can be ignored as it is trivial in comparison with the time of fault injection.

$$cost = \frac{sum * \theta_m + n_{reg} * regpct + n_{mem} * mempct}{sum}. \quad (6)$$

Full fault injection approach has the best performance in identifying PVIns, but the number of fault injections is

**TABLE 6.** Statistics under full fault injection method and the cost of PVInsiden.

| Name | replace | print_tokens | tot_info |
|---|---|---|---|
| $sum$ | 129952 | 37632 | 72256 |
| $n_{reg}$ | 98432 | 29952 | 46656 |
| $n_{mem}$ | 31520 | 7680 | 25600 |
| $cost$ | 34% | 34% | 37% |

tremendous, making full fault injection approach hard to practice. Although fault injection is not necessary for ePVF, the averaged precision recall, and f0.5-score are 66%, 100% and 71% respectively. Besides, ePVF has a non-negligible difference in performance. The maximum difference of averaged precision, recall and f0.5-score of ePVF up to 22%, 0% and 20% among programs. Among different program inputs, they are 20.6%, 0% and 18% respectively. In order to make a better trade-off between the cost of fault injections and the performance, PVInsiden uses SVM to learn a classifier to identify PVIns. It trains a classifier iteratively to determine the minimum number of fault injections. Setting full fault injection as a baseline, PVInsiden only uses 35% fault injections to identify PVIns with the averaged precision, recall and f0.5-score are 80%, 85% and 81% respectively. Furthermore, the maximum difference of precision, recall and f0.5-score of PVInsiden among programs are 8.4%, 7% and 8%. Among different program inputs, they are 7.2%, 6% and 4.9% respectively. These results show that PVInsiden has a better trade-off between the performance and the cost of fault injections in identifying PVIns. And it is more adaptable than ePVF. Besides, The dataset used by PVInsiden is collected from a target program, making PVInsiden light-weighted.

## VII. CONCLUSION

In this paper, PVInsiden is proposed to identify PVIns in programs. To reduce the cost of fault injections, we apply partial fault injection to generate a training dataset to train a classifier to identify PVIns. The training dataset is generated from a target program, making PVInsiden light-weighted.

We explore the relationship between partial fault injection and the performance of a classifier to make sure to what extend fault injection should conduct. Experimental results show that with accurate SDC vulnerability of registers and memories, 20% fault injections is sufficient to learn a classifier, with the averaged precision, recall and f0.5-score are 76%, 85%, and 78% respectively. In general, PVInsiden uses 35% fault injections at most to identify PVIns with 80% precision, 85% recall and 81% f0.5-score on average, outperforming ePVF.

Feature engineering includes selecting features and transforming the selected features into quantifiable representations. Instruction features, register-related features, memory-related features and some other features are considered and selected. Besides, the framework of training a classifier and identifying PVIns is given.

Experiments are also conducted under different program inputs as well as different programs to assess the adaptability of PVInsiden. The results show that compared with ePVF, PVInsiden has a smaller difference in performance, indicating that is more adaptable than ePVF in identifying PVIns.

In future work, new instruction features will be extracted to improve the performance of a classifier. Furthermore, different machine learning models and the improvement of generalization and stability of PVInsiden will be studied.

## REFERENCES

[1] J. Ma, D. Yu, Y. Wang, Z. Cai, Q. Zhang, and C. Hu, "Detecting silent data corruptions in aerospace-based computing using program invariants," *Int. J. Aerosp. Eng.*, vol. 2016, Nov. 2016, Art. no. 8213638.

[2] D. A. G. de Oliveira, L. L. Pilla, T. Santini, and P. Rech, "Evaluation and mitigation of radiation-induced soft errors in graphics processing units," *IEEE Trans. Comput.*, vol. 65, no. 3, pp. 791–804, Mar. 2016.

[3] E. Berrocal, L. A. Bautista-Gomez, S. Di, Z. Lan, and F. Cappello, "Toward general software level silent data corruption detection for parallel applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 12, pp. 3642–3655, Dec. 2017.

[4] C. Chao, E. Greg, W. Matthew, and P. Santosh, "LADR: Low-cost application-level detector for reducing silent output corruptions," in *Proc. 27th Int. Symp. High-Perform. Parallel Distrib. Comput.*, New York, NY, USA, Jun. 2018, pp. 156–167.

[5] S. K. Daya and M. Scott, "Harnessing soft computations for low-budget fault tolerance," in *Proc. IEEE/ACM Int. Symp. Microarchitecture*, Cambridge, U.K., Dec. 2014, pp. 319–330.

[6] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "ePVF: An enhanced program vulnerability factor methodology for cross-layer resilience analysis," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Toulouse, France, Jul. 2016, pp. 168–179.

[7] Q. Lu, K. Pattabiraman, M. S. Gupta, and J. A. Rivers, "SDCTune: A model for predicting the SDC proneness of an application for configurable protection," in *Proc. Int. Conf. Compil., Archit. Synth. Embedded Syst.*, New Delhi, India, Oct. 2014, pp. 1–10.

[8] K. Gokcen, B. P. Ivy, G. Roberto, and K. Sriram, "Understanding scale-dependent soft-error behavior of scientific applications," in *Proc. IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, Washington, DC, USA, Jul. 2018, pp. 483–491.

[9] S. K. S. Hari, R. Venkatagiri, S. V. Adve, and H. Naeimi, "GangES: Gang error simulation for hardware resiliency evaluation," in *Proc. IEEE Int. Symp. Comput. Archit.*, Minneapolis, MN, USA, Jun. 2014, pp. 1–12.

[10] A. Vishnu, H. V. Dam, N. R. Tallent, D. J. Kerbyson, and A. Hoisie, "Fault modeling of extreme scale applications using machine learning," in *Proc. IEEE Int. Conf. Parallel Distrib. Process. Symp.*, Chicago, IL, USA, Jul. 2016, pp. 222–231.

[11] J. Ma, Y. Wang, L. Zhou, C. Hu, and H. Wang, "SDCInfer: Inference of silent data corruption causing instructions," in *Proc. IEEE Int. Conf. Softw. Eng. Service Sci.*, Beijing, China, Sep. 2015, pp. 228–232.

[12] X. Xu and M.-L. Li, "Understanding soft error propagation using efficient vulnerability-driven fault injection," in *Proc. 42nd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2012, pp. 1–12.

[13] V. Sridharan and D. R. Kaeli, "Eliminating microarchitectural dependency from architectural vulnerability," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, Raleigh, NC, USA, Feb. 2009, pp. 117–128.

[14] J. Ma and Y. Wang, "Characterization of stack behavior under soft errors," in *Proc. IEEE Int. Conf. Design, Autom. Test Eur. Conf. Exhib.*, Lausanne, Switzerland, Mar. 2017, pp. 1534–1539.

[15] F. Restrepo-Calle, A. Martínez-Álvarez, S. Cuenca-Asensi, and A. Jimeno-Morenilla, "Selective SWIFT-R: A flexible software-based technique for soft error mitigation in low-cost embedded systems," *J. Electron. Test.*, vol. 29, no. 6, pp. 825–838, Dec. 2013.

[16] F. Restrepo-Calle, S. Cuenca-Asensi, A. Martínez-Álvarez, E. Chielle, and F. L. Kastensmidt, "Application-based analysis of register file criticality for reliability assessment in embedded microprocessors," *J. Electron. Test.*, vol. 31, no. 2, pp. 139–150, Apr. 2015.

[17] T. E. Thomas, A. J. Bhattad, S. Mitra, and S. Bagchi, "Sirius: Neural network based probabilistic assertions for detecting silent data corruption in parallel programs," in *Proc. 35th Symp. Reliable Distrib. Syst. (SRDS)*, Budapest, Hungary, Sep. 2016, pp. 41–50.

[18] J. Liu, M. C. Kurt, and G. Agrawal, "A practical approach for handling soft errors in iterative applications," in *Proc. IEEE Int. Conf. Cluster Comput.*, Chicago, IL, USA, Sep. 2015, pp. 158–161.

[19] F. Hamid, N. A. Behnaz, M. Zahra, B. Bahram, P. Mohammad, and N. A. Omid, "Automated diagnostic system for breast cancer using least square support vector machine," *Amer. J. Biomed. Eng.*, vol. 3, no. 6, pp. 175–181, Mar. 2017.

[20] A. R. Norman and C. Jeronimo, "Trading fault tolerance for performance in an encoding," in *Proc. Comput. Frontiers Conf.*, Siena, Italy, May 2017, pp. 183–190.

**NA YANG** received the B.E. degree in information and engineering from Linyi University, Linyi, China, in 2014, and the M.S. degree in computer science from Inner Mongolia University, Hohhot, China, in 2017. She is currently pursuing the Ph.D. degree with the School of Computer Science and Engineering, Southeast University, Nanjing, China. Her research interest includes software reliability.

**YUN WANG** received the B.S. degree in computer software from Nanjing University, China, in 1989, and the M.E. and Ph.D. degrees in computer networking from Southeast University, China, in 2004 and 2007, respectively. She was a Postdoctoral Researcher with INRIA/IRISA, France, and a Senior Researcher with the University of Texas at Dallas, USA, from 1999 to 2002. She joined Southeast University, in 1997, and is currently a Full Professor with the School of Computer Science and Engineering. She was PI for more than 20 research projects supported by the national and international grants. She has published more than 120 peer-reviewed journal and conference papers, including TOC, JPDC, InfoCom, ICDCS, and IPDPS. Her research interests include distributed systems, fault tolerance, and computer networking. She is an Executive Member of the Council of Jiangsu Computer Society, China. She obtains three Science and Technology Awards from the Ministry of Education, China.