Identifying SDC-Causing Instructions Based on Random Forests Algorithm

LiPing Liu[1*], LinLin Ci[2], Wei Liu[3], Hui Yang[4]
[1] Computer department, Beijing Institute of Technology
Beijing China
[e-mail: liuliping_bit@163.com]
[2] Computer department, Beijing Institute of Technology
Beijing China
[e-mail: cilinlin_bit@126.com]
[3]Computer department, Beijing Institute of Technology
Beijing China
[e-mail: Liuwei_bit@126.com]
[4]Computer department, Beijing Institute of Technology
Beijing China
[e-mail: zlj-1943@163.com]
*Corresponding author: LiPing Liu

Received December 15, 2017; revised April 14, 2018; revised June 16, 2018; accepted July 15, 2018; published March 31 2019

Abstract

Silent Data Corruptions (SDCs) is a serious reliability issue in many domains of computer system. The identification and protection of the program instructions that cause SDCs is one of the research hotspots in computer reliability field at present. A lot of solutions have already been proposed to solve this problem. However, many of them are hard to be applied widely due to time-consuming and expensive costs. This paper proposes an intelligent approach named SDCPredictor to identify the instructions that cause SDCs. SDCPredictor identifies SDC-causing Instructions depending on analyzing the static and dynamic features of instructions rather than fault injections. The experimental results demonstrate that SDCPredictor is highly accurate in predicting the SDCs proneness. It can achieve higher fault coverage than previous similar techniques in a moderate time cost.

Keywords: Fault tolerance, Error detection, Reliability, SDC-Causing instructions, Random forests.

1. Introduction

With the processor design trends towards smaller transistor size, lower core voltage and higher frequency, the threat of soft errors becomes more and more serious. [1]. Soft errors could lead to silent data corruption (SDC) which are difficult to be detected. When SDCs occur, the program executes normally, but the outputs of program are incorrect. Thus, with the sustained effect of Moore's Law, more and more transistors will be integrated into the chips, so that soft errors of hardware will occur more and more frequently [2].Therefore, necessary protection measures should be adopted to prevent SDC errors.

Hardware-based solutions such as triple modular redundant and dual modular redundancy increase the equipment cost greatly. Software-based solutions can also handle soft errors of hardware without any additional cost on hardware. Due to the advantages of cost savings, hardware-independent design, flexibility and implementation simplicity, software-based detection methods are being paid more and more attention in soft error mitigation research [3]. To date some of these methods have already been applied to many fields including astronautics and high performance computing [4].

Compared with the hardware-based techniques, software-based techniques can save more hardware resources; nevertheless, they occurs significant performance overhead. Reducing performance overhead has become to be the top issue of such techniques. To reduce performance overheads, recent works tend to protect these SDC-prone instructions selectively [5-7].

It was recently reported that a small part of programs' instructions are responsible for most of SDCs, and protect these instructions selectively can achieve high coverage against SDCs [8, 9]. These instructions are called SDC-prone instructions. Therefore, how to identify the SDC-prone instructions becomes a key problem.

Recently, a lot of works [10-16] try to improve the static injection framework. CriticalFault [10] applies vulnerability analysis to avoid the derated fault injections. Relyzer [11] employs pruning techniques to decrease the quantity of fault injections by predicting the outcomes of faults. Although the quantity of fault injections is reduced, the statistical fault injection (SFI) experiments are still time-consuming. SmartInjector [12] proposes an intelligent fault injection framework to identify the SDC-prone instructions. It firstly decreases the quantity of fault injections by predicting the outcomes of faults, and then reduces the time for a single fault simulation by predicting the fault outcome prediction technique. Shoestring [13] leverages compiler to analyze and identify the statistically vulnerable instructions. The time cost of Shoestring is low because it does not rely on any SFI injections. However, the compiler analysis technology is static and lack of dynamic analysis of program instructions, resulting in lower error coverage. SymPLIFIED [14] identifies SDC-prone instructions using symbolic execution, which enumerates all potential hardware errors.

The work [16] presents a selective protection technique that allows users to selectively protect these SDC-prone data. The main idea of work [16] is predicting the SDC proneness of a program's data firstly, then selectively protects the most SDC-prone instructions of the program for the user-specified overhead bound. Since the prediction model is built based on the Classification and Regression Tree (CART) algorithm, the process of prediction do not need to perform fault injections. Therefore, it is more time-saving than fault injection based method. However, CART algorithm is easy to cause the over fitting which leads to a poor stability and low accuracy.

In this paper, by employing random regression forest algorithm, an ensemble regression prediction scheme, we propose a novel prediction model, SDCPredictor, to predict the SDC proneness of program instructions. SDCPredictor identifies SDC-causing Instructions depending on analyzing the static and dynamic features of instructions rather than fault injections, thus it can save a lot of time and manpower. Our experimental results demonstrate that SDCPredictor is more accurate in predicting the SDCs proneness than previous similar techniques in a moderate time cost. To summarize, our contributions are as follows:

- We propose an intelligent approach named SDCPredictor to identify the instructions that cause SDCs. SDCPredictor identifies SDC-causing Instructions depending on analyzing the static and dynamic features of instructions rather than fault injections.
- Our method assumes that different features are not equally important. To strengthen the generalization error of SDCPredictor, we employ weight value to represent the importance of the features. That means features with larger weights have high probability to be selected. In order to improve the prediction accuracy, we screen all trees by evaluating their quality. Those trees whose accuracy is lower will be excluded. Only the trees whose accuracy is high enough will be reserved.
- We evaluate the efficiency of proposed approach. The experimental results demonstrate that the proposed approach can acquire higher fault coverage at the same performance overhead bound than previous similar approaches.

The remainder of this paper is organized as follows. We review in brief the works related to identifying SDC-prone instructions in Section 2, while fault model of the proposed approach is presented in Section 3. In Section 4 we introduce the proposed approach. The results of the experiments are reported and analyzed in section 5, and finally section 6 summarizes the paper and points out the field of research in future.

2. Related Work

Considerable research efforts have been done to identify the SDC-causing instructions. A typical technique is modeling the SDC rate of SDC-causing instructions by performing fault injections.

CriticalFault [10] make use of vulnerability analysis to avoid derated injections. Relyzer [11] employs pruning techniques to decrease the quantity of fault injections by predicting the outcomes of faults. Hardware faults with similar behavior are deemed to be equivalent faults and fall into one group. Only one representative fault is selected to implement fault injection for each group. SmartInjector [12] proposes an intelligent fault injection framework to identify the SDC-prone instructions. It firstly decreases the quantity of fault injections by predicting the outcomes of faults, and then reduces the time consumption for an individual fault simulation by predicting the fault outcome prediction technique.

Instructions vulnerability analysis is another important solution to identify SDC-causing instructions. Shoestring [13] uses a static compiler analysis technology to identify SDC-causing instructions, and protect them by inserting redundant instructions. Shoestring only considers the instructions which are possible to cause user-visible errors. These instructions which are possible to lead to SDC errors are left unprotected. Although Shoestring is time-saving, the SDC coverage is low. SymPLIFIED [14] identifies SDC-prone instructions using symbolic execution, which enumerates all potential hardware errors. This might cost more time than SFI due to the state explosion problem caused by symbolic execution.

The work [15] employs genetic algorithm (GA) to identify the most vulnerable sections of a program. By analyzing the dynamic dependencies between the program blocks, the proposed method can identify the most vulnerable basic blocks of a program precisely. However, some instructions of vulnerable blocks may make no contributions to the outputs of program. Protecting these instructions will incur high and unnecessary performance overhead. The work process of work [15] is shown in **Fig. 1**. The step 1 converts a program to a smaller and executable one. In step 2, the most vulnerable blocks of the input program are selected by GA. Step 3 strengthens the identified vulnerable blocks.

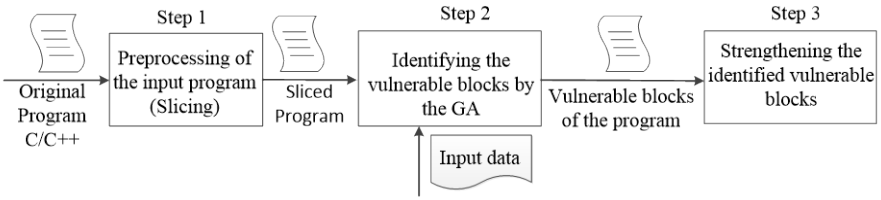


Fig. 1. Block diagram of the work [15]

The work [16] introduces a prediction model named SDCAuto to predict the SDC proneness of a program's instructions. SDCAuto is built using CART algorithm, requiring little to no human intervention. Therefore, it is no need to perform fault injections in the process of instructions selective protection against SDC-causing errors. **Fig. 2** illustrates the diagram of the work [16]. The work [16] first compiles the source code into LLVM IR, and extracts instruction features based on LLVM IR file. Then, it obtains the SDC proneness for each instruction with the help of SDCAuto model.

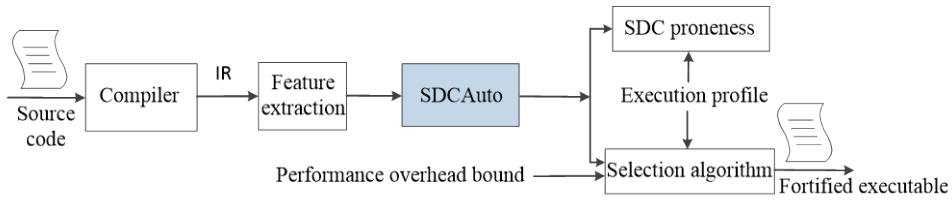


Fig. 2. Block diagram of the work [16]

Finally, detectors are inserted into the source code for protecting the most SDC-prone instructions for the user-specified overhead bound.

3. Fault Model

SEU-induced soft errors fall into two categories: user-visible errors and user-invisible errors. User-visible errors usually cause architecture-level symptoms such as program crash or hang, which can be detected by symptom-based detection techniques. When the user-invisible errors occurs, the program executes normally and do not cause any abnormal symptoms. Nevertheless, the outputs of program are incorrect, i.e. SDC errors. These errors cannot be handled by symptom-based detection techniques. Therefore, we focus on SDC errors in this paper.

Faults in memory and caches are not considered, since these devices are usually protected with ECC. We focus on the faults that occur in processors' functional units and registers, which are not protected by fault-tolerant techniques due to performance reasons. Faults occur in processors' functional units and registers can lead to control flow errors and data flow errors.

We focus only data flow errors and assume that control flow errors are detected by control-flow checking techniques. Faults in the instruction opcode are also not considered, since it always causes illegal opcode exception rather than SDC.

Finally, as in work [16], we assume that at most one fault occurs during a program's execution.

4. Proposed method

SDCPredictor aims to build a intelligent prediction model which can predict the SDC proneness of program instructions accurately without faults injection. For this purpose, we extract some dynamic and static features of instructions and create training data set with the help of faults injection experiments. Based on the training data set, our prediction model is built using random regression forests.

To understand this paper better, some useful definitions which are used in this paper are presented.

SDC coverage: The SDC coverage is defined as the rate of SDC causing errors detected by error detection technology.

SDC proneness per instruction: This is the probability that a fault in instruction I leads to an SDC. This is denoted as $P(SDC)$.

Dynamic count ratio: This is the ratio of the number of dynamic instances of instruction I executed to the total number of dynamic instructions in the program. This is denoted as $D(I)$.

To understand our proposed method on a macro level, we give a brief presentation about its work process. The flow diagram of the proposed method is shown in **Fig. 3**. Details of each element are as follows. We first introduces the extracted program features of instructions that highly correlated with SDC-prone. We then explain the implement of faults injection experiments on selected benchmark programs to generate training data set. Finally, we describe how to build our prediction model and design protected code.

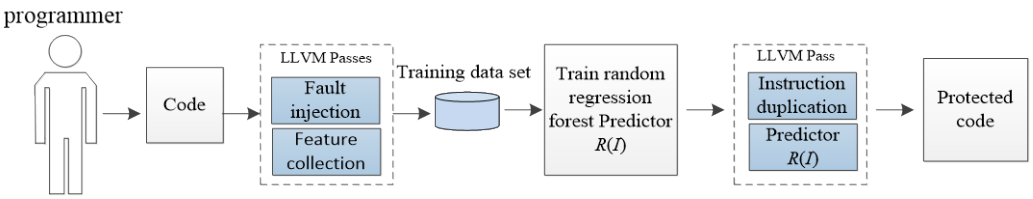


Fig. 3. Flow diagram of the proposed method

4.1 Feature extraction

We extract features of instructions according to our analysis and prior work [12, 13, 16, 17 and 18]. In all, 72 features are extracted. These features of instructions fall into nine categories shown in **Table 1**. Detailed categories results are as follows: (1) Comparison operations related features. Comparison operations may cause the control flow deviation, which cannot be detected by existing control flow checking techniques because such comparison operations are resilient. Some errors may be masked by comparison operations. (2)Memory address calculation related features. Accessed memory address must be in legal address space. If the accessed memory address exceeds the legal bound, this operation will lead to a segmentation fault. If not, the operation is possible to cause SDCs. (3) Sub-word operations related features.

Sub-word operations such as left shift operation and right shift operation often discard some bits of registers. Bit-flipping errors occurred in the discarded bits will be derated and will not cause SDCs. Thus, SDC proneness of the instruction operands will be reduced. (4) Logical operations related features. Logical operations such as AND operations and OR operations are possible to mask errors that occurred in corresponding bit of operands. For example, if a bit value of AND instruction operand is 1, then the bit-flipping errors occurred in the corresponding bit of other instruction operand will be masked. (5) Successor instruction related features. When an error occurs, the error will propagate along the data dependency chain. When a SDC-masked instruction is encountered, the error is possible to be masked; otherwise it will propagate to the end points of data dependency chain and leads to an incorrect output. Therefore, the SDC proneness of precursor instructions in data dependency chain is affected by the successor instructions. (6) Type of end points of data dependency chains related features. Different end points of data dependency chains have different effects on the outputs of program. (7) Code structure related features. The SDC proneness of instructions is highly correlated with the execution probability of run-time. Thus, instructions on the hot paths of the program have higher SDC proneness due to the higher execution probability of run-time. (8) Data width related features. Data width is the effective number of bits of instruction operands. The higher the data width, the higher the SDCs rate of instructions will be. (9) Execution time related features. Instructions executed with a higher frequency read and write resisters frequently. Corrupted data of resisters are most likely to be used by these instructions. Therefore, these instructions usually have a higher SDC proneness.

Table 1. Some features extracted for Model Building

Feature group	Feature	Description
Comparison operations related features	is_cmp	whether the operation is a comparison operation
	is_loop_terminator	whether the comparison result can determine the time of loop execution
	is_cmp_with_zero	whether the comparison is made with zero
	is_cmp_with_address	whether memory address is exist in the operands of comparison operation
Memory access and addressing related features	is_read	whether the operation read data from memory
	is_memory_addressing	whether the result of operation is used to address memory
	is_write	whether the operation write data to memory
Sub-word operations related features	is_shl	whether the instruction is a left shift operation
	is_shr	whether the instruction is a right shift operation
Logical operations related features	is_and	whether the instruction is a logic “and” operation
	is_or	whether the instruction is a logic “or” operation
	is_xor	whether the instruction is a logic “xor” operation
Successor instruction related features	shl_instructions_count	The number of left shift instructions contained in successor instruction
	shr_instructions_count	The number of right shift instructions contained in successor instruction
Type of end points of data dependency chains related features	is_global	whether the operation modify the value of global variable
	is_stack_push	whether the operation push a value to stack
	is_function_call	whether the operation call a function

Code structure related features	number_of_pred _BBs	number of predecessor BBs
	number_of_suc _BBs	number of successor BBs
	is_within_loop	whether the operation is within a loop
	is_accumulative_computation	whether the operation is a cumulative-computation operation
	bb_length	the number of static instructions in the basic block that contains the specific instruction
Data width related features	data_width_Source _operand	the effective number of bits of instruction source operand
	data_width_destination_operand	the effective number of bits of instruction destination operand
Execution time related features	dynamic_count_ratio	dynamic count ratio of the specific instruction

4.2 Fault injection and training data generation

In order to acquire high-quality training samples, we create training data set with the help of faults injection experiments. We use the famous fault injection tool PINFI [19] to implement fault injection experiment. PINFI is built with Intel Pin [20] and uses the API exposed by Pin to inject faults.

Table 2. Characteristics of the training benchmarks.

Program	**Description**	**Benchmark suite**
Bzip2	File compression and decompression program	SPEC benchmarks
Perlbench	SPEC benchmark for perl interpreter	SPEC benchmarks
Blackscholes	Financial analysis program	PARSEC benchmarks
Swaptions	Price portfolio of swaptions	PARSEC benchmarks
TSP	Solving the TSP problem by genetic algorithms	Stanford benchmarks
Qsort	quick-sort algorithm	Stanford benchmarks
IS	Integer sorting, random memory access	NAS benchmarks
EP	Embarrassingly Parallel	NAS benchmarks
BFS	Breadth-First search	Parboil benchmarks
MM	Dense Matrix-Matrix Multiply	Parboil benchmarks

Table 3. Characteristics of the testing benchmarks.

Program	**Description**	**Benchmark suite**
Gzip	File compression program	SPEC benchmarks
Ferret	Similarity search program	PARSEC benchmarks
Queens	Solving the classic n-queens problem	Stanford benchmarks
CG	Conjugate Gradient, irregular memory access and communication	NAS benchmarks
LBM	Fluid dynamics computing program	Parboil benchmarks

We select 15 benchmarks which are drawn from SPEC benchmarks [21], NAS parallel benchmarks [22], Stanford benchmarks [23], Parboil benchmarks [24] and PARSEC benchmarks [25]. These benchmarks are divided into two groups randomly: training group and testing group. **Table 2** and **Table 3** provide a brief description of these benchmarks. We compile these benchmarks using LLVM compiler and provide the executable file to PINFI after linking.

Foregone studies showed that data dependencies among the instructions are important influence factors to SDC proneness and a large part of program instructions have no influence to the outputs of program. To simplify these programs, static-slicing technique [26] are utilized to convert a program to a smaller and executable one. Converted program eliminates those instructions that have no influence to the outputs of program and can be executed normally.

First, we select some instructions as fault injection targets by running PINFI on each converted program.

Second, we inject faults into these selected instructions with the aid of statistical fault injection like Relyzer [11]. To simulate the data corruption faults, we flip one certain bit of instruction's source operand register and each bit flips one time. In each run, a fault, i.e., a single bit flip, is injected into the operands register of the dynamic instruction instance. The outcome of the fault is compared with the fault-free outcome. The fault-free outcome is obtained by executing the original executable program with the same input. The outcome are divided into four categories by program execution results: (1) Crash, the program terminate unexpectedly and threw an exception, (2) SDC, when SDCs occur, the program executes normally, but the outputs of program are incorrect, (3) Hang, which means the execution time of program is much longer than a fault-free execution, and (4) Benign, which means the faults are derated or masked inherently by the program and the outputs of program are correct.

Third, the SDC proneness $P(SDC)$ of each instruction is obtained through the equation (1)

$$P(SDC) = \frac{N_{SDC}}{N_{fault}} \times D(I) \tag{1}$$

where N_{SDC} is the SDC count caused by instruction I, N_{fault} is the total number of initial faults attributed to the instruction I, $D(I)$ is the dynamic count ratio of the instruction I.

Finally, a sample $\{F, C\}$ is generated, where F is the extracted features vector, and C is the annotated class label (i.e., SDC proneness).

4.3 Regression model training

4.3.1 Selecting the classification algorithm

In order to better meet the requirements of our data and problem space, the machine learning algorithm must be selected carefully. We choose the Random Forest (RF) algorithm for the following three reasons:

- Some features we extracted are boolean data types, e.g., is_ stack_push, is_load and is_cmp, while some other features are numerical, e.g., bb_length and data_width. Other regression algorithms might not support such a hybrid data types. RF is competent to such a hybrid data types.
- Other regression models, such as deep neural network and support vector machine (SVM), need to normalize the input data. Besides, In order to achieve accurate prediction results many parameters need to be adjusted. While, RF requires little data preparation and parameters adjustment.

- In recent years, more and more research results prove that RF is one of the most accurate techniques and is currently regarded as the most advanced prediction method. A random regression forest is an ensemble of regression methods which consisting of multiple decision trees. A single decision tree could cause over fitting, while this problem can be largely avoided by multiple randomly trained decision trees. RF can give estimates of what variables are important in the classification. It also can effectively estimate missing data and maintain accuracy.

4.3.2 Training random regression forests

Our prediction model based of random regression forests is trained by samples with 72 feature vectors. The full training samples have 10000 training instructions. Our random regression forest is constructed based on these training instructions. In classification, random forest uses voting mechanism to determine the classification result. In regression, the regression value of forest is obtained by computing the average value of the individual tree predictions. Let $F = \{f_i \in R | i = 1, 2, \ldots, N\}$ denote the feature vectors, and let $C = \{c_1, c_2, \ldots, c_N\}$ denote the labeled regression values of the training samples (i.e., SDC proneness). We build the trees compliance with the random forest framework [27]. For each tree in the random forest, we use the bootstrap resampling from the training samples to select the training subset. For each node split for building the tree, we select the random selection as the node split strategy.

Different features are not equally important. In other words, the importance of the different features to the SDC proneness of instructions is different. The more important the feature is, the more influence on the prediction results it has. Therefore, important features should have high probability to be selected. To do this, we employ weight value to represent the importance of the features. That means features with larger weights have high probability to be selected. We calculate the feature weight using the formula presented in [28]. The formula for calculation the feature weight as follows:

$$\chi^2 = \sum_{i=1}^{m} \sum_{j=1}^{2} \frac{(o_{ij} - e_{ij})^2}{e_{ij}} \tag{2}$$

where m is the number of feature A, o_{ij} is the count of joint event (A_i, C_j), defined as:

$$o_{ij} = count(A = a_i \bigcap C = c_j) \tag{3}$$

e_{ij} is the expected value of joint event (A_i, C_j), defined as:

$$e_{ij} = \frac{count(A = a_i) \times count(C = c_j)}{N} \tag{4}$$

where N is the number of training samples, count($A = a_i$) is the number of samples whose value of feature A is a_i, and count($C = c_j$) is the number of samples whose value of the class feature is c_j. An χ^2 statistic weight is calculated for each feature. From the weights, we select only different subsets of features with high weights to build individual decision trees.

In order to improve the prediction accuracy, we screen all trees by evaluating their quality. Those trees whose accuracy is lower will be excluded. Only the trees whose accuracy is high enough will be reserved. As previously described, we use the bootstrap resampling from the training samples to select the training subset. Samples selected for building a tree are called in-of-bag (IOB) data, while the rest is called out-of-bag (OOB) data. When a tree is built, we exploit the OOB data to evaluate its prediction accuracy through calculating the mean square error (MSE). The mean square error for a given regressor $h_k(x)$ is defined as:

$$MSE = \frac{\sum_{i=1}^{N-N_s} (h_k(x_i) - C_i)^2}{N - N_s} \quad (5)$$

where x_i is a sample in the OOB data and C_i is the class label of the sample x_i. It can be seen from the formula: the smaller MSE of a tree, the higher accuracy is. Hence, we only retain trees whose MSE are below the predetermined threshold. Thus, the prediction accuracy of our random regression forests is increased.

4.3.3 SDC proneness prediction

Once the random forest is built from training data-set, we can use it to estimate the SDC proneness of each instruction of the testing programs. The higher SDC proneness indicates higher SDC probability and greater importance. Therefore, the instructions with high SDC proneness should be given high priority. According to the importance of the instruction, the instructions selective duplication algorithm designs the detector to protect the most SDC-prone instructions for the user-specified overhead bound.

4.3.4 Choose the instructions to protect and design detector

Based on the instruction's SDC proneness, we then select instructions to maximize the SDC coverage for the user-specified overhead bound through a standard dynamic programming algorithm [29]. To protect these protected instructions, we need to insert duplicated instructions and check instructions, which we called detectors. These instructions are inserted immediately after the protected instructions. Our detectors use new registers and memory spaces and do not interfere with the original program semantics. By comparing the original value computed by the protected instruction with the value computed by the duplicated instructions, detectors can detect the errors occurred in protected instructions. If they match, it means that no errors occurred; otherwise, it means that an error has been detected and the control flow of program will be transferred to error handling routine.

5. Experimental evaluation

In this section experiments are designed to evaluate the effectiveness of the proposed approach, we choose five programs, namely Gzip, Ferret, Queens, CG and LBM, which are chosen form SPEC benchmarks [21], NAS parallel benchmarks [22], Stanford benchmarks [23], Parboil benchmarks [24] and PARSEC benchmarks [25] mentioned in section 4.2.

We compile these benchmarks using LLVM compiler and provide and run them in a single threaded mode. It should note that the proposed approach is not only applicable to single threaded mode. We conduct the evaluation experiment on an Intel i7 machine, with 8 GB of RAM and 400 GB Hard drive running Debian Linux Version 6.0.

SDC proneness accuracy, SDC coverage and time efficiency are important metrics for evaluating our approach. Therefore, we carry out a detailed test and analysis to these metrics.

5.1 SDC proneness accuracy

Prediction the SDC proneness of instructions is the key of machine learning based selective protection technique. The prediction accuracy determines the error detection rate. To acquire an accurate prediction effect, parameters of prediction model need to be tuned. For random regression forests model, three parameters play determinative roles in improving the prediction accuracy: (1) maximum number of features in individual tree, and (2) number of trees, and (3)

minimum sample leaf size of an individual tree. Setting a reasonable value for the first parameter can maintain the diversity of the trees and increase the generalization ability of prediction model. We set it to the recommended value $\sqrt{n}$, where n is the total number of features. The second parameter is a decisive parameter for the prediction accuracy. In order to find the optimal value, we gradually increased it from 50 with a step-size 2 until the prediction accuracy becomes stable or decreasing. Finally, we set the optimal value to 215. Setting a reasonable value for the third parameter can avoid over-fitting problem. According to our experimental scene, we set the value to 100.

We evaluate the predicting accuracy of our prediction model by calculating the average squared errors of testing data-set and the accuracy (the percentage of the samples whose SDC proneness estimation error is less than 10%) of SDC proneness estimation.

Table 4. The MSE and accuracy of the testing programs.

Program	MSE	Accuracy
Gzip	0.00548	88.77%
Ferret	0.00249	94.55%
Queens	0.00179	95.29%
CG	0.00362	91.65%
LBM	0.00428	90.13%

The MSE and accuracy are shown in **Table 4**. From the **Table 4** it can be seen that our model achieve high prediction accuracy of SDCs proneness. The high accuracy benefits from the following aspects. Firstly, we give different selection probability to features with different weight. This enhances the generalizing ability of each tree of random forests. Besides, we create training data set with the help of faults injection experiments which acquires high-quality training samples. In addition, the process of optimizing parameters improves the prediction accuracy. More importantly, the extended features such as data propagation distance and the type of SDC-masked instructions included in the successor instructions make a great contribution to the prediction accuracy. Therefore, the proposed approach can guide error detection mechanism to make the best detector placement.

5.2 SDC coverage

In this paper we define the SDC coverage as the rate of errors detected by our error detection technology. We apply our approach to predict the SDC proneness for each instructions of a program. Under the user-specified overhead bound, we select these instructions with the highest SDC proneness, and expand the set of protected instructions under the performance overhead constrains. We inject faults into these protected instructions with the aid of statistical fault injection like Relyzer [11], and then collect the number of errors detected by our error detection technology. Finally, we calculate the SDC coverage according the results calculated. We also compare our results with the work [15] and SDCAuto presented in work [16]. We use our approach to maximize SDC coverage under the user-specified performance overhead.

By computing the SDC coverage of each program under the same performance overhead bound, we compare our method with work [15] and SDCAuto. We select three performance overhead bounds: 10%, 20% and 30%. **Fig. 4** shows the statistical results obtained by our approach (SCDPredictor), work [15] and SDCAuto for each benchmark. As it can be seen in **Fig. 4**, the averages SDC coverage for SCDPredictor, Work [15] and SDCAuto are 34.68%, 25.84% and 32.3% respectively for the 10% performance overhead bound, the corresponding averages SDC coverage are 51.4%, 40.4% and 47.8% for the 20% performance overhead

bound, and 69.0%, 56.04% and 62.16% for the 30% performance overhead bound. It can be seen that the SCDPredictor acquires highest SDC coverage at the same performance overhead bound. That came out of work [15] uses genetic algorithm (GA) to identify the most vulnerable blocks of a program. However, not all the instructions of vulnerable blocks make contributions to the outputs of program and need to be protected. Protecting these instructions will incur high performance overhead. The models of SCDPredictor and SDCAuto are built using machine learning approaches.

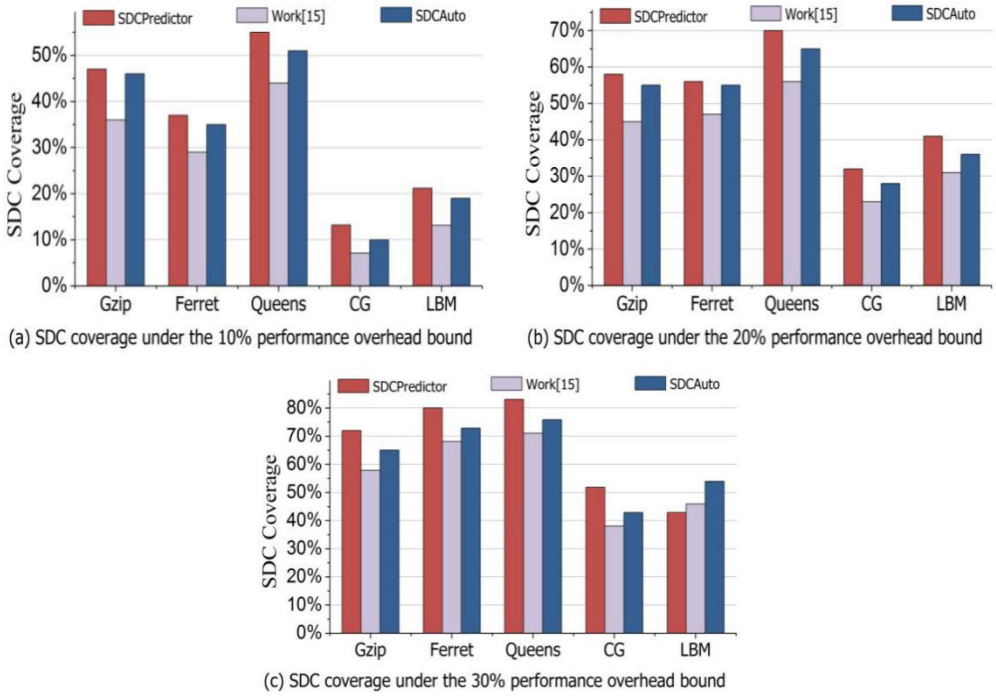


Fig. 4. The comparison of SDC coverage under different performance overhead bounds: 10%, 20% and 30%

These models can predict the SDC proneness of each instruction. Instructions with highly SDC proneness are duplicated and protected. Thus, SCDPredictor and SDCAuto obtain higher SDC coverage than work [15]. Since SDCAuto is built using CART, which is easy to cause the over fitting, it is lack of prediction robustness and stability. Unlike SDCAuto, SCDPredictor is built using random regression forests, which hardly cause over-fitting and are insensitive to noisy data due to it constructs a number of weak learners. More importantly, SCDPredictor can obtain higher accuracy in predicting the SDCs proneness than SDCAuto. Therefore, SCDPredictor acquires highest SDC coverage.

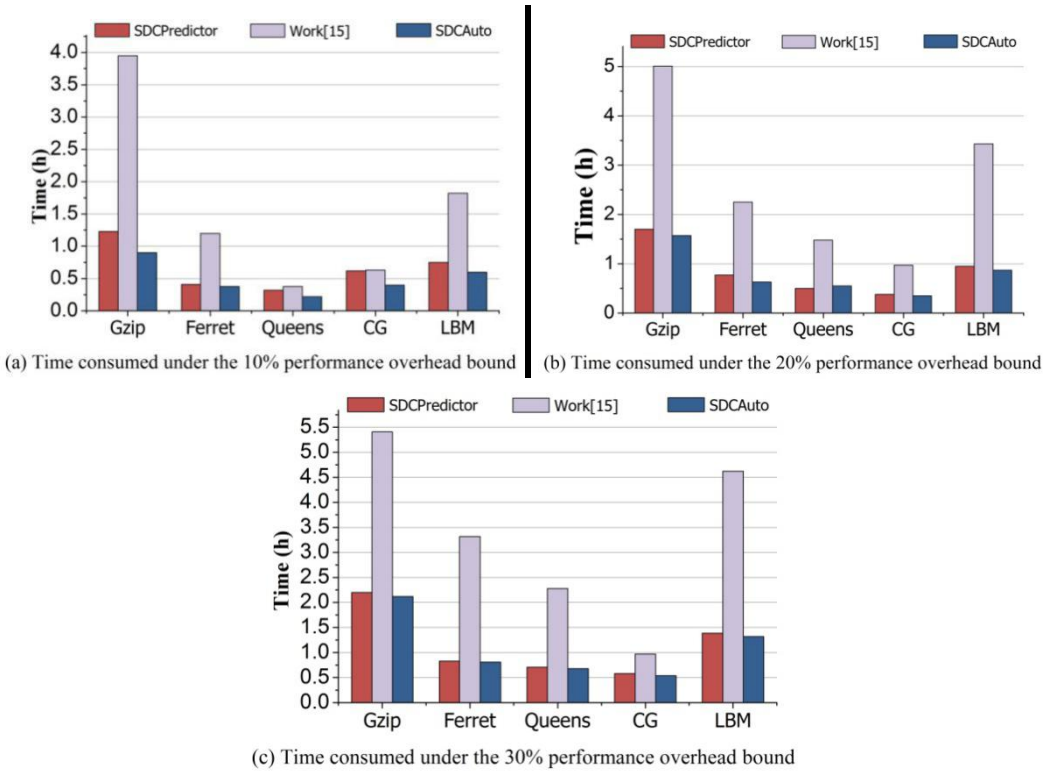
5.3 Time efficiency

In this subsection, the time efficiency of the proposed method is evaluated. The time efficiency is defined as the time consumed for identifying the program instructions which need to be protected. We record the consuming time for identifying instructions of each benchmark under a user-specified overhead bound. The results in **Fig. 5** show that the average consuming times for SCDPredictor, Work [15] and SDCAuto are 0.67h, 1.60h and 0.6h respectively for the

10% performance overhead bound, the corresponding average consuming times are 0.86h, 2.63h and 0.79h for the 20% performance overhead bound, and 1.14h, 3.32h and 1.09h for the 30% performance overhead bound.

We observe that the SCDPredictor model performs worse than the SDCAuto model, but still manages to outperform Work [15] in terms of time efficiency. For SCDPredictor and SDCAuto, most of the times are spent on the feature extraction, SDC proneness prediction and selection algorithm. Feature extraction needs to traverse all instructions of application. Some features also need to execute application multiple times. Thus, the complexity of feature extraction can be approximated as $O(mn)$, where m is the total number of features and n is the total number of instructions. The prediction model of SDCAuto is built using CART, which has a complexity of $O(n)$, where n is the depth of tree. While, the prediction model of SCDPredictor is built using random forests. The complexity of random forests is $O(kn)$, where k is the total number of trees and n is the depth of tree. The complexity of selection algorithm can be approximated as $O(n)$.Therefore, the consuming time gap between SCDPredictor and SDCAuto is mainly due to the process of SDC proneness prediction.

Work [15] exploits the genetic algorithm (GA) to identify the most vulnerable blocks of a program. In Work [15], the BB subsequence in code execution path which is selected in the control flow graph is treated as a chromosome.



(a) Time consumed under the 10% performance overhead bound

(b) Time consumed under the 20% performance overhead bound

(c) Time consumed under the 30% performance overhead bound

Fig. 5. The comparison of time efficiency under different performance overhead bounds: 10%, 20% and 30%

In order to evaluate the fitness of a chromosome, BBs of the chromosome are executed several times. Thus, the complexity of Work [15] can be approximated as $O(n \ln n)$, where n is the total number of BBs. Therefore, the Work [15] is the most time-consuming.

6. Conclusions and Future Research

In order to overcome the shortcomings of instruction selective recalculation, such as huge time cost and low error coverage, a prediction model named SDCPredictor based on random forests is proposed. SDCPredictor does not need to perform fault injections to predict the SDC proneness of each instruction. In order to strengthen the generalization error of SDCPredictor, we choose features according to their weights when building the individual tree of random forests. To acquire high-quality training samples, we create training data set with the help of faults injection experiments. Besides, for better prediction accuracy, we screen all trees by evaluating their quality. Those trees whose accuracy is lower will be discarded. Only those trees whose accuracy is high enough will be kept. Thus, SDCPredictor obtains higher prediction accuracy of SDC proneness for each instruction.

We assess the effectiveness of SDCPredictor from three metrics: SDC proneness accuracy, SDC coverage and time efficiency. The experimental results demonstrate that SDCPredictor is highly accurate in predicting the SDCs proneness. It can achieve higher fault coverage than previous similar techniques in a moderate time cost.

Invariant based detection techniques incur lower overhead than duplication-based detection techniques. Existing invariant based detection techniques have drawbacks in terms of false alarm and low error coverage, which have affected their application. Therefore, developing efficient invariant based detection techniques are the further research directions for our research group.

Acknowledgment

This research was supported by the National Natural Science Foundation of China under grant No. 61370134, the National High Technology Research and Development Program of China (863 Program) under grant No. 2013AA013901.

References

[1] Bhattacharya K, Ranganathan N. RADJAM, “A Novel Approach for Reduction of Soft Errors in Logic Circuits,” in *Proc. of 2009 22nd International Conference on VLSI Design*, 453-458, 2009. Article (CrossRef Link)

[2] Racunas P, Constantinides K, Manne S, et al., “Perturbation-based Fault Screening,” in *Proc. of 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 169-180, 2007. Article (CrossRef Link)

[3] Reis G A, Chang J, Vachharajani N, et al., “SWIFT: software implemented fault tolerance,” in *Proc. of International Symposium on Code Generation and Optimization, IEEE*, 243-254, 2005. Article (CrossRef Link)

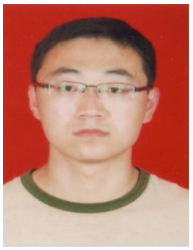
[4] Restrepocalle F, Martnezlvarez A, Cuencaasensi S, et al., “Selective SWIFT-R: A flexible software-based technique for soft error mitigation in low-cost embedded systems,” *Journal of Electronic Testing*, 29(6):825-838, 2013. Article (CrossRef Link)

[5] Chielle E, Azambuja J R, Barth R S, et al., "Evaluating Selective Redundancy in Data-flow Software-based Techniques," *Radiation and ITS Effects on Components and Systems*, 2012. Article (CrossRef Link)
[6] Cong J, Gururaj K., "Assuring application-level correctness against soft errors," in *Proc. of 2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 47(10):150-157, 2011. Article (CrossRef Link)
[7] Sundaram A, Aakel A, Lockhart D, et al., "Efficient fault tolerance in multi-media applications through selective instruction replication," in *Proc. of The Workshop on Radiation Effects and Fault Tolerance in Nanometer Technologies*, ACM, 339-346, 2008. Article (CrossRef Link)
[8] Hari S K S, Adve S V, Naeimi H., "Low-cost program-level detectors for reducing silent data corruptions," in *Proc. of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, 1-12, 2012. Article (CrossRef Link)
[9] Thomas A, Pattabiraman K., "Error detector placement for soft computation," in *Proc. of 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 1-12, 2013. Article (CrossRef Link)
[10] Xu X, Li M L., "Understanding soft error propagation using Efficient vulnerability-driven fault injection," in *Proc. of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, 2012. Article (CrossRef Link)
[11] Hari S K S, Adve S V, Naeimi H, et al., "Relyzer: Application Resiliency Analyzer for Transient Faults," *IEEE Micro*, 33(3):58-66, 2013. Article (CrossRef Link)
[12] Li J, Tan Q., "SmartInjector: Exploiting intelligent fault injection for SDC rate analysis," in *Proc. of IEEE International Symposium on Defect and Fault Tolerance in Vlsi and Nanotechnology Systems*, IEEE, 236-242, 2013. Article (CrossRef Link)
[13] Feng S, Gupta S, Ansari A, et al., "Shoestring: probabilistic soft error reliability on the cheap," in *Proc. of Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ACM, 385-396, 2010. Article (CrossRef Link)
[14] Pattabiraman K, Nakka N M, Kalbarczyk Z T, et al., "SymPLFIED: Symbolic Program-Level Fault Injection and Error Detection Framework," *IEEE Transactions on Computers*, 62(11):2292-2307, 2013. Article (CrossRef Link)
[15] Arasteh B, Bouyer A, Pirahesh S., "An efficient vulnerability-driven method for hardening a program against soft-error using genetic algorithm," *Computers & Electrical Engineering*, 48:25-43, 2015. Article (CrossRef Link)
[16] Rivers J A, Rivers J A, Rivers J A, et al., "Configurable Detection of SDC-causing Errors in Programs," *Acm Transactions on Embedded Computing Systems*, 16(3):88, 2017. Article (CrossRef Link)
[17] Cook J J, Zilles C., "A Characterization of Instruction-level Error Derating and its Implications for Error Detection," in *Proc. of 2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, 482-491, 2008. Article (CrossRef Link)
[18] Laguna I, Schulz M, Richards D F, et al., "IPAS: Intelligent protection against silent output corruption in scientific applications," in *Proc. of IEEE/ACM International Symposium on Code Generation and Optimization*, IEEE, 227-238, 2016. Article (CrossRef Link)
[19] Wei J, Thomas A, Li G, et al., "Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults," in *Proc. of IEEE/IFIP International Conference on Dependable Systems and Networks, IEEE Computer Society*, 375-382, 2014. Article (CrossRef Link)
[20] Luk C K, Cohn R, Muth R, et al., "Pin:building customized program analysis tools with dynamic instrumentation," *Acm Sigplan Notices*, 40(6):190-200, 2005. Article (CrossRef Link)
[21] Henning J L., "SPEC CPU2006 benchmark descriptions," *Acm Sigarch Computer Architecture News*, 34(4):1-17, 2006. Article (CrossRef Link)
[22] Bailey D., "The NAS parallel benchmarks," *International Journal of Supercomputer Applications*, 2(4):158 – 165, 1991. Article (CrossRef Link)

[23] Hsueh M C, Tsai T K, Iyer R K., "Fault injection techniques and tools," *Computer*, 30(4):75-82, 1997. Article (CrossRef Link)
[24] J. A. Stratton, C. Rodrigrues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-M. W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," UIUC, Urbana, Tech. Rep. IMPACT-12-01, 2012.
[25] Bienia C, Kumar S, Singh J P, et al., "The PARSEC benchmark suite:characterization and architectural implications," in *Proc. of International Conference on Parallel Architectures and Compilation Techniques*, IEEE, 72-81, 2017. Article (CrossRef Link)
[26] Weiser M., "Program slicing," *IEEE Transactions on Software Engineering*, SE-10(4):352-357, 1984. Article (CrossRef Link)
[27] Breiman L., "Random Forests," *Machine Learning*, 45(1):5-32, 2001. Article (CrossRef Link)
[28] Ye Y, Li H, Deng X, et al., "Feature Weighting Random Forest for Detection of Hidden Web Search Interfaces," *The Journal of Computational Linguistics and Chinese Language Processing*, 13(4):387-404, 2008.
[29] Pisinger D, Toth P., "Knapsack Problems," *Handbook of Combinatorial Optimization*, v.4 n.1(5):xx, 1998. Article (CrossRef Link)

LiPing Liu received the B.S. degree and M.S. degree from North University of China from Dept. of Computer Science, China, in 2008 and 2011, respectively. Currently, he is studying for his PhD Degree at Computer Science in Beijing Institute of Technology. His current research interests include Secure Wireless Sensor Networks,Pattern Recognition, and Trusted Computing.

LinLin Ci received the B.S. degree in the Dept. of Computer Science from Beijing Institute of Technology, China, in 1976; he received the M.S. degree in the Dept. of Computer Science from Northwestern Polytechnical University, China, in 1985. Currently, he is professor and doctoral supervisor in computer application. His research areas include Secure Wireless Sensor Networks, Pattern Recognition, and Trusted Computing.



Wei Liu received the B.S. degree and M.S. degree from North University of China from Dept. of Computer Science, China, in 2008 and 2011, respectively. Currently, he is studying for his PhD Degree at Computer Science in Beijing Institute of Technology. His current research interests include Secure Wireless Sensor Networks, Pattern Recognition, and Trusted Computing.

Hui Yang received the B.S. degree and M.S. degree from North University of China from Dept. of Computer Science, China, in 2008 and 2011, respectively. Currently, he is studying for his PhD Degree at Computer Science in Beijing Institute of Technology. His current research interests include Secure Wireless Sensor Networks, Pattern Recognition, and Trusted Computing.