SDC-causing Error Detection Based on Lightweight Vulnerability Prediction

Cheng Liu nuaaliucheng@nuaa.edu.cn
Jingjing Gu gujingjing@nuaa.edu.cn
Zujia Yan yanzujia@nuaa.edu.cn
Nanjing University of Aeronautics and Astronautics, Nanjing, China
Fuzhen Zhuang zhuangfuzhen@ict.ac.cn
Key Lab of Intelligent Information Processing of Chinese Academy of Sciences (CAS), Institute of Computing Technology, CAS, Beijing 100190, China

Yunyun Wang wangyunyun@njupt.edu.cn
Nanjing University of Posts and Telecommunications, Nanjing, China

Abstract

Nowadays the system vulnerability caused by soft errors grows exponentially, of which Silent Data Corruption(SDC) is one of the most harmful issues due to introducing unnoticed changes to the original data and error outputs. Thus, the detection of SDC-causing errors is extremely significant to the system reliability. However, most of the current detecting techniques require sufficient data of fault injections for training, which are difficult to achieve in practice because of high resources consumption, such as expensive execution time and code size costs. To this end, we propose a lightweight model named Deep Forest Regression based Multi-granularity Redundancy(DFRMR) to improve the error detection rate and meanwhile decrease the resources consumption. Specifically, first, we employ the program analysis to extract instruction features which are highly related to SDCs. Second, we design the deep forest regression model to predict the SDC vulnerability of instructions. Third, we optimize the error detection procedure by duplicating the critical instructions with different granularity. Finally, we evaluate our DFRMR model on Mibench benchmarks with multiple testing programs. The results show that our method attains better detection accuracy compared to other state-of-the-art methods and keeps the low multi-granularity redundancy.

Keywords: SDC vulnerability prediction, Fault tolerance, Error detection, Deep forests

1. Introduction

Soft errors in a computer's memory system always refer to the changes of an instruction in a program or a data value, which will damage the processing data[1]. Practically, Silent Data Corruption(SDC) is the most harmful issue of soft errors caused by multiple reasons. For instance, the network might suffer undetected corruption, cosmic radiation and many other causes of soft errors in memory. Due to run unnoticedly, SDCs can't be detected by the disk firmware or the host operating system. As a result, SDCs can lead to incorrect program outputs without any explicit error phenomena. Previous studies have exhibited that the error rates on silent corruption are far higher than one in every 10^6 bits Shoshani and

1. https://en.wikipedia.org/wiki/Soft_error

Table 1: Statistics of programs under full fault injection approach

	Qsort	Fast_Fourier_trans	Dijkstra
Lines of source code	45	142	170
Number of static instructions	92	541	325
Number of dynamic instructions	245	13061	6.51×10^6
Number of fault injections	10751	7.07×10^5	3.54×10^8

Rotem (2009). Thus, how to identify and protect instructions against SDCs have become an significant problem in academic and industry.

In recent years, various techniques have been proposed to detect SDCs. They focus on the identifying process of SDC vulnerability instructions, for instance, fault injection based techniques Hari et al. (2012a); Hiller et al. (2002), program analysis based techniques Yang and Gao (2007); Pattabiraman et al. (2008) and machine learning based techniques Laguna et al. (2016); Bronevetsky et al. (2009); Liu et al. (2018); Yang and Wang (2019); Lu et al. (2014, 2017); Liu et al. (2019). To be specific, fault injection based techniques identify the vulnerable instructions by a full fault injection. They inject faults to every bit of an operand of all dynamic instructions, which takes too much time to practice. Tab.1 reports some statistics of three classic programs (i.e., Qsort, Fast Fourier Transform and Dijkstra) under the full fault injection approach. We can observe that a program with only hundreds of code lines may require millions of injections, which is very time-consuming. Program analysis based techniques determine whether an instruction needs to be protected by statically analyzing the propagation path of errors. However, as programs linearly increase in size, rapid exponential growth of computational complexity makes these methods hard to be applied. Machine learning based techniques use instruction SDC vulnerability predictors for predicting which instruction to protect. Generally, these methods can achieve a better prediction accuracy, but they suffer from the demand for large-scale training data and complex manual parameter tuning.

To these ends, in order to lighten the workload of identifying process, we propose a lightweight model called Deep Forest Regression based Multi-granularity Redundancy (DFRMR) to predict the SDC vulnerability of instructions, and select high SDC vulnerability instructions to conduct protection. Generally, considering the time consumption issue, DFRMR identifies vulnerable instructions depending on source code features rather than fault injections. Further, our method allows users to select how many instructions to protect subject to the overhead they are willing to tolerate. To reach above goals, specifically, (1) We extract the features related to SDC vulnerability using Low Level Virtual Machine(LLVM) framework on a small set of benchmark programs. Then we perform a small scale fault injection experiment to generate predictive value of SDC vulnerability. (2) We introduce deep forest regression as SDC vulnerability predictor to relief data scale issue. Compared to other machine learning methods (e.g., deep neural network) that require large-scale training data Zhou and Feng (2017), the deep forest can obtain high accuracy even when there are only small-scale training data, which allows us to lighten the process of data collection. Further, it requires little manual tuning because the cascade level can be adaptively determined. In order to improve the prediction accuracy, we develop cascade regression forest structure and improved sliding window scanning technique. The former is used for generating enhanced features which provide a more accurate description of the data, while the latter is used for exploring feature space as well as extracting relationships' features among sequential samples. (3) We use DFRMR to predict SDC vulnerability of instructions in new programs. The prediction results are used to derive error detectors for instructions, subject to a given redundancy granularity. While the program is running, our detectors re-execute the chosen instructions by duplicating them and compare the re-executed value with the original one. Any deviation between the two values will be treated as successful error detection. Our experimental results demonstrate that DFRMR is more accurate in predicting the SDC vulnerability and achieves a higher SDC error detection rate. The main contributions of our work are summarized as follows:

- We develop a lightweight model named DFRMR to detect SDCs of instructions. DFRMR predicts the SDC vulnerability depending on instruction features without performing fault injections. And instructions are protected partially subject to the given granularity by users.
- We design a deep forest model for vulnerability prediction in order to lighten data collection and processing workloads. To achieve higher accuracy, we apply various regression forests to extract enhanced features, and improved sliding window scanning to explore feature space.
- We develop error detectors by duplicating the instructions depending on a given granularity. The detectors will throw indications if there occur soft errors.
- We evaluate the performance of our approach under two metrics, accuracy of the prediction model and effectiveness of fault tolerance subject to the overhead specified by users. The result shows that DFRMR achieves higher accuracy than state-of-the-art algorithms, as well as acquires high error detection rate with low overhead.

The remainder of this paper is organized as follows. Section 2 introduces the related works. Section 3 presents the overall framework of our approach. Section 4 details the DFRMR method, including SDC vulnerability, feature extraction, deep forest model and instruction redundancy strategy. Eventually, the validation experiments are conducted in Section 5, and the conclusion is summarized in Section 6.

2. Related Work

Error detection techniques are mainly classified into hardware-implemented and software-implemented techniques. The hardware-implemented techniques require on-chip redundancy which means additional hardware circuits for fault tolerance. Typical triple-module redundancy(TMR) provides an efficiency error detection rate, but it also brings huge additional hardware overhead Kastensmidt et al. (2005). Dai et al. (2015); Sato and Arita (2001) proposed methods for repeatedly executing an operation on idle processing units of the processor in order to reduce the hardware overhead of fault tolerance. Although it is proved that hardware-implemented methods are able to achieve higher reliability, they need to modify or add additional circuits to meet the needs of error detection. In contrast, software-implemented methods are flexible enough to solve these problems and have attracted a lot of attentions. Generally, software redundancy techniques are classified into three categories: fault injection based techniques, program analysis based techniques, machine learning based techniques.

Fault injection based techniques have significant overhead to performing a large number of fault injections. Hari et al. (2012b) attempted to reduce the overhead of fault injection by obtaining equivalent faults. Li and Tan (2013) employed the faults equivalent technique and faults outcome prediction technique to reduce the time for a single fault simulation. Xu and Li (2012) compressed fault injection space using a biased injection method, which allows the injection campaign only focuses on likely non-derated faults. However, these methods are not suitable for large-scale programs.

Program analysis based techniques identify the vulnerability instructions through program analysis rather than fault injection. Yang and Gao (2007) analyzed the propagation of transient fault in the program and calculate the SDC vulnerability by the error propagation model. Pattabiraman et al. (2008) used symbolic execution to abstract the state of erroneous values in the program and modeled checking to comprehensively find all errors that evade detection. Although these techniques can achieve a high analysis accuracy, they suffer from high computation complexity due to the complex structures of programs.

To balance the relationship between fault injection space and analysis accuracy, program analysis can be combined with fault injection techniques. The SDC-causing instructions are inferred dynamically based on the information of program analysis and fault injection results, which not only ensures the accuracy but also reduces the cost of fault injections. Recent studies focus on the efficient selection of susceptible instructions, because a valid instruction set is a guarantee of the trade-off between detection accuracy and overhead.

Machine learning based techniques predict the vulnerability by extracting features of instructions in programs. Laguna et al. (2016); Yang and Wang (2019) discussed the performance of Support Vector Machine(SVM) in SDC-prone prediction, and an analysis in Yang and Wang (2019) showed the feasibility of partial fault injection. Lu et al. (2014, 2017); Liu et al. (2019) used Regression Tree based method to predict the SDC vulnerability of instructions. These methods require little human intervention, which allows them to perform no fault injections in the process of instructions selection. Bronevetsky et al. (2009); Liu et al. (2018) applied Neural Network(NN) and Long Short-Term Memory(LSTM) to predict SDC vulnerability, which needs a large scale dataset of fault injections.

3. The Overall Framework

Fig. 1 shows the overall framework of our approach. It mainly contains three components: (1) Data collection. To get our training data, we implement the program analysis to extract features that are highly related with SDCs. Fault injections are conducted on the benchmark programs to calculate SDC vulnerability. (2) Model training. We first introduce a sliding window to extract expanded features of instructions. Then the cascade regression forest model is applied to train the SDC vulnerability predictor. (3) Fault tolerance. The SDC vulnerability of instructions from a new program will be predicted by the SDC predictor.

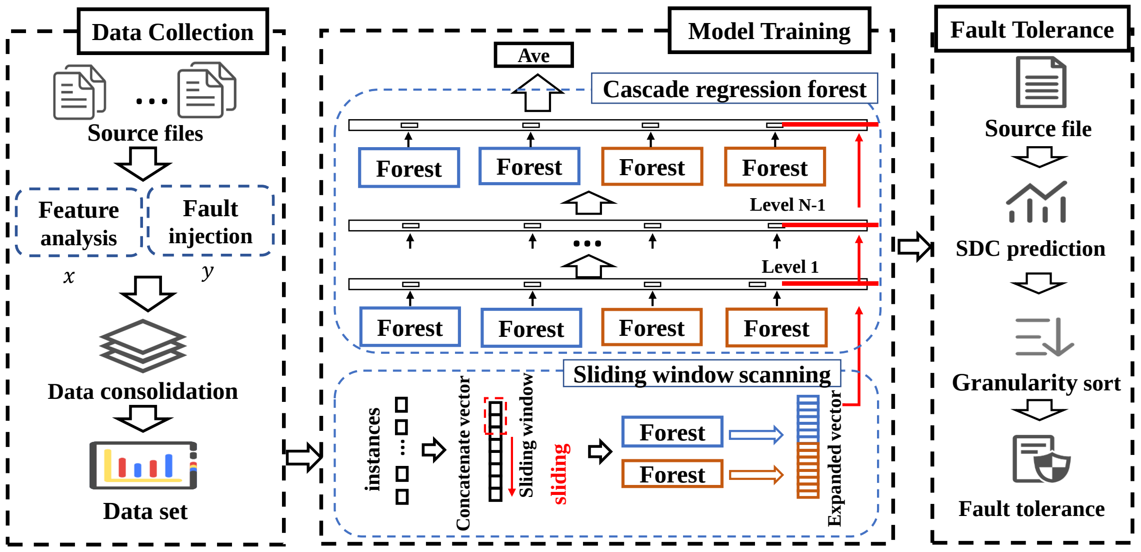


Figure 1: The overall framework of our approach.

We sort the instructions by predictive values and duplicate those which are more vulnerable. Next, we introduce the details of each component step by step.

4. Methodology

4.1. SDC vulnerability

In this paper, instruction SDC vulnerability refers to the probability that an SDC will be generated when a soft error occurs to the instruction. The definition of the instruction SDC vulnerability is described as follows.

Static and dynamic instruction sets are denoted as Eq. 1,

$$\begin{cases} V_{static} = \{I_1, ..., I_i, ..., I_N\}, \\ V_{dynamic} = \{I_1^{d_1}, ..., I_i^{d_i}, ..., I_N^{d_N}\}, \end{cases} \tag{1}$$

where I_i is the i_{th} static instruction, N is the number of static instructions in the program, $I_i^{d_i}$ means the dynamic instructions set of I_i that I_i executes d_i times in a program running process. The SDC vulnerability of I_i is defined as Eq. 2:

$$P_{SDC}(I_i) == \frac{1}{d_i} \times \sum_{I_i^q \in I_i^{d_i}} \frac{S_i^q}{F_i^q} \tag{2}$$

where I_i^q is the q_{th} dynamic instruction of $I_i^{d_i}$, F_i^q is the number of faults injecting to the dynamic instruction I_i^q, S_i^q is the number of the SDC errors caused by fault injections.

4.2. Feature extraction

We extract features according to our analysis and prior works Lu et al. (2014); Ma et al. (2016); Bo et al. (2016). Fig. 2 shows the SDC vulnerabilities of partial instructions. As

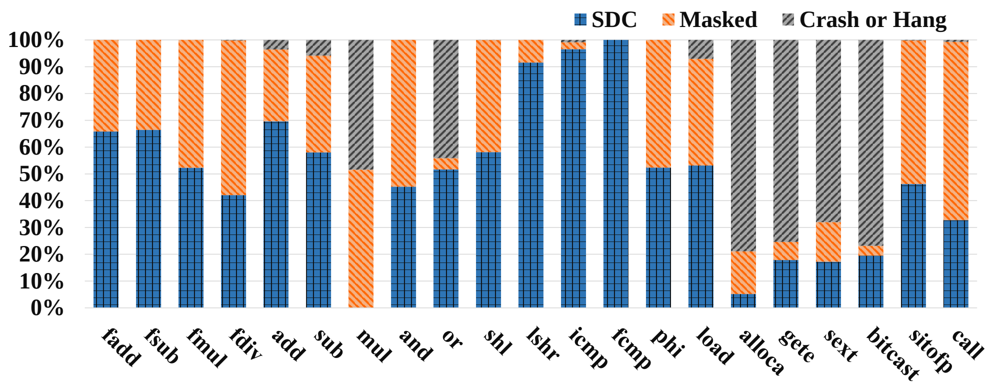


Figure 2: Impacts of instruction type on fault injection results.

we can see in Fig. 2, instructions SDC vulnerability are highly related to their types. Consequently, the extracted features shown in Tab. 2 mainly consist of the following aspects:

Transfer instructions and branch instructions are the critical points of program propagation paths Ma et al. (2016). They are highly related to the data process like parameter passing and value return. Data corruption occurs at transfer or branch instructions is most likely to cause SDCs. So transfer and branch characteristics are concerned in our feature extraction denoted as v_{tran}.

Comparison instructions impact the bits of the flag register, and they determine the jump destination of branch instructions Ma et al. (2016). A propagation through comparison instructions will be affected by the bit-flip, which will incur SDCs. Thus, we take comparison characteristics into consideration denoted as v_{comp}.

An error occurs in the instruction related to the address will lead to a crash or hang in most cases Bo et al. (2016), therefore the SDC will be greatly reduced. In addition, the address width also affects the rate of SDC, since more bits mean much probability of error. Thus, address related characteristics are the key characteristics in feature extraction denoted as v_{addr}.

Mask instructions may mask the error in the instruction operation and reduce the rate of SDCs. For example, AND operation with 0 or OR operation with 1 will not affect the result, Shift Right(SHR) operation may ignore error bits in the variables. Our feature extraction will involve mask related characteristics denoted as v_{mask}.

The loop info is also key information about SDCs. Lu et al. (2014) shows that nested loop depths affect the SDC vulnerability of loops' comparison operations, as the SDC vulnerability of comparison operations in inner loops are generally lower than the comparison operations in outer loops. We denote loop info vector as v_{loop}.

Calculation instructions generate SDCs are relatively higher than that of other instructions. Our fault injection experiments show that the SDC vulnerability of ADD, SUB, etc. are above most instructions. Furthermore, instruction SDC vulnerability depends on the basic block and function info (e.g., block size, function call number), code structure info (e.g., number of predecessor successor block, whether it is in a loop). These characteristics are intended as a supplement to our previous content which are denoted as v_{arith} and v_{basic}.

Table 2: Description of instruction features

Feature cluster	**Feature**	**Description**
Transfer and branch related features	is_branch	whether the instruction generate a branch
	is_fuction_call	whether the operation call a function
	is_return	whether the operation return a value
Comparison operations related features	is_int_cmp	whether the operation is a int comparison operation
	is_float_cmp	whether the operation is a float comparison operation
Address related feature	is_used_in_add	whether the variable is used in address related instruction
	dest_op_width	the width of destnation operation
	is_used_store	whether the varable is used in store instruction
Error Masked related features	is_and	whether the operation is AND
	is_or	whether the operation is OR
	is_sh	whether the operation is a shift operation(e.g., shl,lshr)
	is_conv	whether the operation is conversion operation(e.g., trunc,fptosi)
Loop info related features	is_in_loop	whether the instruction is in loop structure
	loopdepth	the loopdepth of the instruction
Arithmetic related features	is_add	wether the instruction is addition instruction
	is_sub	wether the instruction is subtraction instruction
	is_mul/div	wether the instruction is multiplication or division instruction
Basic block related features	bb_length	basic block size
	bb_remaining_ins_num	the number of instructions to be executed
	pred_bb_num	the number of predecessor basic blocks
	suc_bb_num	the number of successor basic blocks

In summary, we define the origin input vector as Eq. 3,

$$v_{origin} = \{v_{tran}, v_{comp}, v_{addr}, v_{mask}, v_{loop}, v_{arith}, v_{basic}\}. \tag{3}$$

4.3. Deep Forest Model

The deep forest is a decision tree ensemble approach with performance highly competitive to deep neural networks in a broad range of tasks. Our predicting model mainly contains two parts: improved sliding window scanning and cascade regression forest. Cascade regression

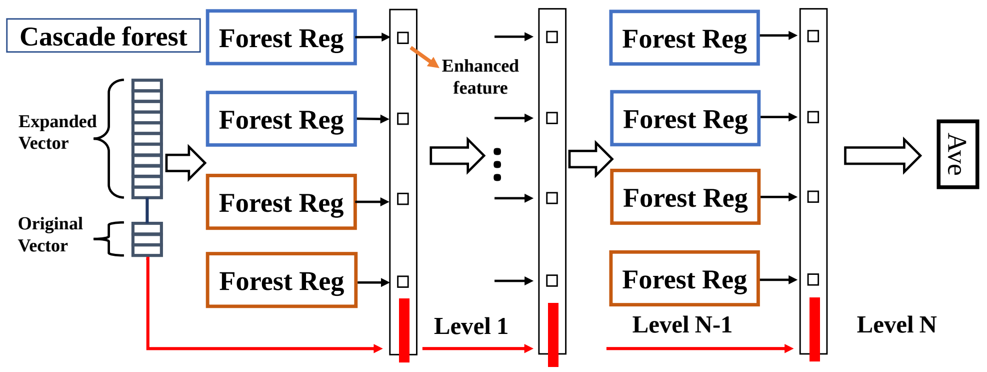


Figure 3: Cascade regression forest structure.

forest is the main structure of the deep forest, using multiple random forest layers to improve prediction accuracy. Improved sliding window scanning is a pretreatment process that is used to get sequential or spatial characteristics.

In order to make the training model adaptive to the data set and problem space, we choose the deep forest algorithm for the following reasons:

- Most of the features we extract are Boolean data type, and the features within one sample are not highly correlated. Random forest based deep forest is competent to such data type.

- Deep forest performs well on samples with few features, while other regression models, such as deep neural network needs a large number of samples. Further, deep forest requires fewer parameters adjustment which will provide better generalization ability.

- Our data set is sequential. SDC vulnerability is susceptible to instruction sequences in propagation path. Extracting instruction sequence features can improve the prediction accuracy. Deep forest is effective on sequence data where sequential relationships are critical by using window sliding.

4.3.1. Cascade Regression Forest Structure

We employ a cascade regression structure to train the predictor, as illustrated in Fig. 3. Each level receives feature information processed by its preceding level and outputs its processing result to the next level.

Each level is composed of multiple random regression forests, in other words, it is a type of ensemble learning. Bronevetsky et al. (2009) has showed sufficient diversity is critical for ensemble learning, therefore, we introduce different types of random regression forests to increase diversity. In this paper, we use two extremely random forests denoted as ϕ_{extr} and two normal random forests denoted as ϕ_{norm} at each level. Each extremely random forest contains 200 decision trees, generated by randomly selecting one feature among all features to split at each node of the tree, and selecting all samples to train each tree. The normal random forest also contains 200 decision trees, generated by randomly selecting $\sqrt{n}$ number of features (n is the number of input features) at each node but chose the one with

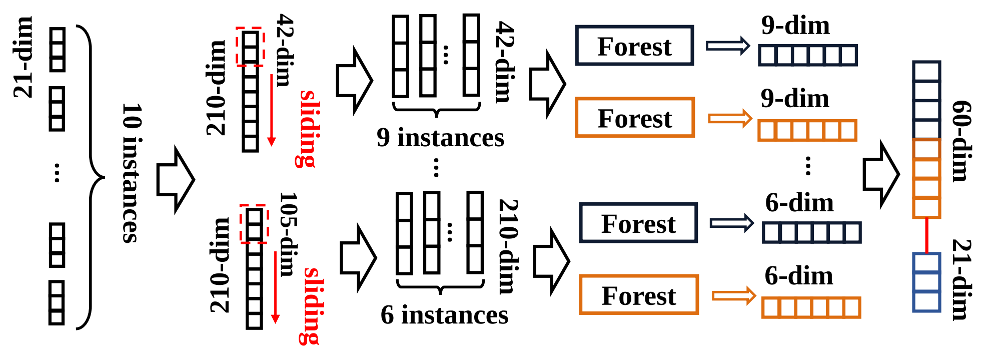


Figure 4: Improved sliding window scanning.

the best MSE value to split, at the same time, randomly selecting part of the samples to train each tree.

Fig. 3 shows the details of the cascade structure. The initial input vector are defined as Eq. 4:

$$v_{input} = [v_{origin}, v_{expanded}], \tag{4}$$

where v_{origin} is defined in Section 4.2, $v_{expanded}$ is generated by the sliding widow scanning that will be introduced in the next sub-section. Each regression forest ϕ will produce a regression result $\phi(v_{input})$ called enhanced feature, by searching for a best-split value on the selected features at each node and calculating the average value at each split space. The final output of this forest is the average value among all trees.

The enhanced features, which is then concatenated with the original feature vector, denoted as the Eq. 5:

$$v_{input}^{n} = [\phi_{extr_1}^{n-1}(v_{input}^{n-1}), \phi_{extr_2}^{n-1}(v_{input}^{n-1}), \phi_{norm_1}^{n-1}(v_{input}^{n-1}), \phi_{norm_2}^{n-1}(v_{input}^{n-1}), v_{origin}], \tag{5}$$

will be used as input to the next cascade level. V_{input}^{n} means the nth level input vector. In our case, we get four enhanced features at the ensemble layer, then the next cascade layer will receive four extra features.

In order to improve the accuracy and reduce the cost of training, the performance of the whole cascade will be estimated on validation after expanding a new layer. Once the performance of the whole model does not improve, the training process will be terminated. In this way, the number of the layer will be automatically determined. The SDC vulnerability prediction function can be denoted as Eq. 6:

$$F(v_{input}) = \frac{1}{m}\sum_{i=1}^{m} \phi_i(v_{input}^{N}), \tag{6}$$

where N represents the total number of cascade forest levels, m represents the number of random forests in the Nth level.

4.3.2. Improved Sliding Window Scanning

The traditional sliding window method can only extract sequence features of one sample, because it slides the window within the sample. However, in our work, correlations of features within one sample are weak, while the correlations between samples are strong. To addressed this problem, an improved sliding window method is illustrated as shown in Fig. 4.

As shown in Fig. 4, a new vector is produced by connecting ten instances. To reduce the impact of sliding on one sample, we set the widow dimension and the step length to be integer multiple of number of features. In our case, the window dimensions are ranged from 21 to 105, and the step length is set to be 21. For example, suppose there is a 21-dimensional window that will slide across the new vector, and take 21 steps forward each time, then 9 instances are produced. The instances extracted from the same size of windows will be used to train an extremely random forest and a normal random forest, then the regression values are generated and concatenated as transformed features. In total, 60-dimensional features will be produced, and lead to a 60-dimensional expanded feature vector denoted as $v_{expanded}$. For the expanded vector is generated by 10 instances, finally, we concatenate the expanded vector with the tenth original feature vector and use the tenth instance's SDC vulnerability as the predictive value. The final vector will be used as the initial input of the cascade forest.

4.4. Instruction Redundancy Strategy

Instructions can be processed at different granularity, where high SDC vulnerability instructions are preferred for redundancy protection. The designed strategy is as follows:

4.4.1. Select target instructions

The instructions will be sorted in a descending order by their SDC vulnerability denoted as I_{sort}, then we select instructions to perform redundancy processing with a granularity value of Z. The instructions set selection is described as Eq. 7:

$$I_{selected} = \{I_1, I_2..., I_s | s = \lceil N \times Z \rceil, 0 \leq Z \leq 1\}, \tag{7}$$

where $I_{selected}$ is the first S instructions selected from I_{sort}, Z is redundant granularity that specified by users, and N is the total number of instructions in the program.

4.4.2. Instruction fault tolerance

The selected instructions set $I_{selected}$ will be duplicated to obtain fault tolerance instructions set I_{dup}.For all instructions in I_{dup}, if I_{dup_i} depends on I_{dup_j} and $i > j$, we insert comparison instruction into the end of the basic block, comparing the execution result of original instruction with redundancy instruction. Besides, if the instruction in I_{dup} can not form a relationship with other instructions, a unique comparison instruction will be inserted. If an error occurs when the program is running, the detector will generate inconsistent results; on the contrary, it means that the program is running normally.

5. Experiment

In this section, experiments are designed to validate our proposed method. Our evaluation experiments use a PC with Intel i7 8750H CPU and 16G memory running Ubuntu Linux

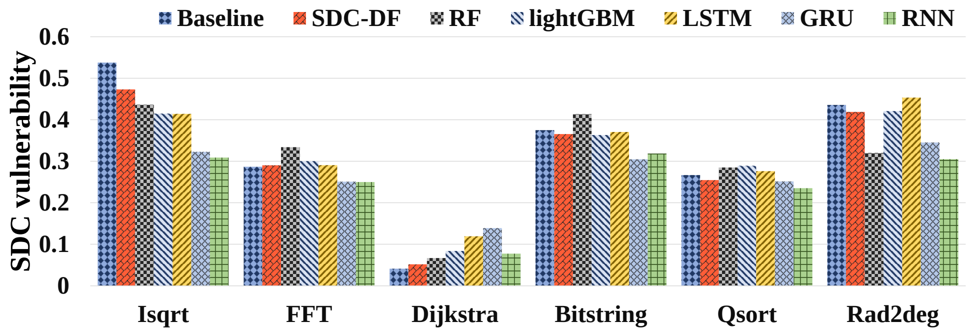


Figure 5: The SDC vulnerability prediction accuracy of testing programs.

16.04. We randomly select part of the benchmarks in Mibench[2] as the training set, and the rest as the test set. The selected Mibench benchmarks are firstly compiled using LLVM to obtain the Intermediate Representation(IR) instructions and then an LLVM Pass is performed to extract instruction features. For each selected program, we conduct automatic fault injections to get SDC vulnerability of each instruction using a script on LLFI. LLFI is an LLVM based fault injection framework and operates at the IR instruction level. In total, we collect 3400 SDC vulnerability training samples.

This paper simulates the impact of soft error by injecting faults into a single bit of instruction destination registers. We only discuss the program running in a single thread mode, and we chose 6 benchmarks, Qsort (Quick sorting), Dijkstra (Shortest path algorithm), FFT (Fast Fourier Transformation), Isqrt (Int square root calculations), Bitstring (Bit pattern of bytes formatted to string) and Rad2deg (Radians to degrees).

In reference to previous works Shivakumar et al. (2002); Wang et al. (2015); Wei et al. (2014), we make the following assumptions for fault injection experiments: (1) We consider single flip error, and assume that only one failure occurs during the execution of the program. (2) This paper focuses on the soft error that occurred in registers (eg, ALU, LSU, GPR, etc.) of a processor, therefore, only the error in the instruction operand is considered.

We evaluate our approach in two aspects: (1) the accuracy of SDC vulnerability prediction; (2) the effectiveness of SDC-causing error detection.

5.1. Instruction SDC vulnerability Prediction Experiment

Predicting the SDC vulnerability of instructions is the most critical part of the machine learning based protection method. The parameters of the model are set as described in section 4.3.1. 200 trees are used in each forest for cascade regression forest and 30 trees for sliding window forest. The calculation method of the instruction SDC vulnerability is shown as Eq. 2, which is used as our prediction criteria.

We compare our proposed method with the random forest Liu et al. (2019) consisting of 800 trees, the lightGBM model with 0.03 learning rate and 800 estimators. We also compare it with deep neural networks represented by LSTM/GRU Liu et al. (2018) with 64 hidden units and sequence length of 10. ReLU, cross-entropy loss, AdamOptimizer and

2. http://vhosts.eecs.umich.edu/mibench

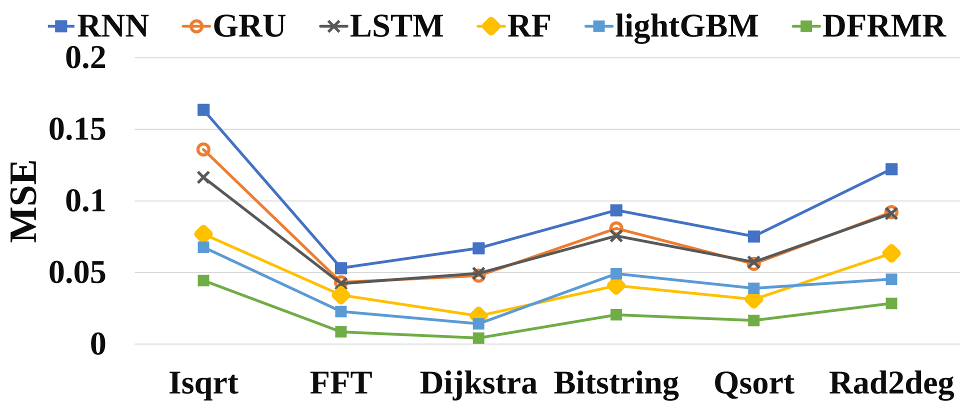


Figure 6: The MES comparison of testing programs.

Table 3: The MSE performance of the testing programs

	Isqrt	FFT	Dijkstra	Bitstrng	Qsort	Rad2deg	Average
RNN	0.1636	0.0531	0.0670	0.0935	0.0752	0.1222	0.0958
GRU	0.1359	0.0432	0.0478	0.0808	0.0560	0.0921	0.0760
LSTM	0.1164	0.0423	0.0494	0.0756	0.0573	0.0912	0.0720
RF	0.0768	0.0342	0.0197	0.0409	0.0313	0.0633	0.0444
lightGBM	0.0677	0.0228	0.0142	0.0491	0.0390	0.0453	0.0397
DFRMR	0.0444	0.0086	0.0042	0.0205	0.0165	0.0284	0.0204

learning rate of 0.02/0.002 are used for training. The SDC-causing error rate obtained by fault injection (FI) is used as a baseline. Experimental results are shown in Fig. 5. Compared to other methods, DFRMR performs more accurate and stable in most cases. We also evaluate the predicting accuracy of our prediction model by calculating the average squared errors(MSE) of the testing data set. MSEs of test programs are shown in Fig.6. As we can see in Fig. 6, DFRMR gets the best MSE score on all test programs. The details are shown in Tab. 3.

We find that the average SDC vulnerability of each test program is highly different. To be specific, the highest and the lowest SDC vulnerability are obtained for Isqrt and Dijkstra, respectively. We analyze the types and SDC vulnerabilities of instructions in each program. The result shows that Isqrt contains more integer and comparison operations which lead to higher SDC vulnerability, while Dijkstra contains more conversion and address related operations. Because the former will be masked when an error occurs and the latter will mainly cause system fault.

5.2. SDC-causing Error Detection Experiment

In this paper, the SDC-causing error detection rate and performance overhead of the proposed method are evaluated. We randomly injected 3000 faults into the benchmarks to verify the DFRMR method.

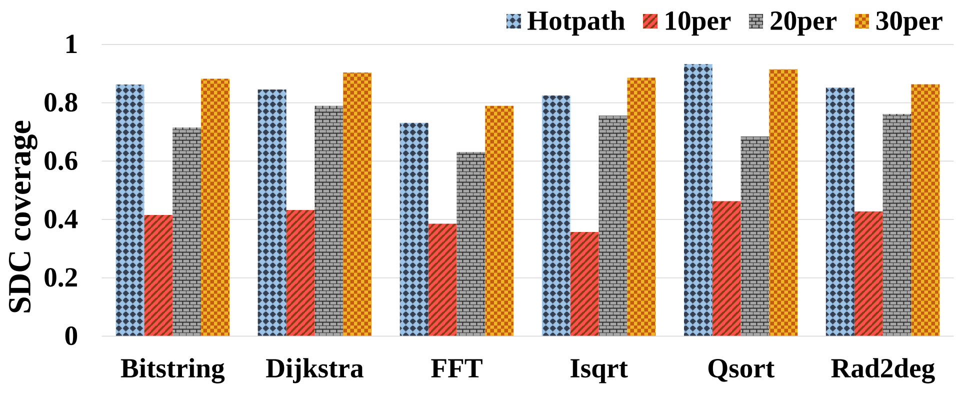


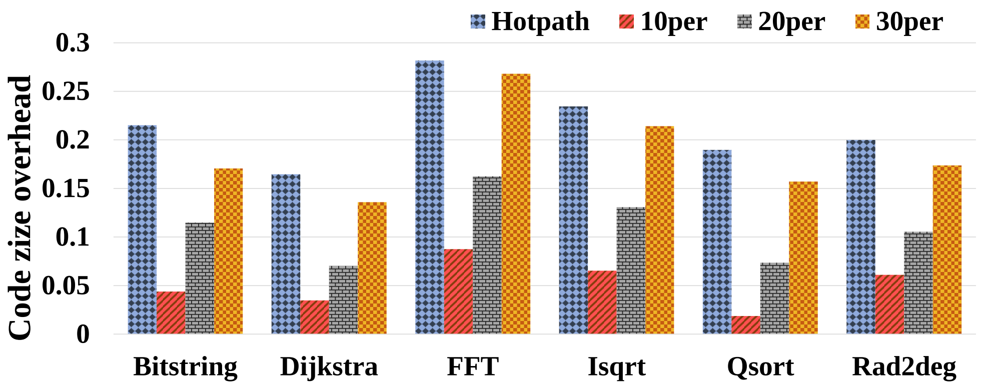
Figure 7: SDC coverage of test programs.

SDC coverage means the rate of errors detected by our error detector. According to specified granularity, we will flexibly select the instructions with high SDC vulnerability, and implement redundancy technology on them. To collect the data of SDC coverage, we inject faults into the protected program using LLFI. Fig. 7 shows the comparison of SDC coverage between our method and Hotpath Lu et al. (2014). 10per, 20per and 30per, which mean the SDC coverage of target program with redundancy granularity of 10%, 20%, and 30%. Hotpath is the SDC coverage of the target program with redundancy of frequent instructions. The frequent instructions are obtained through the LLVM framework by implemented Pass and this paper selects the first 20% of frequently executed instructions. As we can see in Fig. 7, the averages SDC coverage for Hotpath, 10%, 20% and 30% redundancy granularity are 84.5%, 39.5%, 68.3% and 86.2%. Our method acquires high SDC coverage at 20% and 30% redundancy granularity, and achieves higher SDC coverage at 30% than Hotpath.

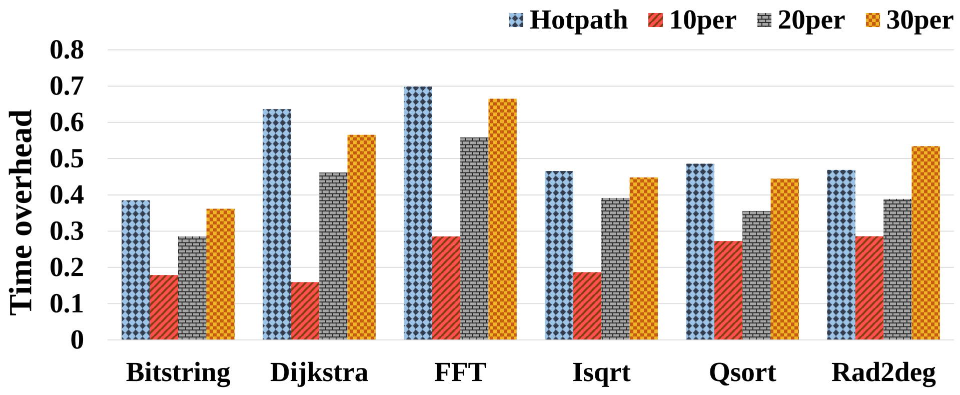
Fig. 8 shows a comparison of performance overhead. The protected programs are compared with the original programs. We record the code size and execution time of each benchmark under a specified granularity protected. As shown in Fig. 8, the average code size overhead of Hotpath, 10%, 20% and 30% redundancy granularity are 23.9%, 5.1%, 10.9% and 18.6%, while the average execution time overhead of Hotpath, 10%, 20% and 30% redundancy granularity are 52.8%, 22.5%, 41.2% and 50.3%.

We find that time performance overhead among benchmarks is not uniform. This is because of benchmark-specific reasons such as the distribution of high SDC vulnerability instructions. Our method needs to protect instructions with high SDC vulnerability, but these instructions may execute at a low frequency. In contrast, duplicating instructions within loops will lead to higher performance overhead. We observe that programs with deep loops may cause high time overhead, such as FFT and Dijkstra. So programs with less depth of loops may benefit more from our method, for they will get an efficiency fault tolerance. Further, the code size overhead mainly depends on the program scale, we will not discuss too much in this paper.

In summary, our method significantly outperforms Hotpath in providing better fault tolerance with a lower performance overhead.



(a) Code size overhead.



(b) Time overhead.

Figure 8: Performance overhead of test programs.

6. Conclusion

This paper proposed a lightweight SDC-causing error detection model named DFRMR to overcome shortcomings of redundancy, such as expensive time cost and code size cost. We used the program analysis technique to extract the features indicating the SDC vulnerability of instructions. Then we developed instructions SDC vulnerability prediction model based on deep regression forest. To improve prediction accuracy, we employed various regression forests and improved the sliding window scanning method. We used DFRMR to guide the selection of instructions under different granularity. Our experiments showed that DFRMR performed a high accuracy in SDC prediction, and the error detector gave an efficiency fault tolerance with lower overhead.

References

Fang Bo, Qining Lu, Karthik Pattabiraman, Matei Ripeanu, and Sudhanva Gurumurthi. epvf: An enhanced program vulnerability factor methodology for cross-layer resilience analysis. In *IEEE/IFIP International Conference on Dependable Systems & Networks*, 2016.

Greg Bronevetsky, B de Supinski, and Martin Schulz. A foundation for the accurate prediction of the soft error vulnerability of scientific applications. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2009.

Hongjun Dai, Chao Yan, Bin Gong, Zhun Yang, and Tianzhou Chen. Exploring predictable redundant instruction parallelism in fault tolerant microprocessors. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 324–329. IEEE, 2015.

Siva Kumar Sastry Hari, Sarita V Adve, and Helia Naeimi. Low-cost program-level detectors for reducing silent data corruptions. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12. IEEE, 2012a.

Siva Kumar Sastry Hari, Sarita V Adve, Helia Naeimi, and Pradeep Ramachandran. Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults. In *ACM SIGPLAN Notices*, volume 47, pages 123–134. ACM, 2012b.

Martin Hiller, Arshad Jhumka, and Neeraj Suri. On the placement of software mechanisms for detection of data errors. In *Proceedings International Conference on Dependable Systems and Networks*, pages 135–144. IEEE, 2002.

F Lima Kastensmidt, Luca Sterpone, Luigi Carro, and M Sonza Reorda. On the optimal design of triple modular redundancy logic for sram-based fpgas. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 2*, pages 1290–1295. IEEE Computer Society, 2005.

Ignacio Laguna, Martin Schulz, David F Richards, Jon Calhoun, and Luke Olson. Ipas: Intelligent protection against silent output corruption in scientific applications. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 227–238. IEEE, 2016.

Jianli Li and Qingping Tan. Smartinjector: Exploiting intelligent fault injection for sdc rate analysis. In *2013 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, pages 236–242. IEEE, 2013.

LiPing Liu, LinLin Ci, Wei Liu, and Hui Yang. Identifying sdc-causing instructions based on random forests algorithm. *KSII Transactions on Internet & Information Systems*, 13 (3), 2019.

Yunfei Liu, Jing Li, and Yi Zhuang. Instruction sdc vulnerability prediction using long short-term memory neural network. In *International Conference on Advanced Data Mining and Applications*, pages 140–149. Springer, 2018.

Qining Lu, Karthik Pattabiraman, Meeta S. Gupta, and Jude A. Rivers. Sdctune: A model for predicting the sdc proneness of an application for configurable protection. In *International Conference on Compilers*, 2014.

Qining Lu, Guanpeng Li, Karthik Pattabiraman, Meeta S Gupta, and Jude A Rivers. Configurable detection of sdc-causing errors in programs. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(3):88, 2017.

Junchi Ma, Dengyun Yu, Yun Wang, Zhenbo Cai, Qingxiang Zhang, and Cheng Hu. Detecting silent data corruptions in aerospace-based computing using program invariants. *International Journal of Aerospace Engineering*, 2016.

Karthik Pattabiraman, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar Iyer. Symplfied: Symbolic program-level fault injection and error detection framework. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 472–481. IEEE, 2008.

Toshinori Sato and Itsujiro Arita. In search of efficient reliable processor design. In *International Conference on Parallel Processing*, pages 525–532. IEEE, 2001.

Premkishore Shivakumar, Michael Kistler, Stephen W Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings International Conference on Dependable Systems and Networks*, pages 389–398. IEEE, 2002.

Arie Shoshani and Doron Rotem. *Scientific data management: challenges, technology, and deployment*. Chapman and Hall/CRC, 2009.

Honghao Wang, Huiquan Wang, and Zhonghe Jin. Bipartite graph-based control flow checking for cots-based small satellites. *Chinese Journal of Aeronautics*, 28(3):883–893, 2015.

Jiesheng Wei, Anna Thomas, Guanpeng Li, and Karthik Pattabiraman. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 375–382. IEEE, 2014.

Xin Xu and Man-Lap Li. Understanding soft error propagation using efficient vulnerability-driven fault injection. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12. IEEE, 2012.

Na Yang and Yun Wang. Identify silent data corruption vulnerable instructions using svm. *IEEE Access*, 7:40210–40219, 2019.

Xue-Jun Yang and Long Gao. Error flow model: Modeling and analysis of software propagating hardware faults. *Ruan Jian Xue Bao(Journal of Software)*, 18(4):808–820, 2007.

Zhi-Hua Zhou and Ji Feng. Deep forest: towards an alternative to deep neural networks. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 3553–3559. AAAI Press, 2017.