Forseti: An Efficient Basic-block-level Sensitivity Analysis Framework Towards Multi-bit Faults

Jinting Ren, Xianzhang Chen*, Duo Liu*, Moming Duan, Renping Liu, Chengliang Wang
College of Computer Science, Chongqing University, Chongqing, China

***Abstract*—The per-instruction sensitivity analysis framework is developed to evaluate the resiliency of a program and identify the segments of the program needing protection. However, for multi-bit hardware faults, the per-instruction sensitivity analysis frameworks can cause large overhead for redundant analyses. In this paper, we propose a basic-block-level sensitivity analysis framework, Forseti, to reduce the analysis overhead in analyzing impacts of modern microprocessors' multi-bit faults on programs. We implement Forseti in LLVM and evaluate it with five typical workloads. Extensive experimental results show that Forseti can achieve more than 90% sensitivity classification accuracy and 6.16× speedup over instruction-level analysis.**

***Index Terms*—Sensitivity Analysis, Multi-bit Faults, Fault Injection**

I. Introduction

Modern microprocessor chips may have unmeant bit flips caused by cosmic radiation, latent manufacturing defects, etc [1], [2]. Such hardware faults may cause Silent Data Corruption (SDC), which unwittingly causes unacceptable or catastrophic system failures, during the execution of programs. With the development of CMOS technology, microprocessor chips are becoming denser and more complex, which makes the chips more vulnerable to hardware faults. Therefore, software or hardware implemented error handling mechanisms [3], [4] become more important for the reliability of microprocessors. Besides the reliability improvement, those methods can also bring significant overhead. A sensitivity analysis framework is proposed to reduce the overhead by identifying the code needing protection.

For multi-bit sensitivity analysis, the previous study proposes a method that speedup the analysis based on the outcomes of single-bit faults simulation [5]. This method needs to inject single-bit faults first and only choose the instruction which can produce detected outcome (detected by the system or software behavior) to simulate multi-bit faults. Hence, this method needs to perform redundant single-bit error analysis and cannot fully take advantages of the properties of multi-bit faults. To further speed up the sensitivity analysis, we need a new sensitivity analysis framework to analyze the program sensitivity at a higher level.

To tackle this challenge, this paper aims to design an automated sensitivity analysis framework using the multi-bit flip model. Inspired by previous researches, we find that most multi-bit flip errors that may cause SDC usually occur within several adjacent instructions [5]. In other words, the errors that can cause SDC have a high chance to occur within one basic block. Hence, basic blocks are suitable for analyzing the program sensitivity towards multi-bit errors instead of individual instructions.

In this paper, we propose a new basic-block-level sensitivity analysis framework, Forseti, to speed up the sensitivity analysis of multi-bit flip errors. We select five workloads from AXBench [6] to evaluate the proposed Forseti framework. Extensive experimental results show that our framework can achieve over 90% accuracy in sensitivity analysis for most benchmarks. Compared with instruction-level analysis, our framework can achieve over 6.16× speedup. To our best knowledge, this paper is the first work to analyze the program sensitivity in the basic-block level for multi-bit faults.

II. Background and Motivation

A. Multi-bit vs Single-bit

There are two typical fault models for hardware fault simulation in sensitivity analysis frameworks. Early studies use the single-bit model for it is simple and can cover most fault cases [7]. However, recent studies show that transistors are increasingly susceptible to soft errors caused by newly-developed ionizing particle technology and voltage scaling. At the same time, the errors that manifest as multi-bit flip errors are also increasing. For example, a recent study [8] finds that only 78% of errors manifest as single-bit flips with RTL level particle-strike model. Hence, many recent studies are investigating the impact of multi-bit errors [2], [9], [10].

Moving from a single-bit model to a multi-bit model will cause substantial increase in the error candidates space. Compared with the single-bit model, the multi-bit model needs to take the combination patterns of the fault bits within a single instruction and the fault locations for multiple instructions into consideration. How to prune the error space is important especially for multi-bit based sensitivity analysis frameworks.

B. Problems of Existing Instruction-Level Mechanisms

A previous work reduces the error space by selecting instructions for multi-bit sensitivity analysis based on single-bit analysis results in advance [5]. However, there are two defects in this solution. First, the solution needs redundant single-bit analysis before the sensitivity analysis of multi-bit errors. Second, this solution needs to inject real single-bit faults first to achieve higher precision. Thus, this approach cannot estimate the number of multi-bit fault injections before finishing all the single-bit fault injections. As a result, the overall overhead of this solution is unpredictable large.

On the contrary, the sensitivity analysis at a level higher than the instruction level is able to reduce the overall overhead and make the estimation of overhead easier. A recent study shows that most SDCs are caused by several adjacent instructions [5]. Those adjacent instructions will occur within one basic block

Corresponding author: Xianzhang Chen and Duo liu, College of Computer Science, Chongqing University, Chongqing, China.
E-mail: {xzchen, liuduo}@cqu.edu.cn.

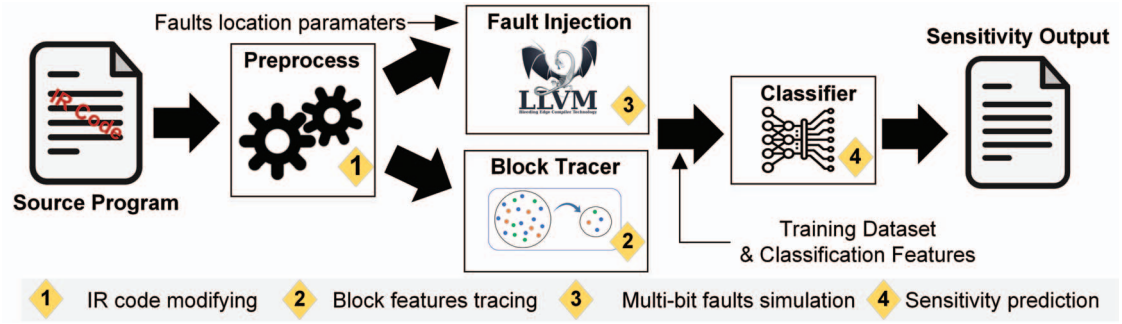

Fig. 1: The overall architecture of Forseti.

with a high probability. Thus, we propose to perform the sensitivity analysis at basic-block level.

III. Forseti Design

This paper proposes an automated sensitivity analysis framework, Forseti, that can directly prune the error space of the multi-bit flip fault model. The overall architecture of Forseti is shown in Fig. 1. It consists of four parts: code pre-processing, basic block tracer, fault injection, and sensitivity classifier. Among them, the code pre-processing part is responsible for indexing instructions and labeling basic blocks for the following parts.

A. Basic Block Tracer

The Basic Block Tracer needs to trace the information of the basic blocks and produce the formatted features for the classifier. To better describe a basic block, we collect the information from various aspects based on the domain knowledge. The traced basic block information is classified into three categories: basic information, control flow information, and data flow information.

The basic information of a basic block includes the ratio of memory operation instructions in the block, whether the block is the entry block or the exit block, and whether the block contains an external call. These information can be directly accessed by the "BasicBlock" interface of LLVM [11]. The other information needed is the detailed types of instructions.

Obtaining the control flow information needs a control flow analysis on the target block. We choose the number of predecessors and the number of successors as the control flow information. The predecessors of the target basic block are the basic blocks that can directly jump to the target block. Similarly, the successors of the target basic block are the basic blocks that the target block can directly jump into. These information is provided by the CFG interface of LLVM. Data flow information is obtained by a data flow analysis algorithm that is close to the live variables analysis. We use the inputs and outputs of the basic block as the features.

B. Fault Injection

Generally, the fault injection module is responsible for injecting faults into the target program to gain the label of basic blocks (Whether this basic block is sensitive). We use a LLVM-based software-implemented fault injection tool to achieve this goal. Since all the actual fault injections are done at instruction level, we need to define the basic block that the injected faults belongs to. If only one instruction meets faults, the fault injection belongs to the basic block of this instruction. Furthermore, if multiple errors occur in different instructions, the fault injection is regarded as belonging to the basic block of the first error instruction. This definition can be changed as long as it can cover all the situations for multi-bit faults simulation.

After defining the basic blocks that all the faults belong to, we still need to simulate most fault situations that may cause SDC within a single basic block. Hence, we use two parameters, "Number of Bits" and "Fault Gap", to control the number of errors and the executed instructions between two injected faults. According to previous studies [5], the errors for several contiguous instructions can cover most conditions that may cause SDC. We set the maximum of "Number of Bits" and "Fault Gap" to three and five, respectively.

Algorithm 1 Multi-bit Fault Injection

Input: Shortest distance D, fault gap F and number of bits N

```
1:  for each instruction I_i in target basic block do
2:    if I_i is binary instruction then
3:      for F from zero to Max_Gap do
4:        for N from one to Max_Bits do
5:          if F is equal to zero then
6:            insert N bits flip function call before I_i and
              replace all uses of it.
7:          else
8:            find N − 1 binary instructions which are the
              next 1 ∗ N ... 1 ∗ (N − 1) instructions after I_i
              according to shortest distance D.
9:            insert single-bit flip function call before the
              N − 1 instructions and replace all uses of
              them.
10:         end if
11:       end for
12:     end for
13:   else
14:     skip this instruction.
15:   end if
16: end for
```

The whole iteration space for one basic block consists of all binary instructions within it. For each instruction, the fault injection module first checks if the fault injection function has been created for the target type, i.e., the type of the destination operand of the current instruction. If the function has been created, we insert the fault injection function call before the target instruction and then replace all the target instruction with the function call.

The detailed multi-bit fault injection is shown in Algorithm 1. All the error cases begin with a specific instruction composing one error candidate. An error candidate needs to explore two parameters configurations possibilities. For one specific configuration, we inject faults with random bit positions for 20 times by default. The output of each fault injection should be compared with the golden outputs. If all the sampling outcomes of a candidate instruction are not SDC, we define the instruction as *insensitive*. Meanwhile, if all instructions within one basic block are insensitive, we define this basic block as insensitive. Otherwise, we label the basic block as *sensitive*.

C. Classifier

As we mentioned in the previous sections, the classifier uses a training dataset collected by the basic block tracer (Features collection) and the fault injection module (Labels collection) to train the model. Then, it classifies the rest of basic blocks instead of doing actual fault injection experiments. The main problem of the classifier is how to select a suitable model.

Firstly, the sensitivity analysis for the basic block is a binary classification task. Meanwhile, as mentioned in the design of the basic block tracer, we sample eight features for classification. Hereby, the overall data amount for a single program in block level is usually in the order of hundreds, which is not a large number. A lightweight machine learning model, such as decision tree, neural network, naive Bayesian model, k-nearest neighbor, is enough for solving the problem.

Among those models, because the distribution of the features whether this basic block is the initial block, whether this block contains external calls and whether this block is exit block are not Gaussian distribution, the naive Bayesian model is not suitable. Furthermore, we need to adjust the amount of collected information to achieve better training accuracy (this part will be discussed in detail in later sections), methods like the k-nearest neighbor algorithm and decision tree are not suitable. Considering all of those factors, we use the neural network as our classifier model in this paper.

IV. Evaluation

The hardware and software configurations of the experimental platform are shown in TABLE I. The benchmarks are chosen from AXBench [6], including Black-Scholes, FFT, Inversek2j, Jmeint, and K-means.

A. Sensitivity Classification

In this section, we want to discuss whether the machine learning method can achieve acceptable accuracy for most benchmarks using different sampling rates. The sensitivity classification accuracy for basic block is shown in Fig. 2. We use two sampling rates to collect training dataset for classification — sampling half of basic blocks and sampling two-thirds of basic blocks. The Fig 2 shows that, when sampling more data, the accuracy will increase for all benchmarks. This is because our machine learning model is underfitting with a few data and collecting more data can help improve the classification accuracy. However, the accuracy improvement using more data can vary between different benchmarks. For the benchmark like K-meanse, the improvement is not significant.

Fortunately, when the sampling rate reaches two-thirds of basic blocks, most benchmarks can achieve acceptable accuracy (over 90%) except K-MEANS which can achieve 83.3% accuracy. When studying this benchmark, we found that this benchmark has many basic blocks that most instructions within it are memory operations. For this reason, the output quality of this benchmark has a limited relationship with faults that occurred in binary instructions. The is also the possible reason why the overall accuracy of this benchmark can not reach an acceptable degree.

TABLE I: Experiments setup.

Hardware	CPU	Intel E5-2630 v3 @ 2.40GHz × 8
	GPU	NVIDIA Corporation Tesla K40c
	DRAM	48GB DDR3
	OS	Ubuntu 16.04 with 4.13 kernel
Software	LLVM	version 4.0.1
	Keras	version 2.2.4

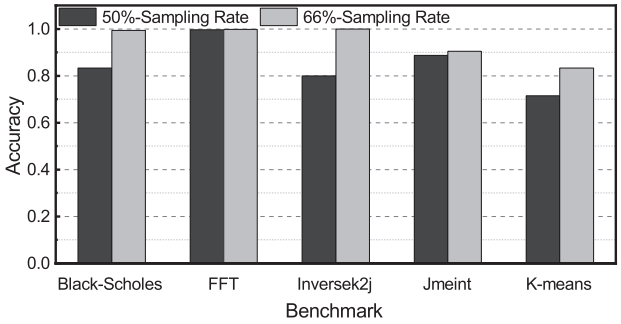

Fig. 2: Sensitivity classification accuracy with different sampling rates.

B. Overall Speedup

In this part, we evaluate the speedup that our Forseti can achieve compared with instruction-level analysis method. In this paper, we use the number of instructions to be analyzed to simply evaluate the whole sensitivity analysis speedup. The sensitivity analysis overhead comparison between Forseti and instruction-level analysis is shown in Fig 3. We also use four configurations to evaluate the effect of different sampling ways. The meaning of each symbol are listed here: 50%-F-Min represents the minimum instructions number to be analyzed sampling 50% basic blocks; 50%-F-Max represents

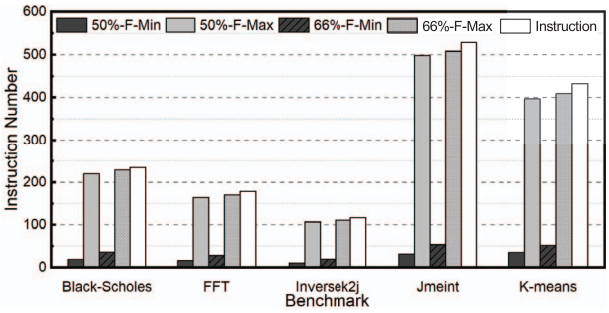

Fig. 3: Speedup comparison between Forseti and instruction-level analysis framework.

the max instructions number with 50% sampling rate; The other symbols have similar definitions. Instruction represents the instruction-level analysis.

From the figure, we can find that using different sampling ways (choose different basic blocks as training dataset but do not change sampling rates) can influence the speedup significantly. For example, for the FFT benchmark, we need can achieve 11.03 times speedup for max cases and 1.08 times for min cases with sampling half of basic blocks. On the other hand, different sampling rates do not have such an impact on speedup. The max difference using different sampling rates is two achieved between 50%-F-Min and 66%-F-Min for the Black-Sholes benchmark.

In the previous section, we do sensitivity classification using random sampling and we found that different sampling ways do not influence the accuracy greatly. In contrast, the sampling rates can have an obvious influence on accuracy. Hence, this paper found that we can use basic blocks that contain little instructions as sampling targets to achieve the most significant performance improvement without sacrificing much accuracy.

We can also see the influence of the program characteristics from the figure. For max speedup, the highest one is FFT with 11.13 times speedup and the lowest one is Inverse2j achieve 6.16 times. Meanwhile, for min speedup, the highest one is also FFT with 1.08 times speedup and the lowest one is Black-scholes with 1.02 times. This is easy to understand, for different benchmarks, the distribution of instruction number among different basic blocks can vary between different programs. Hence, skipping the same rates of basic blocks can result in different speedup because of it.

In conclusion, considering the accuracy limits (over 90% sensitivity classifications), the highest speedup can be achieved is 11.03 times for the FFT benchmark sampling half of the basic blocks compared with instruction-level analysis. The lowest speedup is for the Inversek2j benchmark and our framework can also achieve 6.16 times speedup. Even the worst case for the K-means, our framework can easily turn into the traditional instruction-level analysis because the training dataset collection is same as the instruction-level analysis.

V. Conclusion

Current sensitivity analysis frameworks mostly focus on the single-bit flip model and do not have a specific method to speed up the sensitivity analysis based on the multi-bit model. In this paper, we have proposed an automated sensitivity analysis framework towards multi-bit faults based on LLVM. This framework uses neural networks to classify the basic blocks instead of doing actual fault injection experiments partly. Through the proposed method, our sensitivity classification module can achieve over 90% accuracy for most benchmarks. Based on that, the evaluation finds that our framework can achieve at least 6.16 times speedup over instruction-level analysis.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable feedback and improvements to this paper. This work is partially supported by grants from the National Natural Science Foundation of China (61672115, 61672116, 61601067 and 61802038), Chongqing High-Tech Research Key Program (cstc2019jscxmbdx0063), the Fundamental Research Funds for the Central Universities under Grant (0214005207005 and 2019CDJGFJSJ001), the Funds for Chongqing Distinguished Young Scholars (cstc2020jcyj-jqX0012).

References

[1] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson *et al.*, "Addressing failures in exascale computing," *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014.

[2] A. Chatzidimitriou, G. Papadimitriou, C. Gavanas, G. Katsoridas, and D. Gizopoulos, "Multi-bit upsets vulnerability analysis of modern microprocessors," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2019, pp. 119–130.

[3] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 2005, pp. 243–254.

[4] D. Lin, T. Hong, Y. Li, S. Eswaran, S. Kumar, F. Fallah, N. Hakim, D. S. Gardner, and S. Mitra, "Effective post-silicon validation of system-on-chips using quick error detection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 10, pp. 1573–1590, 2014.

[5] B. Sangchoolie, K. Pattabiraman, and J. Karlsson, "One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 97–108.

[6] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, "Axbench: A multiplatform benchmark suite for approximate computing," *IEEE Design & Test*, vol. 34, no. 2, pp. 60–68, 2016.

[7] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *Ieee Micro*, vol. 25, no. 6, pp. 10–16, 2005.

[8] C.-K. Chang, S. Lym, N. Kelly, M. B. Sullivan, and M. Erez, "Hamartia: A fast and accurate error injection framework," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2018, pp. 101–108.

[9] F. Ayatolahi, B. Sangchoolie, R. Johansson, and J. Karlsson, "A study of the impact of single bit-flip and double bit-flip errors on program execution," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2013, pp. 265–276.

[10] M. Wilkening, V. Sridharan, S. Li, F. Previlon, S. Gurumurthi, and D. R. Kaeli, "Calculating architectural vulnerability factors for spatial multi-bit transient faults," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 293–305.

[11] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.