A non-conservative software-based approach for detecting illegal CFEs caused by transient faults

Rodrigues, Diego; Nazarian, Ghazaleh; Moreira, Álvaro; Carro, Luigi; Gaydadjiev, Georgi

DOI
10.1109/DFT.2015.7315166

Publication date
2015

Published in
Proceedings of the 2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems, DFTS 2015

A non-conservative software-based approach for detecting illegal CFEs caused by transient faults

Diego Rodrigues
Institute of Informatics
Federal University of Rio Grande do Sul
Porto Alegre, Brazil
Email: diegogr@inf.ufrgs.br

Ghazaleh Nazarian
Faculty of Electrical Engineering, Mathematics
and Computer Science
Delft University of Technology
Delft, The Netherlands
Email: g.nazarian@tudelft.nl

Álvaro Moreira
Institute of Informatics
Federal University of Rio Grande do Sul
Porto Alegre, Brazil
Email: afmoreira@inf.ufrgs.br

Luigi Carro
Institute of Informatics
Federal University of Rio Grande do Sul
Porto Alegre, Brazil
Email: carro@inf.ufrgs.br

Georgi Gaydadjiev
Department of Computer Science
and Engineering
Chalmers University of Technology
Gothenburg, Sweden
Email: georgig@chalmers.se

***Abstract*—Software-based methods for the detection of control-flow errors caused by transient fault usually consist in the introduction of protecting instructions both at the beginning and at the end of basic blocks. These methods are conservative in nature, in the sense that they assume that all blocks have the same probability of being the target of control flow errors. Because of that assumption they can lead to a considerable increase both in memory and performance overhead during execution time. In this paper, we propose a static analysis that provide a more refined information about which basic blocks can be the target of control-flow-errors caused by single-bit flips. This information can then be used to guide a program transformation in which only susceptible blocks have to be protected. We implemented the static analysis and program transformation in the context of the LLVM framework and performed an extensive fault injection campaign. Our experiments show that this less conservative approach can potentially lead to gains both in memory usage and in execution time while keeping high fault coverage.**

Keywords*—*Fault tolerance, Reliability, availability, and serviceability.

I. Introduction

The rate of transient faults caused by radioactive particles striking circuits have increased recently. The reasons for this increase are the advances in the technology, leading to smaller and denser transistor, combined with lower voltage levels. And as embedded systems using this more susceptible technology are becoming widely disseminated, the demand for mitigation approaches for transient faults has become a concern for embedded systems developers in general.

However, the costs for developing and deploying highly effective hardware-based mitigation approaches, such as dual or triple modular redundancy, are prohibitive. Because of that, software-based solutions present themselves as viable alternatives for dealing with transient hardware faults in off-the-shelf processors.

The problem is that low-cost and effective software-based mitigation techniques impose, in general, considerable performance overheads, making such techniques usually inadequate for systems that require high performance.

Most of the control-flow error detection methods reported in the literature instrument programs at compiling time. Based on the information available in the program's control-flow graph (CFG), the compiler first assigns a signature to each basic block. The extra protecting code, consists basically in instructions that check the signatures of the basic blocks which are the origin and the destination of a control-flow. If this flow of control from one block to another does not exist in the program's CFG an illegal control-flow error is detected.

These *signature checking* techniques detect illegal inter-block control flow errors and are known to be conservative in the sense that, due to the strict fault model they assume, they add protecting instructions for *every* basic block.

In this paper we propose an effective software-based technique for the detection of certain control flow errors that can potentially led to improvements in performance when compared with similar techniques. With a more relaxed, but still realistic fault model, we show that not every basic block need to be instrumented with protecting code.

We achieve that by means of a systematic analysis of the impact of single bit-flips on the control-flow misbehavior of a given program that allows the identification of all *susceptible* and all *non-susceptible* basic-blocks of a program, i.e. the basic blocks which constitute the potential destinations of faulty transitions, and the basic block that can never be the target of illegal inter-block control flow error, respectively. We use the result of this analysis to instrument the targeted program only at the identified susceptible basic-blocks.

All current CFE detection methods with acceptable fault-coverage and performance overhead, however, update the signature incrementally as they introduce set assertions in all basic blocks. Therefore, omitting set assertions in non-susceptible basic-blocks will corrupt the signature. For this purpose, we propose a novel signature monitoring scheme for CFE detection allowing assertions removal in arbitrary basic-blocks. Our Flexible Control Flow Check (FCFC) has lower performance overhead and sufficient fault-coverage as compared to other proposed methods. We present a straightforward implementation of FCFC and of our bit-flip analysis in a technique we call *partial FCFC* and study its benefits.

We use the x86 processor in our experiments, however, the proposed approach can be easily applied to any other target, general purpose, HPC or embedded

The rest of the paper is organized as follows: next section discusses related work. Section III presents the fault model we assume. The detailed explanation of our bit-flip analysis and of FCFC is presented in Section IV. Section V demonstrates the experimental setup and discusses the results obtained. Finally we present the conclusions and future work.

II. Related Work

Several software-based methods for CFE detection exist: ECCA [1], CFCSS [5], YACCA [2], CEDA [7], ACFC [8], Abstract Control Signatures (ACS) [4] and SWIFT [15]. All the mentioned methods add extra protecting code to all basic blocks.

CFE detection methods can be divided into two main categories: path-asserting and predecessor/successor methods. A path-asserting method adds test in one basic block per control-flow path to assert correct path execution. Predecessor/Successor-asserting methods add tests in all basic blocks to check if the previous (or next) basic block in the execution flow is the correct predecessor (or successor).

The difference between the two categories is in the number of added instructions in the program. Predecessor/Successor methods add more tests, therefore they have higher overheads and also potentially higher fault-coverage. Path asserting methods add less test assertions, have lower overhead but decreased fault-coverage. CFCSS, ECCA, CEDA, YACCA and CCA represent predecessor/successor-asserting methods with high fault coverage and high overhead. ACFC and ACS add one test assertion per group of basic blocks and are categorized as path-based methods.

We divide predecessor/ successor methods into two subgroups: 1) methods with incremental signatures update; 2) methods with local signature updates. Incremental signature update methods require the use of a global signature as input. This means that the value of the global signature, at each basic block, depends on all set assertions in the predecessor basic-blocks along the execution path. Methods for incremental signature update are CEDA, CFCSS and YACCA. On the other hand, local signature updates set the signature at the current basic-block independent of global signature content. ECCA and CCA are examples of methods with local signature update.

Abstract Control Signatures (ACS) [4] is a path-based control flow error detection technique that divides the control flow graph into regions and add only one signature checking by region. This method is closer to ours in that it also is non-conservative and is based on the same fault model.

III. Fault model

The target fault model in our work is bit transitions which can happen due to events such as crosstalk or radiation. As investigated in [6], multiple bit transitions cause the same misbehavior as single bit flips. Therefore, in this work we consider only single bit transitions.

Control-flow errors may occur due to three main reasons: 1) single bit flip leading to branch creation; 2) single bit flip leading to branch deletion and 3) single bit flip leading to error in the address operand of branch instructions

Branch creation may occur if a non-branch instruction converts to a branch, and branch deletion may happen if a branch instruction converts to a non-branch instruction (invalid instructions are included in this category). However, the probability of transforming a non-branch opcode to a branch and vice versa due to a single bit transformation is extremely low and depends on the opcode coding of the instruction set architecture.

It is important to note that branch creation may also happen due to bit-flips in the program counter. However, the probability of control flow error occurrence due to single bit-flip in the program counter is relatively low as it is a small circuitry compared to the rest of processor components.

For all these reasons, in this work, we target only control flow errors caused by single bit-flips in address operands of branch instructions. This same fault model has also been adopted in other recent work on software-based approaches to control flow error detection [4].

Our fault injection experiments, performed following this fault model, will consist of the following: at each program execution, a branch instruction will be selected and a single bit of its address operand will be flipped. The bit to be flipped in randomly selected but the selection of the branch instruction is based on the probability of its execution. Hence, before starting a fault injection campaign we also collect some statistics about the probability of each branch instruction to be executed.

Control flow errors are called *illegal* when the flow of control is not present in the program's CFG. When an error in the control flow leads to a flow of control allowed by the CFG is it called *legal*. Illegal control flow errors can be (i) intra-block, when the source and target of the flow of execution are the same basic block, (ii) inter-block, when source and destination are different program basic blocks, or (iii) they can have as destination areas outside the address space of the program. When the target of an illegal control flow error is an area outside of the program address space usually a segmentation fault is detected by the operating system.

As it is the case with all signature checking mechanisms in the literature, ours aims at detecting only illegal inter-block control flow errors.

IV. Identifying and protecting susceptible basic blocks

Figure 1 shows the schematic view of the first stage of the process for identifying and protecting susceptible basic blocks. In the first step, (a), the source code is compiled to its LLVM intermediate representation (LLVM-IR) by the Clang compiler. In the next step, (b), the FCFC transformation is applied to the LLVM-IR using the infra-structure provided by LLVM for the implementation of program transformations. This step generates a LLVM-IR code with all basic blocks protected with the FCFC technique. In step (c), the LLVM-IR code is compiled to x86 assembly code by the x86 back end of the LLVM static compiler. In step (d) the assembly code is compiled to an executable in x86 architecture by the Clang compiler.

Fig. 1: Step 1

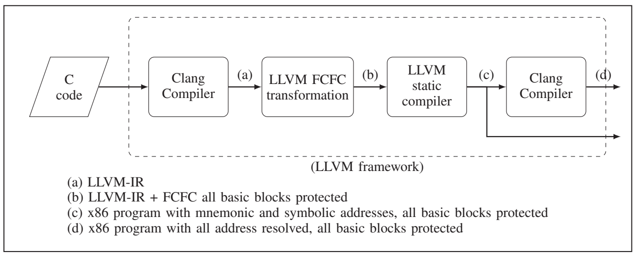

C code
Clang Compiler
(a)
LLVM FCFC transformation
(b)
LLVM static compiler
(c)
Clang Compiler
(d)
(LLVM framework)
(a) LLVM-IR
(b) LLVM-IR + FCFC all basic blocks protected
(c) x86 program with mnemonic and symbolic addresses, all basic blocks protected
(d) x86 program with all address resolved, all basic blocks protected

Figure 2 shows the schematic view of the second stage of the process. The inputs for this stage are the x86 assembly code and also the x86 executable, both corresponding to the same source program and both with all basic blocks protected.

In step (e), the binary-dump [1] of the x86 executable is generated. The binary-dump is a text file that contains all program instructions with the memory addresses of each one of them.

After, in step (f), the systematic bit-flip analysis is performed having as inputs both the binary dump and the x86 assembly code (the details of this analysis are given in subsection B). The result of bit-flip analysis is a list with of all susceptible basic blocks (SBL).

Fig. 2: Step 2

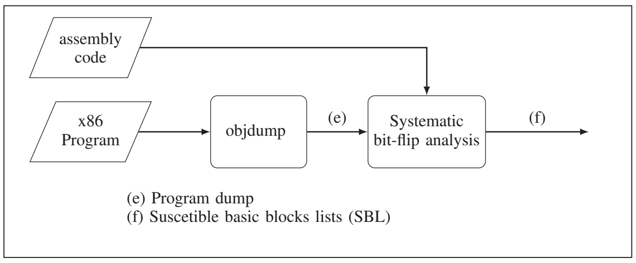
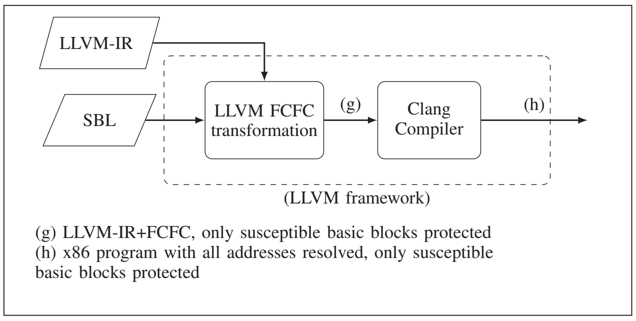

assembly code
x86 Program
objdump
(e)
Systematic bit-flip analysis
(f)
(e) Program dump
(f) Suscetible basic blocks lists (SBL)

Figure 3 shows the third and final stage of the process. The inputs for this stage are the list of susceptible basic blocks and the LLVM-IR program. The LLVM-IR program is then transformed into another LLVM-IR program where only the susceptible basic blocks are protected (step (g)). In step (h), the LLVM-IR code is compiled to x86 by Clang, generating a *non-conservative version* of the program, that is, a new program where only the susceptible basic blocks are protected by FCFC.

[1]This work is done by Linux *objdump* command

The FCFC technique was chosen because it has the necessary properties for working in conjunction with the systematic bit flip analysis: a) it performs *local signature update*, i.e the update of *runtime signature* does not depend on the signature of all basic blocks along the control flow graph, and b) it does not introduce new basic blocks to the program's original control flow graph.

Fig. 3: Step 3


LLVM-IR
SBL
LLVM FCFC transformation
(g)
Clang Compiler
(h)
(LLVM framework)
(g) LLVM-IR+FCFC, only susceptible basic blocks protected
(h) x86 program with all addresses resolved, only susceptible basic blocks protected

In the next subsection we give a precise definition of *suceptible basic block*. In the sequence we provide details about the systematic bit flip analysis and then we explain how the non-conservative protection technique is performed as a program transformation step.

A. Susceptible basic blocks

Figure 4 shows a simple control flow graph with four basic blocks. In this control flow we may notice that there are four legal branches, from B1 to B2 and B3, from B2 to B4 and from B3 to B4.

If a control flow error occurs due to a single bit flip in the address operand of a branch instruction we have the following possibilities: we may have a legal (w.r.t the program's CFG) but wrong flow of control, or we may have an illegal flow of control. An illegal control flow can be *intra-block*, *inter-block* or it can have as target an area outside the program's address space.

An illegal *intra-block* branch may happen if a CFE has as source and target the same basic block. This kind of illegal branch is not in the scope of control flow error detection techniques, thus it is also not considered in this work.

An illegal *inter-block* branch may happen when a control flow error creates a branch between different basic blocks that does not exist in the original CFG, for example, between basic blocks B1 and B4.

Susceptible basic blocks are those that may be the target of illegal inter-block control flow errors. If we identify all the susceptible basic blocks we can reduce the overhead added by CFE detection technique by protecting only susceptible basic blocks and by avoiding to protect non-susceptible basic blocks.

B. Systematic bit-flip analysis

Assuming as our fault model single bit flips in the address operands of branch instructions it is possible to generate all

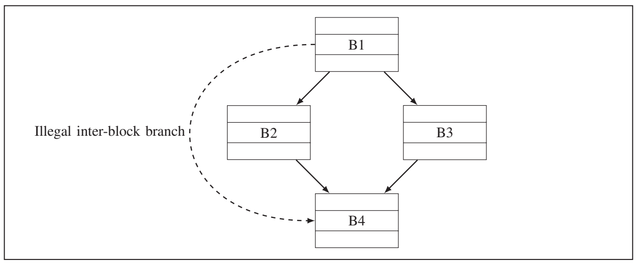


Fig. 4: Susceptible basic blocks

potential destination addresses produced by transient faults. It is important to observe that the proposed scheme considers that all branch instructions have their target addresses calculated statically, so all the addresses are know before the program execution.

To perform the systematic bit-flip analysis the proposed scheme takes the assembly code and the binary-dump of the program as input. From the binary-dump we can obtain all memory addresses used by the basic blocks of the program, all branch instructions, and all original memory addresses which are targets of each branch instruction.

All single bit flips are produced and each single bit flip produces a new address. Considering, for instance, the branch instruction ***jmpq 402475***, and considering 24 bits addressing length (x86), Table I show some possible values that the branch operand can take if a single bit-flips affects it.

Original	Single bit-flip	Result
jne,402475	0000 0000 0000 0000 0000 0001	jne, 402474
jne,402475	0000 0000 0000 0000 0000 0010	jne, 402477
jne,402475	0000 0000 0000 0000 0000 0100	jne, 402471
jne,402475	0000 0000 0000 0000 0000 1000	jne, 40247D
jne,402475	0000 0000 0000 0000 0001 0000	jne, 402465
jne,402475	0000 0000 0000 0000 0010 0000	jne, 402455
...	...	...
...	...	...
jne,402475	0000 0100 0000 0000 0000 0000	jne, 442475
jne,402475	0000 1000 0000 0000 0000 0000	jne, 482475
jne,402475	0001 0000 0000 0000 0000 0000	jne, 502475
jne,402475	0010 0000 0000 0000 0000 0000	jne, 602475
jne,402475	0100 0000 0000 0000 0000 0000	jne, 2475
jne,402475	1000 0000 0000 0000 0000 0000	jne, C02475

TABLE I: Systematic bit-flip analysis

Having produced all possible addresses resulting from single bit flips in all operands of all branch instructions of the program, the identification of susceptible basic blocks can be initiated.

First, all invalid addresses are discarded. Invalid addresses are addresses to areas outside the program's address space. Addresses that can lead to a control flow allowed according to the program's CFG are also discarded since they correspond to legal branches, i.e, branches allowed according to the CFG. From this set we still remove those addresses pointing to the same block of the branch instruction that had bit flips affecting its address operand (these correspond to illegal intra-blocks control flow errors).

After removing addresses corresponding to invalid and to legal branches, and also removing addresses corresponding to illegal but intra-block branches, we are left only with addresses corresponding to illegal inter-block branches

By identifying which basic blocks occupy these addresses we are identifying the list of all susceptible basic blocks of the program (the list SBL of Figure 2).

C. Protecting susceptible basic blocks

Notice that the analysis of which basic blocks need to be protected was performed having as one of its inputs the x86 executable with all memory addresses resolved. In order to avoid the misalignment problem, instead of simply removing protecting instructions of nonsusceptible basic blocks, the memory address where the protecting instructions were positioned are replaced by *NOP* instructions. With this approach it is possible to remove instructions for detecting CFEs in non-susceptible basic blocks while maintaining the same alignment of the original version in the modified version of the program.

Because x86 is a CISC architecture different instructions have different lengths. Because of that, depending on the type of instruction, it might have to be replace by several NOP instructions. For example, a *movq* instruction takes 12 memory addresses, so it needs to be replaced by 12 *NOPs*, because each *NOP* takes only one memory address.

As a final optimization the new NOP instructions inserted at the beginning of basic blocks are moved and positioned after the branch instruction at the end of the block. By doing so, the alignment is still preserved, but the chances that these NOPs instruction are executed are reduced (if they are left at the entrance of basic blocks they will always be executed when the flow of control reaches the block).

If these NOP instructions are positioned after an unconditional branch they will never be executed. If they are placed after a conditional branch instruction they may or may not be executed. It will depend on the outcome of the condition of the branch instruction.

V. Experiments

To investigate the coverage and performance a set of experiments using a subset of representative workload from Mibench [3] embedded benchmark suite was performed. All experiments were dome on a machine with one AMD Turion(tm) II P520 Dual-Core Processor and 4GB memory, using the Operational System Ubuntu GNU/Linux 12.04 LTS x86-64.

In order to simulate CFEs in the binaries at runtime and to evaluate the coverage in fault detection, we implemented a GDB[2]-based fault-injector similar to the one used for previous CFE detection methods evaluation [7]. In these fault injectors the instruction to be affected by the bit flip is selected statically and, at runtime, a single bit flip is injected in the chosen instruction according the fault model presented in Section III.

[2]The GNU debugger

A. Detection coverage

In the first set of experiments, an executable version of each benchmark with all basics blocks protected with FCFC (conservative version) was produced. For each benchmark we injected a thousand faults using the proposed fault injector and model.

Table II shows the result of this set of experiments. The second column (S) represents the faults that resulted in *Segmentation Fault* and were detected by the Operating System. The third column (I) represents the faults that resulted in *Illegal Instructions*, also detected by the Operating System, as well as the faults of the forth columns (OS), which represents other kinds of faults less frequent: Sigkill, Sigtrap, Sigfpe, Sigbus. The fifth column (T) represents the faults that lead the fault injector to timeout and could not be evaluated. The sixth column (EC) represents the faults that were not detected by the Operating System neither by FCFC and resulted in **correct outputs**. The seventh columns (EW) represents the faults that were not detected by the Operating System neither by FCFC and resulted in **wrong outputs**. Finally, the last column (D) represents the faults detected by FCFC.

Benchmark	S	I	OS	T	EC	EW	D
basicmath	70,5%	14,6%	1,6%	0,0%	2,0%	3,0%	8,3%
CRC	76,7%	9,3%	0,0%	0,0%	0,0%	2,2%	11,8%
dijkstra	79,2%	7,2%	2,2%	0,0%	4,0%	0,0%	7,4%
FFT	78,8%	7,2%	0,1%	0,0%	1,3%	5,7%	6,9%
patricia	73,8%	9,3%	0,1%	0,1%	6,4%	1,0%	9,3%
pbmsrch	76,3%	8,6%	0,1%	0,0%	4,2%	0,2%	10,6%
qsort	74,5%	9,4%	0,0%	0,0%	3,7%	2,7%	9,7%
rijndael	74,0%	9,1%	1,4%	0,0%	4,2%	2,8%	8,5%

TABLE II: FCFC: conservative version

In the second set of experiments we first generated the list of susceptible basic blocs for each benchmark. Table III shows the total number of basic blocks of each benchmark, the number of susceptible basic blocks, and the percentage of non-susceptible basic blocks.

Benchmark	Basic blocks		
	Total	Susceptible	% not susceptible
basicmath	63	62	1,59%
CRC	20	16	20,00%
dijkstra	60	56	6,67%
FFT	93	88	5,38%
patricia	179	136	24,02%
pbmsrch	34	34	0,0%
qsort	19	17	10,53%
rijndael	186	172	7,53%

TABLE III: Susceptible basic blocks

With the list of susceptible basic blocks, an executable version for each benchmark, with only susceptible basic blocks protected, was generated. Again, for each benchmark, one thousand faults were injected using the proposed fault injector and model. The faults injected were exactly the same that were injected in the previous experiment. We did that in order to better compare the coverage results obtained with both experiments.

In the third set of experiments, the versions generated in step two had their branches instructions changed and a new

Benchmark	S	I	OS	T	EC	EW	D
basicmath	70,5%	14,5%	1,6%	0,0%	2,2%	3,0%	8,2%
CRC	76,7%	9,3%	0,0%	0,0%	0,0%	2,2%	11,8%
dijkstra	79,2%	7,2%	2,2%	0,0%	4,2%	0,0%	7,2%
FFT	78,8%	7,2%	0,1%	0,0%	1,3%	5,7%	6,9%
patricia	73,8%	9,3%	0,1%	0,1%	6,4%	0,9%	9,4%
pbmsrch	76,3%	8,6%	0,1%	0,0%	4,2%	0,2%	10,6%
qsort	74,5%	9,4%	0,0%	0,0%	3,7%	2,7%	9,7%
rijndael	74,0%	9,1%	0,0%	1,4%	4,2%	2,8%	8,5%

TABLE IV: FCFC: Non-conservative version

version of each benchmark was generated (NOPs inverted). For each benchmark one thousand of faults were injected using the proposed fault injector and model and again using the same faults used to simulate the errors in binaries of Table II and Table IV. Table V shows the result coverage of this set of experiments.

Benchmark	S	I	OS	T	EC	EW	D
basicmath	70,5%	14,6%	1,6%	0,0%	2,1%	3,0%	8,2%
crc	76,7%	9,3%	0,0%	0,0%	0,0%	2,2%	11,8%
dijkstra	79,2%	7,2%	2,2%	0,0%	4,2%	0,0%	7,2%
FFT	79,1%	6,9%	0,1%	0,0%	1,3%	5,7%	6,9%
patricia	73,8%	9,3%	0,1%	0,1%	6,4%	0,9%	9,4%
pbmsrch	76,3%	8,6%	0,1%	0,0%	4,2%	0,2%	10,6%
qsort	74,5%	9,4%	0,0%	0,0%	3,7%	2,7%	9,7%
rijndael	74,0%	9,1%	1,4%	0,0%	4,2%	2,8%	8,5%

TABLE V: FCFC: NOPs inverted

As we can see in the comparative chart of Figure 5, the data converge between the FCFC conservative, FCFC non-conservative and FCFC with *NOPs* inverted are almost the same.

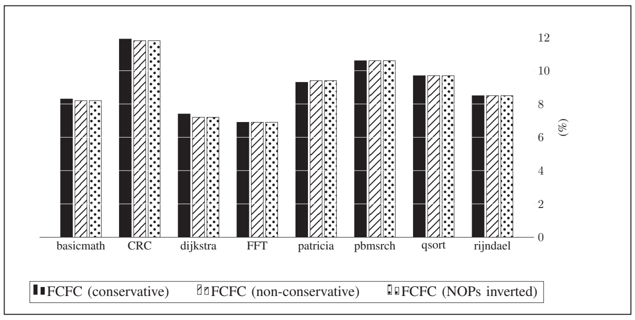


Fig. 5: Comparing FCFC with all basic blocks protected and with only basic blocks protected and branchs changed

B. Performance

To evaluate the performance gain between the different versions of FCFC we have decided not to count the time that each benchmark took to execute. Instead, we decided to count the number of instructions that each version of each benchmark has executed. With the program tracer it is possible to count the number of instructions executed eliminating any time inaccuracy caused by the underline operating system.

Table VI shows the number of instructions executed by each benchmark in FCFC conservative version. Table VII shows the number of instructions executed by each benchmark in FCFC non-conservative version.

Benchmark	Instruction Executed
basicmath	7,422,666
CRC	54,754,761
dijkstra	192,411,876
FFT	7,661,798
patricia	16,667,614
pbmsrch	542,441
qsort	6,854,311
rijndael	47,002,968

TABLE VI: Performance: FCFC conservative

Benchmark	Instruction executed	NOPs executed
basicmath	7,497,816	87,174
CRC	54,754,798	43
dijkstra	304,335,026	130,072,850
FFT	7,661,798	0
patricia	16,758,042	105,092
pbmsrch	542,441	0
qsort	11,310,369	5,178,662
rijndael	48,445,256	1,676,173

TABLE VII: Performance: FCFC non-conservative

Table VIII shows the number of instructions executed by each benchmark in FCFC with *NOPs* inverted. Comparing with Table VII, we may notice a gain in terms of *NOPs* instructions not executed. The gain ranged from 0,006420% (basicmath) to 72,091302% (rijndael). In terms of total number of instructions the gain ranged from 0,000135% (CRC) to 34,447616% (qsort)

Benchmark	Instruction executed	NOPs executed
basicmath	7,497,753	87,111
CRC	54,754,755	0
dijkstra	303,406,576	129,144,400
FFT	7,661,798	0
patricia	16,744,092	91,142
pbmsrch	542,441	0
qsort	8,763,264	2,631,557
rijndael	47,236,876	467,793

TABLE VIII: Performance: FCFC NOPs inverted

As we can see the number of instructions between the conservative and non-conservative version have increased. It happened because in our CISC environment one instruction needs to be replaced by many *NOPs*, for example, a *movq* instruction takes 12 memory addresses, so it needs to be replaced by 12 *NOPs*. In a RISC environment, like MIPS, SPARC, PowerPC or ARM, where all instructions have the same length the number of instructions executed would not increase. Moreover, instead of executing a complex instructions (mov, xor, etc.) we would execute a simple and faster *NOP* instruction. As consequence the time and power consumption of the program would be reduced.

If the experiments were ran in a RISC environment the number of NOPs would be much smaller, as show in Table IX. This table shows a simulation of replacing instructions by NOPs in a RISC environment, where each instruction would be replaced by a single NOP. The second column shows the total number of instruction executed by the conservative version. The third column show the number of NOPs executed by the non-conservative version and the fourth column show the number of NOPs executed in the version with NOPs inverted. As it is possible to notice, the number of NOPs executed is smaller than those shown in the tables VII and VIII.

Benchmark	Instructions Executed Conservative	NOPs executed Non-Conservative	NOPs executed Change NOPs
basicmath	7,422,666	12,024	12,014
CRC	54,754,761	6	0
dijkstra	192,411,876	18,149,700	17,999,950
FFT	7,661,798	0	0
patricia	16,667,614	14,664	12,414
pbmsrch	542,441	0	0
qsort	6,854,311	722,604	426,429
rijndael	47,002,968	233,885	38,985

TABLE IX: Prevision of FCFC in a RISC environment

VI. Conclusion

In this paper, we presented a non-conservative software-based method for detecting certain control-flow errors caused by transient faults. The method is non-conservative in the sense that not all basic blocks need to be protected. Only susceptible basic blocks have to be instrumented with protecting code.

In terms of fault coverage, the technique proved quite successful since it kept the same coverage rate observed in the version of programs with all basic blocks protected.

In terms of performance improvement, we argued that it is possible to obtain considerable gains if we target a RISC architecture instead of a CISC such as x86 used in our experiments.

References

[1] Z. Alkhalifa, V. S. S. Nair, N. Krishnamurthy, and J. A. Abraham. Design and evaluation of system-level checks for on-line control flow error detection. *IEEE Transactions on Parallel and Distributed Systems*, 10(6), June 1999.

[2] O. Goloubeva, M. Rebaudengo, M. Reorda, and M. Violante. Soft-error detection using control flow assertions. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 581–588, 2003.

[3] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE International Workshop on Workload Characterization*, pages 3–14, 2001.

[4] D. S. Khudia and S. Mahlke. Low cost control flow protection using abstract control signatures. In *ACM Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES'13, pages 3–12, 2013.

[5] N. Oh, P. Shirvani, and E. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51(1):63–75, March 2002.

[6] B. Sangchoolie, F. Ayatolahi, R. Johansson, and J. Karlsson. A study of the impact of bit-flip errors on programs compiled with different optimization levels. In *European Dependable Computing Conference*, pages 146–157, 2014.

[7] R. Vemu and J. Abraham. Ceda: Control-flow error detection using assertions. *IEEE Transactions on Computers*, 60(9):1233–1245, Sept 2011.

[8] R. Venkatasubramanian, J. Hayes, and B. Murray. Low-cost on-line fault detection using control flow assertions. In *On-Line Testing Symposium, IOLTS*, pages 137–143, 2003.