

Configurable Tool to Protect Processors against SEE by Software-based Detection Techniques

Eduardo Chielle, Raul Sérgio Barth, Ângelo Cardoso Lapolli, Fernanda Lima Kastensmidt
Instituto de Informática – PPGC - UFRGS
Porto Alegre, Brazil
{echielle, rsbarth, aclapolli, fglima}@inf.ufrgs.br

Abstract—This paper presents a tool capable of automatically adding fault detection capabilities in software to protect the processors against transient faults. The tool implements a set of configurable software-based detection techniques over the assembly code of an unprotected program. The developed tool has been validated for two distinct processors: MIPS and LEON3. But it can be extended to other architectures and organizations by changing the configuration files. A fault injection campaign was performed and simulation results show high detection rates to both processors and a small increase in area and runtime.

Keywords- fault tolerance, soft errors, SEU, SET, software techniques

I. INTRODUCTION

Integrated circuits fabricated with nanometric dimension transistors, operating in high frequency and low voltage supply, are more sensitive to transient faults, known as Single Event Effect (SEE) [1]. SEE is produced by highly energetic particle hits on sensitive circuit regions. When SEE affects a memory element, it is called Single Event Upset (SEU) and it is characterized as a bit-flip in the flip-flop. When this transient effect affects a logic gate of a combinational block, it is called Single Event Transient (SET) and can be perceived as a glitch with variable duration corresponding to the collected charge. SEE is common in circuits operating in radiation environment.

In processors, the effect of SEU and SET faults can be noticed by errors in the program data or errors that change the program flow. In order to protect a processor against SEE faults, the use of fault tolerance techniques capable of detecting and/or correcting these effects is needed. Among the existing fault tolerance techniques, there are hardware-based techniques and software-based techniques. In the hardware-based techniques, the physical system is modified. The circuit is usually duplicate or triplicate and checkers or voters are inserted [2]. Besides, it is possible to use error-correcting code (EDAC) in the memory elements too. According to implemented hardware-based techniques, the system can present variant overheads like reduction in the operating frequency, increase in area and power consumption, besides having a high design and manufacture costs [3].

The software-based techniques do not modify the hardware, allowing to use COTS (Commercial off-the-shelf) processors, which are substantially cheaper than radiation hardened (rad-hard) processors. Only the source code of the program is modified. Techniques which duplicate the variables, like EDDI – Error Detection by Duplicated Instructions [4], or techniques

to protect the control flow, like CFCSS – Control-Flow Checking by Software Signatures [5] or techniques which apply transformation rules over the code [6] are used by modifying the program's source code. These modifications make the software fault tolerant. However, these software-based techniques increase the runtime and the memory occupation of the program. When an error is detected, the processor and the program are restarted or a certain part of the code is recomputed.

The modification of the software-based techniques can be usually done in three different language levels: at high-level, assembly level and machine level languages. One can say that the best choice is applying the techniques at high-level language because it is independent to target processor architecture and organization. However, the compiler optimizes the code, removing the redundancies. It is possible to modify the compiler to do not remove these redundancies, but it makes the tool dependent of the target architecture. The remaining possibilities are the application of the techniques over the assembly or machine code. However, both are linked with the architecture. Consequently, the target architecture must be described to the tool, making the tool, in this way, capable of recognizing the program instructions and applying the selected techniques.

This paper presents a tool capable of protecting processors with diversified architectures against SEE by software-based techniques. The developed tool is configurable and capable of applying a set of software-based fault tolerance techniques that can be specified by the user according with the detection rate and costs in area and runtime. The tool is architecture independent because the information of the target processor architecture and organization is informed in a configuration file. The user can select as well the set of fault tolerant techniques to be implemented.

The paper is organized as follows. Section II presents an overview about the software-based detection techniques and the tools that apply automatically the techniques present in the literature. Section III describes the proposed tool and how it works. A fault injection campaign is presented in section IV. The achieved detection rate of the techniques implemented by the proposed tool to a MIPS processor [7] are compared to the results by using the same techniques applied over the machine code. Results in a LEON3 processor [8] are also presented. In section V, main conclusions and future works are presented.

II. RELATED WORKS

A. Software-based Techniques

There are four main areas in the literature about software-based techniques to protect processors. They are data protection techniques, control-flow protection techniques, hybrids techniques, which aim to protect both data and control-flow, and techniques to protect the memory. The first three are part of this work because they are non intrusive and they can be applied in many abstraction levels.

The data protection techniques aim at protecting against faults that affect the data stored in the memory or in the registers. These techniques duplicate the variables, creating copies and making, over the copy variable, the same operations that are performed over the original variable. Furthermore, instructions to check the program integrity are also inserted in some points of the code. These points depend on the selected technique.

This kind of technique is not designed to detect faults that affect the control-flow, but it is able to detect some errors caused by these faults, because they can make the original variable and the copy have different values, which can be detected by a checker. An example of these techniques is EDDI [4], VAR1, VAR2 and VAR3 [9].

The control-flow protection techniques aim at protecting program flow. These techniques divide the code in basic blocks, which are portions of code without branch or jump instructions, being attributed a signature to each basic block. There are three kinds of faults which affect the control-flow: faults changing the program flow from a basic block to another one; faults changing the flow to the beginning of another basic block; and faults changing the program flow to another position of the same basic block. This last type of error is not detected by the current techniques existing in the literature. Examples of these techniques are CCA [10], ECCA [11], CFCSS [5], SIG and BRA [12].

The hybrid techniques are, in general, transformation rules that are applied to the program code and aim at protecting both data and control-flow. An example of this technique is the SWIFT technique [13], which is a join of the data protection technique EDDI and control-flow protection technique CFCSS with some optimizations.

B. Existing Tools

A program described in a high level language goes through several compilation stages until it is transformed into a machine code level. Fig. 1 shows these stages. First, the code is compiled, being optimized in this process, going from a high-level language description to an assembly description. So, the assembler reads the assembly code and generates the machine code.

The application of the software-based techniques can be usually done in three different language levels. The code can be modified in a high level language, like C. In this case, the process is independent of the processor organization and architecture. However, the source code in high level language is subject to optimizations by the compiler, which can remove the redundancies generated by the techniques, making the

protection ineffective. Another possibility is modifying the compiler so as not to remove the redundancies created by the technique. In this case, each compiler targeting each processor in particular must be modified, which is time consuming and it can generate errors that must be verified. The techniques also can be applied over the assembly code, thus, not being affected by the optimizations of the compiler. In this case, it is not necessary to modify the compiler or assembler. Furthermore, the program can be protected in the machine code. The disadvantage in this case is that it is highly connected with the architecture.

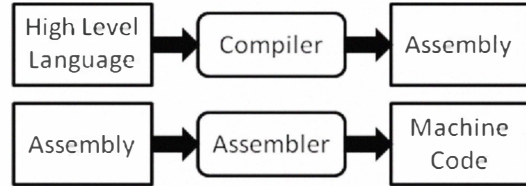


Figure 1. Program's abstraction levels

C2C Translator [14] is a tool that modifies the program's source code in a high level language, C language, before the stages of compiling and assembling. Applying the technique in C makes the tool independent of architecture, however, the compiler optimizes the code, removing the redundancies, which can make the technique do not work.

In reference [15], the compiler OpenIMPACT is modified to apply the SWIFT technique to the Intel Itanium 2 processor. The compiler was modified to not remove the redundancies created by the technique, making it efficient. However, although it gets the benefit of the generality of a high level language, this method is not generic, because for each target processor it is necessary to modify a compiler to that architecture.

The HPCT Suite [9] is a tool that modifies the program's machine code for a MIPS processor. In this level the tool is linked to the processor architecture, and it shows a high efficiency because it modifies the final program, making the technique executed by the processor exactly as it was applied. However, in this level, many calculations are required to correct the addresses of the branches and jump instructions. Furthermore, the tool is not independent of the processor architecture.

III. THE PROPOSED TOOL: CFT

In order to reduce the complexity of modifying the code at the machine level, a solution based on assembly code modifications was developed. The CFT-tool applies over the assembly code, in an automated way, the software-based detection techniques validated by reference [9]. There are some advantages on employing the fault tolerance at assembly level instead of at machine level. For example, in the machine code, the target addresses of the branches and jumps must be corrected. In assembly level, this does not need to be done because the target of branches and jumps are labels and its physical address is defined after the assembling stage. Besides, the branch delay slot consists in reordering the branches and

jumps instructions to gain performance and it is another concern to apply the techniques over the machine code. Furthermore, applying the techniques over the assembly code makes the CFT-tool more portable among different architectures, because the assembler of the target processor takes charge of generating the executable file. For these reasons, the choice was to work over the assembly code, where an existing compiler and assembler can be used without needing any modification.

Fig. 2 presents the stages in which the program goes through until it is protected by the techniques. The code in a high level language is compiled, generating its equivalent assembly code. The CFT-tool reads the configuration about the processor architecture and information about the techniques to be applied over the program code by the SEE Protection Module. The users inform the configuration files containing the processor organization and architecture and the selected software-based techniques. After that, the Verification Module reads the program's assembly code and verifies if the source code matches with the processor architecture informed. Then, the Filler Module generates a new assembly code. Finally, the SEE Protection Module applies the software-based detection techniques selected by the user over the assembly code, generated by the Filler Module, creating a protected assembly code.

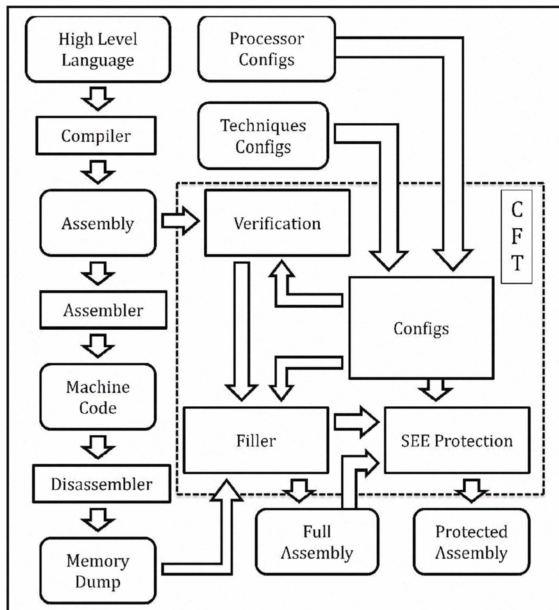


Figure 2. CFT-tool schematics and program stages

A. Configuration Module

As the tool must be independent of the processor architecture, it needs to understand the target processor architecture in some way. For this reason, the processor architecture must be described to the tool by informing features of the processor. This description is made in configuration files, where all required information about the processor is

informed. For example, data about the instructions; which type they are: arithmetic, branches, jumps, loads, stores. Also the format of the instruction must be informed; their mnemonics; which are the source registers and the destiny register; identify the immediate values; and the position of each one of these in the instruction. Data about the registers must be informed, such as name, if it is a special register, if the register is writable, as well others relevant characteristics about the target processor organization.

The tool also needs to know the techniques that must be applied over the code of the program to be protected and which registers must be protected, choosing their priority to get a copy register.

Therefore, the Configuration Module's function is to identify the architecture patterns, the selected techniques and provide the required information to other modules, independently of the target architecture.

B. Verification Module

The function of the Verification Module is to read the program's assembly code generated by the compiler and verifies if the code matches with the information of the specified architecture. The information about the target processor architecture is informed by the Configuration Module. If the program's assembly code matches with the architecture specification, the tool goes to the next stage, the Filler Module, otherwise, an error is reported.

C. Filler Module

In the compiling stage, there are subroutines generated by the compiler. Some of these subroutines are known by the assembler and, for this reason, they are not described in the program's assembly code, but only calls them. So, for the entire program be protected, these subroutines must be in the assembly code. Thereunto, a solution is converting these subroutines from the machine code to assembly code, using memory dump files.

This module aims at verifying if there is any of this kind of subroutine and converts its code to assembly. Firstly, the module search in the assembly code for calls to subroutines and verifies if they are present in the code. If not, the code must be converted. Fig. 3 shows an example of a subroutine called in someplace of the program, but which does not have its code implemented in assembly. The *jal* instruction calls for the *memcpy* subroutine that is not present in the source code.

```
...
main:
...
la $3,$LC1
move $4,$2
move $5,$3
li $6,36
jal memcpy
sw $0,96($fp)
sw $0,100($fp)
...
$L2:
...
$L3:
...
j $31
.end main
```

Figure 3. Call to a subroutine not implemented in the assembly code

<pre> 00000300 <memcpy>: 300: 00a41025 or v0,a1,a0 304: 30420003 andi v0,v0,0x3 308: 14400026 bnez v0,3a4 <memcpy+0xa4> 30c: 00805021 move t2,a0 310: 00064102 srl t0,a2,0x4 314: 00a04821 move t1,a1 318: 30c6000f andi a2,a2,0xf 31c: 1100000d beqz t0,354 <memcpy+0x54> 320: 00803821 move a3,a0 324: 8d220000 lw v0,0(t1) 328: 8d230004 lw v1,4(t1) 32c: 8d240008 lw a0,8(t1) 330: 8d25000c lw a1,12(t1) 334: 2508ffff addiu t0,t0,-1 338: ace20000 sw v0,0(a3) 33c: ace30004 sw v1,4(a3) 340: ace40008 sw a0,8(a3) 344: ace5000c sw a1,12(a3) 348: 25290010 addiu t1,t1,16 34c: 1500ffff bnez t0,324 <memcpy+0x24> 350: 24e70010 addiu a3,a3,16 354: 00064082 srl t0,a2,0x2 358: 11000007 beqz t0,378 <memcpy+0x78> 35c: 30c60003 andi a2,a2,0x3 360: 8d220000 lw v0,0(t1) 364: 2508ffff addiu t0,t0,-1 </pre>	<pre> memcpy: or \$2,\$5,\$4 andi \$2,\$2,0x3 move \$10,\$4 bnez \$2,memcpy1 srl \$8,\$6,0x4 move \$9,\$5 andi \$6,\$6,0xf move \$7,\$4 beqz \$8,memcpy2 memcpy3: lw \$2,0(\$9) lw \$3,4(\$9) lw \$4,8(\$9) lw \$5,12(\$9) addiu \$8,\$8,-1 sw \$2,0(\$7) sw \$3,4(\$7) sw \$4,8(\$7) sw \$5,12(\$7) addiu \$9,\$9,16 addiu \$7,\$7,16 bnez \$8,memcpy3 memcpy2: srl \$8,\$6,0x2 andi \$6,\$6,0x3 beqz \$8,memcpy4 memcpy5: lw \$2,0(\$9) addiu \$8,\$8,-1 </pre>
---	--

Figure 4. Example of a subroutine extrated from a memory dump file (left) and equivalent assembly code generated (right)

After finding the subroutines that are not implemented in the assembly code, the Filler Module searches the subroutines in the memory dump file and creates their equivalent code in assembly and inserts subroutines codes generated in the end of the program's assembly code. To do that, the Filler Module must have information about the architecture, like the specification of the instructions both in assembly and the memory dump format, existence of branch delay slot and the way how the data are organized.

Fig. 4 shows an example of a code in a memory dump file, on the left, and the equivalent assembly code generated by this module, on the right. They are similar but with some important differences. The registers names are not the same. The destination address of the jumps and branches in the memory dump file are physical addresses and in assembly are labels. In some architectures, the branches and jumps are shifted with next instruction to gain performance. For the architecture used as example, these shifts must be undone to keep the program working correctly, but making sure the instructions respect their code block, defined by the labels. This means that no instruction located after a label can be moved to before it and neither an instruction located before a label can be moved to after it. After that, if the architecture admits optimizations, the instructions that are not going to be executed are removed from the code, because the assembler will insert those again if required.

D. SEE Protection Module

The SEE Protection Module is responsible for applying the software-based detection techniques in the program that will be protected. When a detection technique is selected to be applied in the program, the Protection Module performs the transformation in the assembly code, creating a new protected assembly code. The techniques are in the order that the user has informed in the configuration files. All required information is informed in the Configuration Module.

The available techniques are: the ones protecting the data that are called Variables 1 (VAR1), Variables 2 (VAR2) and Variables 3 (VAR3) and the ones to protect branches and control flow effects that are called Branches (BRA) and

Signatures (SIG). The VAR1 technique is based on the transformation rules [15] to protect the data, which say every variable must be duplicated, where every operation over the original variable must be performed over the copy. Instructions to compare the value of the original variables with their copies must be inserted before every instruction that reads the variable. VAR2 is an alternative to the reference [15] data protection technique, based on reference [14] proposal. The only difference between the techniques is the location where the checkers are inserted. In the VAR2 technique, instead of inserting the instructions to check the original and copy variables before it is read, they are inserted after a writing over the variable. The other implemented technique to protect the data is the VAR3 technique. VAR3 is based on the SWIFT technique [13], which also duplicates the variables. In this technique, the checkers are inserted before the variable is read by load, store and branch instructions.

To protect the program flow, the technique SIG was implemented. This technique, like the most of techniques to detect errors affecting the control-flow, divides the source code in basic blocks and attributes signatures to each one. The basic block's signature is attributed to a variable in the beginning of the execution of the basic block. The value is verified at the end of the execution of the basic block to verify if a jump to another basic block did not occur. Besides this technique, there is another called BRA, based on a rule proposed by reference [15] to protect against errors affecting the branch instructions. In this technique, the branches are replicated. Right after where the branch instruction, another branch logically equal to the original, but with the destination addressed to the error treatment subroutine, is inserted. Furthermore, in the branch destination is inserted another branch, but it is logically inverse to the original and the target address is addressed to the error treatment subroutine. Labels and jumps must be inserted to keep the correctness of the program.

IV. VALIDATION RESULTS

A fault injection campaign was performed to validate the CFT-tool. Faults were injected at logical level in MIPS and LEON3 processors, using Modelsim [16]. The location and

time for each injected faults are randomly selected. Only one fault was injected at each execution of the program. Two programs were protected by CFT-tool: a matrix multiplication and a bubble sort. The matrix multiplication has a lot of data processing and a few loops, being ideal to verify the coverage of the techniques in detecting errors affecting data. On the other hand, the bubble sort does not have so much data processing and has a large number of loops, branches and control registers, which brings the faults to affect more the control flow than the matrix multiplication. The programs were protected with four different combinations of the techniques. The first three versions were protected only with data protection technique, VAR1, VAR2 and VAR3, and the last one was a combination of the data protection technique VAR3 with two control-flow protection techniques, BRA and SIG.

The time consumed by the CFT-tool to protect the applications is negligible. The time depends on the code size and the selected techniques. CFT takes less than 5s to protect the case study codes for all techniques and both processors.

Aiming at verifying the efficiency of the software-based detection techniques, only the faults that caused errors were taken into account. The faults masked by the logic of the processor were ignored.

After applying the techniques over the programs, a fault injection campaign was performed where 10,000 faults were injected in each protected program. Table 1 shows the detection rate to the matrix multiplication protected with the different combinations of the techniques to MIPS processor. The programs were protected by CFT-tool and HPCT Suite [9], which was adopted as a basis to comparison as a way to validate the correct application of the techniques by the CFT-tool.

Variables techniques increase the runtime, but also have good error coverage. VAR3 shows the best cost-benefit. The combined techniques presents a small increase in the detection rate with a small increase in the runtime compared to the ones only with variables techniques. The detection rates of the techniques applied over the assembly code by the CFT-tool are quite similar with the results obtained to machine code. They are within the margin of error.

TABLE I. MATRIX MULTIPLICATION (MIPS)

Technique	Exec. Time (μ s)	Memory Occupation (bytes)	Number of Errors	Detection Rate (%)
Unprotected	162	1308	-	-
VAR1 ¹	398 (2.46x)	3220 (2.46x)	2619	73.3
VAR1 ²	332 (2.05x)	3296 (2.52x)	2733	74.7
VAR2 ¹	421 (2.60x)	3452 (2.64x)	2543	71.6
VAR2 ²	365 (2.25x)	3260 (2.49x)	2548	72.5
VAR3 ¹	265 (1.64x)	2524 (1.93x)	2502	69.9
VAR3 ²	290 (1.79x)	2808 (2.15x)	2291	68.0
SIG+BRA+VAR3 ¹	300 (1.85x)	3432 (2.62x)	2708	70.8
SIG+BRA+VAR3 ²	319 (1.97x)	3776 (2.89x)	2462	69.2

¹ Codes protected by CFT-tool
² Codes protected by HPCT Suite

Table 2 presents the results of a bubble sort running in a MIPS processor. The results of the same program with the techniques applied over the machine code are presented below the results obtained by the program protected with the CFT-tool.

TABLE II. BUBBLE SORT (MIPS)

Technique	Exec. Time (μ s)	Memory Occupation (bytes)	Number of Errors	Detection Rate (%)
Unprotected	196	1156	-	-
VAR1 ¹	481 (2.45x)	2744 (2.37x)	2206	69.9
VAR1 ²	396 (2.02x)	2932 (2.54x)	2294	71.2
VAR2 ¹	515 (2.63x)	2996 (2.59x)	2090	66.9
VAR2 ²	448 (2.29x)	2912 (2.52x)	2116	67.6
VAR3 ¹	367 (1.87x)	2220 (1.92x)	2048	67.4
VAR3 ²	343 (1.75x)	2520 (2.18x)	1993	67.3
SIG+BRA+VAR3 ¹	430 (2.19x)	3092 (2.67x)	2133	70.0
SIG+BRA+VAR3 ²	408 (2.08x)	3492 (3.02x)	2152	68.2

¹ Codes protected by CFT-tool
² Codes protected by HPCT Suite

The runtime overhead increases in the bubble sort compared to the matrix multiplication. This happens because code blocks inside the loops have a relative bigger increase compared to a code block of the matrix multiplication. The detection rate is a little smaller because of more faults affecting the control-flow. Furthermore, the variables techniques also protect against faults affecting the program flow by duplicating the control registers.

The difference in the runtime overheads to assembly and machine is because of the branch and jumps treatments. From assembly to machine code, the assembler shifts all branches and jumps one position up, because this processor always executes an instruction after the branches and jumps. Fig. 5 shows an example that explains the difference. The correctness of the machine code is maintained when the program runs without errors, but when an error is detected another one can be created by an instruction that should be not executed. This feature can interfere in the error treatment.

```

01b8: 17ce0222 bne $30,$14,0xa44    01c4: 17cd0235 bne $30,$13,0xa9c
01bc: 14010221 bne $0,$1,0xa44      01c8: 00000000 nop
01c0: afc00060 sw $0,96($30)       01cc: 141a0233 bne $0,$26,0xa9c
01c4: adc10260 sw $1,608($14)      01d0: 00000000 nop
01c8: 17ce021e bne $30,$14,0xa44    01e4: afc00060 sw $0,96($30)
01cc: 1401021d bne $0,$1,0xa44     01d8: 17cd0230 bne $30,$13,0xa9c
01d0: afc00064 sw $0,100($30)     01dc: adba0100 sw $26,256($13)
01d4: adc10264 sw $1,612($14)      01e0: 141a022e bne $0,$26,0xa9c
                                01e4: 00000000 nop
                                01e8: afc00064 sw $0,100($30)
                                01ec: 17cd022b bne $30,$13,0xa9c
                                01f0: adba0104 sw $26,260($13)

```

Figure 5. Machine code (left) and assembly code (right)

The techniques were also applied to a matrix multiplication and a bubble sort running in a LEON3 processor to validate that the CFT-tool is configurable and architecture independent. Table 3 shows the same matrix multiplication with the same software-based techniques applied to a LEON3 processor. LEON3 is a processor with a SPARC V8 architecture that implements the registers in windows, where the registers are divided into local, global, input and output registers.

TABLE III. MATRIX MULTIPLICATION (LEON3)

Technique	Exec. Time (μ s)	Memory Occupation (bytes)	Number of Errors	Detection Rate (%)
Unprotected	99	984	-	-
VAR1	295 (2.98x)	2416 (2.46x)	587	55.4
VAR2	269 (2.72x)	2248 (2.28x)	600	54.8
VAR3	200 (2.02x)	1828 (1.86x)	586	36.7
SIG+BRA+VAR3	199 (2.01x)	2168 (2.20x)	569	31.3

The lower detection rate to the LEON3 processor can be explained because the LEON3 has no sufficient available registers to be used when variables needed to be duplicated. There are only five available registers to the variables techniques and four to the variable technique combined with the control-flow techniques, because the SIG technique requires one register, which can explain the lower detection rate to the combined techniques. Therefore, a lower detection rate is expected. But, despite the lower detection rate due to the lack of register, the same characteristics about the detection happen here, showing that the techniques are also efficient to this processor.

Table 4 shows the detection rate of a bubble sort program, protected with the different combination of the techniques running over a LEON3 processor.

TABLE IV. BUBBLE SORT (LEON3)

Technique	Exec. Time (μ s)	Memory Occupation (bytes)	Number of Errors	Detection Rate (%)
Unprotected	113	588	-	-
VAR1	343 (3.04x)	1660 (2.82x)	514	55.8
VAR2	315 (2.79x)	1540 (2.62x)	563	60.7
VAR3	260 (2.30x)	1264 (2.15x)	524	45.6
SIG+BRA+VAR3	248 (2.19x)	1636 (2.78x)	566	49.6

Also in these simulations we can see the same characteristics to MIPS processor about the detection rate of the techniques and a lower detection rate as in the matrix multiplication running in a LEON3 processor because there are no sufficient free registers to duplicate all used registers. In the bubble sort, there are only two available registers. However, the protected registers have a significant impact in the detection rate. The most read registers were the ones selected to be protected.

V. CONCLUSION AND FUTURE WORKS

In this paper, we presented the CFT-tool, a tool capable of protecting processors against SEU and SET faults. The CFT-tool implements automatically a set of configurable software-based detection techniques over the assembly code of an unprotected program. The tool works independently of the processor architecture. The tool is configured to the target processor by describing its architecture. A set of programs was protected by the CFT-tool to different processors: MIPS and

LEON3. A fault injection campaign was performed and simulation results show high detection rates to MIPS processor, around 70%; and a regular detection rate to LEON3 processor, around 50%. As future work, we intend to identify the best way to select the registers, when more registers than the available in the architecture are needed. Moreover, we will verify the CFT-tool in other architectures, such as ARM and PowerPC, to test the efficiency of the tool and of the techniques.

REFERENCES

- [1] P.E. Dodd, M.R. Shaneyfelt, J.R. Schwank and J.A. Felix, "Current and Future Challenges in Radiation Effects on CMOS Electronics", IEEE Transactions on Nuclear Science, vol. 57, n. 4, august 2010, pp. 1747-1763.
- [2] D. Pradhan, "Fault-tolerant computer system design", Upper Saddle River, USA : Prentice-Hall, 1995.
- [3] S.C. Asensi, A.M. Alvarez, F.R. Calle, F.R. Palomo, H.G. Miranda and M.A. Aguirre, "A Novel Co-Design Approach for Soft Errors Mitigation in Embedded Systems", IEEE Transactions on Nuclear Science, vol. 58, n. 3, june 2011, pp. 1059-1065.
- [4] N. Oh, S. Mitra and E. McCluskey, "ED4I: error detection by diverse data and duplicated instructions", IEEE Transactions on Computers, vol. 51, n. 2, 2002, p. 180-199.
- [5] N. Oh, P.P. Shirvani, and McCluskey, "Control-flow checking by software signatures", IEEE Trans. on Reliability, vol. 51, Issue 1, March 2002, pp. 111-122.
- [6] P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M.S. Reorda and M. Violante, "Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors", IEEE Trans. on Nuclear Science, vol. 47, n. 6 (part 3), Dec 2000, p. 2231-2236.
- [7] L.M.O.S.S. Hangout and S. Jan. The minimips project, available online at <http://www.opencores.org/projects.cgi/web/minimips/overview>, 2010.
- [8] Aeroflex Gaisler, LEON3, available online at <http://www.gaisler.com>, 2010.
- [9] J.R. Azambuja, A. Lapolli, L. Rosa, F.L. Kastensmidt, "Evaluating the efficiency of software-only techniques to detect SEU and SET in microprocessors", IEEE Latin American Symposium on Circuits and Systems, 2010.
- [10] L.D. Mcfearin and V.S.S. Nair, "Control-flow checking using assertions", Proceedings of the IFIP International Working Conference Dependable Computing for Critical Applications (DCCA-05), Urbana-Champaign, IL, USA, September 1995.
- [11] 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 Trans. on Parallel and Distributed Systems, vol. 10, Issue 6, June 1999, pp. 627-641.
- [12] J.R. Azambuja, A. Lapolli, L. Rosa, F.L. Kastensmidt, "Non-Intrusive Hybrid Signature-Based Technique to Detect SEU and SET Faults in Microprocessors". Proceedings of the European Conference on Radiation and Its Effects on Components and Systems, 2010.
- [13] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan and D.I. August, "SWIFT: software implemented fault tolerance", Proceedings of the Symposium on Code Generation and Optimization, 2005, p. 243-254.
- [14] B. Nicolescu and R. Velazco, "Detecting soft errors by a purely software approach: method, tools and experimental results", Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, 2003.
- [15] M. Rebaudengo, M.S. Reorda, M. Torchiano and M. Violante, "Soft-error detection through software fault-tolerance techniques". Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, pages 210, 218, 1999.
- [16] Mentor Graphics, <http://www.model.com/content/modelsim-support>, 2010.