Non-Intrusive Hybrid Signature-Based Technique to Detect SEEs in Microprocessors

José Rodrigo Azambuja[1], Mauricio Altieri[1], Fernanda Lima Kastensmidt[1], Michael Hübner[2], Jürgen Becker[2]

[1] Instituto de Informática – PPGC – PGMICRO
Universidade Federal do Rio Grande do Sul (UFRGS)
Av. Bento Gonçalves 9500, Porto Alegre - RS - Brazil
{jrfazambuja, mascarpato, fglima} @ inf.ufrgs.br

[2] Institut für Technik der Informationsverarbeitung (ITIV)
Karlsruhe Institute of Technology (KIT)
Vincenz-Prießnitz-Str. 1, Karlsruhe, Germany
{azambuja, michael.huebner} @ kit.edu

Abstract— This paper presents a hybrid technique based on software signatures and a non-intrusive hardware module to detect SEE faults in microprocessors. These types of faults have a major influence in the micro**processor's control** flow, causing **incorrect jumps in the program's** execution flow. In order to protect the system, a non-intrusive hardware module is implemented in order to spoof the data exchanged between the microprocessor and its memory. Since the hardware alone is not capable of detecting all control flow errors, it is enhanced to support a new software-based technique. A fault injection campaign is performed using a MIPS microprocessor. Simulation results show high detection rates with a small amount of performance degradation and area overhead.

Index Terms — Fault tolerance, Hybrid fault tolerance techniques, Microprocessors, SEEs

I. INTRODUCTION

MICROPROCESSORS working in harsh environments can be upset by energized particles presented on space or secondary particles such as alpha particles, generated by the interaction of neutron and materials at ground level [1]. One of the major effects that may occur when a single radiation ionizing particle strikes the silicon is known as Single Event Effect (SEE). When a transient pulse is generated in a memory cell, it is classified as Single Event Upset (SEU), while Single Event Transient (SET) when it occurs in a sensitive node of a combinational logic cell. The use of fault tolerance techniques is mandatory to detect or correct these types of faults. Fault tolerance techniques for microprocessors can be based on software or hardware redundancy.

The first one is based on adding instruction redundancy and comparison to detect or correct faults [2-4]. They may be able to detect faults that affect the data and control flow. Software-based techniques provide high flexibility, low development time and cost, since there is no need to modify the hardware. In addition, new generations of microprocessors that do not have RadHard versions in hardware can be used. As a result, aerospace applications can use commercial off the shelve (COTS) microprocessors with RadHard software. However, software-based techniques cannot achieve full system protection against soft errors due to control flow errors [5-6]. This limitation is due to the inability of the software in protecting all the possible control flow effects that can occur in the microprocessor.

Hardware-based techniques usually change the original architecture by adding logic redundancy, error correcting codes and majority voters. But they can be based on hardware monitoring and in this case are non-intrusive. They exploit special purpose hardware modules, called *watchdog processors* [7], to monitor memory accesses. Watchdogs usually can have access to the data and code memory connections. Since they only have access to the memory, watchdogs do not detect faults that are latent inside the microprocessor, as well faults in the register bank.

Combining software-based techniques and watchdog seems interesting in order to increase the fault detection coverage. In this paper, the authors present a method to detect soft errors in microprocessors using a non-intrusive hardware module that works in tandem with the software-based technique. A new method using hybrid signature is proposed. A fault injection campaign is performed in a MIPS microprocessor running two test-case applications. Experimental results prove both the effectiveness and the feasibility of the proposed technique.

II. PROPOSED HYBRID SIGNATURE-BASED TECHNIQUE

This proposed technique was initially based on [8] and its high fault detection rates for control flow errors and [9] with its full fault tolerance using a non-intrusive hardware module. By combining and improving these two works, we intend to increase the detection rates and performance, while decreasing the area overhead.

In order to better explain this technique, we will first introduce some terminology based on [8] that is used in this paper.

Program Graph (P): $P = \{V, E\}$ is a control flow graph with a set of nodes, $V = \{N1, N2, N3, ..., Nm\}$ and a set of directed edges, $E = \{e1, e2, ..., en\}$.

Node (N): A sequence of instructions in a program for which execution always begins with the first instruction and ends with the last instruction of the sequence. There is no branching instruction inside the node except possibly the last instruction and there is no possible branching into the node except to the first instruction of the node.

Edge: A directed edge between nodes *Ni* and *Nj* (denoted $Ni \rightarrow Nj$) representing a possible execution of *Nj* after execution of *Ni* in the absence of any errors.

Successor set (*Succ*): $Nj \in Succ(Ni) \Longleftrightarrow Ni \rightarrow Nj \in E$

Predecessor set (*Pred*): $Ni \in Pred(Nj) \Longleftrightarrow Nj \in succ(Ni)$

Node type (*NT*): A node is of type A if it has multiple predecessors and at least one of its predecessors has multiple successors. A node is of type X if it is not of type A.

Signature register (*S*): A run-time register which is continuously updated to monitor the execution of the program.

Node Ingress Signature (*NIS*): The expected value of S on ingressing the node on correct execution of the program.

Node Signature (*NS*): The expected value of S at any point within the node on correct execution of the program.

Node Exit Signature (*NES*): The expected value of S on exiting the node on correct execution of the program.

The proposed technique's main objective is to protect the system against control flow errors, which comprises incorrect jumps between different ndoes and inside the same node. In order to do that, a software-based technique is combined with a hardware module. The interaction between them is performed through store instructions statically inserted in the program code, which can be interpreted by the hardware module and the data being stored can be read.

A. Software-based Technique

The software-based technique can be divided in four steps, which parse the program code, analyze it and add static instructions to it. The first step parses the program code and generate a program graph, dividing the program into nodes and edges connecting them. On a second step, each node is analyzed and receives a NT value, according to its incoming and outgoing edges. The third step assigns *NIS*, *NS* and *NES* statically to each node, according to some rules discussed later. The fourth and final step adds instructions to the original program code, according to the analyses performed by the previous steps.

Fig. 1. shows the generated program graph and both types of nodes (NT type A and NT type X) generated after the first step. As one can notice, the node where NT equals to A has multiple predecessors and one of them has a successor. The other node has one single predecessor.

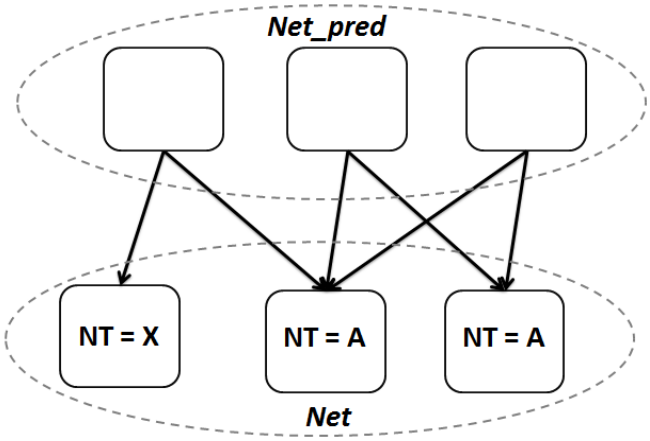

Figure 1. Program graph with both NT types

1) Technique Overview

Instructions are statically inserted into the program code to continuously, at run-time, update the value of S, as to monitor the program flow. Whenever the control flow leaves a node, S is assigned with the node's NES. When entering a new node, its value is assigned with NIS. During the execution of the instructions inside a node, S is assigned with NS. By using the NES and NIS values, one can detect control flow errors caused by jumps between different nodes [8]. NS is responsible for detecting control flow errors with incorrect jumps inside the same node.

At run-time, S can be updated up to three times in each node transition. It varies because NIS and NS, and also NES and NIS values may be the same and, therefore, not require an update on S. The updates on S follow this sequence: (1) NS to NES, (2) NES to NIS and (3) NIS to NS. The updates 1 and 3 (NS to NES and NIS to NS, respectively) are a straight transformation based on the XOR operator, performed by the following instruction:

$S = S$ XOR *invariant(Ni)*

The update 2 (NES to NIS), on the other hand, depend on the NT value of the current node. When NT equals to A, the instruction performed is an XOR, otherwise, the instruction performed is an AND, according to the following rule:

$S = S$ AND *invariant(Ni)* if $NT(Ni) = A$

$S = S$ XOR *invariant(Ni)* if $NT(Ni) = X$

At certain points of the code, which can be defined by the user, consistency check can be added, through store instructions. Such instructions store the value of S in a given preset memory address that can be identified by the hardware module. The hardware module is then responsible for informing an error to the system.

2) Algorithm for Signature Checking

This section gives the algorithms used to assign the values of NIS, NS and NES, for each node. Three terms are introduced here, based on [8].

Network (*Net*): A network is a non-empty set of nodes such that $Ni \in Net \Rightarrow (\forall Nj : pred(Ni) \cap pred(Nj) \_= \varphi : Nj \in Net)$, i.e., all the successors of each of the predecessors of *Ni* are also in the network, and is minimal, i.e., no non-empty subset of *Net* follows the above property. Each node in the program belongs to one and only one network. It can be seen on Fig. 1.

Network predecessors (*Net_pred*): The set *network predecessors* is the union of *predecessors* of all its elements. $net\ pred(Net) = \{\cup pred(Ni) : Ni \in Net\}$. It can be seen on Fig. 1.

Related signature set (*A_sig*): This set is the union of *NES* of all the nodes in the network predecessors set and the *NIS* of all nodes of type A in the network.

$A\_sig(Net) = \{\cup NES(Ni) : Ni \in net\ pred(Net)\}$

$\cup \{\cup NIS(Ni) : Ni \in Net \&\& NT(Ni) = A\}$

In order to allow the hardware module to perform consistency checks, NS must be assigned with a value that can be calculated by the module. It also must be a unique value, preventing aliasing. Therefore, each node's NS is assigned with the XOR of all its instructions and the memory address of its first instruction. Depending on the microprocessor's

architecture, only a set of the instruction can be used to generate the NS value, in order to avoid aliasing. As an example, only the 16 less significant bits of the instructions can be used to generate NS. Considering that S is never reset, but always updated, the hardware module can also detect incorrect values affecting NIS and NES.

NES and NIS values are divided in two parts, the upper half and the lower half. Each part is calculated differently and has a different objective. Therefore, their sizes can vary according to the program code requirements to avoid aliasing.

The upper half is used to identify the program networks (*Nets*), generating an unique value to each *Net*, called *A_sig(Net)*. That means that its minimum size, in order to avoid aliasing is *log2(#networks)* bits. Once generated, *A_sig(Net)* is assigned for every node's NIS that belong to Net and every node's NES that belong to network predecessor *Net_pred*. By doing so, one guarantees the detection of control flow errors that generate an incorrect jump between different networks.

The lower half is used to identify the nodes inside the same network, or node with the same upper half. The algorithm to generate these values, described in Fig. 2., is more complex, because it has to guarantee that:

- The NES value must be accepted by all the successor nodes;
- An incorrect jump from a node to one of its successor node must be detected.

B. Hardware Module

The proposed approach based on hardware relies on a module that combines a watchdog with a decoder. The watchdog is used to detect incorrect jumps to unused memory addresses and control flow loops (which cannot be detected by software-based techniques), while the decoder spoofs data and address buses and the read/write signal between the microprocessor and the memory in order to perform the instructions sent by the software-based transformation rules. The decoder spoofs the buses searching for two instructions: (1) Reset XOR, which resets the internal module's register that store the current XOR value and (2) Check XOR, which performs a consistency check, by verifying the value in the data bus with the internal module's registers storing the current XOR value.

```
For each Network Net {
  For each node N from Net_pred {
    For each node F from Net that is not successor of N {
      If F.NIS has a bit in 1
        Set this bit position to 0 in P.NES and in P's
successors' NIS
      Else if P.NES has any bit in 0
        Set this bit position to 1 in F.NS and in F's
predecessors' NES
      Else {
        Set 1 to a free bit position from F.NS and in its
predecessors NES
        Set 0 to the same bit postion in its successors'
P.NES and NS
      }
    }
    Set the free bit from P.NES to 1
  }
  For each  node N with NT=A
    Set the free bits from NIS to 1
  For each  node N with NT=A
    Set NIS to its predecessor NES
}
```

Figure 2. Algorithm for lower half signature

The hardware module, besides having access to the memory buses, must implement a XOR operator and a decoding module, which identifies instructions by spoofing the data bus from the memory. The buses used by some microprocessors which use on-chip embedded cache memories may not be accessible by the hardware module. In such cases, another approach should be used. It also must implement an error module to inform the system when an error is detected.

III. IMPLEMENTATION

The chosen case-study microprocessor is a five-stage pipeline microprocessor based on the MIPS architecture. The miniMIPS microprocessor is described in [10].

A. Hardware Module

The hardware module was implemented in VHDL language based on a watchdog timer that signals an error if not reset. Its enhancement was performed by adding a 16-bit register to store the real-time calculated XOR value and a decoder module to identify an XOR instruction performed over a preset register and a store instruction preset in a given memory address. Tab. 1. shows the size and performance of the implemented microprocessor and the hardware module. The hardware implementation has a total of 64 flip-flops and is not protected against faults.

Table 1: Original and modified architecture characteristics

Source	miniMIPS	HW Module + miniMIPS
Area (µm)	24261.32	2717.26 (+11%)
Frequency (MHz)	66.7	66.7

B. Hardening Post Compiling Translator Tool

A tool called *Hardening Post Compiling Translator* (HPC-Translator) was implemented to automate the software transformation, since this is a complex process. It receives as input the program's binary code and therefore is compiler and language independent. The tool is then capable of implementing the necessary instructions and modifying the code in order to control the hardware module. The implemented tool outputs a binary code, microprocessor dependent, which can be directly interpreted by the target microprocessor.

In order to evaluate both the effectiveness and the feasibility of the presented approaches, two applications based on 6x6 matrix multiplication and a bubble sort classification algorithms were chosen to be hardened.

Original	Hardened
beq r1, r2, 6	1: beq r1, r2, 6
add r2, r3, 1 ***NT = X***	**2: xor S, invariant** 3: add r2, r3, 1 **4: store S** **5: xor S, invariant**
add r2, r3, 4 ***NT = A***	**6: and S, invariant** **7: xor S, invariant** 8: add r2, r3, 4 **9: store S** **10: xor S, invariant**
jmp end	11: jmp end

Figure 3. Program transformation

Fig. 3 shows the transformation performed when this technique is applied to an unprotected code. On the left, one can see the original program code divided into 2 nodes, one where NT equals to X and another where it equals to A. On the right side, one can see the instructions added to the original code. Instructions 2 and 7 perform the update on S from NES and NIS, respectively, to NS. Instruction 6 performs the update on S from NES to NIS. Instructions 4 and 9 were added to send the real-time value of S to the hardware module and perform the consistency check. In this implementation, every node performs a consistency check.

Instructions 2, 7, 4 and 9 are parsed by the hardware module. The first two reset the stored XOR value, while the last two force the hardware module to check the value of S being stored with the internal XOR value.

One hardened program for each case study was generated using the *Hardening Post Compiling Translator* implementing the proposed technique. Tab. 2. shows the overhead in execution time, code size and data size, when compared to the original unhardened programs.

Table 2: Hardened programs' overheads

Source	Matrix Multiplication	Bubble Sort
Execution Time	10,7%	20%
Code Size	79,1%	25%
Data Size	3%	3%

IV. Fault Injection Experimental Results

In order to start the fault injection campaign, faults were injected in all signals of the non-protected microprocessor (including registered signals), one by program execution. The SEU and SET types of faults were injected directly in the microprocessor VHDL code by using ModelSim XE/III 6.3c. SEUs were injected in registered signals, while SETs were injected in combinational signals, both during one and a half clock cycle. The fault injection campaign is performed automatically. At the end of each execution, the results stored in memory were compared with the expected correct values.

The injected faults were classified according to their effect on the system. We continuously compared the Program Counter (PC) of a golden microprocessor with the PC of the faulty microprocessor. Faults that did not cause an incorrect jump in the program flow were classified as faults with data flow effect and discarded. The remaining faults were classified as faults with control flow effect and considered in the results.

Results presented in Tab. 3. for the matrix multiplication and bubble sort algorithms show that the hardening by the proposed technique could achieve 100% fault detection for control flow errors, where 41.000 and Y faults were injected, respectively.

Table 3: Fault injection results

Source	Faults Injected	Wrong Results	Faults Detected
Matrix Multiplication	41.000	5.021	5.021
Bubble Sort	47.000	5.146	5.146

V. Conclusions and Future Work

In this paper, the authors presented a non-intrusive hybrid technique based on signatures and implemented partially over a watchdog processor. Then, a tool was used to automatically harden binary programs. A fault injection campaign based on simulation was performed on the implemented techniques. Results showed that the proposed techniques could achieve full fault detection against control flow errors with performance overhead up to 20% and area overhead of 11%.

We are currently working on decreasing the impact on execution time and memory overhead of the proposed technique while keeping the same fault detection rates. As future work, we also intend to verify the feasibility and effectiveness of the proposed techniques applied to a real time operating system.

References

[1] International Technology Roadmap for Semiconductors: 2005 Edition, Chapter Design, pp. 6–7, 2005.

[2] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, M. Violante. "Soft-error detection using control flow assertions". In: Proceedings of the 18th IEEE international symposium on defect and fault tolerance in VLSI systems—DFT 2003, November 2003, pp 581–588.

[3] K. H. Huang, J. A. Abraham. "Algorithm-based fault tolerance for matrix operations". In IEEE Trans Comput 33:518–528, Dec. 1984.

[4] N. Oh, P. P. Shirvani, E. J. McCluskey. "Control flow Checking by Software Signatures". In IEEE Trans Reliab 51(2):111–112 (Mar), 2002.

[5] C. Bolchini, A. Miele, F. Salice, and D. Sciuto. "A model of soft error effects in generic ip processors". In Proc. of the 20th IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems, pages 334–342, 2005.

[6] J. R. Azambuja, F. Sousa, L. Rosa, F. L. Kastensmidt, "The limitations of software signature and basic block sizing in soft error fault coverage", IEEE LATW, 2010.

[7] A. Mahmood and E. McCluskey, "Concurrent error detection using watchdog processors", IEEE Trans. on Comp., 1988.

[8] R. Vemu, J. A. Abraham, "CEDA: Control-flow Error Detection through Assertions". In Proc. IEEE International On-Line Testing Symposium, 2006.

[9] J. R. Azambuja, A. Lapolli, L. Rosa, F. L. Kastensmidt, "Detecting SEEs in Microprocessors Through a Non-Intrusive Hybrid Technique". In IEEE Trans. on Nuclear Science, 2011.

[10] L. M. O. S. S. Hangout, S. Jan. "The minimips project", 2009, available online at http://www.opencores.org/projects.cgi/web/minimips/overview.