2021 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress

Impact of Selective Implementation on Soft Error Detection Through Low-level Re-execution

Mohaddaseh Nikseresht*§, Brent De Blaere*§, Jens Vankeirsbilck§, Davy Pissoort‡ and Jeroen Boydens§
*These authors contributed equally to this work
§Department of Computer Science, KU Leuven
‡Department of Electrical Engineering, KU Leuven
Spoorwegstraat 12, 8200 Bruges, Belgium
{Mohaddaseh.Nikseresht, Brent.DeBlaere, Jens.Vankeirsbilck, Davy.Pissoort, Jeroen.Boydens}@kuleuven.be

***Abstract*—Software-implemented hardware fault tolerance techniques can be used as a cost-effective alternative to hardware-implemented techniques to enhance the resilience of microprocessor systems. As with many of these software-implemented techniques, our recently developed low-level re-execution-based DETECTOR strategy suffers from high execution time overhead. Recently, error detection techniques using the concept of selective hardening to reduce this execution time overhead have been researched. Based on this concept, this paper proposes a selective implementation of DETECTOR called S-DETECTOR, which only protects a part of the target program based on its most vulnerable registers. Experimental results show that a considerable overhead reduction is obtained with a minor drop in fault coverage.**

***Index Terms*—Embedded Systems, Fault Detection, Selective Hardening, Soft Errors**

1. Introduction

Nowadays, embedded systems are used in a variety of industrial, military, medical and commercial applications. Over the past decades, industry has been moving towards smaller and increasingly more complex systems, making the design of an embedded system a critical engineering challenge. One of the recent challenges is managing the vulnerability of these complex systems to external disturbances, such as radiation-induced single event effects (SEE). Such disturbances can be caused by radiation effects in electronic devices, energized particles in space or by secondary particles, such as alpha particles generated by neutron-material interactions at ground level [1]. Single Event Upsets (SEUs) or soft errors are one kind of these SEEs, which are the main focus of this paper. Soft errors are nondestructive events that can cause a microprocessor to operate in an incorrect and unpredictable manner, generating a wrong result or an incorrect control flow. These sorts of errors are known as soft errors or transient faults because they do not cause permanent damage to the micro-controller and may be mitigated when damaged data is replaced.

Hardware solutions are often used to increase the resilience of embedded systems. These solutions can range from shielding parts of the system with various materials to duplicating or even triplicating the hardware components [2], [3]. However this approach increases the hardware design complexity and makes the system more expensive to produce. As a cost-effective alternative to traditional hardware protection methods, in recent years, several proposals focus on redundant software, also known as Software Implemented Hardware Fault Tolerance (SIHFT). When applying SIHFT, the business code of a target program is implemented in alternative ways and executed with majority voting to give it the ability to detect and sometimes to correct soft errors [4]–[7]. One recent example is RACFED, proposed by Vankeirsbilck et al. [8]. This state-of-the-art technique is designed to detect control flow errors (CFEs). These are soft errors that disturb the control flow of a program. A second type of soft errors are data flow errors (DFEs). These types of errors corrupt the input, output or intermediate data of a program. They are typically protected by data flow error detection techniques, like FDSC, recently proposed by Thati et al. [9]. To maximize the protection of a program, a hybrid technique like nZDC, proposed by Didehban et al. can be used [10].

One major limitation of data flow error detection techniques, and therefore also hybrid techniques, is that they require at least the same amount of registers used in the business logic of a program to be dedicated to the technique. In other words, only half of the available CPU registers can be used by the compiler for the business logic of a program. This can cause the register allocation process of the compilation process to fail for more advanced and industrial grade programs. To address this issue, we recently developed a novel software-implemented low-level technique, called DETECTOR [11]. DETECTOR uses only three CPU registers, while effectively providing protection against both CFEs and DFEs.

Despite their advantages, SIHFT strategies have substantial code and performance overheads, limiting their use. The increased code size can exceed the processor and memory resources, which are typically limited in embedded systems. To address these shortcomings, selective implementation of SIHFT has been suggested, with promising results on the design of dependable applications [12]. By protecting only

978-1-6654-2174-4/21/$31.00 ©2021 IEEE
DOI 10.1109/DASC-PICom-CBDCom-CyberSciTech52372.2021.00031

a subset of elements, e.g. CPU registers or simple blocks of instructions, it is possible to achieve significant overhead reductions with minimal effects on overall reliability. The parts to protect should be selected based on their criticality or contribution to overhead.

Based on this concept, this paper proposes a selective implementation of DETECTOR, called S-DETECTOR. First, the most vulnerable registers are detected using a register analyzer framework [13]. Next, S-DETECTOR selectively protects these registers. The experimental results demonstrate that with a small decline in fault coverage, a considerable overhead reduction is accomplished.

The remainder of this paper is structured as follows. In Section 2, we briefly discuss how other techniques approach the selective implementation of CFE and DFE detection. Next, in Section 3, the principle of DETECTOR is briefly revisited and the approach to make this technique selective will be discussed. The validation of S-DETECTOR is then discussed in Section 4, after which the future work is explored in Section 5. Finally, our conclusions are drawn in Section 6.

2. Related Work

In recent years, several selective error detection techniques have been proposed to reduce execution time overhead. This entails protecting only the most vulnerable parts of the code while limiting the impact on the error detection capabilities of the technique.

One state-of-the-art selective CFE detection technique is S-SETA [14]. Within S-SETA, no extra instructions are inserted in smaller basic blocks to decrease the imposed overhead. As a drawback, the detection ratio decreases as CFEs may not be detected within the disregarded basic blocks. Khudia et al. chose a different approach with their Abstract Control Signatures (ACS) method [15]. In this method, the basic blocks are divided into regions, each with a unique signature. In each basic block of a region, signature update instructions are placed, whereas signature check instructions are placed exclusively within so-called fundamental blocks. Alternatively, SETA-C only inserts verification instructions in large basic blocks [14]. Although ACS and SETA-C are promising, their implementation can be difficult. The overhead time reduction may be restricted for algorithms with minor change in the length of basic blocks. To overcome this shortcoming and to make the implementation more consistent and easy, Vankeirsbilck et al. suggested to only place the signature verification instructions before return statements [16]. The basic blocks in which these return statements reside represent final fundamental blocks on every conceivable CFG path, enabling most CFEs in the control flow graph to be detected.

Multiple selective DFE detection approaches have also been proposed in the past. The most common approach is to only protect a subset of basic blocks. One way of doing this is to find the most vulnerable basic blocks. This was proposed by Arasteh et al., whose GA method attempts to identify a subset of basic blocks that maximize the vulnerability [17]. Others, like Thati et al., propose to protect the basic blocks of the most vulnerable data paths instead. With their method, called VDP, the vulnerable path is defined as the longest path in a control flow graph [18].

Since DETECTOR relies on neither signature checking, nor instruction duplication, none of these selective approaches are suited to selectively implement DETECTOR. In this paper, the concept of the SHARC technique is used [19]. This technique can be used to determine the most vulnerable registers which is the base of our Register Analyzer Framework [13].

3. Selective Detector

This section starts with a brief description of the full DETECTOR technique, after which DETECTOR is transformed to only protect those registers deemed most vulnerable.

3.1. DETECTOR

The general idea behind DETECTOR is shown in Figure 1. Within the program, several checkpoints are inserted. These checkpoints are added before the parts in the program where it must be ensured that the register values are correct, such as: memory write operations, calls to a subroutine, and return statements. These parts were named *Vulnerable sections*. Before each checkpoint, the register values are written to memory. Next, a part of the program gets executed. Following this, the results of the executed part of the program are written to memory and the original register values are restored. Next, the program jumps back to the checkpoint, re-executing the same part of the program. When the second execution finishes, the program jumps to the comparison stage. In this stage, each register value is individually compared to its counterpart stored in memory. This entails reading the data stored in memory one by one and comparing each loaded value with the current register value. When a mismatch occurs, control is transferred to an error handler. If all values match, the *vulnerable section* is executed, after which the current register values are again stored and the same process repeats for the next part of the program.

The main drawback of DETECTOR is its large imposed execution time overhead (ETO). This is determined by the ratio of the execution time after and before the implementation of the technique, like shown in Equation (1).

$$\text{ETO} = \frac{t_{\text{technique}}}{t_{\text{unprot}}} \tag{1}$$

This can be contributed to a number of factors. Firstly, with DETECTOR, every instruction of the protected code is re-executed. This already doubles the ETO. Next, because DETECTOR re-executes parts of the code, the registers values at the various checkpoints should be stored and restored. For this, DETECTOR writes and reads the register values to memory. This does not come cheap, for this

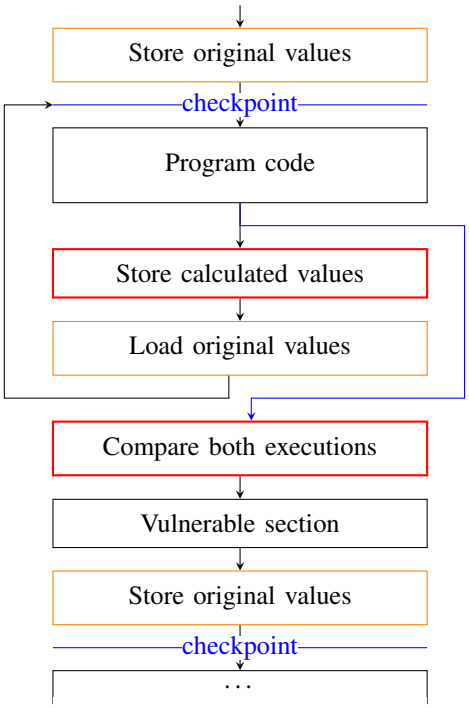

Figure 1: The control flow of a program protected with DETECTOR

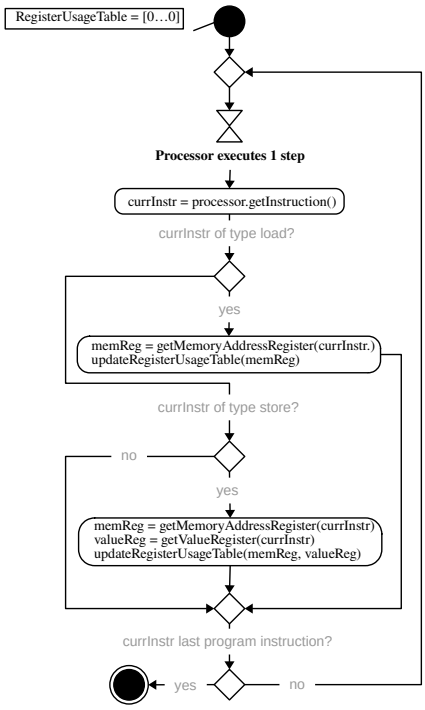

Figure 2: The Register Analyzer framework

can cause a pipeline stall, further increasing the execution time overhead. Finally, at the end of the re-execution, all calculated values of both executions are compared to one another. This requires a lot of instructions, including again memory operations, further increasing the ETO. These factors combined, make that the execution time overhead ranges from 2.09 up to 11.95, depending on the case study on which the technique was applied [11].

3.2. Selective Implementation

In this paper, we address this issue of a high ETO by proposing S-DETECTOR, which only protects the most vulnerable registers. The method used to identify the most vulnerable registers is based on the idea described in [19], which states that memory is more sensitive to ionizing particles than other circuit components since it is designed to be as dense as possible. As a result, the registers that interact with memory are the first candidates in a design to be hardened. With this in mind, we created a dynamic framework that evaluates and prioritizes registers depending on their interaction with memory. By stepping through the program and recording which registers interact with memory, a vulnerability list is created in which the register that interacts most with memory is deemed most vulnerable [13]. Figure 2 shows the inner working of our register analyzer framework.

Focusing on these vulnerable registers reduces the number of instructions needed in the comparison stage of the

```
1: ldr r11, [r5], #4 @r0
2: cmp r0, r11
3: bne 38
4: ldr r11, [r5], #4 @r1
5: cmp r1, r11
6: bne 38
        ...
33: ldr r11, [r5], #4 @r12
34: cmp r12, r11
35: bne 38
36: ldr r11, [r5], #4 @lr
37: bx lr
38: bl <errorHandler>
```

(a) Full DETECTOR

```
1: ldr r11, [r5], #4 @r0
2: cmp r0, r11
3: bne 6
4: ldr r11, [r5], #4 @lr
5: bx lr
6: bl <errorHandler>
```

(b) S-DETECTOR

Figure 3: Difference between the comparison stages of the full and selective DETECTOR using the ARMv7-M ISA

technique. In Figure 3, the comparison stage of the full DETECTOR implementation is compared to the selective implementation when only protecting one register (r0 in this case). Notice that the number of instructions for this comparison has been reduced by a factor of 6. Moreover, a large amount of nonconsecutive memory read instructions (`ldr`) are removed. Eliminating these instructions removes the pipeline stall that comes with memory operations, further decreasing the CPU cycles.

4. Evaluation

To assess our proposed approach, several fault injection campaigns were performed to determine the fault tolerance capabilities of S-DETECTOR. This section first goes into the experimental setup used to validate S-DETECTOR. Next, the employed case studies are discussed, and finally, the experimental results are evaluated.

4.1. Experimental setup

In order to evaluate the proposed technique, it is implemented manually in the assembly code of a number of case studies. In a further stage of our research, this implementation will be automated. Secondly, the fault coverage of the approach was assessed using extensive CFE and DFE fault injection experiments. These experiments were performed on a simulated ARM Cortex-M3 provided by the Imperas Instruction Set simulator [20]. For both the CFE and DFE fault injection, the used process works as follows. The target program is executed step-by-step and for each encountered instruction, all possible single-bit bit-flips are injected. When a CFE must be injected, the bit-flip is caused in the program counter register. In case a DFE is desired, it is injected in a register used by the current instruction. This injection process assures all possible soft errors are injected. For more details about the fault injection process, the reader is referred to [16].

The Imperas simulator is, however, not a cycle accurate simulator. It is, therefore, not suited to perform timing measurements. Hence, the execution time measurements on the case studies are performed on the NXP LPC1768 microcontroller [21].

4.2. Case Studies

As case studies, we selected an implementation of the bubble sort (BS), quicksort (QS), matrix multiplication (MM) and cyclic redundancy check (CRC) algorithms. These algorithms were selected since firstly they are used in different domains of embedded systems and secondly they have varying program structures, enabling a thorough validation of S-DETECTOR. To know which register(s) to protect of each case study, they were analyzed by the in-house built Register Analyzer Framework. The result of this analysis is shown in Table 1, in which the selected registers are indicated with *.

4.3. Overhead Reduction

The execution time overhead is, like previously shown in Equation (1), determined by the execution time before and after the implementation of the SIHFT techniques. The results of the measurements are shown in orange in Figure 4. This figure shows the ETO of all case studies protected by the S-DETECTOR and DETECTOR techniques, relative to the unprotected version of the code. It is apparent that the

TABLE 1: Results of the register analysis

Register	Case studies			
	BS	QS	CRC	MM
r0	66	835*	1,251*	101
r1	65	65	250	1,190
r2	64,874*	422	289	2,254*
r3	130,056*	643	8	108
r4	6	775*	6	6
r5	3	3	3	3
r6	0	0	0	0
r7	9	0	9	1,109
r8	0	0	0	0
r9	0	0	0	0
r10	0	0	0	0
r11	0	0	0	0
r12	0	0	0	0
sp	6	324	6	162
lr	0	0	0	0

*Registers protected by S-DETECTOR

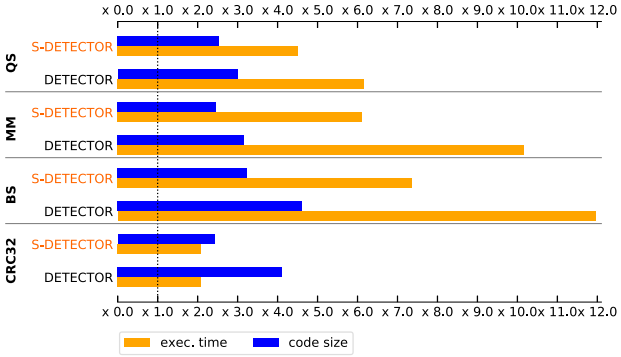

Figure 4: The measured overhead of S-DETECTOR compared to DETECTOR

selective approach of S-DETECTOR significantly reduces the ETO of the case studies. The overhead is still highly dependent on the case study on which the technique is applied. For some case studies - like BS and MM - the ETO is reduced significantly (from x11.95 to x7.37 and from x10.27 to x6.11, respectively), while the CRC32 case study only has a minor ETO decrease (from x2.09 to x2.08). The overhead increases significantly if the added memory operations reside inside a hot spot, i.e. a regularly executed instructions. Therefore, even with S-DETECTOR, the ETO is still high for some case studies.

Similarly, the code size overhead (CSO) is determined by the number of instructions before and after implementing the SIHFT techniques. The calculation of the CSO is shown in Equation (2), with N being the number of instructions.

$$\text{CSO} = \frac{N_{\text{SIHFT}}}{N_{\text{unprot}}} \tag{2}$$

These measurements are shown in blue in Figure 4. As is to be expected, the effect of the selective approach of S-DETECTOR is less significant when looking at code size overhead. This is because the comparison stage in Figure 3

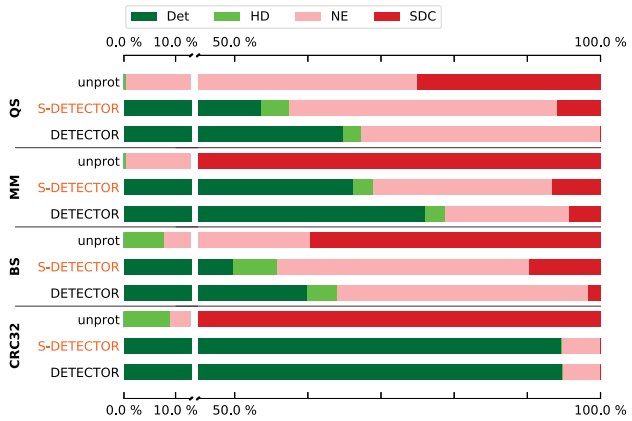

Figure 5: The results of the CFE fault injection

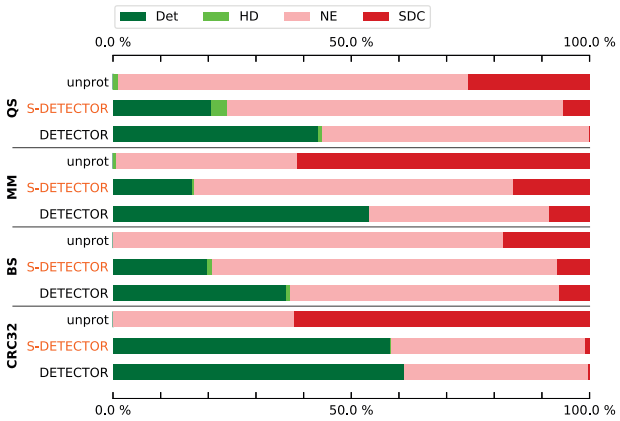

Figure 6: The results of the DFE fault injection

is implemented as a single subroutine. Every time the results of two executions are checked, this subroutine is called. Reducing the size of this subroutine reduces the dynamic instructions, and therefore the ETO, significantly. However, this does not affect the static instructions nearly as much. Therefore, the CSO reduction is also bigger for smaller programs, like for CRC32, where the CSO is decreased from x4.11 to x2.42.

4.4. Fault Injection Results

The results of the CFE and DFE fault injections are shown in Figure 5 and Figure 6, respectively. For each case study, the results of the unprotected code, the results of the code protected with S-DETECTOR and the results of the code with DETECTOR are shown. The results can be categorized as follows:

- **Det:** The error detection ratio - the percentage of injected faults that were detected by the technique.
- **HD:** The percentage of errors detected by the internal hardware fault detectors of the ARM Cortex-M3 processor [21].
- **NE:** The percentage of errors that had no effect on the program results and that were not detected by the technique.
- **SDC:** The percentage of injected faults that were not detected and resulted in a corrupted output of the program.

The selective implementation of DETECTOR implicitly has a lower error detection ratio, has a lower error detection ratio, as well as a higher SDC ratio. Analyzing the results, it is clear that the error detection ratio is significantly reduced, especially for the DFE fault injection results. However, the SDC ratio doesn't increase nearly as much. This indicates that most of the faults that are left undetected by the S-DETECTOR technique have no effect on the output of the program. This is to be expected, since data written to memory defines the program output and the registers that write to memory are protected. A good balance between the number of register to protect and the imposed overhead can most likely be found.

The increased SDC ratio is expected when selectively implementing a technique. However, analyzing the injected errors more closely shows how this selective approach could be improved. The CFE detection of DETECTOR, however, relies on the fact that different operations will be performed on the registers when the program takes an invalid path. Not all registers are checked by S-DETECTOR, which is why more control flow errors are left undetected. The same can be seen for data flow errors. A wrong branch can be taken if a DFE corrupts a register used in a *compare* (`cmp`) instruction. These new insights can help us to further improve the technique.

5. Future Work

The initial results of S-DETECTOR seem promising. However, fine-tuning this technique is still a work in progress. To fully establish its applicability, more research is yet to be done. The technique should be tested on more case studies, both data processing case studies and I/O driven case studies. Only when enough data has been collected, can we definitively determine whether or not the technique can be considered useful for industrial grade applications. In addition, we want to experiment on real-time systems, like audio-processing systems, to evaluate the real-time effects of the imposed execution time overhead. Moreover, the effect of combinations of protected registers should be explored to find the best balance between hardening capabilities and overhead for each case study.

Next, like already mentioned in section Section 3.2, analyzing the fault injection results shows how S-DETECTOR and DETECTOR could possibly be improved. Additionally, alternative methods to determine the most vulnerable registers could be explored based on these results.

Finally, the implementation of S-DETECTOR can be automated to be applied automatically on the program, for

example, inside a GCC or LLVM plugin. This will massively reduce the implementation effort of the technique, making it more applicable and flexible.

6. Conclusions

In this paper, we presented a selective version of our software-based technique DETECTOR, which is called S-DETECTOR. This technique is built in light of the selective hardening concept. In the first phase of the implementation, the Register Analyzer Framework will dynamically analyze all the registers inside the microprocessor register file. After this analysis, the most vulnerable registers are chosen and the S-DETECTOR technique is selectively applied to those registers. This technique is appropriate for low-cost dependable applications which use COTS micro-controllers. The S-DETECTOR evaluation results so far indicate that it has a reduced execution time overhead compared to the DETECTOR technique, with only a slight reduction in fault coverage as a side effect.

References

[1] R. Edwards, C. Dyer, and E. Normand, "Technical standard for atmospheric radiation single event effects, (SEE) on avionics electronics," in *2004 IEEE Radiation Effects Data Workshop (IEEE Cat. No.04TH8774)*. Atlanta, GA, USA: IEEE, 2004, pp. 1–5. [Online]. Available: http://ieeexplore.ieee.org/document/1352895/

[2] S. Geetha, K. K. Kumar, C. R. Rao, M. Vijayan, and D. C. Trivedi, "EMI shielding: Methods and materials - A review," *J. Appl. Polym. Sci.*, vol. 112, no. 4, pp. 2073–2086, may 2009. [Online]. Available: www.interscience.wiley.com

[3] P. Samudrala, J. Ramos, and S. Katkoori, "Selective triple Modular redundancy (STMR) based single-event upset (SEU) tolerant synthesis for FPGAs," *IEEE Transactions on Nuclear Science*, vol. 51, no. 5, pp. 2957–2969, Oct. 2004. [Online]. Available: http://ieeexplore.ieee.org/document/1344451/

[4] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, "SWIFT: Software Implemented Fault Tolerance," in *International Symposium on Code Generation and Optimization*. San Jose, CA, USA: IEEE, 2005, pp. 243–254. [Online]. Available: http://ieeexplore.ieee.org/document/1402092/

[5] S. A. Asghari, M. Binesh Marvasti, and M. Daneshtalab, "A software implemented comprehensive soft error detection method for embedded systems," *Microprocess. Microsyst.*, vol. 77, p. 103161, sep 2020. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0141933120303288

[6] B. Arasteh, "ReDup: A software-based method for detecting soft-error using data analysis," *Comput. Electr. Eng.*, vol. 78, pp. 89–107, sep 2019. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0045790617338806

[7] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Compiler-Directed Soft Error Detection and Recovery to Avoid DUE and SDC via Tail-DMR," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 2, pp. 1–26, apr 2017. [Online]. Available: https://dl.acm.org/doi/10.1145/2930667

[8] J. Vankeirsbilck, N. Penneman, H. Hallez, and J. Boydens, "Random Additive Control Flow Error Detection," in *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 11093 LNCS. Springer Verlag, sep 2018, pp. 220–234. [Online]. Available: https://doi.org/10.1007/978-3-319-99130-6_15

[9] V. B. Thati, J. Vankeirsbilck, N. Penneman, D. Pissoort, and J. Boydens, "An Improved Data Error Detection Technique for Dependable Embedded Software," in *2018 IEEE 23rd Pacific Rim Int. Symp. Dependable Comput.* IEEE, dec 2018, pp. 213–220. [Online]. Available: https://ieeexplore.ieee.org/document/8639681/

[10] M. Didehban and A. Shrivastava, "nZDC: A Compiler technique for near Zero Silent data Corruption Moslem," in *Proc. 53rd Annu. Des. Autom. Conf.*, vol. 05-09-June. New York, NY, USA: ACM, jun 2016, pp. 1–6. [Online]. Available: https://dl.acm.org/doi/10.1145/2897937.2898054

[11] B. De Blaere, J. Vankeirsbilck, and J. Boydens, "Soft Error Detection Through Low-level Re-execution." in review at ICSRS 2021, 2021.

[12] E. Chielle, J. R. Azambuja, R. S. Barth, F. Almeida, and F. L. Kastensmidt, "Evaluating Selective Redundancy in Data-Flow Software-Based Techniques," *IEEE Transactions on Nuclear Science*, vol. 60, no. 4, pp. 2768–2775, Aug. 2013. [Online]. Available: http://ieeexplore.ieee.org/document/6560441/

[13] M. Nikseresht, J. Vankeirsbilck, D. Pissort, and J. Boydens, " A Selective Soft Error Protection Method for COTS Processor-based Systems." accepted at ET 2021, 2021.

[14] E. Chielle, G. S. Rodrigues, F. L. Kastensmidt, S. Cuenca-Asensi, L. A. Tambara, P. Rech, and H. Quinn, "S-SETA: Selective Software-Only Error-Detection Technique Using Assertions," *IEEE Transactions on Nuclear Science*, vol. 62, no. 6, pp. 3088–3095, Dec. 2015. [Online]. Available: http://ieeexplore.ieee.org/document/7322301/

[15] D. S. Khudia and S. Mahlke, "Low cost control flow protection using abstract control signatures," in *Proceedings of the 14th ACM SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*. Seattle Washington USA: ACM, Jun. 2013, pp. 3–12. [Online]. Available: http://dl.acm.org/doi/10.1145/2465554.2465568

[16] J. Vankeirsbilck, J. Van Waes, H. Hallez, and J. Boydens, "Impact of selectively implementing control flow error detection techniques," *Internet of Things*, vol. 12, p. 100260, Dec. 2020. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2542660520300949

[17] B. Arasteh, A. Bouyer, and S. Pirahesh, "An efficient vulnerability-driven method for hardening a program against soft-error using genetic algorithm," *Computers & Electrical Engineering*, vol. 48, pp. 25–43, Nov. 2015. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0045790615003419

[18] V. B. Thati, J. Vankeirsbilck, and J. Boydens, "Comparative study on data error detection techniques in embedded systems," in *2016 XXV International Scientific Conference Electronics (ET)*. Sozopol, Bulgaria: IEEE, Sep. 2016, pp. 1–4. [Online]. Available: http://ieeexplore.ieee.org/document/7753517/

[19] J. Isaza-González, F. Restrepo-Calle, A. Martínez-Álvarez, and S. Cuenca-Asensi, "SHARC: An efficient metric for selective protection of software against soft errors," *Microelectronics Reliability*, vol. 88-90, pp. 903–908, Sep. 2018. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0026271418305493

[20] Imperas Software, "ISS - The Imperas Instruction Set Simulator." [Online]. Available: https://www.imperas.com/iss-imperas-instruction-set-simulator

[21] NXP Semiconductors, "LPC1769/68/67/66/65/64/63 Product data sheet," sep. [Online]. Available: https://www.nxp.com/docs/en/data-sheet/LPC1769_68_67_66_65_64_63.pdf