An Extensive Soft Error Reliability Analysis of a Real Autonomous Vehicle Software Stack

Vitor Bandeira*, Isadora Oliveira*, Felipe Rosa*, Ricardo Reis*, Luciano Ost[‡]

* UFRGS — Porto Alegre — RS — Brazil

{vbandeira, isoliveira, frdarosa, reis}@inf.ufrgs.br

[‡] Loughborough University — United Kingdom

l.ost@lboro.ac.uk

Abstract—Automotive systems are integrating artificial intelligence and complex software stacks aiming to interpret the real world, make decisions, and perform actions without human input. The occurrence of soft errors in such systems can lead to wrong decisions, which might ultimately incur in life losses. This paper focuses on the soft error susceptibility assessment of a real automotive application running on top of unmodified Linux kernels, and considering two commercially available processors, and three cross-compilers. Results collected from more than 29 thousand simulation hours show that the occurrence of faults in critical functions may cause $2.16\times$ more failures on the system.

Index Terms—Automotive Systems, Soft Error Analysis, Fault Injection, Virtual Platforms.

I. INTRODUCTION

Market predictions estimate that autonomous vehicles will result in up to US\$ 1.3 trillion savings for the US economy, mainly, due to the reduction in accidents, fuel utilisation, traffic and parking congestion [1]. Such systems are incorporating safety (e.g., automatic emergency braking) and emerging technologies (e.g., artificial intelligence, machine learning) to enable vehicles to efficiently react to hazards and unpredictable modifications in their environment [2]. Although using reliable hardware components is key to its success, the software side of autonomous vehicles is where the most challenging design challenges lie. The occurrence of soft errors can dramatically affect the functionality of an automotive system, which ultimately, can lead to death or severe injury to human lives.

To ensure autonomous vehicles failsafe operation, designers must be able to assess the soft error rate of complex software stacks with hundreds of billions of instructions, early in the design phase. Due to the complexity of such stacks (e.g., kernels, drivers and heavy applications), analysing their soft error reliability may take several months if conducted at low level simulations (e.g., gate-level). For this reason, this work utilises an open-source fault injection framework [3] to perform an in-depth and extensive soft error susceptibility evaluation of a realistic automotive software stack. Different from other works, this paper *contributes* with an extensive soft error assessment that employs a technique that isolates the critical functions of a real automotive application while considering different inputs (e.g., images), kernels, compilers and processor architectures. This is the first work that uses such an approach and number of variants, thus enabling to identify not only the soft error occurrence but also the specific

software characteristics that contribute more directly to their appearance.

II. RELATED WORKS

The occurrence of soft errors in automotive systems is a growing reliability issue. With that in mind, researchers have started to investigate the soft error reliability of applications and algorithms available in today's cars and emerging technologies that are expected to integrate the new generation of vehicles [4]. For instance, Li et al. [4] investigate the impact of soft errors on a convolutional neural network (CNN) accelerator. Results describe the CNN robustness to a few fault injection campaigns; however, it considers only a bespoke ASIC implementation that is not present in real automotive systems. Further, the adopted CNN is relatively simple, i.e., 17-layer, when compared to the 32-layer used in the present work. Furthermore, while this approach evaluates the Deep Neural Network (DNN) application behaviour under the presence of faults, our work considers a real software stack with multiple variants. Libano et al. [5] explore the soft error reliability of different neural network (NN) implementations using an FPGA-based fault injection tool, which emulates the occurrence of faults by modifying the bitstream configuration. Although interesting, this paper presents a limited amount of experiments that consider a single case study with only nine neutron-induced soft errors.

Santos et al. [6] test a Darknet + YOLO object detection algorithm under a neutron radiation beam to uncover its susceptibility to soft errors when running on three GPU architectures. Such a kind of approach enables the collection of real radiation-induced errors, but, due to the low observability (i.e., difficult to access internal components), conclusions are built based on fewer results, considering a limited error classification. In contrary, this paper evaluates the same algorithm but isolating its main functions while considering different images, compilers and processor architectures under 210 fault injection campaigns. The Darknet + YOLO algorithm is also used in [3] to validate new fault injection techniques. In this work, we gathered the soft error reliability of Darknet + YOLO algorithm considering only one input image, the Clang compiler and an ARMv7 processor architecture. Differently, this work considers five input images, two ISAs, and three compilers, therefore, exploring a broader number of aspects and scenarios.

III. FAULT INJECTION FRAMEWORK

Rather than developing a fault injection (FI) framework from scratch, we evaluated the performance of two open-source FI approaches [7], [3] when executing the target automotive benchmark. Due to the complexity of the underlying benchmark (i.e., up to 104 billions of instructions executed), conducting the soft error evaluation using the gem5-based FI framework [7] would restrict the number of explorations. While a single execution of the Darknet benchmark can take up to 12 hours with the gem5-based approach [7], SOFIA [3] requires less than twenty minutes to execute the same scenario. SOFIA also offers a set of FI techniques that enable bespoke inspections targeting a specific critical application, operating system or API structure/function. The underlying flexibility and simulation performance allows a fast and more effective soft error vulnerability analysis, justifying the use of SOFIA.

Fault Injection Model (FIM): SOFIA emulates the occurrence of single-bit-upset (SBU) by injecting one bit-flip in a single register or memory address during the execution of a given soft stack. In our setup, SBU target only storage elements due to its higher susceptibility to radiation events when compared with logic elements [8]. The fault injection configuration (e.g., bit location, injection time) relies on a random uniform function that has a low computational cost and is a well-accepted fault injection technique, which covers the majority of possible faults on a system. Fault injections occur during the target application lifespan, i.e., the OS startup is not subject to faults but includes OS system calls and parallelisation API subroutines arising during this period. This approach allows identifying unexpected application execution errors (e.g., segmentation fault), which correlate with adopted OS components or API libraries.

Fault Classification: after each fault injection, it is necessary to categorize the detected soft errors. For this purpose, we adopted two classification from the literature. The first proposed by Cho *et al.* [9]: **Vanished**, no fault traces are left;

Application Output Not Affected, the resulting memory is not modified; however, one or more remaining bits of the architectural state is incorrect; **Application Output Mismatch**, the application terminates without any error indication, and the resulting memory is affected; **Unexpected Termination**, the application terminates abnormally with an error indication; **Hang**, the application does not finish requiring a preemptive removal. Depending on the application's nature, the above classification may be inadequate to express possible misbehaviour. SOFIA enables the creation of new classifications to achieve a customised soft error analysis, which also justifies its adoption in this work.

After some experiments, we identified that Cho's classification falls short when it comes to defining object detection algorithms, which produce outcomes based on probabilities, not on just absolute "yes" or "no". For this reason, this work also adopts the classification proposed in [3], which includes the following conditions: **correct output** when the outputs (golden and FI) match, i.e., masked fault; **incorrect** if at least one object or probability is different. Further, the incorrect result can be split into **incorrect probability** when all objects

are correct, but at least one has a different percentile of confidence—in most cases this would not influence the action of an autonomous vehicle; **wrong detection**, i.e., false positive or missing of an object; and **no prediction**, if no object is in the image. The last two can represent a life-threatening failure, by forcing a full stop on a highway (false positive, an object in the path) or **an accident** by missing of an object. There is no clear 1:1 comparison between the two fault classes. For example, a OMM fault could be either **correct outputs** or any of the **incorrect** sub-classifications. However, we can say that a combination of all Vanished, ONA, and some OMM is equal to the **correct outputs**. The remainder of OMM would be classified as **incorrect**, while **crashes** is the summation of UT and Hangs.

Simulation Flow: the fault injection flow of SOFIA comprises five phases: First, the cross-compilation of the application source code results in an ELF object file. The faultless execution (phase 2) simulates the target application (using the ELF file) without fault influence to verify its correctness and extract the reference information (i.e., registers context and final memory state). During the faultless execution, SOFIA may acquire additional information depending on the selected fault injection technique (e.g., function and variable addresses). The third phase deploys the fault generation tool considering the injection time, the register name, and the mask (i.e., the target bit) for each fault injection. In the fourth and most complex phase, SOFIA starts by configuring an instruction counter event, which defines the insertion time. At this event, the component reads the fault characteristics and introduces the flipped bit according to the fault injection techniques. After the application conclusion, the FIM compares the application outcome (e.g., number of executed instructions, registers context, and memory state) under fault influence with the information acquired during phase 2. The last phase assembles all the individual reports to create a single file while performing several statistical analysis (e.g., average, worst, and best cases).

IV. ADAS BENCHMARK

The *European Commission* defines Advanced Driver-Assistance Systems (ADAS) as vehicle-based intelligent safety systems that improve road safety in terms of crash avoidance, mitigation, protection, and post-crash phases [10]. ADAS comprises multiple individual critical functionalities, which can be executed onto different platforms that vary across the multiple vendors (e.g., Waymo, Bosch, Harman) available in today's market. ADAS functionalities include, among others, obstacle detection, guidance system control, and adaptive cruise control. Such functionalities are combined to accomplish a reliable ADAS operation. For example, while a guidance system controls the overall vehicle movement, the collision detection module scans the car surroundings to avoid obstacles. Ensuring the correct and accurate execution of object detection is vital for the robustness of ADAS, even when its execution is exposed to the influence of radiation-induced soft errors. With the above in mind, we adopted the Darknet [11] + YOLO [12] as a case study to demonstrate the efficiency of our approach to inspect the soft error susceptibility of such a complex benchmark executing on the top of Linux kernels.

TABLE I: Darknet Execution (24 frames)

Platform	Avg. Time (seconds)		Objects Detected	
	Total Runtime	Prediction	Correct	Wrong
DE1-SoC A9 @200MIPS	231.77	165.99	142	11
OVPSim A9 @67.45MIPS	733.05	55.70	142	11

Darknet is an open-source neural network framework, while YOLO [12] is a Real-Time Object Detection system that makes predictions with a single network evaluation. Darknet + YOLO can be executed on multiple architectures (e.g., x86-64, GPUs, Arm), and evaluating its performance and results prediction consistency is necessary. In this regard, we compare the prediction results while using the same source code, input images, network weights, and the platform configurations described in Table I. Note that Darknet + YOLO needs 1.5GB of RAM and the board has 750MB, thus, a swap file that negatively affects performance was used. Total runtime includes application initialization and network weight loading; prediction time considers only the evaluation, i.e., from reading the input to the reporting the results.

Asserting the experimental setup accuracy is paramount to obtain meaningful and useful results. Table I also compares the results of adopted benchmark gathered using SOFIA with those obtained from a real board (i.e., DE1-SoC) execution. Results show that the Golden Execution of SOFIA presents no difference in the predictions (last two columns of Table I) for the selected images w.r.t. the physical platform, thus validating the utility of adopted framework.

V. EXPERIMENTAL SETUP

Table II shows the experimental setup for the 90 (architecture $\times 2$, compiler $\times 3$, input image $\times 5$, target register $\times 3$) fault injection campaigns in Table III and the 120 (architecture $\times 2$, compiler $\times 3$, input image $\times 5$, target function $\times 4$) in Table IV.

Different processor architectures: due to the gains in object detection (Section IV), and its current adoption in state-of-the-art cars (i.e., Cortex-A72 — included in Tesla Full Self-Driving Chip), we chose the ARMv7 (Cortex-A9) and the ARMv8 (Cortex-A72) processor architectures as our test platforms. Both processor architectures have 32 floating-point (FP) register, but different number of integer registers: ARMv7 has 16 integers and 32 ARMv8 has 33. Due to the different number of integer registers in the chosen architectures, we perform different fault injection campaigns considering only integer, only FP, and all registers.

Fault injection in register bank: fault analyses are obtained by injecting faults (i.e., bit-flips) in the general-purpose registers (e.g., r0-r12, PC, SP, LR). This approach is highly used by academics and industrial sectors.

Multiple input images: this work evaluates the Darknet + YOLO behaviour in the presence of faults using five images from the KITTI suite [13]. Each image contains relevant objects (e.g., cars, people, road signs) and different compositions (e.g., light, contrast, density of objects). The use of different images improves the quality of our results, as the different inputs may lead to different decision paths inside the evaluation engine (i.e., neural network).

TABLE II: Experimental Setup, where Exec. = Hardware Execution and FI = Virtual Platform Fault Injection

Processors (ISAs)	A9 (ARMv7) and A72 (ARMv8)
Linux Kernel	3.13 (A9) and 4.3 (A72)
Cross-Compilers	GCCv4.9, GCCv7, and Clang v6
Target Registers	Integer, Floating-point, Both
Application	Darknet (YOLO)
Input Frames	Exec. (24); FI (5)
Function FI	4
FI Campaigns	210
Total Fault Injections	172×10^3
Total Sim. Time (hours)	29.9×10^3
Total Sim. Instructions	8.20×10^{15}

Impact of different compilers: aiming to represent the ecosystem diversity, this work performs fault campaigns using three distinct cross-compilers: a legacy GCC version (v4.9.4), a more recent GCC release (v7.3), and the new compiler infrastructure (LLVM/Clang 6).

Fault Injection classification: each fault injection is classified according to the well-known Cho et al. [9], and to the *Bespoke Classification* from [3], which provides a complementary categorisation that relies on custom information extracted from the target application internal structure (i.e., output, quality of result).

Reliability evaluation: for each fault injection campaign we compute two *Reliability Metrics* the Mean Work Time to Failure (MWTF) [14] and the Extrapolated Absolute Failure Counts (EAFC) [15]. While the MWTF shows the average work an application can perform between failures (higher values are better), the EAFC compares an unhardened and a hardened application (lower values are better). This work employs the GCC 4.9 on the ARMv7 as our unhardened application (i.e., our baseline). Both metric equations have the number of clock cycles, which were gathered with the cycle accurate gem5 simulator [16] using same binary, Linux kernel, and processor architecture as the SOFIA setup for each compiler/platform combination. While MWTF shows the overall resiliency of a given fault campaign setup (i.e., processor architecture and compiler) the EAFC metric also considers the memory usage aspect thus providing information on the trade-off between the setups.

VI. DARKNET + YOLO SOFT ERROR ASSESSMENT

Table III shows results for 90 fault injection campaigns considering the whole execution of the application, i.e., a fault can be injected at any time during the execution of the application.

Impact of Processor Architectures on the Soft Error Results: when comparing the two processor architectures (ARMv7 and ARMv8), there are two main difference between them: the register bank, and the ISA. The latter will be discussed later as the compiler has direct impact on what instructions are included in the binary, and thus executed. The register bank for ARMv8 has 17 more integer registers, and all of them are 64bit. The increase in the register pool allows compilers to optimise the code and reduce the number of load/store instructions. Table III shows that ARMv8 executes $\sim 153M$ less instructions (for the same compiler) and $\sim 8.55M$

TABLE III: Object Detection Soft Error Reliability Assessment: Average of all five images. Column 3 shows the difference in number of instructions and cycles w.r.t. GCC 4.9 extracted from gem5. Reliability Metrics are normalized using GCC 4.9 values.

Arch	Compiler	Instructions (Cycles)	Target	Soft Error (%)					Bespoke Classification (%)					Reliability Metrics	
				Vanish	ONA	OMM	UT	Hang	Correct	No Pred	Inc. Probs	Wrong	Crash	MWTF	EAFC
1	GCC 4.9	104×10^9 (115×10^9)	Int	43.00	0.10	27.80	27.48	1.62	67.20	0.83	1.05	1.83	29.10	1.00	1.00
2			FP	60.18	0.00	39.83	0.00	0.00	99.18	0.00	0.03	0.80	0.00	1.00	1.00
3			Int+FP	55.00	0.07	35.59	8.63	0.71	89.31	0.15	0.44	0.76	9.34	1.00	1.00
4	GCC 7.3	-130×10^3 (-196×10^3)	Int	43.85	0.15	28.60	26.20	1.20	69.17	1.30	1.20	0.93	27.40	1.15	0.94
5			FP	56.50	0.03	43.43	0.03	0.03	99.05	0.00	0.00	0.90	0.05	0.79	1.15
6			Int+FP	52.11	0.07	38.73	8.65	0.44	88.95	0.44	0.27	1.25	9.09	0.67	1.03
7	Clang 6	-89 (-304)	Int	52.95	0.05	22.70	23.40	0.90	73.25	0.48	1.15	0.83	24.30	1.85	0.82
8			FP	67.65	0.00	32.35	0.00	0.00	99.28	0.00	0.03	0.70	0.00	1.30	0.88
9			Int+FP	61.79	0.05	30.25	7.60	0.32	90.64	0.22	0.37	0.86	7.92	1.06	0.88
10	GCC 4.9	-153×10^6 ($+8.36 \times 10^6$)	Int	61.19	1.53	13.82	22.76	0.70	74.96	0.27	0.90	0.41	23.47	3.08	3.15
11			FP	51.45	0.00	48.48	0.03	0.05	99.33	0.00	0.03	0.58	0.08	1.68	1.65
12			Int+FP	55.93	0.88	31.08	11.64	0.47	86.70	0.07	0.62	0.50	12.12	0.92	3.37
13	GCC 7.3	-153×10^6 ($+8.36 \times 10^6$)	Int	58.93	1.55	14.67	23.78	1.07	72.80	0.46	1.09	0.80	24.85	1.90	3.42
14			FP	48.90	0.00	51.03	0.08	0.00	99.53	0.00	0.00	0.40	0.08	3.60	1.16
15			Int+FP	53.11	0.88	32.78	12.78	0.45	85.47	0.12	0.45	0.73	13.23	0.76	3.68
16	Clang 6	-153×10^6 ($+8.35 \times 10^6$)	Int	64.32	1.67	11.56	21.43	1.02	75.68	0.53	0.66	0.68	22.45	2.67	3.06
17			FP	51.30	0.00	48.50	0.08	0.13	99.30	0.00	0.03	0.48	0.20	1.93	1.70
18			Int+FP	58.79	0.88	29.21	10.72	0.40	87.88	0.28	0.33	0.38	11.12	1.20	3.07

more cycles across the board (Table III, lines 10/13/16 for instructions and 12/15/18 for cycles). With the larger integer register pool the compiler does not need to perform as many swap operations, i.e., save the value from a register to a memory address to “free” the register and load another variable to it. As a side-effect, if the register is not saved to memory there are three possibilities: (i) fault propagation (OMM, UT, or Hang); (ii) the register is overwritten (Vanished); or (iii) the register is neither read nor overwritten, thus preserving the bit-flip until the end of the application execution (ONA).

Table III shows that a reduction in swaps leads to a decrease in OMM/UTs and an increase in ONA/Vanished (lines 1/4/7 vs. 10/13/16). A swap operation includes saving the value in the register to a memory position, the register that suffered a fault injection (i.e., bit-flip) can propagate its value to the memory in a store instruction (leading to OMM or Hang if later used for control-flow) or the register could be used to compute the memory address, e.g., array position, leading to a potential UT. Analogously, Table III also shows an increase in *Correct* results, decrease in *Incorrect* (No Prediction, Incorrect Probabilities, and Wrong Detection) results and Crashes. Further, the MWTF metric supports the findings that the ARMv8 suffers less from soft error although EAFC is higher, i.e., might not have the best trade-off.

Influence of Cross-compilers on the Soft Error Reliability: compilers are an intrinsic part of the software development cycle, playing a fundamental role as the instructions order and selection impacts on the application’s performance, power-efficiency and reliability. This work explores three distinct cross-compilers (i.e., GCC 4.9, GCC 7.4, and Clang 6) to conduct the soft error analysis shown in Tables III and IV. Considering fault injection on all available register (Table III, lines 3/6/9/12/15/18), Clang has higher MWTF and lower

EAFC (for the same architecture) w.r.t. both GCC versions. This result coupled with the difference in number of instructions executed (153M, line 1/10 vs. 7/16) and number of cycles (lines 3/12 vs. 9/18) lead us to believe that Clang’s choices during compilation better leverage the larger register pool and the different ISA towards a more reliable binary.

VII. DARKNET + YOLO SOFT ERROR ASSESSMENT CONSIDERING THE CRITICAL FUNCTION TECHNIQUE

This section explores the criticality of four core functions that are the most executed and also tightly coupled with the neural network engine, thus the object detection algorithm. Note that the injections are conducted only when the target function is within the processor context as described in [3], thus the Linux kernel is not affected. A convolutional network is a combination of, among other features, the number/order of the layers, weights, and their values. As such, the creation of these layers, performed by function F1 (**make_convolutional_layer**) is directly responsible for the network definition, and therefore, for the prediction results. Functions F2 and F3 (**add_bias**, **scale_bias**, resp.) are reward functions that increase or decrease the likelihood for prediction of a given object. As most tasks heavily rely on matrix multiplication (e.g., layers and weights) the function F4 (**compute_matrix_multiplication**) is also considered in this investigation.

Table IV shows that the most common result for both architectures are *Correct* (average of 82.11% (ARMv7) and 84.78% (ARMv8), lines 13/26); this results support the claim that neural networks (i.e., the image detection engine) are resilient to perturbations. However it also exposes the application dependency on control flow structures, thus leading to a higher number of *Crashes* (11.59% and 12.24%, for ARMv7

TABLE IV: Object Detection Soft Error Reliability Assessment: Function Evaluation. Average of all five images considering Int+FP registers.

			Detailed Inspection (%)				
			<i>Correct (%)</i>	<i>No Pred</i>	<i>Inc. Probs</i>	<i>Wrong</i>	<i>Crash</i>
1	GCC 4.9	F1	84.61	2.16	0.00	0.49	12.75
2		F2	77.72	1.00	3.43	6.72	11.13
3		F3	78.63	1.62	3.31	6.20	10.25
4		F4	86.30	0.05	1.35	0.91	11.40
5	GCC 7.3	F1	84.90	2.35	0.10	0.96	11.69
6		F2	76.08	1.42	3.26	7.16	12.08
7		F3	76.57	2.21	3.14	6.25	11.84
8		F4	87.25	0.05	1.74	1.10	9.85
9	Clang 6	F1	86.79	2.03	0.00	0.71	10.47
10		F2	78.63	0.32	2.84	3.97	14.24
11		F3	80.00	0.27	2.13	3.95	13.65
12		F4	87.82	0.02	1.69	0.69	9.78
13 AVG			82.11	1.13	1.92	3.26	11.59
14	GCC 4.9	F1	89.99	0.90	0.00	0.50	8.62
15		F2	83.55	0.66	1.56	2.84	11.38
16		F3	80.57	0.47	1.94	2.91	14.11
17		F4	86.98	0.09	0.64	0.36	11.93
18	GCC 7.3	F1	88.78	0.57	0.02	0.62	10.01
19		F2	81.85	0.66	1.16	3.91	12.43
20		F3	80.05	0.33	1.37	3.41	14.84
21		F4	85.68	0.05	0.83	0.47	12.97
22	Clang 6	F1	90.37	0.76	0.00	0.24	8.64
23		F2	81.25	0.33	1.37	1.82	15.22
24		F3	80.19	0.26	1.33	2.08	16.14
25		F4	88.05	0.02	0.83	0.50	10.60
26 AVG			84.78	0.43	0.92	1.64	12.24

and ARMv8 resp.). Further, the findings from Section VI are also observed for the critical functions: (i) ARMv8 has more occurrence of *Correct* results, lines 1–12/13 vs. 14–25/26; (ii) Clang is more reliable, presenting consistent lower values for *Incorrect* results (No Prediction, Incorrect Probabilities, and Wrong Detection), lines 1–8/14–21 vs. 9–12/22–25.

Considering the purpose of each function above mentioned, the distribution of *Incorrect* results are as expected: F1 has more occurrences of *No Prediction*, as the creation of the layers is a critical step to assure the expected behaviour of the application; F2 and F3 that compute the bias (i.e., confidence of object detection) predominantly have *Incorrect Probabilities* and *Wrong Detection*; F4 presents a distribution similar to F2 and F3, but with smaller absolute numbers.

Finally, results show that the number of *Incorrect* results and *Crash* for the critical functions (Table IV) is usually higher than the value found for the whole application (Table III, lines 3/6/9 and 12/15/18). This higher occurrence indicates that the chosen functions are indeed critical as they contribute to the occurrence of failures, thus they are also good candidates to hardening techniques.

VIII. FINAL REMARKS AND CONCLUSION

The use of virtual platforms enables engineers to boost the soft error analysis of authentic autonomous vehicles software stacks, considering not only compilers and ISAs but also state-of-the-art processors, which are rarely available to users. The

unmatched simulation performance of SOFIA (e.g., 10 to 100 thousand times faster than a register transfer level or gate-level simulator) enables us to produce a higher error percentage whenever compared to, for example, a netlist-based fault injector, as microarchitectural elements (e.g., flip-flops, wires, latches) have a more significant masking effect over faults than the adopted approach. This paper presented the soft error susceptibility assessment of a real automotive software stack running on unmodified Linux kernel, and considering multiple commercially available processors, and multiple cross-compilers. Results show that the Clang and ARMv8 provide better reliability (MWTF in Table III, and Table IV, lines 13 vs. 26). Further, the occurrence of faults in the chosen critical functions of target application has a significant impact in the overall application reliability as evidence by the higher occurrence of Crash and Incorrect results w.r.t. fault injection campaign on the whole application. Such analysis was only possible due to the robust fault injection framework of SOFIA, consisting of a bespoke fault classification and a fault injection technique that allows to isolate critical functions of the target automotive application.

REFERENCES

- [1] R. Shanker, A. Jonas, S. Devitt, et al., “Autonomous Cars: Self-Driving the New Auto Industry Paradigm”, Morgan Stanley Research, Technical Report, 2013.
- [2] T. Litman, “Autonomous Vehicle Implementation Predictions: Implications for Transport Planning”, Victoria Transport Policy Institute, Technical Report, 2018.
- [3] V. Bandeira, F. da Rosa, R. Reis, and L. Ost, “Non-intrusive Fault Injection Techniques for Efficient Soft Error Vulnerability Analysis”, VLSI-SoC, 2019.
- [4] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, “Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications”, SC, 2017.
- [5] F. Libano, P. Rech, L. Tambara, J. Tonfat, and F. Kastensmidt, “On the Reliability of Linear Regression and Pattern Recognition Feedforward Artificial Neural Networks in FPGAs”, IEEE Transactions on Nuclear Science, vol. 65, no. 1, pp. 288–295, 2017.
- [6] F. F. d. Santos, L. Draghetto, L. Weigel, L. Carro, P. Navaux, and P. Rech, “Evaluation and Mitigation of Soft-Errors in Neural Network-Based Object Detection in Three GPU Architectures”, DSN-W, 2017.
- [7] M. Didehban and A. Shrivastava, “nZDC: A compiler technique for near Zero Silent Data Corruption”, DAC, 2016.
- [8] N. Seifert, B. Gill, S. Jahinuzzaman, J. Basile, V. Ambrose, Q. Shi, R. Allmon, and A. Bramnik, “Soft Error Susceptibilities of 22 nm Tri-Gate Devices”, IEEE Transactions on Nuclear Science, vol. 59, no. 6, pp. 2666–2673, 2012.
- [9] H. Cho, S. Mirkhani, C.-Y. Cher, J. Abraham, and S. Mitra, “Quantitative evaluation of soft error injection techniques for robust system design”, DAC, 2013.
- [10] “Advanced driver assistance systems”, European Commission, Technical Report, 2018.
- [11] J. Redmon, “Darknet: Open Source Neural Networks in C”, <http://pjreddie.com/darknet/>, 2013.
- [12] J. Redmon and A. Farhadi, “YOLOv3: An Incremental Improvement”, CoRR, vol. abs/1804.02767, 2018.
- [13] A. Geiger, P. Lenz, and R. Urtasun, “Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite”, CVPR, 2012.
- [14] G. Reis, J. Chang, N. Vachharajani, R. Rangan, D. August, and S. Mukherjee, “Design and Evaluation of Hybrid Fault-Detection Systems”, ISCA, 2005.
- [15] H. Schirmeier, C. Borchert, and O. Spinczyk, “Avoiding Pitfalls in Fault-Injection Based Comparison of Program Susceptibility to Soft Errors”, DSN, 2015.
- [16] N. Binkert, B. Beckmann, G. Black, et al., “The gem5 simulator”, SIGARCH Comput. Archit. News, vol. 39, no. 2, pp. 1–7, 2011.