






Using Hardware-In-Loop-Based Fault Injection to Determine the Effects of Control Flow Errors in Industrial Control Programs

Jens Vankeirsbilck^(✉) , Hans Hallez , and Jeroen Boydens 

KU Leuven Bruges Campus, Bruges, Belgium
{jens.vankeirsbilck,hans.hallez,jeroen.boydens}@kuleuven.be

Abstract. Embedded systems, which are at the core of many small scale and large scale machines, are affected by external disturbances which can introduce control flow errors. These control flow errors can affect the control program executing on the embedded system, potentially causing sensor signals to be misinterpreted or actuators being miscontrolled. Software-implemented control flow error detection techniques have existed for many years, although there is little literature about these techniques being tested on input/output-driven programs. This paper presents a hardware-in-loop-based fault injection campaign performed on a typical industrial setting, i.e. a small scale factory. Thanks to hardware-in-loop simulation, we can perform the fault injection campaign without the risk of breaking a mechanical or an electrical part. For our fault injection campaign, we considered both the unprotected control program and the version protected with our RACFED error detection technique. The results show that up to 58% of the injected control flow errors can affect the unprotected control program in a dangerous manner. Implementing RACFED clearly lowers this percentage to less than 4%, showing this technique can be used in industrial settings.

Keywords: Hardware-in-the-loop simulation · Fault injection · Control flow error

1 Introduction

Today, industry is becoming more and more data-driven, also known as Industry 4.0 [8]. While the Internet-of-Things makes this shift possible, it also creates a much harsher working environment for embedded systems that are at the core of many small and large scale machines. By interconnecting all these machines, often using wireless communication, electromagnetic interference is now a major form of disturbance for those embedded systems [3, 5, 12]. Combined with other technology trends such as decreasing transistor feature sizes and lowering supply voltages, embedded systems are inherently more susceptible for external

disturbances [1, 7, 11, 13, 16, 24]. These disturbances range from electromagnetic interference and high-energy particles to temperature fluctuations and they may introduce bit-flips in the system’s hardware [10, 14]. In turn, these bit-flips can cause control flow errors (CFE), unwanted jumps in the system’s software. This can lead to misinterpreting sensor readings, erroneously controlling actuators or even crashing programs [9].

To protect embedded systems, many software-implemented CFE detection techniques have been proposed [2, 15, 18, 20, 22]. Such techniques add extra control variables and their update instructions to the target programs. At run time, the added instructions are executed and calculate a run-time value for the control variable. At certain points in the target program, the run-time value and the compile-time value of the control variable are compared to one another and a mismatch indicates a CFE has occurred and has been detected. However, these measures are often only validated using data-driven case studies, such as matrix multiplication, fast fourier transform or cyclic redundancy check calculations. To expand the validation of such techniques, we created a small scale factory that enables the validation of CFE detection mechanisms using an input/output-driven case study [23]. In that work, we merely performed a preliminary study on the effects of CFEs on the control programs of the small scale factory. This paper builds upon that work by proposing a hardware-in-loop-based (HIL) fault injection setup. This allows us to execute the control programs of the small scale factory in a simulated environment and enables an extensive fault injection study, without the risk of breaking mechanical or electrical parts.

The remainder of this paper is structured as follows. Section 2 describes the small scale factory and how a CFE detection technique is added to its control programs. Following, Sect. 3 presents the built fault injection setup. Next, the effects of the injected CFEs on those control programs are discussed in Sect. 4. Then, the drawback of this HIL simulation is described in Sect. 5. Finally, future work is presented in Sect. 6 and conclusions are drawn in Sect. 7.

2 Case Study

This section presents our small scale factory and discusses how its control programs are protected against CFEs by implementing our Random Additive Control Flow Error Detection (RACFED) technique.

2.1 Small Scale Factory

Our small scale factory consists of three stations from the Festo-Didactic MPS® series: a distribution station, a testing station and a sorting station [6]. Combined, they represent a closed process, in which workpieces are pushed out of a stacked magazine and transported to the testing area where only the good workpieces are moved to the final station, which in turn sorts them by color. In total, the small scale factory is able to distinct between six types of workpieces.

Regarding the color, it recognizes three types: silver, red and black, and for each color, there are correct workpieces and wrong workpieces.

As the stations' names imply, each station performs a part of that process. The setup is shown in Fig. 1, with the distribution station on the left, the testing station in the middle and the sorting station on the right. To drive each station and thus to execute the control program, we selected an NXP LPC 1768 which is an ARM Cortex-M3 driven microcontroller. We selected the ARM Cortex-M3 because it is an industry leading 32-bit processor. For more information about the functionality of each station and how the control programs are developed, the reader is referred to our previous work [23].

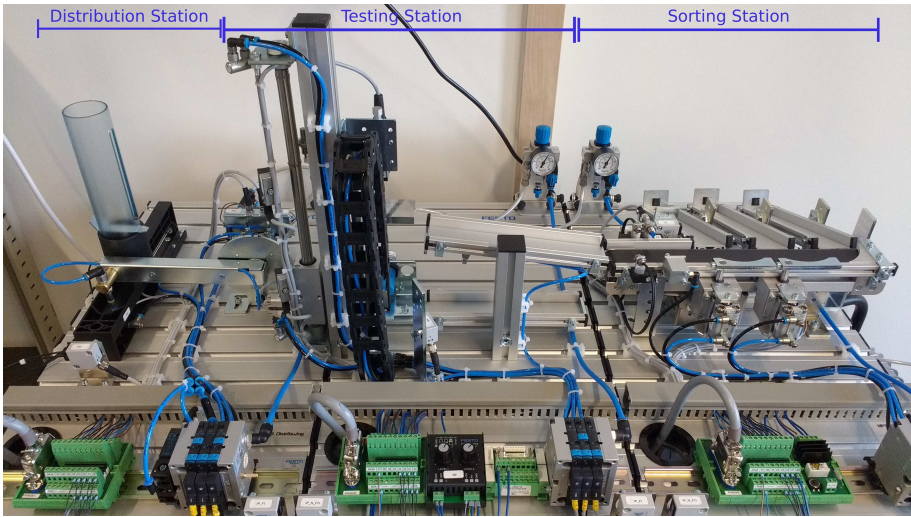


Fig. 1. The small scale factory. On the left the distribution station, in the middle the testing station and on the right the sorting station. The workpiece flow goes from left to right.

2.2 Adding CFE Detection

To make the control programs more fault tolerant, the first step is to apply a CFE detection technique. For this paper, we opted to implement our in-house developed RACFED technique [22]. RACFED detects CFEs by inserting a control variable and its update instructions in the target code. As shown in Fig. 2, RACFED is implemented in the assembly code of a control program, using a basic block as implementation unit. A basic block is a sequence of consecutive instructions with exactly one entry and one exit point. Together, basic blocks and edges which show the intentional paths between basic blocks, a program can be visualized in a control flow graph. In fact, Fig. 2 shows the control flow graph of a sample program, with RACFED implemented. Shown in the normal font are

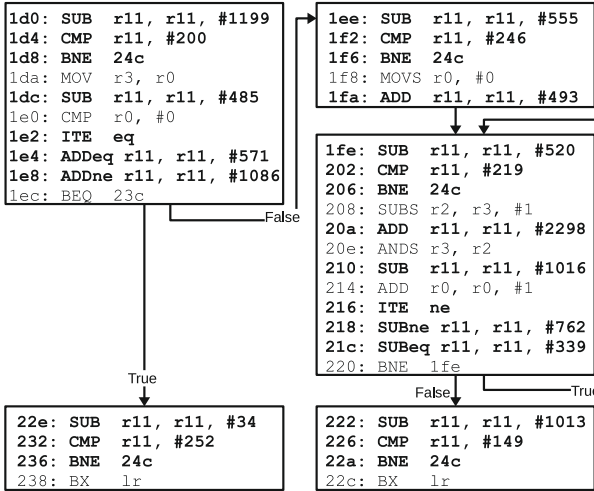


Fig. 2. RACFED implemented for a sample program in the assembly code.

the original program instructions and shown in bold are the instructions inserted by RACFED to detect CFEs, with the control variable held in register r11.

At the beginning of each basic block, a control variable update instruction is inserted. This update instruction is followed by a verification instruction that compares the run-time value of the control variable with its compile-time value. If there is a mismatch between the two values, a CFE is detected and control is transferred to an error handler, here located at address 0x24c. This error handler is defined by the user and is out of scope for this paper. Next, control variable update instructions are inserted after each non-branch instruction. The final control variable update instruction is executed conditionally when the basic block ends in a conditional branch.

Implementing RACFED in the assembly code of the control programs manually would be too time consuming and too error-prone. Therefore, we adjusted the compiler flags of the control programs to use our in-house developed GCC plugin that can automatically add the supported CFE detection techniques to the assembly code [21]. This GCC plugin supports up to ten CFE detection techniques, including RACFED, and currently two instruction set architectures, i.e. ARMv6-M and ARMv7-M. Figure 2 uses ARMv7-M as assembly code, since this is the instruction set of the selected NXP LPC 1768. For more information on RACFED or the GCC plugin¹, the reader is referred to [22] and [21], respectively.

Of course, adding RACFED to the control programs increases the instruction count of those programs, leading to an increase in code size and execution time. Although not important for the small scale factory, since memory is abundant

¹ Available as open-source project on <https://github.com/MGroupKULeuvenBrugesCampus/CFED.Plugin>.

and there are no deadlines to be met, we measured both types of overhead. The code size overhead was measured by using the GNU *size* tool on the compiled program, i.e. on the produced *.elf* file. This shows the amount of extra instruction memory needed to store the protected program, relative to the unprotected program. The execution time overhead is measured using an on-board hardware timer of the NXP LPC 1768 and shows the extra time it took for the protected control program to process one workpiece relative to the unprotected control program.

The measurement indicate that using RACFED to detect CFEs has increased the code size of the control programs by a factor of $\times 1.66$ for each station. Nonetheless, this has no impact on the execution time overhead. The time necessary to process one workpiece for the protected control program and the unprotected program is the same. This is because the control program is mainly waiting for the mechanical parts to have moved. During this wait, no instructions are executed, leading to the same execution time for the unprotected and the protected control program. These measurements show that the impact of implementing a software-based CFE detection methods depends from use case to use case.

3 Fault Injection Setup

In order to analyse the effects of CFEs on the control program of each of the stations, we have built a HIL-based fault injection setup. Hardware-in-loop means that the LPC 1768 is removed from its station and is plugged into a hardware simulation of its respective station. This hardware simulation will provide the necessary input signals to the LPC 1768 in order to execute the station control program, and it will analyse the output signals of the LPC 1768. We opted to perform fault injection using a HIL setup to avoid breaking mechanical or electrical parts of the actual stations, as we do not know all potential effects of CFEs on the control programs. This section will first discuss the hardware setup, later the software execution is presented, concluding with the executed experiments.

3.1 Architecture

To inject control flow errors in the control program of each station, we created the architecture illustrated in Fig. 3. As shown, the built setup has four major parts: a computer, a USB-hub, the target LPC 1768 and another LPC 1768 which executes the HIL code to simulate the sensors and actuators of the actual hardware.

Our in-house developed software-implemented fault injection (SWIFI) tool executes on a computer, which is connected to the on-chip debugger of the target LPC 1768 through a USB-hub [19]. Using the on-chip debugger, the SWIFI tool has access to the program counter register of the microcontroller. By injecting bit-flips in this register, CFEs are introduced into the control program. The target LPC 1768 is connected to the computer through the controllable USB-hub to enable hard-resetting the target. From time to time, the communication

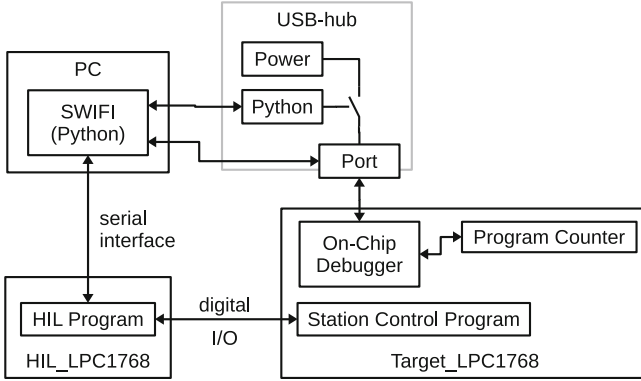


Fig. 3. The hardware setup built for the HIL-based fault injection campaign.

between the computer and the on-chip debugger of the target can get corrupted. To solve this, the SWIFI tool can issue to the USB-hub to power down and re-power the USB-port to which the target is connected. This power-cycling of the USB-port resets the target and enables to establish a new connection between the SWIFI tool and the on-chip debugger of the target.

To make sure that the station control program executes, we developed a HIL program that provides the necessary digital inputs to the target LPC 1768 and that analyses the received digital outputs send by the target. This HIL program also executes on an LPC 1768, indicated as HIL_LPC1768 in Fig. 3. To provide the necessary digital inputs and to analyse the digital outputs of the target, both LPC 1768 are connected through several digital I/O connections. To be able to report the status of the station control program, and to know when to start providing inputs to the target, the HIL LPC 1768 is connected to the computer, and in fact the SWIFI tool, through a serial interface.

3.2 Injecting a CFE

To inject a CFE the following steps are executed:

1. The SWIFI tool determines an origin program counter value for the CFE. Based on that origin value, a destination value is created by flipping a single-bit of the origin value. To make sure the destination value is valid for the current control program, the disassembly file of the control program is provided to the SWIFI tool. The disassembly file holds all valid program counter values for the current program and can thus be used by the tool to select a destination program counter value.
2. Once both the origin and destination of the CFE are selected, a thread is started to inject the defined CFE. This thread waits until the program counter holds the origin value and then corrupts it to the destination value.

3. Once the thread to inject a CFE is started, the SWIFI tool creates a second thread which sends the command to the HIL LPC to start the process of providing the necessary digital inputs to the target LPC to simulate a workpiece that must be processed. Once the HIL LPC has received this command, it will sequentially provide the necessary digital inputs and for each provided digital input, it will check to see if the target LPC provides the correct digital outputs.

Once both threads have executed, the effect of the injected CFE is analyzed. This is done by analyzing the state of the control program and by sending a command to the HIL LPC, which then sends back whether or not all digital outputs from the target LPC were correct. Based on the response of the HIL LPC and the state of the control program, the effect of the CFE is categorized in one of the following four categories:

- **Detected (Det):** The CFE is detected by a software-implemented CFE detection mechanism.
- **Hardware Detected (HD):** The CFE is detected by a hardware measure present in the LPC 1768. Many microcontrollers have error detection mechanisms implemented in their hardware. Such error detection mechanism enable the detection of improper bus usage, stack corruption, etc. This category indicates that the CFE was detected by such a hardware error detection mechanism.
- **Silent Data Corruption (SDC):** The CFE remained undetected and was able to corrupt the execution of the station control program. This means that the HIL LPC responded with an error code, indicating that while processing a simulated workpiece, the target LPC provided incorrect outputs. This is the most dangerous effect a CFE can have and should be avoided as much as possible.
- **No Effect (NE):** The CFE remained undetected but had no effect on the execution of the station control program. This is an indication of the inherent CFE resilience of the control program.

3.3 Executed Experiments

In this paper, we conducted two types of experiments for each station control program. The first type of experiment injected CFEs contained within a control program function. This means that both the origin program counter value and the destination program counter value belong to the same control program function. The results of this type of experiment will be indicated as *IntraFunc* for the remainder of this paper. The second type of experiment injected CFEs that jumped between two control program functions. In these experiments, the origin program counter value belongs to one function, while the destination program counter value belongs to another program function. The results of this type of experiment will be indicated as *InterFunc* in the following sections.

To analyze the effects of CFEs on the unprotected control programs, we injected 1000 *IntraFunc* and 1000 *InterFunc* CFEs for each control program

function. For the testing station, we executed these experiments twice: once simulating a correct workpiece and once simulating a wrong workpiece. Similarly, we performed these experiments three times for the sorting station: once simulating a silver workpiece, once simulating a red workpiece and finally, once simulating a black workpiece.

To determine the efficiency of RACFED, we repeated this fault injection campaign for the protected version of the control programs. To compensate for the increase in instructions, we injected 2000 IntraFunc and 2000 InterFunc CFEs for each protected control program function.

4 Impact of the Injected CFEs

The results of the fault injection experiments are shown in Fig. 4 and Fig. 5, in which WP stands for workpiece. The two figures show the results of the IntraFunc fault injection campaign and that of the InterFunc fault injection campaign, respectively. In dark-green, the faults detected by RACFED are indicated, in light-green the HD category is represented, the NE category is depicted in orange and because the SDC category represents the worst possible effect of a CFE, it is illustrated in dark-red.

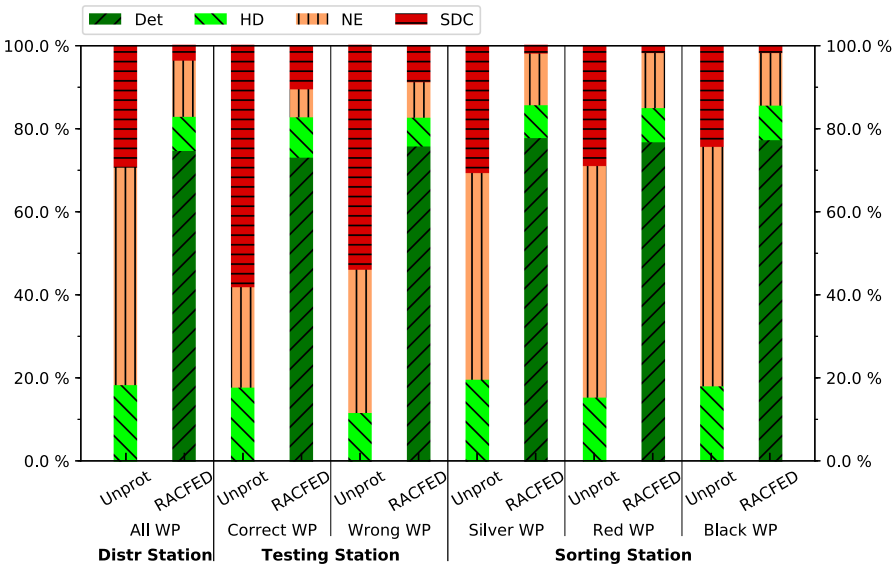


Fig. 4. Results of the IntraFunc fault injection campaign. (Color figure online)

When analyzing the IntraFunc results, it is clear that the unprotected control programs are vulnerable to these types of CFEs. Furthermore, the testing station shows to be even more sensitive to IntraFunc CFEs than the other two stations.

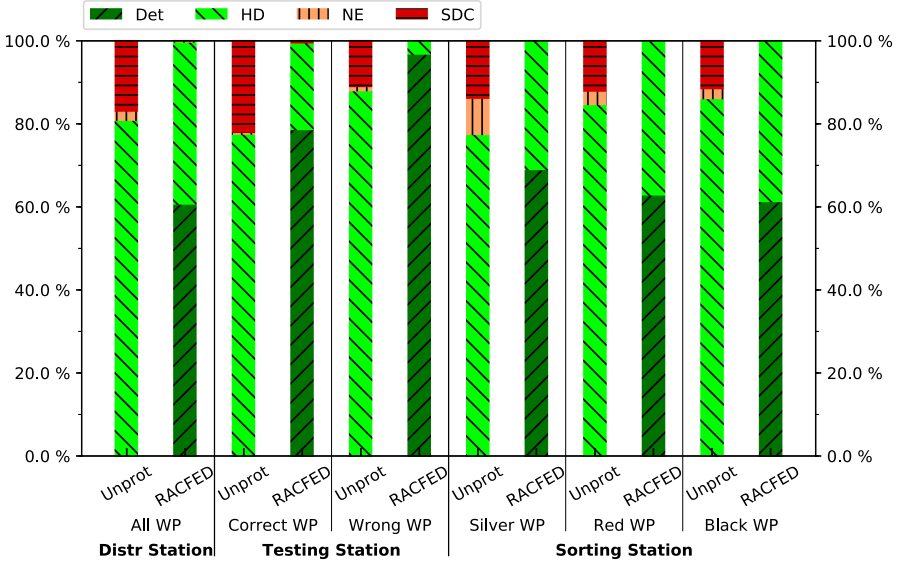


Fig. 5. Results of the InterFunc fault injection campaign. (Color figure online)

As shown in Fig. 4, the IntraFunc CFEs resulted in an SDC ratio of 29.4% for the distribution station and in an average SDC ratio of 28.0% for the sorting station, the testing station reports an average SDC ratio of 56.1%. The data does not show any discrepancies compared to the other two stations. We therefore conclude that the control program for the testing station is just more susceptible to IntraFunc CFEs than its distribution station and sorting station counterparts.

Regarding the global results of the fault injection campaign, the figures indicate that the IntraFunc CFEs resulted in a higher SDC ratio than the InterFunc CFEs for the unprotected control programs. This is due to the hardware error detection mechanisms present in the LPC 1768. A jump between two different control program functions is more likely to corrupt the stack or to be a larger jump, and the hardware error detection mechanisms are implemented to detect such occasions. This is shown in Fig. 5, as the HD category has a minimum value of 77.2%. Therefore, of all injected InterFunc CFEs, only 22.8% or less are not caught by a hardware error detection mechanism and are able to corrupt the workpiece processing. As shown in the figure, most of these InterFunc CFEs do corrupt the processing of a workpiece and are thus categorized as SDC. IntraFunc CFEs on the other hand are smaller jumps and less likely to cause stack corruption, so less likely to trigger the hardware error detection mechanism and thus remain undetected. This is shown in Fig. 4 in which the HD category has a maximum value of 19.7%. This means that 80% of the injected IntraFunc CFEs remain undetected and potentially be registered as SDC.

When looking at the protected control programs, RACFED detects most of the injected CFEs, achieving 60% or higher error detection ratio, and significantly reduces the SDC ratio for all stations, as expected. For the distribution station and the sorting station, the SDC ratio drops below 4% which is within the design limit of the RACFED technique. As explained in the literature, a technique will never reach an SDC ratio of 0% for IntraFunc CFEs, as there are always IntraFunc CFEs that can defeat the technique [22]. To give an example, consider again Fig. 2. A CFE originating at address `0x1d4` and landing at address `0x1dc`, skips the program instruction located at `0x1da` but does not skip a control variable update and thus remains undetected. While such InterFunc CFEs do exist, they are much more rare and thus a fault injection campaign can report an SDC ratio of 0% for these type of CFEs as is the case for the sorting station.

The IntraFunc results of the testing station differ from those of the other two stations. While RACFED detects 72% of the injected CFEs, the SDC ratio is still 10.5% which is high. Analysis of the data revealed that many of the CFEs causing SDC exploit the weaknesses of RACFED, such as the type of CFE mentioned in the previous paragraph. Another reason for the high SDC ratio is that, due to the memory layout of the program, multiple single-bit bit-flips of the program counter, result in the program jumping to a wait sequence causing an indefinite wait in the program. These wait sequences are used to make sure the mechanical parts are driven for the correct amount of time. When a CFE jumps to such a wait sequence, the waiting time is not initialized or the mechanical part is not actually driven, causing an indefinite wait in the program, which is then categorized as SDC. This shows that, although a CFE detection mechanism is implemented, CFEs can still have devastating effects on the control program.

5 Drawback of the Created Fault Injection Setup

Using the created HIL-based fault injection, we were able to perform an in-depth CFE study on the control programs for our small scale factory. This has revealed that 30% or more of the injected IntraFunc CFEs can corrupt the execution of these control programs, which is much more than our estimate from our preliminary study [23].

The numbers presented in this paper are, however, a *Worst Case* scenario since any deviation from the normal workpiece processing flow is categorized as an SDC. This process does not take the inherent error resilience of the station control programs into account. As described in Sect. 3.2, the created HIL simulation sequentially produces the necessary inputs for the target LPC1768 to correctly process one workpiece, independent from the output of the target. Once all signals to process one workpiece have been produced, no further signals are produced, regardless of the state of the station control program. In the actual small scale factory, however, the sensors might produce signals for the controlling LPC1768 indicating that something went wrong. In turn, the control program can react to those signals and try to correctly process the workpiece.

As an example of this inherent error resilience, consider that the testing station is processing a correct workpiece and that it is in the state **put correct workpiece on airslider**, as shown in Fig. 6. If due to a CFE, the ejection cylinder is not activated, the workpiece will remain on the lift table. Once the remainder of the code is executed, the station will be in the state **wait for workpiece**. Since the workpiece is still on the lift table, the HIL simulation would classify this behavior as an SDC. In the actual small scale factory, however, the workpiece detection sensor detects that the workpiece is still on the lift table and hence sends the signal *new workpiece*, causing the control program to process the workpiece correctly. This means that despite the CFE, the workpiece is processed correctly albeit with a delay. This shows the inherent error resilience of the control program.

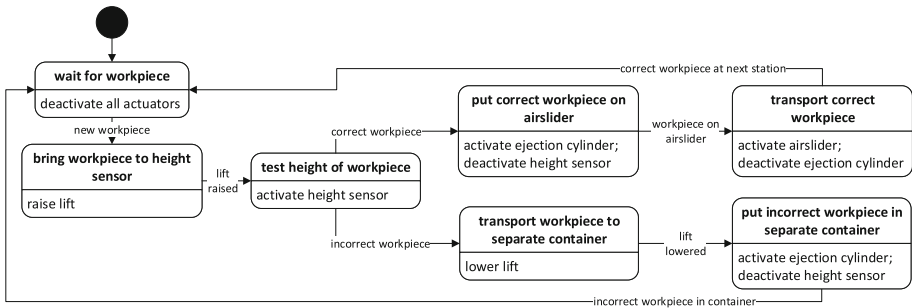


Fig. 6. Flow chart showing the functionality of the testing station.

To have a more realistic simulation of the hardware and to have more realistic fault injection results, another HIL simulator needs to be created. In this new HIL simulator, each major part of each stations should be simulated independently and be reactive to the inputs provided by the target LPC 1768. Where in the current HIL simulation the HIL LPC 1768 is the master and the target LPC 1768 is the slave, these roles should be reversed in the more realistic HIL setup. The current HIL setup has as advantage that detecting whether or not the workpiece was processed correctly is easy. This can now be done by analysing the outputs of the target LPC 1768 for each step in the sequential providing of the inputs. In the more realistic and reactive HIL simulation this would be more difficult, as each simulated part of the station needs to be analysed.

Moreover, such a reactive HIL simulation would allow to test our crude recovery method. As described in our previous work, we implemented a crude recovery method for each of the stations which, simply put, re-executes a certain part of the control program depending on when the CFE is detected. With our current HIL setup, we could not test this recovery mechanism, but with a more reactive HIL simulation this becomes possible. We are currently working on the architecture, hardware selection, etc., for this more reactive HIL simulation.

6 Future Work

As described in Sect. 3.2, non-valid single-bit bit-flip CFE destination values for the program counter are filtered from the pool of possible CFEs to inject. Analysis revealed that 0% to 30% of the single-bit bit-flip values are valid program counter values, i.e. those single-bit bit-flip values are program counter addresses valid for the target program. The other 70% to 100% are non-existing for the target program and CFEs using such values as destination address would be caught by the hardware error detection mechanisms. Although the injected CFEs result in silent data corruption, these values reveal that the program counter is not that sensitive to erroneous bit-flips introduced by external bit-flips.

Therefore, a further study will include bit-flips introduced in the other CPU registers, which are better known as data flow errors. Using the developed HIL setup, bit-flips can be injected in the remaining CPU registers to cause data flow errors risk free. Similarly, the effectiveness of data flow error detection techniques can be evaluated when applied to a more input/output-driven case study [4, 17]. Once known, the best performing data flow error detection technique can be combined with RACFED to develop a technique that is able to detect both control flow errors and data flow errors.

7 Conclusions

In this paper, we presented our HIL-based fault injection setup to be able to inject CFEs into the control programs of our small scale factory without the fear of braking anything. For our HIL simulation, we selected the NXP LPC 1768 as hardware platform and developed a HIL control program for each of the three stations in the small scale factory. This HIL control program sequentially provides the inputs to the target LPC 1768 to mimic a workpiece being processed correctly. During this sequential process, the outputs of the target are monitored. At the end, the HIL control program reports back to the fault injection framework whether or not the target LPC 1768 provided the correct outputs or not.

Once set up, we used the HIL-based fault injection setup to inject 1000 IntraFunc and 1000 InterFunc CFEs in each control program function. The results show that, when no CFE detection is present, up to 58% of the injected CFEs can result in the corruption of the processing of a workpiece. This means that due to the CFE, the target LPC 178 produced wrong outputs, which is here classified as a corruption of processing. To analyse the effect of adding a CFE detection technique, we implemented the RACFED technique in each of the control programs and then repeated the fault injection experiments, using 2000 CFEs of each type. Now, a minimum of 60% of the CFEs is detected and the corruption of processing a workpiece is reduced to less than 4% in most cases. This clearly shows the increase in resilience due to the CFE detection technique.

However, due to the sequential and nonreactive nature of the created HIL setup, the numbers shown in this paper represent the *Worst Case* scenario.

Each deviation of the normal procedure to process a workpiece is classified as dangerous. In reality, however, some CFEs are handled by the inherent error resilience of the control program and can result in the correct processing of the workpiece. Unfortunately, the built HIL simulator does not allow for the inherent error resilience to be executed. We are currently looking into new ways to implement a HIL simulator for the small scale factory that is reactive and does allow for this inherent error resilience to take place.

References

1. Abella, J., et al.: Towards improved survivability in safety-critical systems. In: 2011 IEEE 17th International On-Line Testing Symposium. pp. 240–245 (2011). <https://doi.org/10.1109/IOLTS.2011.5994536>
2. Choi, K., Park, D., Cho, J.: SSCFM: separate signature-based control flow error monitoring for multi-threaded and multi-core environments. *Electronics* **8**(2), 199 (2019). <https://doi.org/10.3390/electronics8020166>
3. Claeys, T., Catrysse, J., Pissoort, D., Arien, Y.: Stripline set-up for characterizing the effect of corrosion and ageing on the shielding effectiveness of emi gaskets with improved repeatability. In: 2018 International Symposium on Electromagnetic Compatibility (EMC EUROPE), pp. 725–729 (2018). <https://doi.org/10.1109/EMCEurope.2018.8485135>
4. Didehban, M., Shrivastava, A.: nZDC: a compiler technique for near zero silent data corruption. In: 2016 53rd ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 1–6. IEEE (2016)
5. Estep, N.A., Petrosky, J.C., McClory, J.W., Kim, Y., Terzuoli, A.J.: Electromagnetic interference and ionizing radiation effects on cmos devices. *IEEE Trans. Plasma Sci.* **40**(6), 1495–1501 (2012). <https://doi.org/10.1109/TPS.2012.2193600>
6. Festo-Didactic: Mps the modular production system. <http://www.festo-didactic.com/int-en/learning-systems/mps-the-modular-production-system/stations/>
7. Hashimoto, M., Liao, W.: Soft error and its countermeasures in terrestrial environment. In: 2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 617–622 (2020)
8. i-SCOOP: Industry 4.0: the fourth industrial revolution - guide to industry 4.0. <https://www.i-scoop.eu/industry-4-0/>
9. Ibe, E.H., et al.: VLSI design and test for systems dependability. *Radiation-Induced Soft Errors* (2019). https://doi.org/10.1007/978-4-431-56594-9_3
10. Jagannathan, S., et al.: Temperature dependence of soft error rate in flip-flop designs. In: 2012 IEEE International Reliability Physics Symposium (IRPS), pp. SE.2.1–SE.2.6 (2012). <https://doi.org/10.1109/IRPS.2012.6241927>
11. Kanekawa, N., Ibe, E.H., Suga, T., Uematsu, Y.: *Dependability in Electronic Systems: Mitigation of Hardware Failures, Soft Errors, and Electro-Magnetic Disturbances*. Springer, Heidelberg (2011). <https://doi.org/10.1007/978-1-4419-6715-2>, <https://www.springer.com/gp/book/9781441967145>
12. Kim, K., Iliadis, A.A.: Critical bit errors in cmos digital inverters due to pulsed electromagnetic interference. In: 2007 International Conference on Electromagnetics in Advanced Applications, pp. 217–220 (2007). <https://doi.org/10.1109/ICEAA.2007.4387276>

13. Riera, M., Canal, R., Abella, J., Gonzalez, A.: A detailed methodology to compute soft error rates in advanced technologies. In: 2016 Design, Automation Test in Europe Conference Exhibition (DATE), pp. 217–222 (2016)
14. Sierawski, B.D., et al.: Effects of scaling on muon-induced soft errors. In: 2011 International Reliability Physics Symposium. pp. 3C.3.1–3C.3.6 (2011). <https://doi.org/10.1109/IRPS.2011.5784484>
15. So, H., Didehban, M., Shrivastava, A., Lee, K.: A software-level redundant multi-threading for soft/hard error detection and recovery. In: 2019 Design, Automation Test in Europe Conference Exhibition (DATE), pp. 1559–1562 (2019). <https://doi.org/10.23919/DATE.2019.8715089>
16. Team, M.M.: International roadmap for devices and systems - 2018 update: More moore. Technical Report, IEEE IRDS (2018). https://irds.ieee.org/images/files/pdf/2018IRDS_MM.pdf
17. Thati, V.B., Vankeirsbilck, J., Penneman, N., Pissoort, D., Boydens, J.: An improved data error detection technique for dependable embedded software. In: IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC), pp. 213–220. IEEE (2018)
18. Tsai, T., Huang, J.: Source code transformation for software-based on-line error detection. In: 2017 IEEE Conference on Dependable and Secure Computing, pp. 305–309 (2017). <https://doi.org/10.1109/DESEC.2017.8073852>
19. Vankeirsbilck, J., Cauwelier, T., Van Waes, J., Hallez, H., Boydens, J.: Software-implemented fault injection for physical and simulated embedded CPUs. In: IEEE XXVII International Scientific Conference Electronics (ET), pp. 1–4 (2018). <https://doi.org/10.1109/ET.2018.8549630>
20. Vankeirsbilck, J., Penneman, N., Hallez, H., Boydens, J.: Random additive signature monitoring for control flow error detection. *IEEE Trans. Reliab.* **66**(4), 1178–1192 (2017). <https://doi.org/10.1109/TR.2017.2754548>
21. Vankeirsbilck, J., Hallez, H., Boydens, J.: Automatic implementation of control flow error detection techniques. In: Accepted at IASED International Conference on Wireless Networks and Embedded Systems (ICWNES) (2019)
22. Vankeirsbilck, J., Penneman, N., Hallez, H., Boydens, J.: Random additive control flow error detection. In: Gallina, B., Skavhaug, A., Bitsch, F. (eds.) SAFECOMP 2018. LNCS, vol. 11093, pp. 220–234. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99130-6_15
23. Vankeirsbilck, J., Van Waes, J., Hallez, H., Pissoort, D., Boydens, J.: Control flow errors in an industry 4.0 setup: a preliminary study. In: IEEE International Conference on Systems, Man and Cybernetics (SMC), pp. 2305–2310 (2019). <https://doi.org/10.1109/SMC.2019.8914545>
24. White, M., Chen, Y.: Scaled cmos technology reliability users guide. Technical Report 20100014217, National Aeronautics and Space Administration (NASA) (2010). https://nepp.nasa.gov/files/16361/08_102_4%20new%20del.White.pdf