Automatic Implementation of Control Flow Error Detection Techniques

Jens Vankeirsbilck*, Hans Hallez and Jeroen Boydens
Dept. of Computer Science
KU Leuven Bruges Campus
Spoorwegstraat 12, 8200 Brugge, Belgium
jens.vankeirsbilck@kuleuven.be

ABSTRACT

Modern embedded systems are prone to erroneous bit-flips introduced in its hardware by external disturbances such as alpha particles, electromagnetic interference or intentional external attackers. In order to protect embedded systems against these disturbances, a wide variety of software-implemented detection techniques have been proposed, a.o. by the authors of this paper. Implementing those techniques, however, can be arduous and error-prone since they have to be implemented in low-level code, e.g. assembly. To overcome this problem we propose a compiler extension, in the form of a plugin, that can automatically add any supported technique to the low-level code of the target program. We discuss the internal working of our compiler extension and conclude with a demonstration using an example program and validate the effectiveness of the introduced countermeasures by running a fault injection campaign.

CCS Concepts

•**General and reference** → **Reliability;** •**Computer systems organization** → **Reliability;** *Embedded software; Redundancy;* •**Software and its engineering** → **Compilers;**

Keywords

Automatic Implementation, Compiler Extension, GCC Plugin, Control Flow Error, Software-Implemented Error Detection

1. INTRODUCTION

The reliability of embedded systems in ever harsher working environments is becoming ever more important, especially in safety-critical domains such as medical, automotive, avionics, etc. These systems are, however, vulnerable to external disturbances ranging from high energy particles striking the hardware to electromagnetic interference and intentional attackers [19, 7, 14, 8]. These external disturbances create extra charges into the struck hardware component, causing a bit-flip. In turn, such a bit-flip can corrupt data, cause a jump through the executing program or even wrongly control an actuator [12, 15].

A jump through the target program is better known as a control flow error (CFE). A CFE is a violation against the control flow graph (CFG) of the program. The CFG is the representation of a program's execution order using basic blocks and edges. A basic block is a list of branch-free instructions following each other. This means that a basic block has exactly one entry and one exit point. An edge represents a valid path between two basic blocks. A CFE can be one of two types: either an inter-block CFE or an intra-block CFE. An inter-block CFE is an invalid jump between two different basic blocks, while an intra-block CFE is an invalid jump within the same basic block. Both types of CFE can lead to hazardous situations by causing the affected program or system to halt, to crash or to provide erroneous output.

To increase the reliability of embedded systems, several software-implemented CFE detection techniques have been proposed [18, 10, 1, 16, 5, 6, 17, 21, 22]. Such techniques add extra control variables at compile time. At run time, these control variables are calculated and compared to the expected compile-time value. A mismatch between both of these values indicates that an error has occurred. Depending on which type of CFE the technique was designed to detect, intra-block and/or inter-block CFEs, the introduced control variables are more frequently or less frequently updated.

Implementing a CFE detection technique can prove to be difficult, because they only achieve their reported error detection ratio when implemented in low-level code, e.g. assembly. As we describe further, this is mainly due to the optimisation of the compiler when translating high-level code, e.g. C++, to low-level code. To solve this problem, we propose a compiler extension which allows to automatically add the instructions of a CFE detection technique to the low-level code. The advantages of our compiler extension are not only that it can use all internal states created by the compiler, such as the CFG, but moreover it reduces the implementation time and effort.

The remainder of this paper is structured as follows. Section 2 provides more background about the need for a com-

*This work is supported by a research grant from the Baekeland program of the Flemish Agency for Innovation and Entrepreneurship (VLAIO) in cooperation with Televic Healthcare NV, under grant agreement IWT 150696.

ICWNES '19 July 26–28, 2019, Rome, Italy

DOI: 10.475/123_4

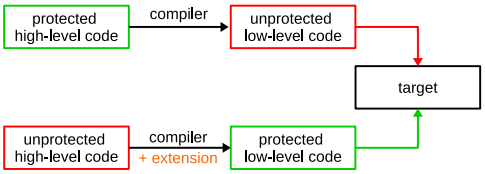

Figure 1: The problem with high-level protection and the solution provided by our compiler extension.

piler extension and Section 3 explains the internal working of the selected compiler, i.e. GCC. Our implementation of the compiler extension, i.e. a GCC plugin, is discussed in Section 4. Next, Section 5 shows how to use our plugin with an example algorithm. Following, Section 6 presents the performed experiments. Finally, future work is presented in Section 7 and conclusions are drawn in Section 8.

2. BACKGROUND

Literature describing software-implemented CFE detection techniques use high-level language instructions to explain the working of their technique and only provide a high-level implementation example. This could lead an embedded systems engineer to believe such techniques have to be implemented in high-level code, e.g. C++. When doing this however, and taking the appropriate measures to assure the compiler does not optimize away the added instructions, experiments have shown that the CFE detection techniques detect around 65 % of the occurring CFEs. In contrast, literature describes an error detection ratio of 75 % and higher [18, 10, 1, 16, 5, 6, 17, 21, 22]. This mismatch in error detection ratios is caused by the fact that human-readable high-level code is not mapped one-to-one to machine-readable low-level code. A high-level instruction is often mapped to multiple low-level instructions and the many compiler optimizations often generate a completely different CFG for the low level code than the CFG constructed for high-level source code. Fig. 1 visualizes how the compiler produces unprotected low-level code from the protected high-level code.

The solution to this problem is implementing the CFE detection techniques in low-level code. Performing this manually, however, is arduous and error-prone. Therefore, we propose a compiler extension that can automatically implement a variety of software-implemented CFE detection techniques in the low-level code of the program. We support both pre-existing techniques [18, 10, 1, 16, 5, 6, 17] and our in-house developed techniques [21, 22]. As Fig. 1 shows, the compiler extension allows to create protected low-level code from unprotected high-level code. We selected the GCC toolchain for bare-metal ARM development *arm-none-eabi-gcc* as compiler and as an extension, we developed a plugin. To the best of the author's knowledge, this is the first paper proposing a GCC plugin that enables the automatic implementation of a variety of CFE detection techniques in low-level code.

Before discussing the structure of our plugin, the internal working of GCC and the plugin execution point are described.

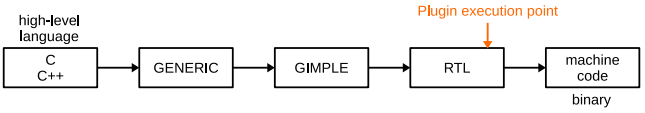

Figure 2: The different intermediate languages of GCC and the plugin execution point.

3. GCC INTERNAL WORKING

GCC compiles high-level source code to low-level code through several intermediate representations, as depicted in Fig. 2 [20]. First high-level language is translated into GENERIC, then into GIMPLE, followed by a translation into REGISTER TRANSFER LANGUAGE (RTL) to finally end in MACHINE CODE. This process is done via several passes. A pass is a set of instructions that perform a specific part of the compilation process, e.g. dead code removal, building the CFG or loop optimization. As shown, GCC does not generate assembly code, but gradually transforms the high-level code to machine-level code using several intermediate representations. Therefore, our plugin will interact with one of those intermediate representations to implement the CFE detection techniques.

Our plugin executes after *pass_free_cfg*, which is an RTL pass and is only executed once per compiled function of the program. This pass indicates that the CFG will not change anymore during the further compilation process, making it the ideal point for the plugin to insert extra instructions. As shown in Fig. 2, this pass is at the end of the compilation process which forces us to work with hardware registers instead of RTL pseudo-registers. Since each supported CFE detection technique needs one or more registers to be implemented, these must be reserved during compilation using the GCC option *-ffixed-r<number>* .

4. OUR GCC PLUGIN

The abstract execution flow of our plugin is as follows. The first time our plugin is executed, it registers itself as a compilation pass. Once registered, it will be executed for each program function GCC compiles. For each function, the plugin determines if it needs to execute or not, depending on the provided information. Which function to protect and which function to leave unprotected is up to the user of the plugin, as this is very application dependent. For some applications it might be necessary to protect all defined functions, while for others it might be sufficient to only protect a critical section of the program. If this analysis finds that the current function must be protected, the selected technique is implemented.

4.1 Implementation

The implementation of our plugin is shown in more detail in Fig. 3. The shown UML class diagram depicts the classes and methods to implement a CFE detection technique. Our plugin supports the following CFE detection techniques: CFCSS [18], YACCA (2 versions) [10], ECCA [1], RSCFC [16], SEDSR [5], SCFC [6], SIED [17], RASM [21] and RACFED [22]. However, to keep the diagram in Fig. 3 as clear as possible, only two of all the supported techniques are shown. The `Plugin` class contains two methods that are used by GCC to execute our plugin and all other classes contain the methods to effectively implement the selected

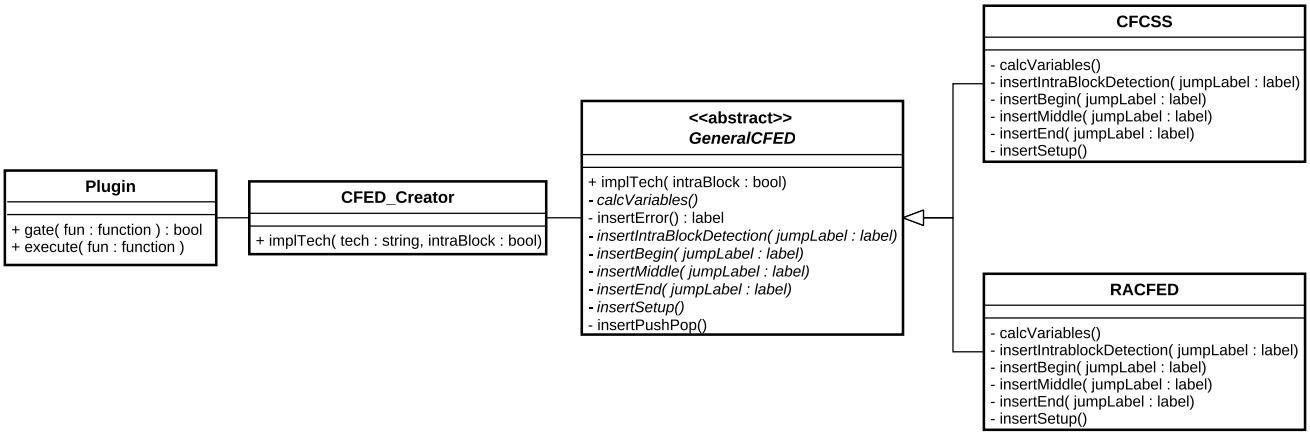

Figure 3: UML class diagram showing the implementation of our plugin is pseudo-code.

Algorithm 1 Pseudo-code describing the high-level execution flow of the GCC plugin.

```
1: registerPass()
2: for all functions do
3:   if gate( ) is True then
4:     intraBlock ← readArgument(techniqueType)
5:     technique ← readArgument(technique)
6:     cfedCreator ← new CFEDcreator()
7:     cfedCreator.implTech(intraBlock, technique)
```

technique for the current function.

Once the plugin has registered itself as a compilation pass, the `gate` method of the `Plugin` class is called for each function. This method checks whether or not the current function needs to be protected with a CFE detection technique. This is determined by reading the *function* plugin argument and verifying if the function attribute `noProtection` is set. If the *function* argument specifies one function, then only that function is protected. When the *function* argument is empty, all functions are protected unless the user has set the `noProtection` function attribute. The `gate` method returns either True or False indicating whether or not the function needs to be protected.

When the `gate` method returns True, the `execute` method is called. This method contains the actual compilation pass and will implement the selected CFE detection technique, as shown in Algorithm 1. It determines whether or not intra-block CFE detection should be implemented by reading the *techniqueType* plugin argument and it determines which of the supported techniques should be implemented by reading the *technique* plugin argument. Finally, it uses those two arguments to call the `implTech` method of the `CFEDcreator` class.

The `CFEDcreator` is a class that knows which CFE detection techniques are supported and how to build them. As Algorithm 2 shows, first the instruction set architecture (ISA) of the target processor is determined. With this information, the correct registers to be used can be provided to the technique that must be implemented. At the moment, two ARM ISAs are supported: ARMv6-M and ARMv7-M [2, 4]. These are the most used ISAs in the envisioned embedded

Algorithm 2 Pseudo-code describing the flow to implement a technique in the `CFEDcreator` class.

```
1: function CFEDcreator::implTech(intraBlock,
                                  technique)
2:   isa ← getIsaTarget()
3:   GeneralCFED genCFED
4:   switch technique do
5:     case "CFCSS" :
6:       genCFED ← new CFCSS(isa)
7:     case "RACFED" :
8:       genCFED ← new RACFED(isa)
9:     default :
10:      raise error("Requested technique not supported!")
11:  genCFED.implTech(intraBlock)
```

systems. Next, it creates an instance of the selected techniques and finally calls its `implTech` method. As the name gives away, that method makes sure the selected technique is implemented for the function.

The `implTech` method of the `GeneralCFED` class implements the selected technique and is shown in Algorithm 3. First all necessary variables, e.g. signatures and other auxiliary compile-time variables, are calculated in the `calcVariables` method. Next, the local error branch to the general error handler is inserted. Because recovering from a CFE is too application specific, we let its implementation to the user. Different applications or application domains often require a different strategy once a CFE has been detected. Possible options are 1) transitioning to a safe state, 2) doing a system reset, 3) starting an automatic recovery method or 4) halting and triggering an alarm signal or escalating to a higher level. All we define, is the handler name which is CFED_Handler. For each function, the `insertError` method adds a branch to the CFED_Handler function and returns a label to itself. That label is used in the remaining `insert`-methods to jump to the local error branch. Next, the technique itself is implemented. First the intra-block CFE detection instructions are inserted if needed. Secondly, the instructions that need to be added in the middle of the basic block are added. Thirdly, the instructions to be added

Algorithm 3 Pseudo-code describing the procedure to implement the different instructions of the selected technique.

```
1: function GeneralCFED::implTech(intraBlock)
2:   calcVariables()
3:   jumpLabel ← insertError()
4:   for all basic blocks in the CFG do
5:     if intraBlock is True then
6:       insertIntraBlockDetection(jumpLabel)
7:     insertMiddle(jumpLabel)
8:     insertBegin(jumpLabel)
9:     insertEnd(jumpLabel)
10:  insertSetup()
11:  insertPushPop()
```

at the beginning of each basic block are inserted, ending with adding the necessary instructions at the end of the basic block. Next, the `insertSetup` method adds the setup procedure of the technique to the beginning of the first basic block. This setup procedure makes sure that the needed registers are filled with the expected values to allow a correct verification of the run-time variables in the first basic block of the CFG. Finally, the `implTech` method ends by calling the `insertPushPop` method, which adds PUSH and POP instructions to place the necessary run-time variables on and remove them from our run-time variable stack. To make sure the regular stack remains valid, this run-time variables stack is a second stack next to the regular stack of the used microcontroller.

Although the `GeneralCFED` class implements the `implTech` method and thus is in control of the execution order of the `calcVariables` and `insert`-methods, their implementation is provided in the specific classes of the supported techniques, as indicated in the bottom Fig. 3. As depicted, the specific technique classes `RACFED` and `CFCSS` implement the needed methods but nothing more [22, 18]. This makes it easy to support new techniques, as they only need to implement the six abstract methods defined by the `implementTechnique` function of the `GeneralCFED` class.

4.2 The Need For insertPushPop

As discussed, the `insertPushPop` method is used to store and restore the value of the run-time variables. This is needed when multiple functions of a program have to be protected or when a recursive function has to be protected. Illustrating this with an example, consider functions `f1` and `f2` of Fig. 4. These are two functions from one fictitious program and `f1` calls `f2`, indicated with the `BL` instruction. When both functions have to be protected and no PUSH - POP sequence is implemented, the situation above the black separator line is created. Function `f1` assigns the value of 10 to the run-time variable held in register `r11` and verifies it at the end. Function `f2` does the same, but with the value of 25. In this situation, `f1` will always detect a false CFE once `f2` has been called. As can be seen, `r11` is 10 when calling `f2`, but has the value of 25 once `f2` has executed. Next, `r11` is verified in function `f1` and that verification falsely detects a CFE and would call the appropriate error-handler.

Our solution to this problem is inserting a PUSH instruction at the beginning of each function and a POP instruction at the end of each function. These instructions store the value of the run-time variables and allows to restore them.

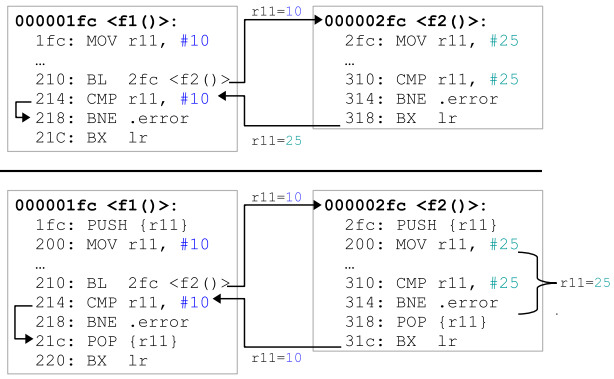

Figure 4: ARMv7-M code showing the need for extra push and pop instructions.

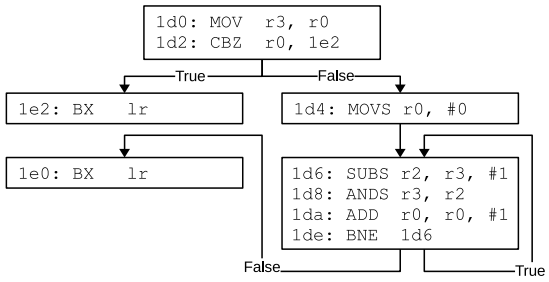

Figure 5: The CFG of the bit_count algorithm when compiled for an ARM Cortex-M3 using *arm-none-eabi-gcc7.3*

Applied to the example, the situation below the black separator line of Fig. 4 is now created. Both functions now start by pushing the current value of the run-time variable held in register `r11` to the stack, and end by popping that value from the stack and storing it back into `r11`. As can be seen, the run-time value of 25 is now local to function `f2` and the value of 10 is restored back into `r11` once `f2` has executed. Since `r11` now matches the expected value within `f1`, its verification instruction does not detect a CFE and execution resumes as expected.

In order not to corrupt the stack used by the program, we defined a second stack and PUSH run-time variables to and POP them from this second stack. Practically, this can be realised by adjusting the *linker*-script and startup procedure of the used microcontroller.

5. EXAMPLE USAGE

To show the usage and outcome of our plugin, we'll use the bit_count (BC) algorithm of the MiBench benchmark suite [11]. The BC algorithm counts the numbers of bits set in a given code word. When compiled with the *arm-none-eabi-gcc7.3* toolchain for an ARM Cortex-M3, which uses the ARMv7M ISA, it has the CFG shown in Fig. 5. It is a small algorithm, containing nine instructions, and therefore has a rather small and easy CFG which can be manually validated for correct execution.

```
1 extern "C"{
2     void __attribute__((noProtection))
      CFED_Handler(void){
3         while(1);
4     }
5 }
```

Listing 1: Adjustment to be made to the source code to use our plugin

5.1 Needed Adjustments

To use our plugin, both the source code and the compiler flags have to be adjusted. As shown in Listing 1, the `CFED_Handler` function has to be defined in the source code. As discussed in the previous section, this handler is the function that will be executed once a CFE has been detected. When working with C++, it is necessary to define the handler in an `extern "C"` environment, as shown on the first line. This tells the compiler to keep the function name as defined and not to mangle it, as is often the case with C++ functions. Name mangling is the encoding of function and variable names into unique names and is most commonly used to facilitate the overloading feature and to facilitate visibility within different scopes. We do not want this for our `CFED_Handler` function, hence the need for the `extern "C"` environment. The handler can have our `noProtection` function attribute set to tell our plugin the function must not be protected. In our example we implement the `CFED_Handler` function as an infinite loop instructing the target to wait there, but any alternative functionality can be provided.

The changes to be made in the compiler flags are shown in Listing 2. In total, six extra compiler flags have to be added, i.e. four plugin-related flags and two global compilation flags. The global compilation flags, `-ffixed-r11` and `-ffixed-r7`, tell the compiler not to use registers `r11` and `r7` during compilation. The plugin will use the first register to implement the chosen technique and the latter as stack pointer for the run-time variable stack. We selected register `r7` as stack pointer, due to limitations of the ARMv6-M ISA. For ARMv6-M, register `r7` is the highest register that can be used as stack pointer and to keep the common part between the different supported ISAs as large as possible, we defined register `r7` as the stack pointer for all supported ISAs. We selected register `r11` to implement the chosen technique because it is the highest general purpose register that can be reserved with the `-ffixed-r<number>` option.

The four plugin-related flags specify where to find the plugin (line 4), which function to protect (line 6), what type of technique to apply (line 8) and which technique to implement (line 10). In our example, the flags have been set to only protect the BC algorithm itself, i.e. `function=bit_count`, and to implement the RACFED technique with both intra-block and inter-block CFE detection, i.e. `techniqueType=fullCFED, technique=RACFED` [22].

5.2 Protected Example

Using the adjusted source code and compiler flags, compiling the BC algorithm with the *arm-none-eabi-gcc7.3* toolchain for an ARM Cortex-M3, now generates the CFG presented in Fig. 6. Indicated in bold are the instructions added

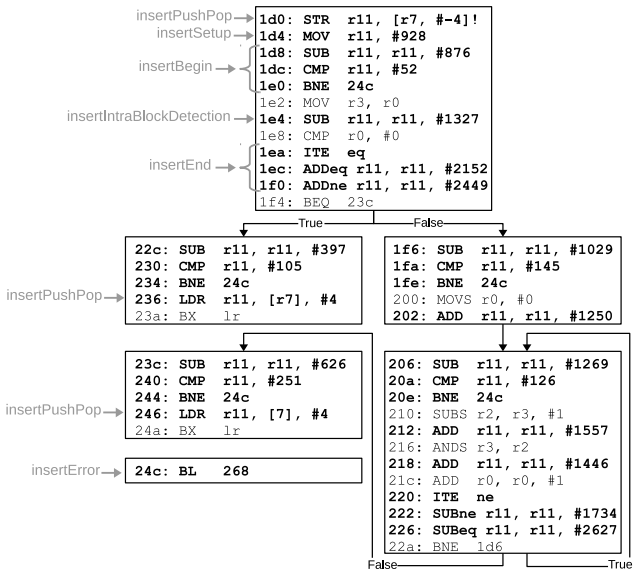

Figure 6: The CFG of the bit_count algorithm when compiled for an ARM Cortex-M3 using *arm-none-eabi-gcc7.3* and our plugin

by our plugin with the needed run-time variable stored in register `r11`. For the first basic block and the two exit blocks, the two basic blocks containing the `BX lr` statement, we indicated which plugin method inserted which instructions in light-gray.

The first instruction of the bit_count algorithm is now the PUSH of the run-time variable. As indicated by the compiler flags, register `r7` is used as stack pointer and is used to store the run-time variable stored in register `r11` (`STR`). Next, the setup procedure has been inserted. For the chosen RACFED technique, this means storing a specific value in `r11` (`MOV`). Since the plugin argument `techniqueType` specified that both intra-block and inter-block CFE detection must be inserted, the first and lower-right basic blocks have the necessary intra-block detection updates inserted. For RACFED, these are additions or subtractions between the run-time variable and a random value. These are the instructions located at addresses `0x1e4`, `0x212` and `0x218`. Then, the instructions of the `insertBegin` method are inserted, which in the case of RACFED are three instructions. First, an update of `r11` is inserted (`SUB`), then the run-time verification between the run-time variable and its compile-time value is inserted (`CMP`) and finally the branch to the local error handler in case an CFE has occurred is inserted (`BNE 24c`). As shown in Algorithm 3, the local error handler is added to the function by the `insertError` method. In this example, it is added at address `24c` and calls the function located at `0x268`, which is the `CFED_Handler` function. Next, the instructions of the `insertEnd` method are inserted, which in case of RACFED is a final update of the run-time variable. To conclude the final instruction of the bit_count algorithm, before exiting, is now the POP of the run-time variable (`LDR`).

```
# 1) Define the plugin name
PLUGIN_NAME = CFEDplugin
# 2) Specify where to find the plugin
COMPILER_FLAGS += -fplugin=<pathToPlugin>/$(PLUGIN_NAME).so
# 2) Only protect the bit_count algorithm
COMPILER_FLAGS += -fplugin-arg-$(PLUGIN_NAME)-function=bit_count
# 3) Implement both inter-block and intra-block CFE detection
COMPILER_FLAGS += -fplugin-arg-$(PLUGIN_NAME)-techniqueType=fullCFED
# 4) Implement RACFED
COMPILER_FLAGS += -fplugin-arg-$(PLUGIN_NAME)-technique=RACFED
# 5) RACFED needs one register (r11)
COMPILER_FLAGS += -ffixed-r11
# 6) The run-time variable stack needs a stack pointer (r7)
COMILER_FLAGS += -ffixed-r7
```

Listing 2: Adjustment to be made to the compiler flags to use our plugin

6. EXPERIMENTS

This section presents experiments performed to validate the working of the plugin. First the experiment setup is described, then the results are shown.

6.1 Experiment Setup

To prove the validity of our plugin, we implemented the RACFED technique for 8 case studies both manually in high-level code and using the plugin in low-level code. As case studies we selected the following algorithms: BC, bubble sort (BS), cyclic redundancy check (CRC), cubic function solver (CU), Dijkstra's algorithm to find the shortest path (DIJ), fast fourier tranform (FFT), matrix multiplication (MM) and quick sort (QS). Many of the selected implementations are based on the MiBench benchmark suite [11]. These case studies were selected because they are highly used in the embedded systems domain, are often used in the literature to validate CFE detection techniques and have varying CFGs to allow a thorough validation of our plugin.

Next, we performed a fault injection campaign using our in-house built fault injection tool to test the error detection capabilities of the implementations. The used fault injection process gradually steps through the target program and injects all possible CFEs for each program step. We repeated this process five times for each case study, as we provided five different input datasets per case study. As hardware target, we selected a simulated ARM Cortex-M3 [13].

For each injected fault, we categorized its effect in one of the following four categories:

- **Detected (Det.):** This category represents the faults that were detected by the implemented CFE detection technique. So, for this paper, these are the faults detected by RACFED.
- **Hardware Detected (HD):** Current microcontrollers have several hardware mechanisms that detect for instance improper bus usage or stack corruption. This category represents the faults that were detected by such mechanisms.
- **Silent Data Corruption (SDC):** These are the faults that were not detected, neither by RACFED nor by the hardware and corrupted the outcome of the case study. This is the group of faults that needs to be minimized.
- **No Effect (NE):** This final category represents the faults that were not detected but did not corrupt the output of the case study.

6.2 Results

The results of the fault injection campaign are shown in Fig. 7. For each of the eight case studies, we have three variants: HL_NoVol, HL_WithVol and LL_Plugin. The HL_NoVol variant represents the high-level implementation of RACFED without taking any measures to avoid the compiler optimize away the instructions of the technique. HL_WithVol represents the high-level implementation of RACFED with taking appropriate measures to make sure the compiler keeps the necessary instructions. These two variants are called NoVol and WithVol because using the C++ keyword `volatile` when defining the needed variables proved sufficient to prevent the compiler from removing the instructions inserted to implement RACFED. Finally, LL_Plugin represents the low-level implementation of RACFED using the GCC plugin and is indicated with the orange label.

The green part of each bar represents the faults detected by RACFED, thus the Det. category and the red part represents the non-detected errors that corrupted the output, thus the unwanted and dangerous SDC category. The HD and NE categories are not in our control, and are therefore shown together in the gray part of the bar.

Analyzing the chart in Fig. 7, it can be seen that when no measures are taken to prevent the compiler optimizing away the implemented technique (HL_NoVol), no CFEs are detected. A small exception occurs for the FFT case study in which some of the protecting instructions remained and were able to detect 1 % of the injected CFEs. When comparing the HL_WithVol and LL_Plugin variants, the chart shows that the low-level implementation detects more errors than the high-level implementation for each case study. While the high-level implementation error detection ratio varies between 55 % and 75 %, the low-level implementation error detection ratio varies between 70 % and 86 %. Next to the error detection ratio, the SDC ratio is important as it represents undetected CFEs that were able to corrupt the output of the case study. For this category, the low-level implementation outperforms the high-level implementation for seven out of eight case studies with a varying SDC ratio between 0 % and 12 %. In contrast, the high-level implementation SDC ratio ranges from 1 % to 23 %.

To summarize, the experiments show that the low-level implementation outperforms the high-level implementation.

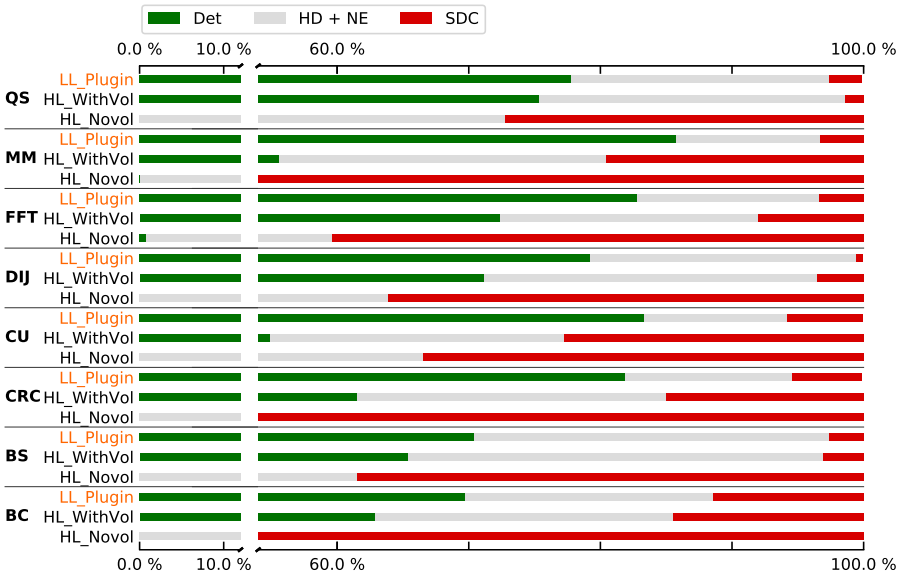

Figure 7: The results of the fault injection campaign.

On average, the low-level implementation has an error detection ratio of 79 % and an SDC ratio of 4 %, while the high-level implementation has an average detection ratio of 65 % and an SDC ratio of 11 %.

7. FUTURE WORK

Next to the academic case studies, we also applied the plugin on an industrial case study, i.e. a small scale factory. The small scale factory consists of three stations from the Festo-Didactic MPS® series: a distribution station, a testing station and a sorting station [9]. 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. Each station is controlled by an ARM Cortex-M3. Using the GCC plugin, approximately 480 functions are protected per station. Thanks to the plugin, only the changes listed in Listings 1 and 2 are needed. Protecting each station manually would be arduous and error-prone, while using the plugin makes it possible with little effort. Further details about the small scale factory and a preliminary fault injection study are currently under review [23].

The plugin has been mainly used to protect code on microcontrollers, such as the ARM Cortex-M0 and ARM Cortex-M3. Currently, we are adding support for the ARMv7-A ISA, which is used in application processors such as the ARM Cortex-A7 [3]. A first validation for this added support will be to use the plugin on bare-metal code executing in the trusted-world, next to an untrusted-world, using the TrustZone support on an ARM Cortex-A7.

In our research group, research is also being performed on software-implemented data flow error (DFE) detection. A data flow error is the corruption of data due to an erroneous bit-flip. In this research track, focus is being given to developing new and better software-implemented detection techniques and to the automatic implementation of those techniques in the form of another GCC plugin. Once this second GCC plugin has been thoroughly validated, the goal is to merge both plugins into one plugin. This single plugin can then provide a more complete protection against erroneous bit-flips, i.e. implement both CFE and DFE detection techniques and their error handlers to specify a recovery mechanism.

8. CONCLUSIONS

In this paper, we discussed our compiler extension which is able to implement a variety of software-implemented CFE detection techniques. First, the need for such a compiler extension was shown. When implementing the CFE detection techniques in high-level code, they detect approximately 65 % of the occurring CFEs. This problem arises due to the mapping of high-level code to machine-level code, which is not a one-to-one mapping. Therefore, software-implemented CFE detection techniques must be implemented in low-level code. Since performing this manually is arduous and error-prone, a compiler extension is needed.

Next, we presented our implementation of such compiler extension in the form of a GCC plugin. Our plugin works on a low-level intermediate language of GCC, called RTL, and can implement the following methods: CFCSS, YACCA, ECCA, RSCFC, SEDSR, SCFC, SIED, RASM and RA-CFED.

Then, the internal working of the compiler extension was shown and discussed using on the bit_count algorithm of the MiBench benchmark suite. Finally, we demonstrated that the low-level implementation of a CFE detection method always achieves a higher error detection ratio and a lower silent data corruption ratio when compared to the high-level implementation using fault injection experiments on eight representative case studies.

9. REFERENCES

[1] Z. Alkhalifa, V. S. Nair, N. Krishnamurthy, and J. A. Abraham. Design and evaluation of system-level checks for on-line control flow error detection. *IEEE*

Transactions on Parallel and Distributed Systems, 10(6):627–641, 1999.

[2] ARM, 110 Fulbourn Road Cambridge, England CB1 9NJs. *ARMv6-M Architecture Reference Manual*, ddi 0419d edition, May 2017.

[3] ARM, 110 Fulbourn Road Cambridge, England CB1 9NJs. *ARMv7-A and ARMv7-R Architecture Reference Manual*, ddi 0406c.d edition, March 2018.

[4] ARM, 110 Fulbourn Road Cambridge, England CB1 9NJs. *ARMv7-M Architecture Reference Manual*, ddi 0403e.d edition, June 2018.

[5] S. A. Asghari, A. Abdi, H. Taheri, H. Pedram, S. Pourmozaffari, et al. SEDSR: soft error detection using software redundancy. *Journal of Software Engineering and Applications*, 5(09):664–670, 2012.

[6] S. A. Asghari, H. Taheri, H. Pedram, and O. Kaynak. Software-based control flow checking against transient faults in industrial environments. *IEEE Transactions on Industrial Informatics*, 10(1):481–490, 2014.

[7] S. Baffreau, S. Bendhia, M. Ramdani, and E. Sicard. Characterisation of microcontroller susceptibility to radio frequency interference. In *Proceedings of the Fourth IEEE International Caracas Conference on Devices, Circuits and Systems (Cat. No.02TH8611)*, pages I031–1–I031–5, 2002.

[8] R. De Keulenaer. Softwarebeveiliging van smartcards tegen laseraanvallen. Master's thesis, Universiteit Gent, 2013.

[9] Festo-Didactic. Mps the modular production system.

[10] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante. Soft-error detection using control flow assertions. In *Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on*, pages 581–588. IEEE, 2003.

[11] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. IEEE, 2001.

[12] E. H. Ibe, S. Yoshimoto, M. Yoshimoto, H. Kawaguchi, K. Kobayashi, J. Furuta, Y. Mitsuyama, M. Hashimoto, T. Onoye, H. Kanbara, H. Ochi, K. Wakabayashi, H. Onodera, and M. Sugihara. *VLSI Design and Test for Systems Dependability*, chapter Radiation-Induced Soft Errors. Springer Japan, 2019.

[13] Imperas. Revolutionizing embedded software development. Online, 2018.

[14] S. Jagannathan, Z. Diggins, N. Mahatme, T. D. Loveless, B. L. Bhuva, S. J. Wen, R. Wong, and L. W. Massengill. Temperature dependence of soft error rate in flip-flop designs. In *2012 IEEE International Reliability Physics Symposium (IRPS)*, pages SE.2.1–SE.2.6, April 2012.

[15] M. Kishani, M. Tahoori, and H. Asadi. Dependability analysis of data storage systems in presence of soft errors. *IEEE Transactions on Reliability*, 68(1):201–2015, Jan 2019.

[16] A. LI and B. Hong. Software implemented transient fault detection in space computer. *Aerospace science and technology*, 11(2):245–252, 2007.

[17] B. Nicolescu, Y. Savaria, and R. Velazco. SIED: Software implemented error detection. In *Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on*, pages 589–596. IEEE, 2003.

[18] N. Oh, P. P. Shirvani, and E. J. McCluskey. Control-flow checking by software signatures. *IEEE transactions on Reliability*, 51(1):111–122, 2002.

[19] B. D. Sierawski, R. A. Reed, M. H. Mendenhall, R. A. Weller, R. D. Schrimpf, S. J. Wen, R. Wong, N. Tam, and R. C. Baumann. Effects of scaling on muon-induced soft errors. In *2011 International Reliability Physics Symposium*, pages 3C.3.1–3C.3.6, April 2011.

[20] R. M. Stallman and the GCC Developer Community. *GNU Compiler Collection Internals for GCC version 7.3.0*. Free Software Foundation, 2017.

[21] J. Vankeirsbilck, N. Penneman, H. Hallez, and J. Boydens. Random additive signature monitoring for control flow error detection. *IEEE Transactions on Reliability*, 66(4):1178–1192, Dec 2017.

[22] J. Vankeirsbilck, N. Penneman, H. Hallez, and J. Boydens. Random additive control flow error detection. In B. Gallina, A. Skavhaug, and F. Bitsch, editors, *Computer Safety, Reliability, and Security*, pages 220–234, Cham, 2018. Springer International Publishing.

[23] J. Vankeirsbilck, J. Van Waes, H. Hallez, D. Pissoort, and J. Boydens. Control flow erros in an industry 4.0 setup: a preliminary study. In *accepted at IEEE SMC*, 2019.