PhD Thesis: Advancing Control Flow Error Detection Techniques for Embedded Software using Automated Implementation and Fault Injection

Thesis · January 2020

CITATIONS
3

READS
729

1 author:

Jens Vankeirsbilck
KU Leuven
58 PUBLICATIONS **312** CITATIONS

SEE PROFILE

KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Technology

Advancing Control Flow Error Detection Techniques for Embedded Software using Automated Implementation and Fault Injection

ing. Jens Vankeirsbilck

Supervisors:
Prof. dr. ing. J. Boydens
Prof. dr. ir. H. Hallez

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Technology (PhD)

January 2020

Advancing Control Flow Error Detection Techniques for Embedded Software using Automated Implementation and Fault Injection

ing. Jens VANKEIRSBILCK

Examination committee:
Prof. dr. ir. M. Vergauwen, chair
Prof. dr. ing. J. Boydens, supervisor
Prof. dr. ir. H. Hallez, supervisor
Prof. dr. T. Holvoet
Prof. dr. V. Naessens
dr. ir. N. Penneman
Prof. dr. ir. K. De Bosschere
(Universiteit Gent, Belgium)
Prof. dr. C. W. Fetzer
(Technische Universität Dresden, Germany)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Technology (PhD)

January 2020

Uitgegeven in eigen beheer, ing. Jens Vankeirsbilck, Spoorwegstraat 12, bus 7923, B-8200 Brugge (Belgium)

Preface

After graduating my M.Sc. in Engineering Technology at the KU Leuven Technologycampus Ostend, prof. dr. ing. Jeroen Boydens asked me if I was interested in doing research and pursuing a PhD. After meetings with other PhD researchers on campus, dr. ing. Dries en dr. ing. Tim (now both holders of the PhD degree and colleagues), and hesitating for a while, I decided to roll the dice and accept Jeroen's offer. Today, nearing the end of a five year pursuit, I can say I could not have made a better decision. I have thoroughly enjoyed the research topic of *Resilient Embedded Software* , the wisdom and freedom given to me by my supervisors Jeroen and prof. dr. ir. Hans Hallez, and the general working environment both at the former campus in Ostend and at the new campus in Bruges.

First of all I want to thank my supervisors, Jeroen and Hans, not only for having given me the opportunity to pursue a PhD degree but also for the countless efforts they have made to help me. These efforts include, but are not limited to, scientific guidance, proofreading publications, discussions and encouragements.

Next to my supervisors, my deepest gratitude goes out to Televic Healthcare NV who co-funded years 2 through 5 of my PhD research through a Baekeland mandate (IWT 150696). Next to all the colleagues in Izegem, I want to specifically thank dr. ir. Niels Penneman and dr. ir. Pieter Crombez for their guidance during the course of the PhD. Niels, this thesis and the general research would not have reached the current level without your knowledge on the ARM architecture and without you sharing your experiences from your pursuit to a PhD. I also want to thank you for being part of my supervisory committee. Pieter, discussions with you always broadened my view on the research and have without doubt lifted the various results of the research to a higher level.

I would also like to thank prof. dr. Vincent Naessens and prof. dr. Tom Holvoet to be part of my supervisory committee. Together with the supervisory

committee, I am very grateful to prof. dr. ir. Maarten Vergauwen, prof. dr. ir. Koen De Bosschere and prof. dr. Christof W. Fetzer to be part of the examination committee and for their valuable feedback on the thesis.

This five year journey would not have been as enjoyable as it was now, without the colleagues at Ostend and Bruges. Babu, Jonas, Sotirios, Kristof, Bozheng, Chandu, Dejana and Piet, thank you very much for the many enjoyable coffee breaks, exquisite culinary trips to the kebab place, interesting meetings and entertaining social events. Piet, thank you for showing me the ropes when I first started in October 2014. Thanks to you I was able to get started quickly and was the transition from master student to PhD researcher more smoothly. While all colleagues contributed to the pleasant environment, a special mention goes out to Jonas Van Waes. Jonas, you deserve a special mentioning and a special token of gratitude, in the first place for listening to me when I had to vent some of my frustrations, but also for accompanying me to events, to drive me around with your car, and last but not least to be my friend.

I would not be the person I am today without the support of my family and friends. Special thanks to my parents to have raised me to the person I am now, for their support during these years and for providing me with the time and space to pursue my PhD, both literally and figuratively. A sincere thank you towards my friends, for having stayed my friends even though I was often mentally absent during our get-togethers.

Finally, I would like to thank the many students, both master students of KU Leuven Bruges Campus as well as students who did an international internship at the campus, for their help in developing different parts of this research.

January 2020
Jens Vankeirsbilck

Abstract

This thesis focuses on the selection and implementation of software-implemented countermeasures designed to detect control flow errors in embedded systems. A control flow error is an erroneous jump throughout an executing program induced by external disturbances. These disturbances, such as electromagnetic interference, can introduce bit-flips in different components of a system's hardware. In turn, these introduced bit-flips affect the executing program by corrupting the execution order of instructions. This phenomenon is known as a control flow error and can cause the program to hang or to crash, possibly creating dangerous situations. An introduced bit-flip can also manifest itself as a data flow error, by corrupting data needed by the program. These are, however, out of scope for this research.

By adding extra control variables and inserting update instructions that modify that control variable in the low-level code of the target program, software-implemented techniques are able to detect if a control flow error has occurred. Since multiple options are possible to create this type of protection, numerous techniques have been proposed in literature. With many options, and no guideline on how to select a technique, the following question arises: what is the best technique? To solve this question, solutions to the following problems had to be found: I) ease the implementation of the techniques in the low-level code of a target program; II) objectively characterize each technique; and III) develop a new and better technique.

To solve the first problem, we developed a compiler extension. While it is possible to implement each of the selected techniques into the low-level code of a target program manually, this is arduous and error-prone. The compiler extension we developed solves these issues as it is capable of automatically implementing the discussed techniques in low-level code. By simply adding a few extra parameters when compiling the target program, a control flow error detection technique can be added. This eliminates both the need to know the low-level language of the embedded system and the need to know about the

internal operations and added functionality of the technique. Using the compiler extension thus saves time and effort.

Next, we defined three criteria to objectively characterize each technique: 1) error detection ratio, 2) execution time overhead and 3) code size overhead. The error detection ratio indicates which percentage of control flow errors a technique detects. To measure this, we use fault injection experiments. Because there were no fault injection tools and no deterministic control flow error injection processes available, we developed our own software-implemented tool and processes. This tool can execute three different deterministic injection processes and supports multiple targets, both physical hardware targets and simulated targets. The execution time overhead indicates how much longer the protected program needs, compared to the unprotected program, in an error-free run. We measured this using an on-board hardware timer of the target embedded system. Finally, the third criterion, code size overhead, indicates how much more memory the protected program needs, compared to the unprotected program. This criterion is determined by measuring how much memory the compiled program needs.

Using the developed tools and selected criteria, a comparative study between eight established control flow error detection techniques is presented in this thesis. By implementing the techniques for the same case studies, executing them on the same hardware, subjecting them to the same fault injection campaign and measuring their overhead with the same tools, an objective comparison was made. The study revealed that the technique called *Control Flow Checking by Software Signatures* is the best established technique to use so far, as it achieves a high error detection ratio while imposing a low overhead.

The study also revealed that there was room for improvement. Using the collected data, we derived five guidelines to build an optimal control flow error detection technique. To demonstrate their validity, we developed a detection technique that complies with all five guidelines, called *Random Additive Control Flow Error Detection*, and submitted it to the same fault injection campaign as used during the aforementioned comparative study. These experiments revealed that our technique outperforms the selected state-of-the-art techniques. Our technique achieves a higher error detection ratio and imposes a lower overhead then the state-of-the-art techniques.

This thesis concludes by presenting the application of the different research outputs on industrial case studies, such as a small scale Industry 4.0 setup. These final experiments verify that the research can indeed be used in an industrial setting.

Beknopte samenvatting

Deze thesis handelt over het selecteren en implementeren van softwaregebaseerde technieken die ongewenste sprongen doorheen een programma kunnen detecteren. Deze ongewenste sprongen treden op nadat een of meerdere geheugenlocaties van een ingebed systeem getroffen worden door een storing van buitenaf. Zo een storing, vb. electromagnetische interferentie, kan een opgeslagen bit doen omslaan in waarde. Dit kan tot gevolg hebben dat de volgorde waarin instructies worden uitgevoerd, wijzigt. Dit fenomeen uit zich in een ongewenste sprong doorheen het uitvoerende programma en kan leiden tot een vastgelopen of gecrashet programma.

Softwaregebaseerde detectietechnieken voegen extra controlevariabelen en extra instructies die deze controlevariabelen veranderen toe aan de low-level code van een programma om ongewenste sprongen te detecteren. Omdat er meerdere mogelijkheden zijn om dit gedrag te bekomen, zijn er dan ook al vele detectietechnieken voorgesteld in de literatuur. Dit roept de vraag op: wat is nu de beste techniek? Om het antwoord op deze vraag te vinden, moesten drie problemen overkomen worden: I) de implementatie van de technieken in de low-level code van een programma vereenvoudigen; II) elke techniek objectief karakteriseren; en III) een betere techniek ontwikkelen.

Als oplossing voor het eerste probleem is er een compileruitbreiding ontworpen. Hoewel het mogelijk is om elke techiek manueel aan de low-level code van een programma toe te voegen, is dit proces tijdrovend en leidt veelal tot foutieve implementaties. De ontwikkelde compileruitbreiding lost dit probleem op, daar deze in staat is om elk van de gekozen technieken automatisch toe te voegen aan de low-level code. De gebruiker hoeft dan slechts enkele extra parameters mee te geven aan het compileerproces om een detectietechniek toe te voegen. Dit betekent dan ook dat een gebruiker geen kennis meer nodig heeft van de low-level syntax van het gekozen ingebedde systeem, noch over de interne werking van de geselecteerde techniek.

Daarnaast werden drie criteria gedefinieerd om elke techniek objectief te karakteriseren: 1) het foutdetectiepercentage, 2) de extra benodigde uitvoeringstijd en 3) het extra benodigde codegeheugen. Het foutdetectiepercentage geeft weer welk percentage van de ongewenste sprongen een techniek kan detecteren. Om dit te bepalen, gebruiken we foutinjectie experimenten. Omdat er geen foutinjectie raamwerken en geen deterministische foutinjectie processen beschikbaar waren, hebben we deze zelf ontwikkeld. Ons raamwerk beschikt op dit moment over drie deterministische foutinjectie processen en ondersteunt meerdere ingebedde systemen, zowel fysieke hardware systemen als gesimuleerde systemen. De extra uitvoeringstijd geeft een indicatie over hoeveel langer het beschermde programma nodig heeft om uit te voeren wanneer er geen fout optreedt, vergeleken met het originele, onbeschermde programma. Om dit te meten hebben we gebruik gemaakt van een hardware timer die aanwezig was op de fysieke hardware systemen. Het laatste en derde criterium, extra codegeheugen, duidt aan hoeveel meer geheugen de code van het beschermde programma nodig heeft, vergeleken met het originele, onbeschermde programma.

Gebruikmakend van de ontwikkelde hulpmiddelen en de geselecteerde criteria presenteert deze thesis een vergelijkende studie uitgevoerd op acht reeds bestaande detectietechnieken. Door deze technieken te implementeren voor gelijke gevalstudies, ze uit te voeren op gelijke hardware, ze gelijke foutinjectie experimenten te laten ondergaan, en hun extra uitvoeringstijd en extra codegeheugen op gelijke wijze op te meten, kan er een objectieve vergelijking gemaakt worden. In deze studie komt de techniek *Control Flow Checking by Software Signature* als de beste bestaande techniek naar boven. Dit omdat de techniek een hoog foutdetectiepercentage koppelt aan een lage extra uitvoeringstijd en laag extra codegeheugen.

Uit de studie bleek ook dat er nog ruimte was voor verbetering. Gebruikmakend van de verzamelde data, hebben we vijf vereisten opgesteld om een ideale detectietechniek te ontwikkelen. Om hun kwaliteiten aan te tonen, hebben we een techniek ontwikkeld die aan alle vijf vereisten voldoet. Deze techniek, genaamd *Random Additive Control Flow Error Detection*, hebben we dan onderworpen aan dezelfde foutinjectie experimenten zoals gebruikt tijdens de hiervoorgenoemde vergelijkende studie. Deze experimenten tonen inderdaad aan dat deze techniek beter presteert dan de bestaande technieken. Onze techniek haalt een hoger foutdetectiepercentage, en heeft zowel minder extra codegeheugen als minder extra uitvoeringstijd nodig dan de andere besproken technieken.

Deze thesis sluit af met enkele toepassingen van het onderzoek op industriële gevalstudies, waaronder een Industrie 4.0 machine op kleine schaal. Deze laatste experimenten tonen aan dat het onderzoek en zijn resultaten effectief gebruikt kunnen worden in een industriële omgeving.

List of Abbreviations

ACS	Abstract Control Signatures
ALU	Arithmetic Logic Unit
BC	Bit Count
BS	Bubble Sort
CFC	Control Flow Checking
CFCSS	Control Flow Checking by Software Signatures
CFE	Control Flow Error
CFG	Control Flow Graph
CRC	Cyclic Redundancy Check
CU	Cubic Solver
DFE	Data Flow Error
DIJ	Dijkstra's algorithm
ECC	Error Correction Code
ECCA	Enhanced Control Flow Checking using Assertions
EMI	Electromagnetic Interference
FFT	Fast Fourier Transform
GCC	GNU Compiler Collection
HD	Hardware Detected Error
HIL	Hardware-in-the-Loop

ISA	Instruction Set Architecture
MM	Matrix Multiplication
NE	No Effect Error
PC	Program Counter
QS	Quicksort
RACFED	Random Additive Control Flow Error Detection
RASM	Random Additive Signature Monitoring
RSCFC	Relationship Signatures for Control Flow Checking
RTL	Register Transfer Language
S-RACFED	Selective Random Additive Control Flow Error Detection
S-SETA	Selective Software-only Error detection Technique using Assertions
SCFC	Software-based Control Flow Checking
SDC	Silent Data Corruption
SEDSR	Soft Error Detection using Software Redundancy
SETA-C	Software-only Error detection Technique using Assertions minus Checkers
SIED	Software-Implemented Error Detection
SOTA	State-Of-The-Art
SWIFI	Software-Implemented Fault Injection
TA	Trusted Application
TEE	Trusted Execution Environment
YACCA	Yet Another Control Flow checking using Assertions
YACCA_CMP	The YACCA technique using simple comparisons

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Research Objectives

Embedded systems are being used more and more, with their application domain ranging from infotainment systems to Industry 4.0 machinery and safety-critical systems. This increase in usage has been driven by the technological trends to decrease the feature size of transistors and to lower the supply voltages. By decreasing the feature size of transistors, more transistors can be put on a similar area, creating more powerful and often faster microcontrollers. Microcontrollers, and microprocessors, are now able to perform complex tasks at lightning speeds. Lowering the supply voltages has prolonged the battery life of embedded systems, which has driven the deployment of principles like the Internet of Things and Industry 4.0 [1].

Despite the many advantages, those technological trends have one major disadvantage as they have increased the inherent vulnerability of embedded systems against erroneous bit-flips. Such erroneous bit-flips, also known as soft errors, can be introduced by many different external sources [2]. Research shows that particles, such as alpha particles, neutrons and muons, have the energy to strike microcontroller components, such as internal memory cells, and change their state [3–7]. Different sources from literature show that the soft error rate of different devices increases for each new technology generation [4–6]. While White et al. show that with the scaling down of the features sizes from 180 nm to 90 nm the soft error rate increases 8 % per bit, Kanekawa et al. even estimate that the soft error rate for 22 nm devices could be trice the soft error rate of 90 nm devices! Ibe et al. compiled a list of recent failure reports by

Table 1.1: Failure reports by terrestrial neutrons in various industries. *This is a summary of the compiled list of Ibe et al.(Table 3.3 of [8])*

Field	Application	Failure symptom
Avionics	fly-by-wire	reboot
Railway	transistors	out-of-service
Automotive	brake-by wire	no stop / sudden stop
	power steering	stuck / unexpected rotation
	engine control	sudden acceleration / no operation
	pedestrian detection	missing pedestrians
Super computer	calculations	unrecognizable wrong calculations
Networking	server	data corruption / reboot
	router	address change / reboot
	power supply	out-of-service
Personal Devices	smartphone	freeze / mail address corruption

terrestrial neutrons in various industries [8]. Table 1.1 is a summary of their compiled list (Table 3.3 of [8]) and the reader is referred to the full source for detailed information and references to the reported failures. As can be seen, both safety-critical domains such as avionics, railway, and automotive and mission-critical domains such as networking and super computing are affected by soft errors. Even infotainment systems, e.g., smartphones and tablets, can fail due to soft errors.

Next to particles, a second source of bit-flips is electromagnetic interference (EMI), which accumulates charges on PCB traces, on transistors, etc., and changes their state [9–11]. EMI is becoming a greater reliability concern, as more systems are being used close to each other and wireless communication is being increasingly used. Jagannathan et al. demonstrate that the sensitivity of transistors and memory cells increases with temperature. They show that a higher microcontroller temperature increases the effect generated by a striking particle or by EMI and decreases the drive strength of transistors used in the components [12]. And, although not the scope of this thesis, research from the security domain also shows that external attackers can introduce bit-flips in order to extract critical data, such as PIN codes of smart cards and cryptographic keys [13, 14]. The combination of these sources has increased the number of erroneous bit-flips to more than one per day for each device and thus makes them a real concern for all embedded system engineers [8]. These reliability issues have also been recognized by the IEEE International Roadmap for Devices and Systems (IRDS™) 2018 edition. In its *More Moore* report, they list:

> Reliability is an important requirement for almost all users of integrated circuits. The challenge of realizing the required levels of reliability is increasing due to (1) scaling, (2) new materials and devices, (3) more demanding mission profiles (higher temperatures, extreme lifetimes, high currents), and (4) increasing constraints of time and money. (More Moore Team, p. 18 [15])

Furthermore, in Table MM-14 they list *Reliability of embedded electronics in extreme or critical environments* as a near-term (2018 - 2025) reliability issue and *Muon induced soft error rate* as a long-term (2026-2034) reliability issue for electronics, and therefore for embedded systems. This shows that the whole electronics and embedded systems community is aware of the reliability problems concerning the components and devices posed by external disturbances.

The introduced bit-flip(s) can manifest itself as a control flow error (CFE) or as a data flow error (DFE). A CFE is the corruption of the execution order of instructions and a DFE is the corruption of input, intermediate or output data. Both types of errors can cause an actuator to be controlled erroneously and potentially dangerously, or cause the system to hang or crash. While this is undesired in any application, CFEs and DFEs are of major concern for safety-critical systems. A safety-critical system is a system of which a failure results in either death or serious injury to people, loss or severe damage to equipment or property, or harm to the environment. These systems are often regulated through laws and their development has to adhere to international standards to make sure the devices are as safe as reasonably possible. Such laws and standards help to identify potential reliability or safety risks and provide guidelines and requirements on how to mitigate those risks. Examples of these laws and standards are European Medical Device Regulations, IEC 62304 for medical device software, ISO 26262 for automotive systems, EN 50128 for railway system software and IEC 61508 as a more general standard [16–20]. A safety-critical system can only be put on the market when compliance with the imposed laws and standards can be shown.

In order to mitigate the effect of an erroneous bit-flip, and thus to increase the reliability of embedded systems, hardware-based countermeasures used to be the go-to protection mechanism. An example of such a countermeasure is Triple Modular Redundancy in which three copies of the same module and a voting system are used to drive a system [21]. While effective, hardware-based countermeasures represent a high cost to implement as they need extra components, power and room. A second disadvantage is that they are not flexible and cannot be changed once the system has been deployed. A third disadvantage is that hardware and thus also hardware-based fault tolerance measures suffer from wear, or aging. Several reports show that soft error rates

increase exponentially when hardware begins to wear [4, 5]. Abella et al. claim that 65 nm chips are unable to achieve the expected long life-times of several domains, e.g., 10 years in automotive and 50 years in avionics, because their soft error rate raises exponentially after five to six years of use, posing high reliability issues [5]. Also EMI measures, such as gaskets, suffer from aging, which renders them useless, giving higher probability to EMI to affect the system [22].

Therefore, software-implemented countermeasures have gained importance the past years [23–26], since they offer a flexible and often a more low-cost implementation than most hardware-based countermeasures can offer. Our research, and thus this thesis, focuses on CFEs and its software-implemented countermeasures. In order to detect CFEs, such countermeasures insert extra control variables and instructions into the target program at compile time. At run time, the added instructions recalculate these control variables and compare them to their expected compile-time value. A mismatch indicates that an error has occurred. The main differences between the various techniques proposed in literature are the operations used to update and verify these control variables and the overhead they impose [27–33].

Although many different CFE detection techniques already exist, it is difficult to know which technique to use for the target program. This is because, in their respective literature, each technique claims to be the best either regarding error detection ratio or regarding trade-off between error detection ratio and error detection cost, i.e. the imposed overhead. Therefore, a first objective of this thesis is to conduct a thorough comparison between various software-implemented CFE detection techniques. This means implementing them for the same case studies, execute the protected programs on the same hardware and subject them to the same fault injection campaign. Fault injection is a technique in which CFEs are deliberately and deterministically caused in the target hardware in order to verify the implemented detection technique. Such a comparison will either reveal which technique is the best, which technique makes the best trade-off between error detection and imposed overhead or which technique works best for which program type.

However, implementing the various techniques for various case studies is difficult. 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. However, as we show in this thesis, effectively implementing those techniques in high-level code reduces their error detection capability. To achieve their maximum error detection ratio, the CFE countermeasures have to be implemented in low-level code. This means that to implement a detection technique, knowledge about both the technique and the low-level code of the selected embedded system is needed. To solve this issue, a second objective of this thesis is to lower the required implementation

effort. This is accomplished by 1) providing an example implementation of several CFE detection mechanisms in low-level code and 2) discussing how the implementation of the techniques can be done automatically, with a compiler extension as a concrete example.

The third objective of this thesis is to advance the state-of-the-art by proposing new CFE detection techniques. Using the data of the performed comparative study, cfr. the first objective of this thesis, we developed the RASM technique. RASM has a higher error detection ratio and imposes a lower overhead than state-of-the-art techniques. In a next phase, we extended RASM into RACFED in order to detect even more CFEs. Finally, we propose a new approach to selectively implement CFE detection techniques which enables to reduce the imposed overhead with a minimal impact on the error detection ratio.

By presenting the work performed to achieve the three objectives, this thesis presents a practical approach to software-implemented CFE detection.

1.2 Contributions

The main contributions of this thesis are as follows:

- To verify the CFE detection techniques and measure their error detection ratio, we have developed a software-implemented fault injection tool, which supports multiple fault injection processes and multiple targets [77–79].
- To ease the implementation of the researched CFE detection techniques, we have developed a compiler extension that enables to automatically implement those techniques into the target program. This not only eliminates the need to implement the techniques manually, but also reduces the need of in-depth knowledge of the techniques [80, 81].
- We have performed a comparative experimental study of eight existing CFE detection techniques. We implemented the techniques for eight case studies and submitted them to a fault injection campaign. This enabled to compare the techniques on error detection ratio and error detection cost [82, 83].
- Using the data gathered during the comparative study, we proposed five guidelines to build an optimal CFE detection technique. These guidelines define what instructions to use and which program points to take into account when developing a CFE detection technique. [83]

- We developed a CFE detection technique, called RASM, which uses our derived guidelines to build an optimal CFE detection technique [83].
- We extended RASM into RACFED, another in-house developed CFE detection technique. RACFED extends RASM by inserting more error detection instructions into the target program which increases its CFE detection ratio [84].
- In an effort to lower the imposed overhead, we proposed a new approach to selectively implement CFE detection techniques. By applying it to our RACFED technique, creating S-RACFED, we show that a decrease in overhead with a minimal impact on error detection ratio is possible [79].

1.3 Structure

The remainder of this thesis is organized as follows. Chapter 2 presents more background on the envisioned embedded systems, on the CFE and the investigated software-implemented CFE detection techniques. Chapter 3 discusses the developed tools and selected case studies used for all experiments throughout this thesis. Next, Chapter 4 presents the performed experimental comparative study on established techniques. Following, Chapter 5 shows how we advanced the state-of-the-art with our RASM, RACFED and S-RACFED techniques. Next, Chapter 6 presents two industrial use cases used in this research to further enhance and to verify the developed CFE detection techniques. Finally, future work is presented and conclusions are drawn in Chapter 7.

Chapter 2

Background And State-Of-The-Art

As mentioned in the Introduction, this research has focused on increasing the reliability of embedded systems against CFEs introduced by external disturbances. Before discussing the CFE in more detail, this chapter will further scope our research. To conclude, this chapter presents several state-of-the-art (SOTA) software-implemented detection techniques.

2.1 Setting the Scene

An embedded system is a combination of hardware and software components that interact with the physical environment through sensors and actuators to perform a dedicated task [21]. The software component can consist of two subcomponents: the application and an optional (real time) operating system. Whether or not an operating system is needed, depends on the type of applications the embedded systems needs to execute. Embedded systems that do not use an operating system are called bare-metal embedded systems, to indicate that the application directly interacts with the hardware, as shown in Fig. 2.1. The hardware component of an embedded system is either a microcontroller or an application processor. While both perform the same operation, they are

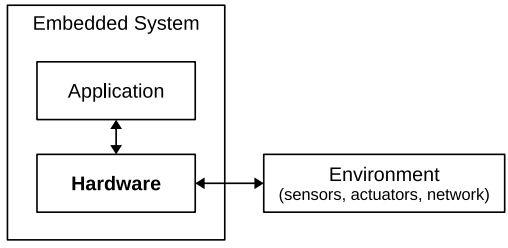


Figure 2.1: Abstract representation of a bare-metal embedded system.

meant for other use cases. Typically an application processor is used when a general purpose operating system is needed to support the desired application(s), whereas microcontrollers are designed to execute bare-metal applications or real time applications supported by a real time operating system. In this research, we have focused on bare-metal microcontroller-driven embedded systems because a recent market study indicates that 33 % of the produced embedded systems are bare-metal and because of our cooperation in this research with Televic Healthcare NV [34].

By taking a closer look at the hardware components of a microcontroller, we can identify where external disturbances can introduce bit-flips, how they would affect the embedded system and how to deal with them. Fig. 2.2 represents a simplified overview of the different components of a microcontroller and enables to identify four major components where bit-flips can be introduced.

The first two major components that can be affected by an erroneous bit-flip are the memory and the peripherals. Erroneous bit-flips in these components can supply false information, be it data, instructions or interrupt signals, to the CPU core. In turn, this often results in wrong output of the application executing on the embedded system. This type of erroneous bit-flip, however, is known to occur for decades and can easily be solved by using error correction codes (ECC) to protect the memory and the peripheral registers. Such codes add redundant bits to the stored data that enables to correct the data if a bit-flip has occurred.

A third major component that can be affected by an erroneous bit-flip are the buses used to transmit the data between the different hardware components. While data resides on the bus, it can be influenced by an external disturbance and be corrupted. Although the bus itself cannot be protected with an ECC, another research track within our research group is investigating how ECCs can be used and improved to protect the data being transmitted [36], [88–93].

The final major component that can be affected and which is the focus of our

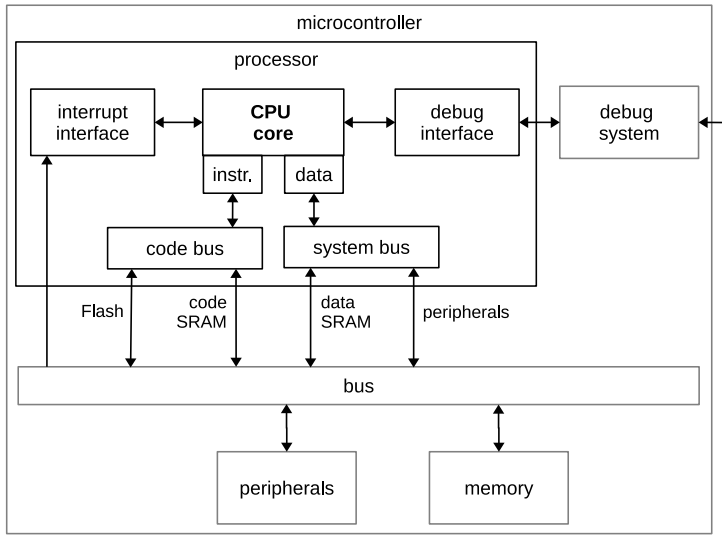


Figure 2.2: Simplified overview of the different hardware components of a microcontroller [35].

research, is the CPU core itself, which is shown in more detail in Fig. 2.3. Depending on which component is corrupted, the bit-flip will manifest itself in the application as a CFE or a DFE. If the register bank, the arithmetic logic unit (ALU) or the data interface are corrupted, this will result in a DFE as the CPU core will execute the current instruction with corrupted data. If the erroneous bit-flip occurs in the instruction register, which is the register that stores the next instruction to execute, either the instruction type, the indications of which registers to use or an encoded value will be corrupted. This corruption can either result in a CFE or in a DFE depending on how the instruction is affected. If the instruction is a branch instruction, e.g. `B 0x1a0`, a bit-flip affecting the branch address, changing the instruction to `B 0x1e0` results in a CFE. A corruption of the indication of which registers to use, can also result in a CFE. For instance, the typical return instruction `BX lr` instructs the microcontroller to branch to the address held in the `lr` register. A bit-flip changing the instruction to `BX r12`, instructs the microcontroller to branch to the address held in register `r12`, thus causing a CFE. While changing the instruction type often results in a DFE, it can result in a CFE in some cases. How to deal with DFEs using software-implemented techniques is not the focus of this thesis, but this research track is being investigated by another researcher in our research group [94–99]. The final element that can be affected is the program counter (PC) register, which indicates the address of the next

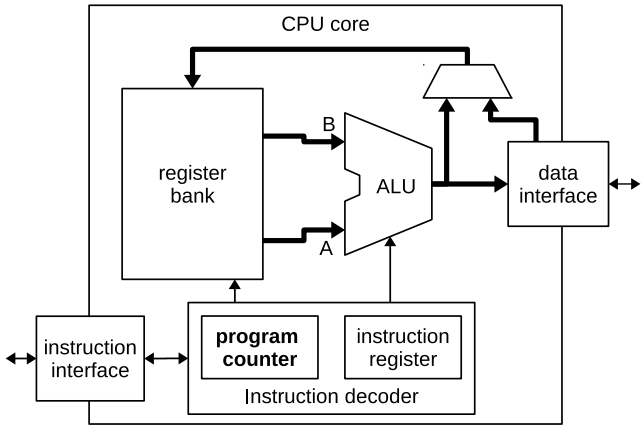


Figure 2.3: Simplified overview of the different hardware components of a CPU core [37].

instruction to execute. An erroneous bit-flip in this register always results in a CFE. A corruption of the instruction interface results in either a corrupted program counter or a corrupted instruction register and is thus covered by the previous two cases.

2.2 Control Flow Error

In Ch. 1, we defined a CFE as *the corruption of the execution order of instructions*. A more accurate definition is that a CFE is a violation against the control flow graph (CFG) of the program. The CFG is the representation of the program's execution order using basic blocks and edges. A basic block is a sequence of consecutive instructions with exactly one entry and one exit point. An edge is an intentional path between basic blocks. Multiple edges can originate from one exit point and multiple edges can end in one entry point. An example of a CFG is shown in Fig. 2.4. On the left, the program is shown in assembly code, using ARMv7-M syntax, and the corresponding corresponding CFG is depicted on the right [38].

To construct the CFG, the program is traversed and all branch instructions are considered as basic block exit points. For the program shown in Fig. 2.4, these are the instructions located at `0x1d2, 0x1de, 0x1e0` and `0x1e2`. Next, the destinations of those branch instructions are indicated as basic block entry points. For the example, these are the instructions located at `0x1e2` and

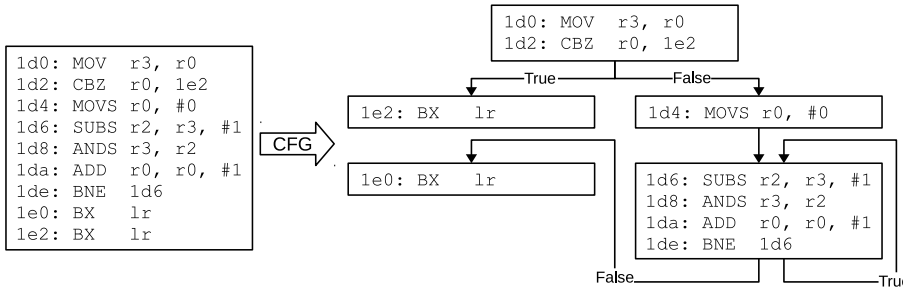


Figure 2.4: An example CFG of a program, shown in ARMv7-M assembly code.

`0x1d6`. The first instruction of the program is also a basic block entry point, instruction `0x1d0` for the example. Then, basic blocks are constructed with consecutive instructions between an entry point and an exit point, e.g., the basic blocks [`0x1d0, 0x1d2`], [`0x1d6, 0x1d8, 0x1da, 0x1de`], [`0x1e0`] and [`0x1e2`]. Next, the remaining instructions construct basic blocks on their own, e.g., the basic block [`0x1d4`]. Finally, the edges between all basic blocks are constructed. For the CFG of Fig. 2.4, the following edges are made: `0x1d2` → `0x1e2`, `0x1d2` → `0x1d4`, `0x1d4` → `0x1d6`, `0x1de` → `0x1e0` and `0x1de` → `0x1d6`.

Using the CFG, three types of CFE can be defined:

1. **inter-block** CFE: This type represents an erroneous jump between two different basic blocks. An example is shown in Fig. 2.5 as the jump between the instructions `0x1d0` and `0x1e2`.

2. **intra-block** CFE: In contrast to an inter-block CFE, an intra-block CFE is an erroneous jump within the same basic block. This kind of error is depicted as the jump between instructions `0x1d6` and `0x1de` in Fig. 2.5.

3. **out-of-CFG** CFE: The last type of CFE is an erroneous jump out of the CFG. Such CFE can either end in another program or in unused code memory. Fig. 2.5 indicates this type of error as the jump between instructions `0x1d4` and `0x2d4`.

To protect embedded systems against CFEs, software-implemented detection techniques have been proposed [27–33]. Inter-block CFEs are typically detected with a technique called *signature monitoring*, while intra-block CFEs are detected via a technique called *instruction monitoring*. Both methods insert extra control variables at compile time. At run time, these control variables are recalculated and compared to the expected compile-time values. A mismatch between both values indicates an error has occurred. The main difference

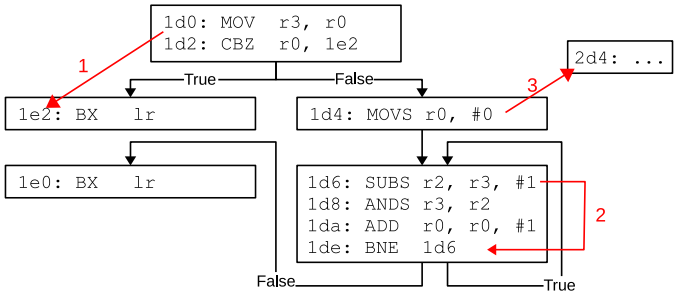


Figure 2.5: The three different types of CFE depicted on a CFG.

between inter-block and intra-block CFE detection is the frequency of updating the control variables. Out-of-CFG CFEs can be detected by one out of two methods, depending on where it ends. If the out-of-CFG CFE ends in another program, it can be detected by that program's inter-block and intra-block CFE countermeasure. To detect an out-of-CFG CFE that ends in unused code memory, that code memory has to be filled with jumps to a predefined error handler. For the rest of this thesis, out-of-CFG CFEs are out of scope, since its detection methods are either the countermeasures used to detect the other two types of CFEs or its detection method is either filling up the unused code memory, which is the job of the linker.

2.3 Software-Implemented Detection Techniques

This section describes several SOTA CFE detection techniques. For each technique we summarize its detection mechanism and present our implementation using the ARMv7-M syntax [38]. For each implementation, we show an example CFG in which the instructions depicted in bold are the instructions inserted by the CFE detection technique. The needed compile-time variables are shown either above, below or next to each basic block.

2.3.1 CFCSS

The Control Flow Checking by Software Signature (CFCSS) technique, proposed by Oh et al., is an inter-block CFE detection technique [27]. To implement CFCSS, two compile-time variables are assigned to each basic block. The first variable is the compile-time signature s_i which is a unique randomized bit sequence. The second variable d_i represents the valid branch of the first

predecessor to the current basic block. It is calculated as follows $d_i = s_i \oplus s_{pred_1}$ where s_{pred_1} is the compile-time signature of the first predecessor basic block. The $\oplus$ symbol refers to the EXCLUSIVE OR operation.

The run-time signature G is updated at the beginning of each basic block as follows $G = G \oplus d_i$. Next, the result of the update is verified. In an error-free run, the run-time signature should now hold the same value as the compile-time signature of the current basic block. When a basic block has multiple incoming edges, the signature update uses an extra variable D. This variable is updated by the predecessor basic blocks and assures that the run-time signature can be updated to the correct value, regardless of which predecessor has executed. For the first predecessor D is updated to the value zero, for all other predecessors D is updated with the result of the EXCLUSIVE OR operation of the signature of the current predecessor and the first predecessor. Then, the run-time signature update is given as $G = G \oplus d_i \oplus D$.

Using the CFG of Fig. 2.4, our implementation of CFCSS is shown in Fig. 2.6. To store and use the needed run-time variables, registers are used. In order for these registers to be available, they must be reserved at the start of the compilation process. Regarding Fig. 2.6, the run-time signature G is held in register `r11` and the run-time value for D is held in register `r10`. The first instruction each basic block executes is the update of the run-time signature, implemented by the EXCLUSIVE OR instruction (`EOR`). The upper-right basic block shows how the update using D is implemented. Here, the run-time update is split into two instructions: 1) the regular signature update using d_i, instruction `0x1f4`, and 2) the extra update with D, instruction `0x1f8`. Next, each basic block executes the verification of the run-time signature, by comparing its value with the compile-time signature. This verification is implemented as the comparison instruction (`CMP`). When there is a mismatch between both values, control is transferred to the error handler located at address `0x250`. This transfer is implemented by the branch-not-equal instruction (`BNE`). The last CFCSS-related instruction each basic block executes, is the update for the variable D implemented as the move instruction (`MOV`). We make an exception here for basic blocks that exit from the current function or program. The two lower basic blocks are exit blocks and therefore do not update register `r10`. We do not update the run-time variables in exit blocks because they have no successor that can verify that update.

2.3.2 YACCA

Goloubeva et al. proposed the Yet Another Control Flow Checking using Assertions YACCA inter-block CFE technique [28]. The technique uses three

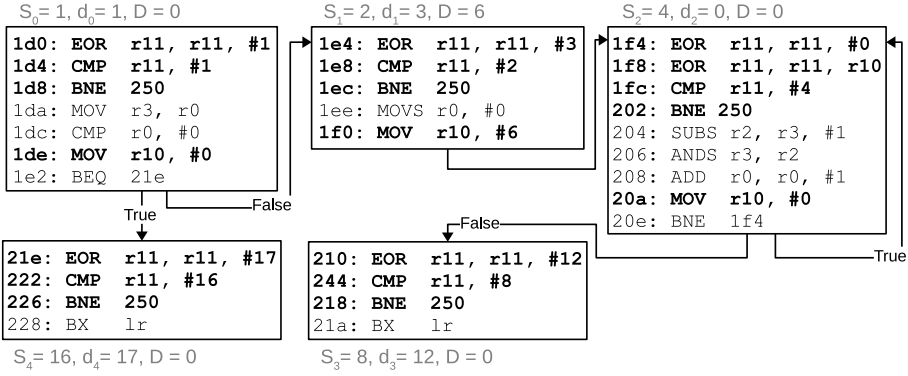


Figure 2.6: An example of our implementation of CFCSS using the ARMv7-M instruction set.

or four compile-time variables for each basic block. The first variable is the unique compile-time signature B_i. The second variable *Previous* represents all predecessor blocks of the current basic block. It is calculated by multiplying the compile-time signatures of all predecessor blocks of the current basic block. The third compile-time variable is $M1$, but is only assigned to basic blocks with two predecessors. It is calculated using the unique compile-time signatures of those predecessors: $M1 = B_{pred1} \overline{\oplus} B_{pred2}$. The $\overline{\oplus}$ symbol refers to the EXCLUSIVE NOR operation. The final compile-time variable is $M2$, which represents both the current block and its predecessors. For basic blocks with only one predecessor $M2 = B_{pred} \oplus B_i$, and for basic blocks with two predecessors $M2 = B_{pred1} \,\&\, M1_i \oplus B_i$. The & symbol refers to the BITWISE AND operation.

The run-time signature *code* is updated at the end of each basic block. In case of an error-free run, the final value of *code* is the compile-time signature B_i of the current basic block. How the update is performed, depends on the number of predecessors of the current basic block. For basic blocks with just one predecessor $code = code \oplus M2_i$, and for basic blocks with two predecessors $code = code \,\&\, M1_i \oplus M2_i$.

The run-time signature is verified both at the beginning and at the end of each basic block. The assertion validates whether or not the previous basic block was a valid predecessor of the current basic block. Goloubeva et al. discuss two methods to verify the run-time signature. The first method is by performing a modulo operation between the *Previous* value of the current basic block and the run-time signature, which returns zero in an error-free run. Any other result indicates a CFE has occurred. While this first method is an effective and easy method to verify the run-time signature, the modulo operation can be

computationally expensive. To avoid this expensive operation, Goloubeva et al. proposed a second method to verify the run-time signature. In this second method, the run-time signature is compared to the compile-time signature of each predecessor one by one. If no match is found, a CFE has occurred. For the remainder of this thesis, this second method will be indicated as YACCA_CMP.

Fig. 2.7 shows our implementation of YACCA. The run-time signature *code* is held by register `r11` and the run-time value of *Previous* is stored in register `r10`. Because the ARMv7-M instruction set does not have a modulo instruction, we implemented the module operation using an integer division (`UDIV`) and a multiplication (`MUL`). The `UDIV` instruction divides `r10` by `r11` and stores the integer result in `r9`. To express this integer division with the variables of YACCA, the `UDIV` performs the following $r9 = \frac{Previous}{code}$. Next, the `MUL` instruction multiplies `r9` with `r11` and stores the result back in `r9`. In YACCA terms, this is $r9 = r9 \times code$. In an error-free run, `r9` should now hold the *Previous* value of the current basic block. This is verified by the comparison instruction (`CMP`), which transfers control (`BNE`) to the error handler located at address `0x27a` when `r9` and *Previous* differ. Regarding the run-time signature update at the end of each basic block, the needed operations are mapped to their corresponding instructions. The BITWISE AND operation between *code* and $M1$ is implemented with the `AND` instruction, and the EXCLUSIVE OR operation between *code* and $M2$ is implemented using the `EOR` instruction.

We also implemented the computationally less expensive version of YACCA, shown in Fig. 2.8. As described, YACCA_CMP does not use the modulo operation but comparison operations to verify the run-time signature. For basic blocks with only one predecessor, this comparison is straightforward: register `r10` is loaded with the compile-time signature B of the predecessor and the `CMP` instruction compares its value to the run-time signature, held by register `r11`. If they are not equal, control is transferred to the error handler located at address `0x27e`. When a basic block has two predecessors, such as the right side basic block of Fig. 2.8, the comparison is more complicated. In this case, both compile-time signatures of both predecessors are evaluated, and an error flag (`r9`) is updated in case of any mismatch. Instructions `0x1fe` to `0x206` verify if the run-time signature has the value of the first predecessor and instructions `0x20a` to `0x212` verify whether or not the run-time signature has the value of the second predecessor. Next, instructions `0x216` and `0x218` verify if the error flag has the expected value and transfer control to the error handler when needed. Finally, the instruction at address `0x21a` clears the error flag, preparing it for the next run-time signature verification.

Both our implementations treat basic blocks that only contain an exit statement, e.g., `BX lr`, with special care. Such basic blocks have only one run-time signature verification, instead of two such as regular basic blocks. Neither do they update

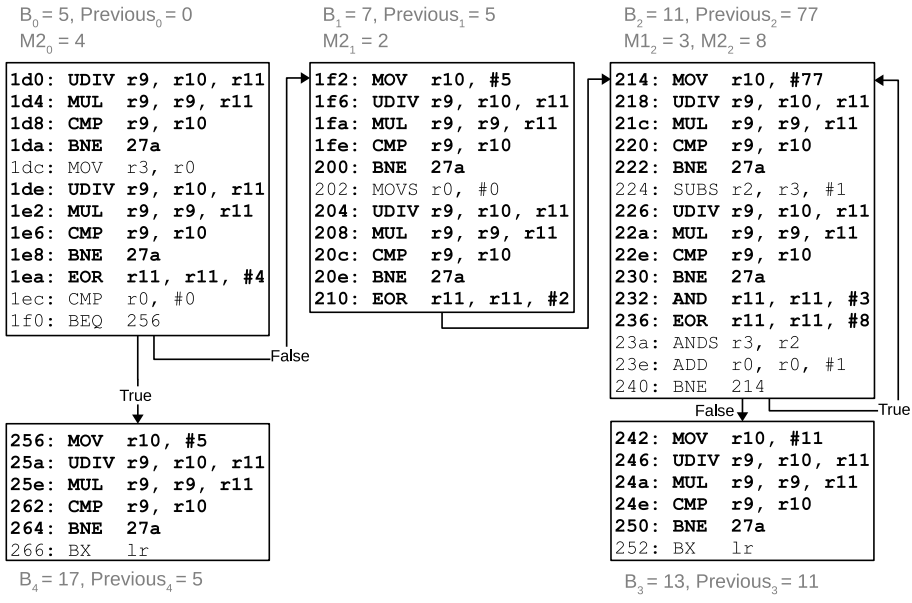


Figure 2.7: An example of our implementation of YACCA using the ARMv7-M instruction set.

the run-time signature.

2.3.3 ECCA

Enhanced Control flow Checking using Assertions (ECCA) was proposed by Alkhalifa et al. as an inter-block CFE detection technique [29]. To implement ECCA, three compile-time variables are assigned to each basic block: BID, $NEXT1$ and $NEXT2$. BID is the unique compile-time signature and has to be a prime number larger than 2. $NEXT1$ and $NEXT2$ hold the compile-time signatures of the successor blocks of the current basic block.

At run time, two variables are needed, $v1$ and $v2$. $v1$ is the run-time signature and $v2$ is a helper variable used during two updates of $v1$. ECCA adds four operations to each basic block to detect CFEs:

1. $v1 = (v1 - BID) \times (v2 - BID)$
   This multiplication updates $v1$ to zero, because an error-free run either $v1$ or $v2$ holds the value of BID.

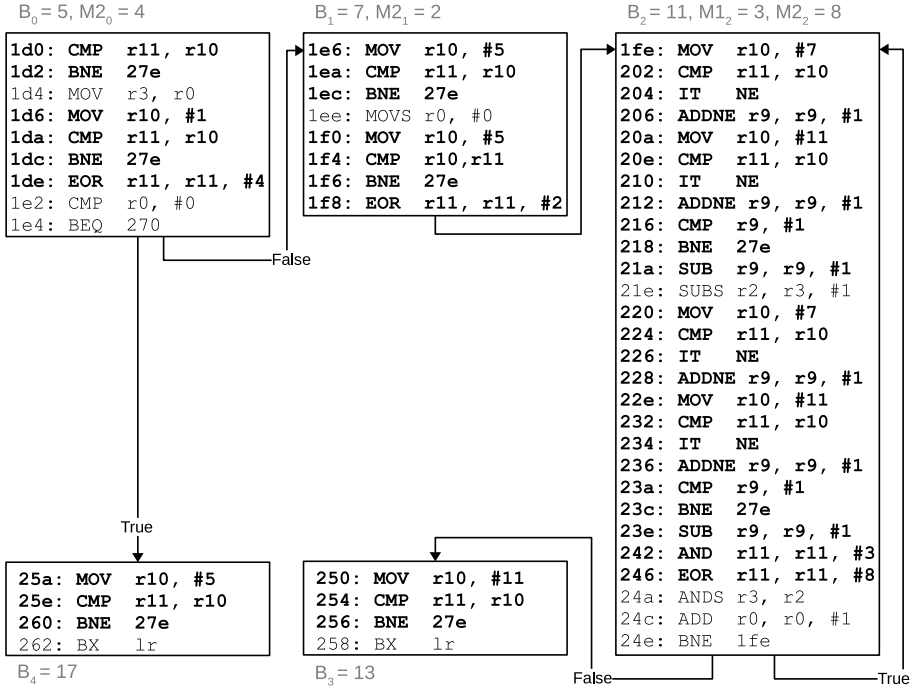


Figure 2.8: An example of our implementation of YACCA_CMP using the ARMv7-M instruction set.

2. $v1 = \frac{BID+1}{(v1+1)/(2v1+1)}$
   This division assigns the value of $BID + 1$ to $v1$ in an error-free run. When no CFE has occurred, the previous operation updated $v1$ to zero, so the division becomes $v1 = \frac{BID+1}{(0+1)/(0+1)} = BID + 1$.

3. $v2 = (v1 - BID) \times NEXT2$
   The branch to one of the successor blocks is prepared, by assigning the value $NEXT2$ to $v2$. In an error-free run, $v1 = BID+1$ at this point in the program, thus the multiplication is $v2 = (BID + 1 - BID) \times NEXT2 = NEXT2$.

4. $v1 = (v1 - BID) \times NEXT1$
   To allow the branch to the other successor block, $NEXT1$ is assigned to $v1$.

The first two operations are added at the beginning of each basic block and the last two are added at the end of the basic block. A CFE is detected during the

second update instruction through a division-by-zero exception. The division-by-zero exception is caused by the division $\frac{v1+1}{2v1+1}$. This is an integer division and results in zero if $v1$ is non-zero, which is the case when a CFE has occurred. In more detail, the second updated for $v1$ becomes $v1 = \frac{BID+1}{(v1+1)/(2v1+1} = \frac{BID+1}{0} \rightarrow$ division-by zero exception.

Our implementation of ECCA is presented in Fig. 2.9. The run-time variable $v1$ is stored in register `r11` and the run-time variable $v2$ is stored in register `r10`. The first update of $v1$ is mapped on three instructions, two subtractions (`SUB`) and one multiplication (`MUL`). The next update of $v1$ is implemented using six instructions. First, $2v1$ is calculated by performing a left-shift of register `r11` and storing the result in register `r10`. Secondly, $v1+1$ and $2v1+1$ are calculated using two additions (`ADD`). Next, $(v1+1)/(2v1+1)$ is executed, by dividing the results of the previous additions by each other and storing that result in register `r11` (`UDIV`). Finally, $BID+1$ is stored into register `r10` (`MOV`) and the final division to update $v1$ is performed and its result stored in register `r11` (`UDIV`). As discussed, a CFE should be detected by a division-by-zero exception during the second division. Small scale experiments, however, showed this was not the case. In fact, some microcontrollers do not raise a division-by-zero exception unless explicitly instructed to do so. In order for our implementation of ECCA to detect CFEs regardless of the fact whether or not the microcontroller has been set up to throw the exception, we insert a run-time signature comparison (`CMP`) between the first and second update of $v1$. An example is given at address `0x1dc`. This comparison verifies whether or not the run-time signature has the value of zero and transfers control to the error handler located at address `0x2bc` when a non-zero value is detected (`BNE`).

Because the ARMv7-M instruction set does not support a multiplication between a register and a value, we had to modify the final updates of $v1$ and $v2$. First, we subtract $BID+1$ from register `r11`, updating $v1$ to zero (`SUB`). Next, we add $v1$ to $NEXT1$ and $NEXT2$ to update $v1$ and $v2$ to the correct values (`ADD`).

With our implementation of ECCA, exit basic blocks are treated differently from regular basic blocks. In such basic blocks, only the first update of $v1$ and the run-time signature verification instructions are inserted.

2.3.4 RSCFC

The Relationship Signatures for Control Flow Checking (RSCFC) technique, proposed by Li et al., is designed to detect both inter-block and intra-block CFEs [30]. The technique assigns three compile-time variables to each basic block: the compile-time signature s_i, the CFG locator L_i and the cumulative

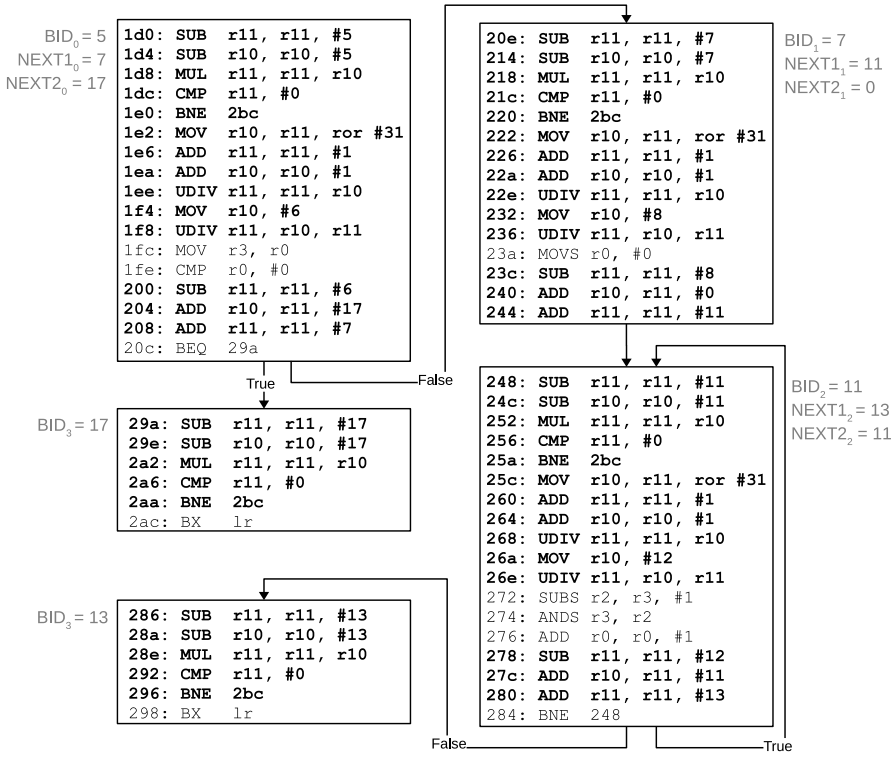


Figure 2.9: An example of our implementation of ECCA using the ARMv7-M instruction set.

signature m_i. The compile-time signature indicates the successor blocks of the current basic block and is generated according to the following process:

1. If there are n basic blocks in the CFG, the signature is $n + 1$ bits wide;
2. Bit $n + 1$ is set to 1;
3. The bits of the successor blocks are set to 1.

The CFG locator indicates the position of the basic block in the CFG. The variable L_i is again $n + 1$ bits wide, but only the bit associated with the basic block is set to 1. E.g., for the first block in the CFG, only the least significant bit is set. For its successor block, only the next bit is set, and so on. The final compile-time variable m_i is a bit sequence that indicates how many instructions

must be executed for each basic block. m_i has as many bits as the basic block has instructions and all are set to 1.

The run-time signature S is updated at the beginning of each basic block as follows $S = S \,\&\, L_i$. Next, the result is verified. In an error-free run, the result of the BITWISE AND operation is L_i and thus non-zero. Next, the intra-block updates are executed. First, the run-time cumulative signature N is updated with the m_i of the current basic block. Next, N is updated after each instruction with the result of the EXCLUSIVE OR operation between itself and a bit pattern indicating which instruction has executed. In an error-free run, N is updated to zero once all instructions have executed. Finally, the run-time signature S is updated at the end of each basic block. The update to perform is shown in eq. (2.1).

$$S = s_i \,\&\, (S \,\overline{\oplus}\, L_i) \,\&\, (-!N) \tag{2.1}$$

Breaking the update down:

- $(S \,\overline{\oplus}\, L_i)$ performs the EXCLUSIVE NOR operation between S and the CFG locator L_i of the current basic block. In an error-free run S currently holds the value L_i, therefore the result of the operation is an all 1 bit pattern, often shown as the value -1.
- $(-!N)$ results in the BITWISE INVERSE of N. In an error-free run N has the value of zero, so the result of the operation is an all 1 bit pattern.
- Finalizing for an error-free run, eq. (2.1) becomes $S = s_i \,\&\, -1 \,\&\, -1 = s_i$. The update stores s_i in S, so the run-time signature shows the valid successors of the current basic block.

For our implementation of RSCFC, the run-time signature S is stored in register `r11` and the run-time cumulative signature N is stored in register `r10`. As shown in Fig. 2.10, the first update of S and the according verification are implemented rather straightforward. The BITWISE AND operation is mapped to the `AND` instruction and the verification to the `CMP` instruction. When a CFE is detected, control is transferred to the error handler located at address `0x288` (`BNE`).

Next, the intra-block part of RSCFC is implemented. The initial update of N is implemented as a `MOV` instruction and is inserted after the run-time signature verification. Then, update instructions (`EOR`) are inserted after each original program instruction.

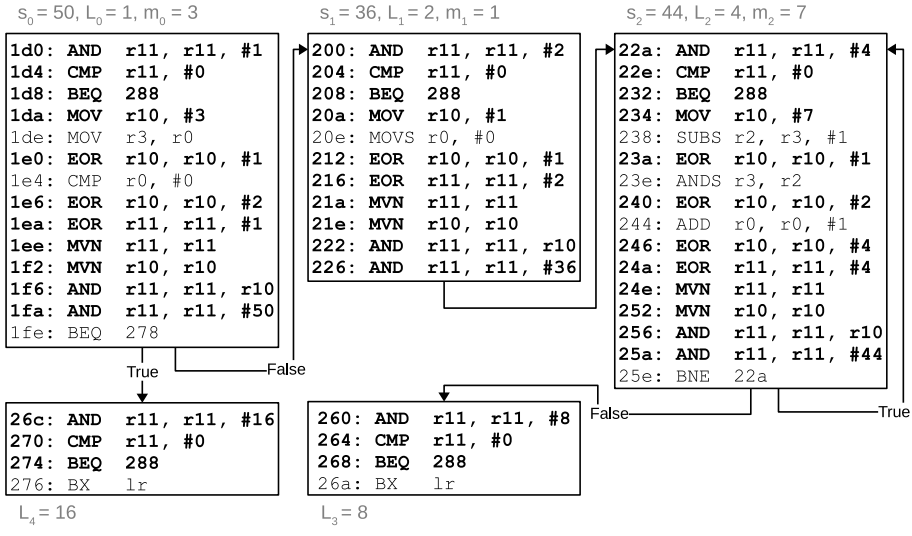


Figure 2.10: An example of our implementation of RSCFC using the ARMv7-M instruction set.

Finally, the last update of S is implemented. Eq. (2.1) is broken down into five instructions. First $(S \overline{\oplus} L_i)$ is calculated, using an EXCLUSIVE OR (`EOR`) instruction and a BITWISE INVERSE (`MVN`) instruction. Then $(-!N)$ is calculated, using an `MVN` instruction. Finally, two `AND` instructions complete the update.

Our implementation treats exit basic blocks differently from regular basic blocks. Exit basic blocks only implement the first run-time signature update and the verification.

2.3.5 SEDSR

Soft Error Detection using Software Redundancy (SEDSR) is an inter-block CFE detection technique proposed by Asghari et al. [31]. SEDSR assigns just one variable to each basic block at compile time, i.e. the compile-time signature s_i. It is a bit sequence that shows the valid successor basic blocks of the current basic blocks. If there are n basic blocks in the CFG of the program, s_i is n bits wide and the bits of the successor blocks are set to 1.

The run-time signature S is verified at the beginning of each basic block. The verification checks whether or not the current basic block is a valid successor of the previous basic block. In case of an error-free run, the bit on the position

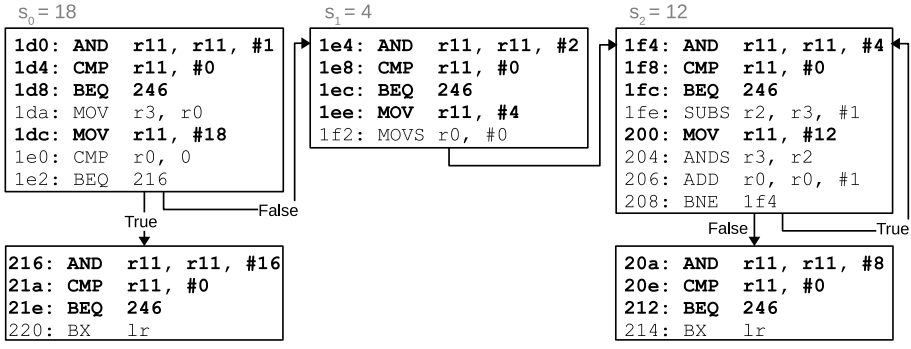


Figure 2.11: An example of our implementation of SEDSR using the ARMv7-M instruction set.

associated with the current basic block should be set. If that bit is zero, a CFE has occurred. The run-time signature S is updated in the middle of each basic block and assigns the compile-time signature s_i of the current basic block to S.

Fig. 2.11 shows our implementation of SEDSR. The run-time signature is stored in register `r11`. We implemented the run-time signature verification with a BITWISE AND operation (`AND`), between the run-time signature and the expected set bit. Next, the result of the BITWISE AND operation is verified (`CMP`). In an error-free run, the bit at the wanted position is set, so control is transferred to the error handler located at address `0x246` if the run-time signature is zero (`BEQ`). The signature update in the middle of each basic block is implemented using a move instruction (`MOV`), except in exit basic blocks.

2.3.6 SCFC

To further increase the error detection ratio, Asghari et al. proposed the Software-based Control Flow Checking (SCFC) technique as an extension to their SEDSR technique [32]. SCFC uses the same compile-time signature s_i as in SEDSR. The difference with SEDSR is the run-time verification process. SCFC uses two run-time variables, the run-time signature S and the run-time identifier ID.

A first run-time verification is performed at the beginning of each basic block. This verification checks whether or not ID holds the compile-time identification number (id) of the current basic block. A mismatch indicates a CFE has occurred. The second run-time verification, i.e. the verification of the run-time signature S, is performed in the middle of each basic block. This verification

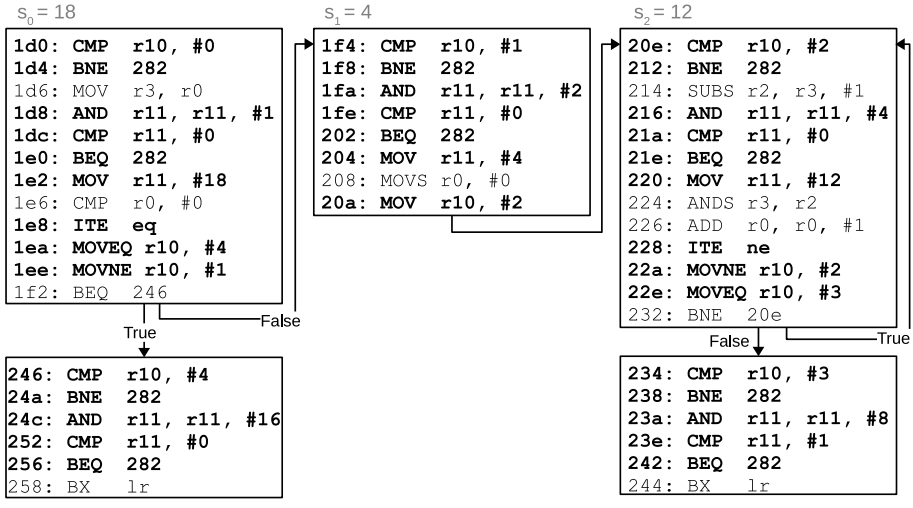


Figure 2.12: An example of our implementation of SCFC using the ARMv7-M instruction set.

is the same as with SEDSR, and checks if the current basic block is a valid successor of the previous basic block.

To maximize the error detection ratio, ID and S are updated at different locations in the basic block. S is update in the middle of each basic block, after it has been verified, and ID is updated to the id of the successor block at the end of each basic block.

Our implementation of SCFC is shown in Fig. 2.12. The verification of ID, stored in register `r10`, is implemented using a comparison (`CMP`) and a branch to the error handler (`BNE`). The verification and update of the run-time signature S are implemented in the same way as in SEDSR. Finally, the update of ID is implemented using the simple move instruction (`MOV`). For basic blocks that end in a conditional branch, this update is executed conditionally. The ARMv7-M instruction set allows instructions to be executed conditionally, i.e. the instruction is only executed when its according condition has been met. The upper-left and upper-right basic blocks of Fig. 2.12 show this conditional update. As with all other techniques, the exit basic blocks do not update the run-time variables.

2.3.7 SIED

The final SOTA technique we discuss in this chapter is Software-Implemented Error Detection (SIED) proposed by Nicolescu [33]. SIED is capable of detecting both inter-block and intra-block CFEs. At compile time, each basic block is assigned a unique identifier IDB, a list containing the compile-time signatures of all successor basic blocks and a variable n that indicates how many instructions must be executed in the basic block.

The run-time signature X is updated and verified at the beginning of each basic block. The update consists of storing the result of the addition between the unique identifier of the current basic block and the status condition branch (SCB) in X. In other words, the update is the following $X = IDB_i + SCB$. The status condition branch is updated each time a conditional branch is taken and indicates whether the false or true path should have been taken. Next, the run-time signature verification compares X with a second run-time variable Y. In an error-free run, both should hold the same value, thus a mismatch indicates a CFE has occurred.

Next, the intra-block updates are inserted. To detect intra-block CFEs, SIED uses the run-time variable *checkpass*. Once the run-time signature has been verified, *checkpass* is updated with the n_i variable of the current basic block. After each original program instruction, *checkpass* is decremented. At the end of the basic block, a verification instruction is inserted to validate whether or not *checkpass* is now zero. If that is not the case, a CFE has occurred. Finally, the run-time variables SCB and Y are assigned their new values, after the intra-block verification. Y is updated with the IDB_i of the successor basic block and SCB is updated to 1 if the true path of a conditional branch has to be taken, otherwise it is updated to 0.

For our implementation of SIED, register `r10` stores X or SCB depending on the location in the basic block, register `r9` stores Y and register `r11` stores *checkpass*. Fig. 2.13 shows that the implementation is rather straightforward. The update $X = IDB_I + SCB$ is implemented with the addition instruction (`ADD`) and the according verification, i.e. $X == Y$, is implemented using the comparison instruction (`CMP`). When a mismatch is detected, control is transferred to the error handler located at address `0x28a` (`BNE`). Next, the intra-block updates are inserted. The initialization of *checkpass*, i.e. $checkpass = n_i$, is implemented using the simple move instruction (`MOV`). Then a decrement is implemented after each original original program instruction (`SUB`). The last thing each basic block does, is update SCB and Y. We implemented this using the `MOV` instruction. When a basic block ends with a conditional branch, these last two updates are executed conditionally. As with all our implementation, exit basic blocks do

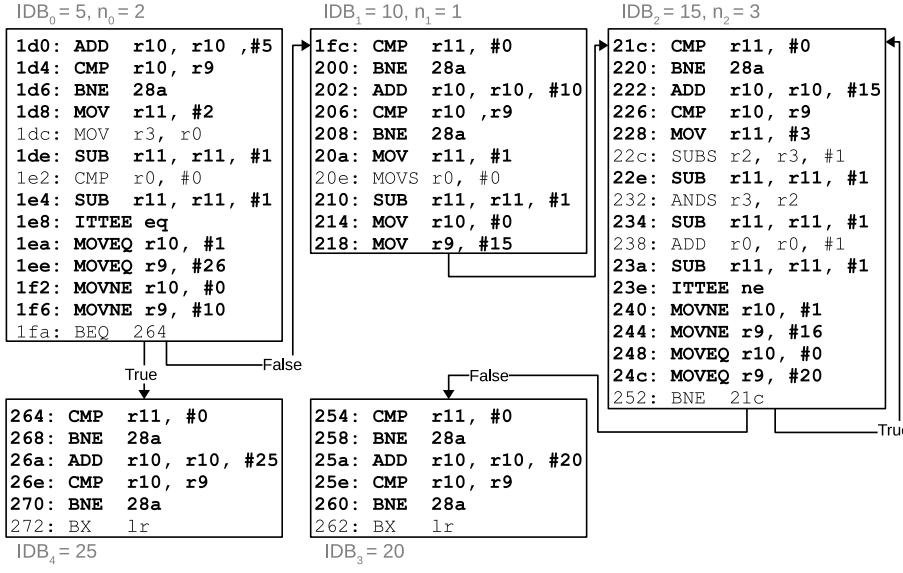


Figure 2.13: An example of our implementation of SIED using the ARMv7-M instruction set.

not perform these last updates.

The verification *checkpass* == 0 is inserted at the beginning of each basic block, except for the first one. We implemented it at the start of the basic block, instead of at the end of each basic block as proposed by Nicolescu et al., to assure the correct control flow through the program. When a comparison instruction is inserted at the end of a basic block, it might overwrite the system flags needed to take the correct path of a conditional branch. To avoid this problem, we insert the verification and branch to the error handler at the beginning of each basic block.

2.4 Conclusion

This chapter discussed the CFE and several SOTA detection techniques in detail. We presented how CFEs can be divided into inter-block and intra-block CFEs and how each type has its own detection mechanism, *signature monitoring* and *instruction monitoring* respectively. For each detection mechanism, extra

control variables, update instructions and verification instructions are needed in order to detect CFEs.

Regarding the discussed SOTA techniques, we've shown that each of them are clearly different concerning the types of updates they perform and concerning the locations in the basic block where they add the needed instructions. To update their control variables, the SOTA techniques use a wide variety of operations, EXCLUSIVE OR, BITWISE AND, division, etc. The locations to insert the extra instructions also span a wide range of options: at the beginning of the basic block, in the middle of the basic block, at the end of the basic block or a combination of the previous locations.

While a wide variety of SOTA techniques exist, it is difficult to ascertain which technique has the highest error detection ratio or the lowest error detection cost, i.e. impact on execution time or impact on code size. In their respective literature, each technique claims to perform better in either error detection ratio or error detection cost than other SOTA techniques. However, to evaluate the techniques, different case studies, different hardware and different fault injection processes have been used, making a comparison virtually impossible. Therefore, one of the first objectives of this research was to evaluate all of the discussed SOTA techniques on an equal basis. This means implementing them for the same case studies, executing them on the same hardware and submit them to CFEs using the same fault injection process.

Chapter 3

Experiment Prerequisites

The content and results of this chapter have been published in [77–81, 83]. Since their publication, the results presented in this chapter have been updated by using updated versions of the tools and our latest insights.

This chapter presents our developed software tools and the used hardware, used throughout the different experiments presented in this thesis, such as the comparative study of the SOTA techniques and experiments performed to verify our own developed techniques. First, we present the selected case studies. Then we describe the criteria selected to objectively characterize the techniques. Following, our developed fault injection tool is discussed, which was developed because no other tool was available to us. This chapter ends by presenting our compiler extension that enables automatic implementation of the different CFE detection techniques.

3.1 Case Studies

We selected eight academic case studies, and five data sets for each case study, to perform experiments with:

- **Bit Count (BC):** The bit count algorithm counts the bits set, i.e. the *1*'s, in the given data word, also known as the hamming weight. This functionality is, amongst others, used in the communication domain to calculate a parity bit or in the cryptographic domain to calculate keys [39].

- **Bubble Sort (BS) and Quicksort (QS):** The bubble sort and quick sort algorithms were selected because the sorting of information is used in different systems, e.g., to assign priorities or to enable faster analysis of data [40].
- **Cyclic Redundancy Check (CRC):** The cyclic redundancy check algorithm is mainly used to add error detection information in data transmissions [41]. When using CRC, additional bits are added to the transmitted data which enables to analyze whether or not the received information is correct.
- **Cubic Solver (CU):** The cubic function solver algorithm is used in physics related applications, such as calculating the speed of a car or calculating the power density in wind turbines [42].
- **Dijkstra's algorithm (DIJ)**: Dijkstra's algorithm calculates the shortest path between different nodes, which is a highly used functionality in routing applications [43].
- **Fast Fourier Transform (FFT):** The fast fourier transform is used in a wide variety of applications, such as file compression, voice recognition and vibro-acoustic analysis in cars [44].
- **Matrix Multiplication (MM):** The matrix multiplication algorithm was selected because matrices and matrix multiplication are used in many different embedded domains such as image processing, e.g., CAT and MRI scan, robotics and data compression [45].

While the BS, MM, and QS case studies use our implementation, the implementation of the remaining case studies was selected from MiBench version 1.0 [46]. MiBench is free commercially representative embedded benchmark suite. This suite contains 35 embedded applications for benchmarking purposes which are divided into six categories: Automotive and Industrial Control, Consumer Devices, Office Automation, Networking, Security and Telecommunications.

Next to being highly used in the embedded systems domain, the selected case studies also have varying basic block and edge distributions. The distribution of the basic blocks for each target algorithm is shown in Fig. 3.1. The chart shows how many basic blocks of each length each case study possesses, when compiled with the *arm-none-eabi-gcc-7.3* compiler and optimization level *-O2*. Total height of each bar indicating the total amount of basic blocks. As can be seen, each case study has a different distribution of basic blocks.

Next to the basic block profile, we also analyzed the edge profile of each case study. Table 3.1 shows the amount of unconditional and conditional

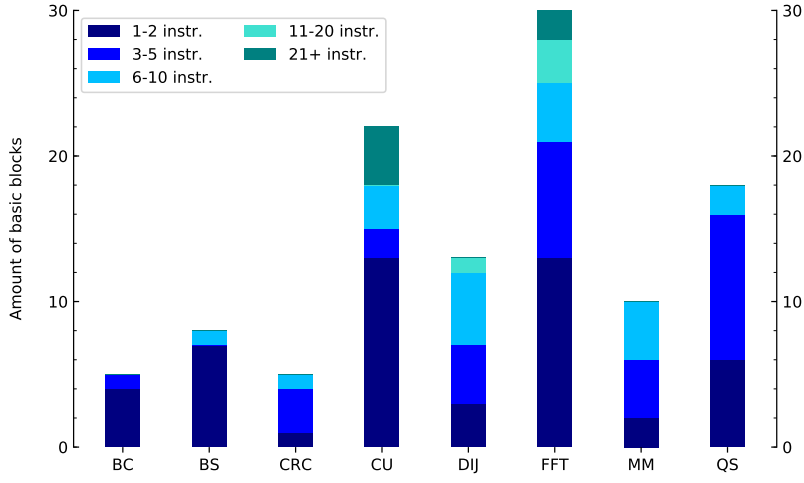


Figure 3.1: The distribution of basic blocks for each case study.

branches. The table indicates that each case study has a different amount of both conditional and unconditional branches. Combined with a varying basic block distribution, these selected case studies assure a thorough validation of any implemented CFE detection technique. Furthermore, the five selected data sets for each case study assure each path through the CFG is taken. This means that for the

- BC case study, we also provide the value zero as input;
- BS and QS case studies, we provide an ordered data set, a reverse ordered data set and several randomized data sets as input;
- CRC, MM, DIJ, FFT case studies, we provide randomized data sets as this was the best way to assure that all paths were taken;
- CU case study, data sets were provided that resulted in three solutions and data sets were provided that resulted in one solution, as these are the two cases the algorithm produces.

Table 3.1: The edge distribution for each case study.

	Conditional Branches	Unconditional branches
BC	3	4
BS	4	8
CRC	3	4
CU	12	20
DIJ	6	14
FFT	16	28
MM	4	12
QS	9	18

3.2 Technique Characterization Criteria

To compare the different techniques to one another, we measured two metrics: their error detection ratio when submitted to errors and the cost of the protection in error-free circumstances. The error detection ratio is measured because this is the desired gain. CFE detection techniques are implemented to improve the resilience of the target program to CFEs and this criterion reveals how good a technique is at detecting CFEs.

As shown in Section 2.3, CFE detection techniques add extra code to the target program in order to detect CFEs. This means that the target program is now larger and needs more memory to be stored on the target and needs more time to execute. Therefore, we measured these types of overhead for each technique. Moreover, this second criterion enables further comparison between the different techniques. For example, when two techniques have a similar error detection ratio, but one technique achieves this ratio while imposing a moderate overhead and the other technique achieves this ratio by imposing a high overhead, we can conclude that the first technique is better than the second despite them having the same error detection ratio.

In addition, these two criteria are also highly used in literature to characterize and compare CFE detection techniques with one another, which validates our approach.

3.2.1 Fault Injection Result Categories

Although most literature concerning CFE detection techniques use fault injection to evaluate their technique, some concerns about this used approach have been raised [47]. Schirmeier et al. raised some concerns about fault-injection based

comparison of different fault tolerance techniques. One of their identified pitfalls is:

> *Pitfall 3: Fault-Coverage Percentages for Benchmark Comparison*
> Subsequently, our third and most important pitfall is the usage of *fault-coverage percentages* for benchmark comparison. Unless the fault space dimensions of two program variants are identical – which is practically never the case when effective software-based fault-tolerance mechanisms are in place –, their fault coverages are measured in percent relative to different fault-space areas, and are *by definition not comparable* .... (Schirmeier et al., p. 327 [47])

In order not to fall victim to this pitfall, we do not characterize a technique solely on fault-coverage. Instead, we categorize the effect of each injected error using the following four categories:

- **Detected (Det.):** The error was detected by the implemented detection technique. This is the desired result.
- **Hardware Detection (HD):** Many processors have several internal fault handlers that are able to detect specific hardware faults, such as improper bus usage or memory access violations. This category represents the errors detected by such fault handlers.
- **Silent Data Corruption (SDC):** This means the error was neither detected by the implemented technique nor by the hardware and caused the protected case study to produce a wrong result.
- **No Effect (NE):** Finally, this category indicates that the error was not detected and did not affect the outcome of the protected algorithm. This is also known as the inherent resilience of the case study.

Throughout this thesis, the fault injection results will be depicted in percentages for each of the four categories. While showing the fault-coverage alone, our Det. category, could indeed give a false indication about the technique, showing the percentages for all four categories gives a complete overview of the capabilities of a CFE detection technique. Furthermore, the shown percentages are calculated based on the number of injected errors and are not based on the fault space dimensions, thus eliminating the difference in fault space dimensions from the comparison.

3.2.2 Control Flow Error Detection Cost

Software-implemented CFE detection techniques insert extra instructions into the code they need to protect. This means that the code size and the execution time of the protected code will be higher than that of the unprotected code. For each discussed technique, we measured both its code size and execution time and compare the results to the unprotected code using eq. (3.1) and eq. (3.2). The code size is measured using the *text* output of the *arm-none-eabi-size* tool, when used on the *.elf* file of the compiled case study. The execution time is measured using a hardware timer of the target.

$$\text{code size relative to unprot. code} = \frac{\text{code size protected}}{\text{code size unprotected}} \tag{3.1}$$

$$\text{exec. time relative to unprot. code} = \frac{\text{exec. time protected}}{\text{exec. time unprotected}} \tag{3.2}$$

3.3 Software-Implemented Fault Injection Tool

In order to validate the SOTA techniques and our developed techniques, CFEs need to occur in the selected target. Since at the start of the research most fault injection tools either were not available to us or did not support our target, we developed our own software-implemented fault injection tool (SWIFI).

First, we discuss the architecture of our tool. Next, the in-house developed CFE injection processes are presented, to end with an overview of the supported targets.

3.3.1 Architecture

First, we developed a tool to inject faults in our hardware target, an NXP LPC 1768. Because support to interact with this and other targets was better in Python, we developed a Python-based SWIFI-tool for hardware targets. In a later phase, we moved to a simulated target due to the need for faster and more exhaustive CFE injection [48]. Because the API of the Imperas simulator to interact with the targets is provided in C++, we developed a C++-based SWIFI-tool for simulated targets.

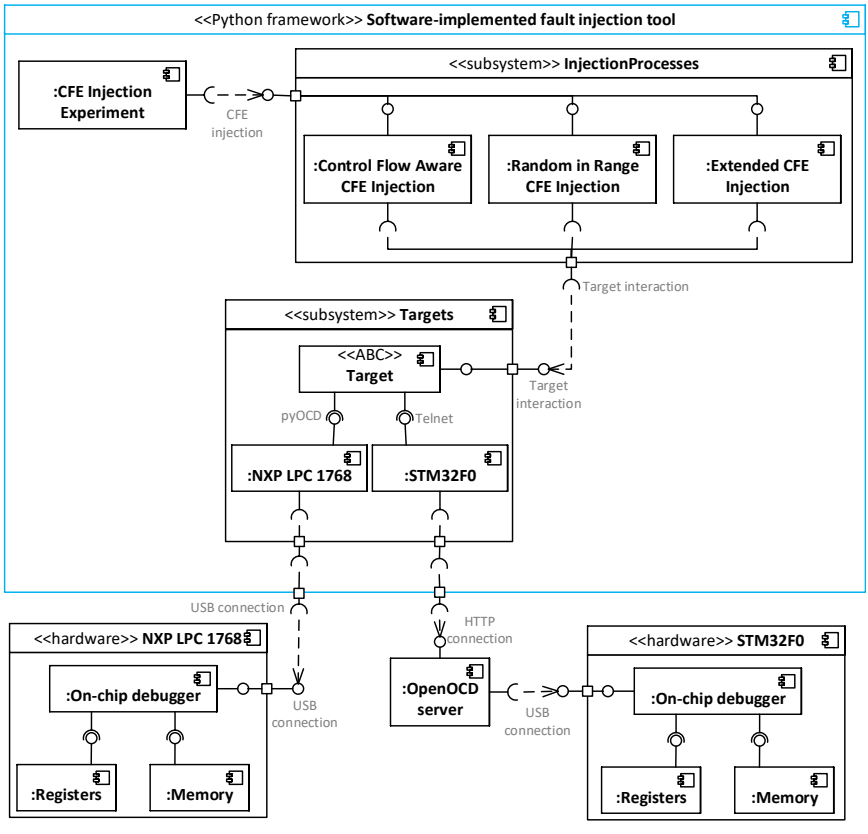


Figure 3.2: Overview of the Python framework to enable CFE injection in hardware targets.

Python Framework to support Real Targets

The architecture of the Python framework to enable fault injection in real hardware targets is shown in Fig. 3.2. The framework contains two subsystems to ease the definition of new CFE injection processes and to ease the support of new targets.

Targets subsystem: This subsystem of the Python framework contains the drivers for the supported hardware targets. The Target component provides the interface between the InjectionProcesses subsystem and the hardware targets. It is implemented as an Abstract Base Class (<< ABC >>) and defines a number of abstract methods that are used by the injection processes. These abstract methods have to be implemented by each target in order for the target to be

supported. This approach enables to define target independent CFE injection processes and enables easy integration of new targets.

Currently, two targets are implemented, the NXP LPC 1768 and the STM32F0. These two targets were selected, because market studies show the NXP LPC and STM32 families are highly used in industry [34, 49]. On the left side, the driver of the NXP LPC 1768 is shown, which uses the PyOCD package to implement the Target interface [50]. This PyOCD package provides an interface to connect to the on-chip debugger of a real target using USB. The provided interface allows to halt and resume the target, set and remove breakpoints, read and write the CPU registers, etc. On the right side, the driver for the STM32F0 is shown, which uses the Telnet package to implement the Target interface. A telnet connection is setup with an openOCD server, which in turn connects to the on-chip debugger of the target [51]. Once connected to the openOCD server, the STM32F0 driver sends the necessary commands to it in order to inject the requested CFE.

InjectionProcesses subsystem: This subsystem uses the interface provided by the Targets subsystem to define three CFE injection processes. The *control flow aware* injection process enables the user to specify which part of the program has to be tested and which type of CFEs to inject [77]. The major advantage of this process is that it enables the user to deterministically inject inter-block and intra-block CFEs.

The *random in range* injection process injects random CFEs in a user defined range. An advantage of this process is that it enables the user to inject CFEs in a particular part of the program. This injection process not only injects inter-block and intra-block CFEs but also CFEs that jump between functions, if multiple functions are defined in the specified range.

The *extended CFE* injection process enables the user to target a specific range of the program and then injects all possible CFEs for that range [78, 79]. The process gradually steps though the program, and for each program counter (PC) value that is within the user-defined range, all possible CFEs are injected. This process enables users to achieve a high code coverage for the fault injection. All three injection processes are discussed in detail in Section 3.3.2.

CFE Injection Experiment component: This component defines the custom CFE injection experiments. In these Python scripts, the user defines which injection processes to use and for which target. Combined, these components and subsystems enable the user to implement a wide variety of CFE injection experiments for different targets.

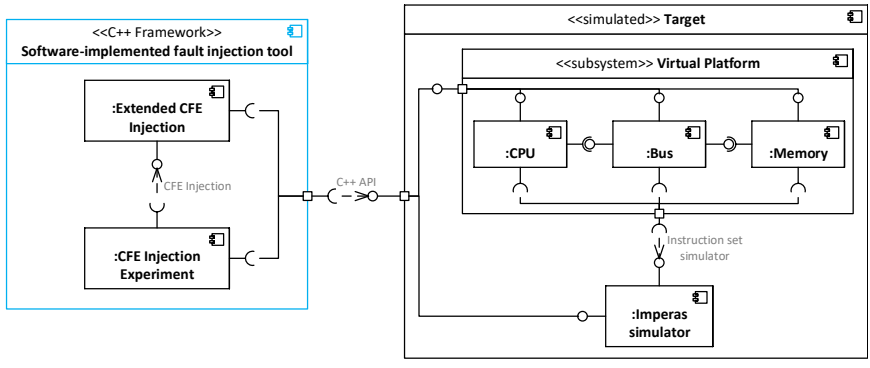


Figure 3.3: Overview of the C++ framework that enables CFE injection in simulated targets.

C++ Framework to support Simulated Targets

The C++ tool to support simulated targets, using the Imperas simulator, is shown in Fig. 3.3. As shown, a simulated target consists of a virtual platform, which defines the hardware to be simulated, and the Imperas simulator. The Imperas simulator is an instruction set simulator which enables to execute the target instructions at host (computer) speed [48]. It supports ARM, MIPS, PowerPC and many other targets. This fault injection tool has been built in C++ in order to use the provided C++ API that allows to control and influence the simulated target. Its architecture is simpler than that of the Python tool and only consists of two components, as the target interface and the supported targets are provided by the C++ API.

Extended CFE Injection component: This component provides the functionality for the *Extended CFE* injection process [78, 79]. As shown, the process uses the C++ API to interact with the simulated target. This interaction is needed to read and write the CPU registers.

CFE Injection Experiment component: This component contains the custom CFE injection experiment and interacts with the Imperas simulator to halt and progress the virtual platform.

3.3.2 Fault Injection Processes

This section discusses our three in-house developed fault injection processes, *control-flow-aware* injection, *random in range* injection and *Extended CFE*

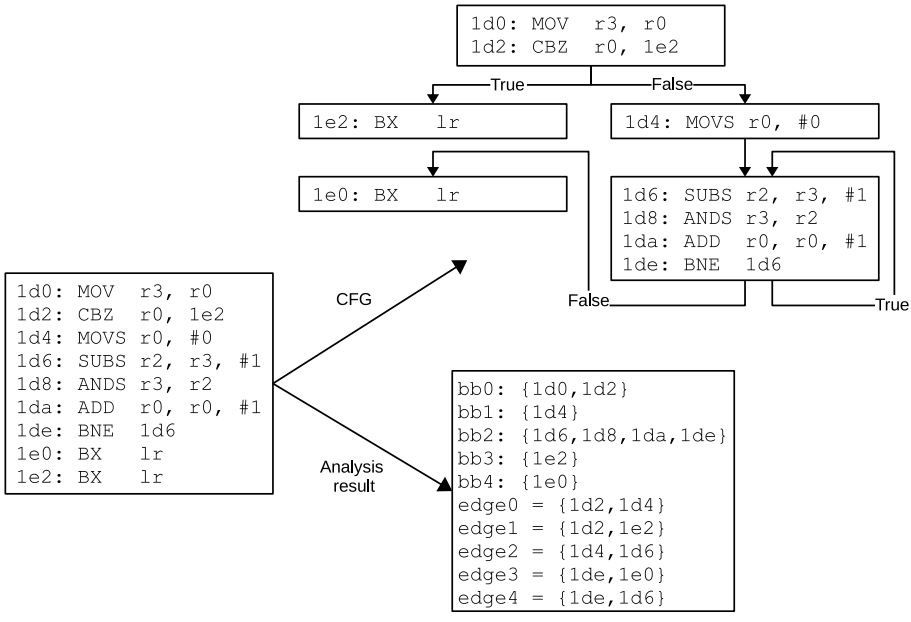


Figure 3.4: Illustration of the CFG analysis for the control-flow-aware injection process.

injection, in more detail. All processes use the PC of the target to inject CFEs as this is the most deterministic way to make them occur.

Control-flow-aware Injection Process

This process injects CFEs deterministically in one program function using the CFG of that target function. The CFG is created in memory by analyzing the disassembly of the target function. As depicted in Fig. 3.4, this analysis results in PC values mapped to basic blocks and edges converted to PC pairs. By using the in-memory CFG, the control flow of the target can be corrupted deterministically. The process is capable of injecting both inter-block and intra-block CFEs.

The following presents the process to deterministically inject an inter-block CFE. First, the PC of the target is read out and the basic block of the read out value is determined. Based on that information, a new value is calculated. This is done by first creating a list of all possible new values, flipping a bit at each bit position. Next, the non-existing PC values and the intra-block CFE values

are discarded. If this filtered list of possible new values contains more than one value, the new PC is chosen at random. This increases the fault coverage of the fault injection experiment. Finally, the new value is written to the PC.

Using the function of Fig. 3.4, injecting an inter-block CFE using this process occurs as follows.

1. The PC is read out: its value is `0x1d2`
2. Create a list of all possible values, by flipping one bit for each position. The list contains: {`0x1d3, 0x1d0, 0x1d6, ..., 0x9d2`}
3. Discard all non-existing values. The list still contains: {`0x1d0, 0x1d6, 0x1da`}
4. Discard all intra-block values. The list still contains: {`0x1d6, 0x1da`}
5. Choose a value of the remaining list randomly: `0x1da`
6. Write the new value to the PC: the PC now has the value of `0x1da`

The same process is used to deterministically inject intra-block CFEs. Instead of discarding the intra-block values, the inter-block values are discarded.

Random In Range Injection Process

This process injects random CFEs throughout a user-defined range of the program. It can be used to quickly setup a fault injection experiment that injects many different CFEs in order to get a first idea about the resilience of the covered code. The process needs two user-defined variables to execute:

- *cfeOriginList*: This variable contains all PC values that can be used as starting points of the CFEs to inject.
- *cfeDestList*: This list contains all PC values that can be used as destinations points of the CFEs to inject.

The execution flow to inject one CFE is shown in Fig. 3.5. The process starts by determining the origin address of the CFE by randomly selecting a PC value from the user-defined *cfeOriginList*. Once an origin address has been selected, the destination addresses is determined. First, all possible single-bit bit-flip values are calculated for the chosen origin address. Then only the values that

are also contained in the user-defined *cfeDestList* are kept. If this filtered list of destination addresses contains multiple values, then a random value from the list is selected as destination address. Depending on the selected origin address, the filtered list of destination addresses can be empty. In that case, the process starts over and selects another origin address.

Now that the CFE to inject has been defined, since the origin and destination addresses are selected, the actual injection can start. First, a breakpoint is set in the target at the origin address and the target resumes execution. After a certain amount of time, the target is halted and its PC is read out. The set breakpoint is also removed. Then, the read out PC is verified. Normally, it should be the origin address of the CFE to inject. However, it is possible that the selected origin address cannot be reached due to the control flow of the program. For example, if the selected origin PC belongs to the *true* path of a conditional branch, but during the current execution, the *false* path was executed, then the origin address of the CFE cannot be reached. When this is the case, the injection process starts over and chooses a new origin address. When the read out PC matches with the origin address, the CFE is injected. This means that the destination address is written to the PC register of the target, which is then resumed. Finally, the effect of the CFE is determined.

Based on the PC values provided in both the *cfeOriginList* and the *cfeDestList*, intra-block CFEs, inter-block CFEs and even CFEs that jump between program functions can be injected using this process.

Extended Control Flow Error Injection Process

This process gradually steps through a user-defined range of the program and injects all possible CFEs for that range. This process needs four user-defined variables to execute:

- *disasmFile*: This variable holds the path to the disassembly file of the target program. The file is needed to know which PC values are valid for the program.
- *range*: This variable indicates for which PC range the target program must have CFEs injected. The variable enables to target a specific function (or functions) to be covered, instead of the entire program.
- *endAddress*: This is the PC value that indicates the end of the current fault injection experiment. When this value is reached during the fault injection campaign, all desired CFEs have been injected and the campaign stops.

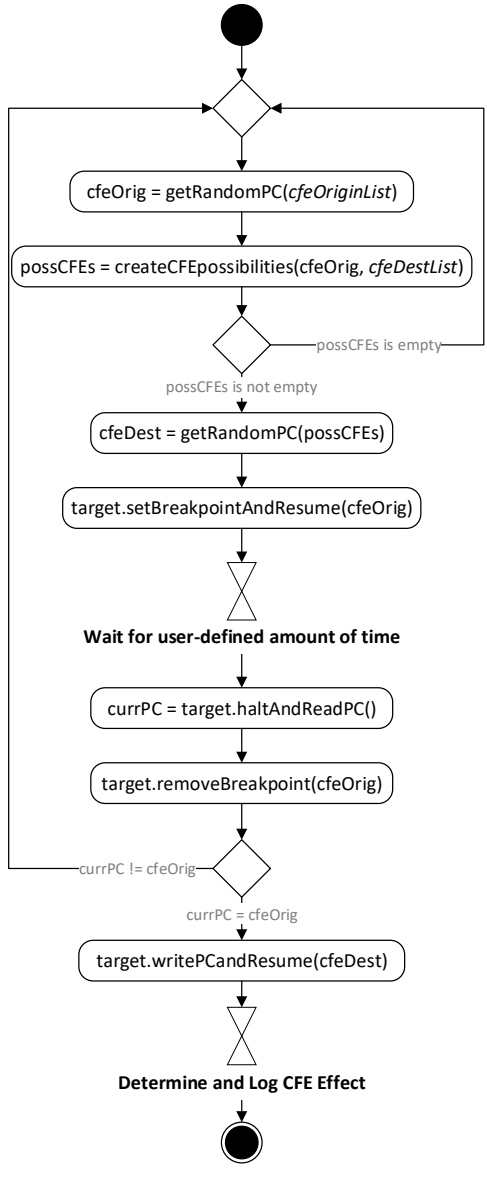


Figure 3.5: The execution flow of the *random in range* injection process.

- *minCFEs*: This final user-defined variable is mainly used to break the injection process out of case studies that have loops with lots of iterations. The variable indicates how many CFEs must be injected before the loop can be skipped to progress to the instructions passed it.

The detailed injection flow is shown in Fig. 3.6 using a simulated target. It starts with an initialization phase that initializes variables and creates all possible CFEs. Those possible CFEs are constructed from the user-defined options *disasmFile* and *range*. All created possible CFEs are within the user-defined *range*.

Once the possible CFEs are created, the execution part of the fault injection process starts. This phase begins with performing a check to determine whether or not the previous loop injected a CFE. In case a CFE was injected, a new instance of the virtual platform is created and a handler to the processor is obtained. Next, the processor executes the number of steps indicated by *execSteps* variable. The *execSteps* variable indicates how far into the case study the fault injection testing has already progressed. When no CFE was injected, the processor of the current instance of the virtual platform is progressed one step further and the corresponding variables are updated.

Next, the current PC of the processor is read out and the loop variable `CFE_Injected` is updated. The read PC value is compared to the user specified *endAddress* value, to determine whether or not the fault injection experiment should stop.

The next step in the algorithm is to determine whether or not the read out PC falls within the user specified *range*. When the PC is not in that range, a new CFE attempt is started and the execution phase restarts. In case the PC does fall within the range, its according CFE possibilities are extracted from all possibilities that were created at the start of the algorithm.

Finally, the process analyses whether or not for the current PC a new CFE must be injected. This is determined by comparing the current wanted possibility index (*possIndex*) with the number of possibilities for this PC. If a CFE must be injected, it is injected, analysed for its effect on the target program, the according variables are updated and the current instance of the virtual platform is deleted. Then, the execution part of the fault injection restarts.

When no CFE must be injected, a final check is executed. This final check determines whether or not the current PC must be removed from the *range* of allowed PC values. It does this by checking if the total amount of injected CFEs exceeds the limit provided by the user via the *minCFEs* option. When the total amount of injected CFEs exceeds the limit, the PC is removed. The *minCFEs*

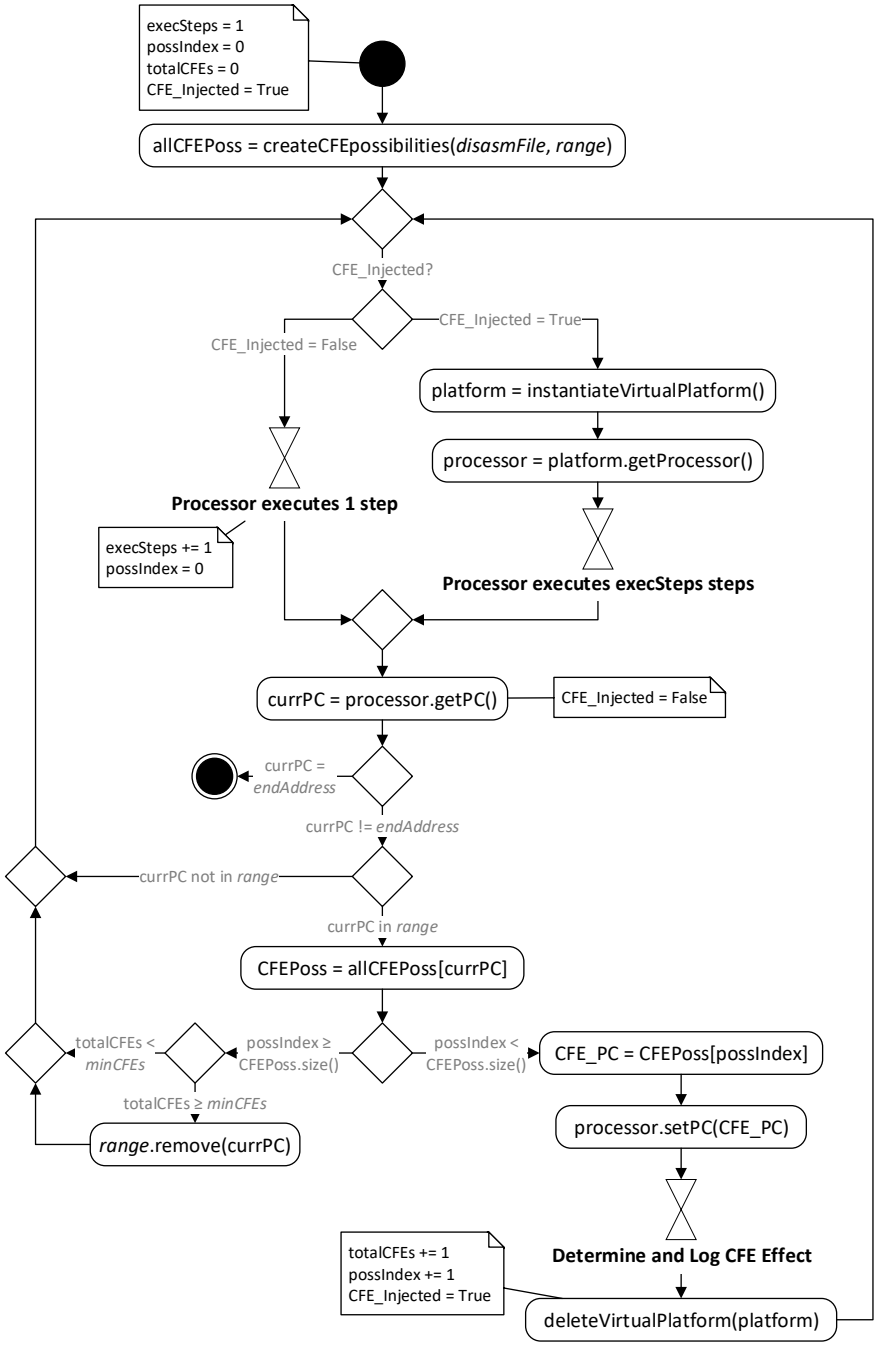


Figure 3.6: The execution flow of the *extended CFE* injection process.

option is mainly used to break the process out of case studies that have loops with a lot of iterations. In this case, the loop iterations will inject the same CFEs until the *minCFEs* limit is reached. Once reached, the PC values part of the loop will be removed from the allowed PC *range*, causing the fault injection experiment to skip them and test the remainder of the target program.

This process has also served as a basis to develop a similar DFE injection process. This extended DFE injection process also gradually steps over a user-defined *range*, but instead of injecting all possible CFEs, it injects all possible DFEs. For each instruction, the process analyses the used registers and flips all bits for that register. For example, if an instruction uses two 32-bit registers, the process will inject 64 independent DFEs for that instruction. This process has been used to validate the idea published in [99]. In this paper, a first experiment on an in-house built technique that can detect both CFEs and DFEs is presented. During that experiment both the extended CFE and DFE injection processes were used to measure the techniques error detection ratio and SDC ratio.

3.3.3 Supported Targets

As discussed previously, the Python framework currently has two implemented targets, i.e. the NXP LPC 1768 and the STM32F0. However, these targets use the PyOCD Python package and the telnet Python package to interact with the on-chip debugger of a hardware target. This means that in theory, any hardware target supported by either the PyOCD package or that can be accessed through a Telnet session, such as an openOCD server, can be implemented and thus be supported by our fault injection tool. Currently, the PyOCD package supports up to sixty targets and openOCD supports up to 190 targets through more than seventy debug adapters [50, 51]. To conclude, many extra targets can be added to our Python framework using the existing supported packages.

Regarding the C++ framework, as the provided C++ API is a generic API, it supports all targets of the Imperas simulator [48]. Our latest used version, 20181114, supported up to sixty processor families.

3.4 Automatic Implementation of CFE Detection Techniques

This section discusses our developed GNU Compiler Collection (GCC) plugin, the tool that automatically implements the discussed CFE detection techniques for the provided source code. The GCC plugin is available as open-source

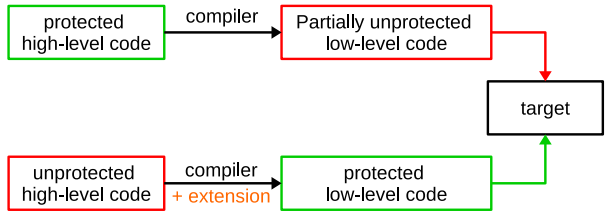


Figure 3.7: The problem with high-level protection and the solution provided by our compiler extension.

project on Github, licensed under the MIT Expat license: `https://github.com/MGroupKULeuvenBrugesCampus/CFED_Plugin`.

3.4.1 Purpose

Because the SOTA techniques describe their operations with high-level instructions, this could lead an embedded systems engineer to believe such techniques have to be implemented in high-level code. 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 CFE. In contrast, SOTA literature describes an error detection ratio of 75 % and higher. Upon inspection of the low-level code, generated with the *objdump* tool, we noticed that the inserted instructions were only protecting a small part of the program. This problem is caused by the fact that the human-readable high-level source 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 in high-level source code. Fig. 3.7 visualizes how the compiler produces partially unprotected low-level code from the high-level code.

The solution to this problem is implementing the CFE detection techniques in the low-level code. Performing this manually, however, is arduous and error-prone. Therefore, we created a compiler extension that can automatically implement all CFE detection techniques discussed in this thesis in the low-level code of the program. As Fig. 3.7 shows, the extension enables to create protected low-level code from unprotected high-level code. As the GCC toolchain is one of the most widely used toolchains for embedded system development, we created an extension for this toolchain. More specifically, we selected the GCC toolchain

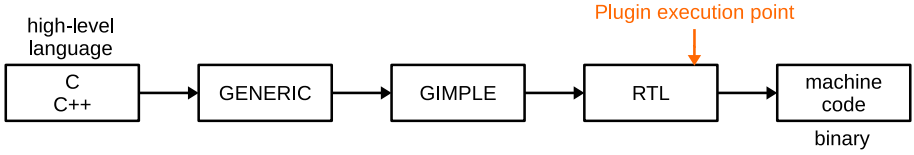


Figure 3.8: The different intermediate languages of GCC and the plugin execution point.

for the bare-metal ARM development *arm-none-eabi-gcc* and as extension, we developed a plugin.

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

3.4.2 GCC Internal Steps

GCC compiles high-level source code to low-level code through several intermediate languages, as depicted in Fig. 3.8. First high-level language is translated into GENERIC, then into GIMPLE, followed by a translation into REGISTER TRANSFER LANGUAGE (RTL) to end in MACHINE CODE. This process is done via several so called 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. Our plugin executes after *pass_free_cfg*, which is an RTL pass and is only executed once per compiled function of the program. It indicates that the CFG will not change anymore during the further compilation process, making it the ideal point for the plugin to insert the extra instructions. As shown in Fig. 3.8, this is rather 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, these must be reserved using the GCC option *-ffixed-r<number>* during compilation.

3.4.3 GCC Plugin

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, by using one of our plugin arguments or by using our function attribute, a function must have a CFE detection technique implemented

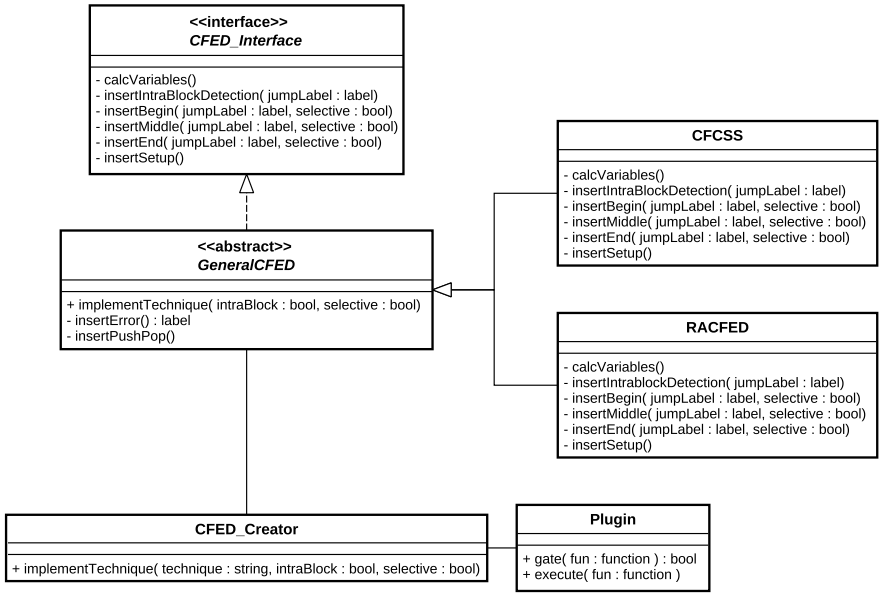


Figure 3.9: UML class diagram showing the implementation of our plugin in pseudo-code.

or not. If this analysis finds that the current function must be protected, the selected technique is implemented.

The implementation of our plugin is shown in more detail in Fig. 3.9. The shown UML class diagram depicts the classes and methods to implement a CFE detection technique. To keep the diagram as clear as possible, only two of all 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 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. The 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. The *function* argument either specifies one function that has to be protected or is empty, which means all functions need to be protected. A function with the `noProtection` attribute set, will be skipped. The `gate` method returns either True or False indicating whether or not the function needs to be protected. When the `gate` method returns True,

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:         bool intraBlock ← readArgument(techniqueType)
5:         string technique ← readArgument(technique)
6:         bool selective ← readArgument(selective)
7:         cfedCreator ← new CFEDcreator()
8:         cfedCreator.implementTechnique(technique, intraBlock, selective)
```

the `execute` method is called. This method implements the actual compilation pass and will implement the selected CFE detection technique, as shown in Alg. 1. It determines if intra-block CFE detection should be implemented or not by reading the *techniqueType* plugin argument. Then it determines which of the supported techniques should be implemented by reading the *technique* plugin argument. Next, it reads the *selective* plugin argument to determine whether or not the selected technique should be implemented selectively. Finally, it uses those three arguments to call the `implementTechnique` function of the `CFEDcreator` class.

The `CFEDcreator` is a class that knows which CFE detection techniques are supported and how to build them. As Alg. 2 shows, first the instruction set architecture (ISA) of the target processor is determined. With this information, the correct registers can be provided to the technique that must be implemented. Next, it creates an instance of the selected techniques and finally calls its `implementTechnique` method. As the name gives away, that method makes sure the selected technique is implemented for the technique.

The `implementTechnique` method of the `GeneralCFED` class implements the selected technique and is shown in Alg. 3. First the 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) continue operation with reduced functionality, better known as graceful degradation. 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

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

```
1:  function CFEDcreator::implementTechnique(technique, intraBlock,
                                                          selective)
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.implementTechnique(intraBlock, selective)
```

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 at the beginning of each basic block are inserted, ending with adding the necessary instructions at the end of the basic block. Next to the label of the error handler, these methods are also provided with the value indicating whether or not the technique has to be implemented selectively. Then, 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 enable 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. This stack is a second stack next to the regular stack of the used microcontroller, to make sure the regular stack remains valid.

Although the `GeneralCFED` class implements the `implementTechnique` 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 at the right side of Fig. 3.9. As depicted, the specific technique classes `RACFED` and `CFCSS` implement the needed methods but nothing more. This makes it easy to support new techniques, as they only need to implement the six needed methods defined by the interface called `CFED_Interface`.

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

```
1: function GeneralCFED::implementTechnique(intraBlock, selective)
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, selective)
8:         insertBegin(jumpLabel, selective)
9:         insertEnd(jumpLabel, selective)
10:    insertSetup()
11:    insertPushPop()
```

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. 3.10. 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 in the top half of the figure 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 enables to restore them. Applied to the example, the situation in the bottom half of Fig. 3.10 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 desired.

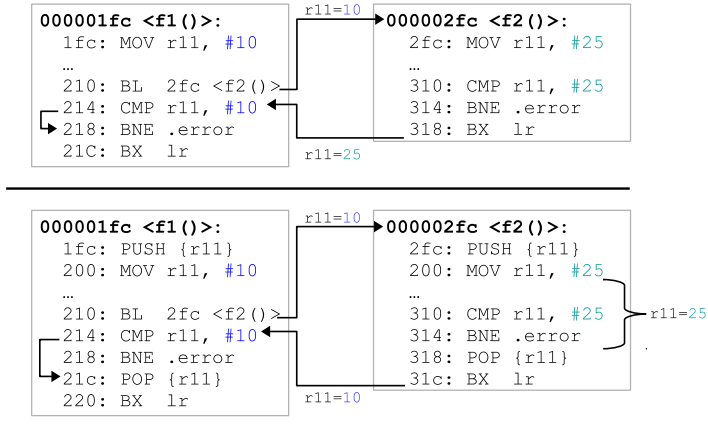


Figure 3.10: ARMv7-M code showing the need for extra PUSH and POP instructions.

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.

3.4.4 Example Usage

To show the usage and outcome of our plugin, we used the BC algorithm of the MiBench benchmark suite [46]. When compiled with the *arm-none-eabi-gcc7.3* toolchain for an ARM Cortex-M3, it has the CFG shown in Fig. 3.11. It is a small algorithm, containing nine instructions, and therefore has a rather small and easy CFG, making it ideal to explain the workings of the plugin.

Necessary Adjustments

To use our plugin, both the source code and the compiler flags have to be adjusted. As shown in Listing 3.1, 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

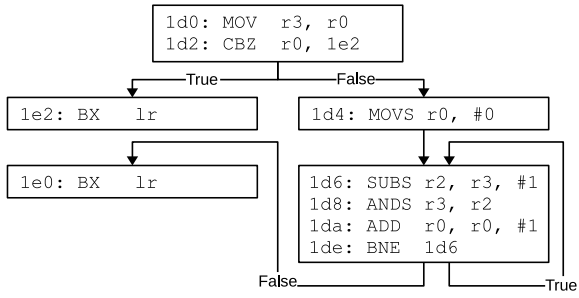


Figure 3.11: The CFG of the BC algorithm when compiled for an ARM Cortex-M3 using *arm-none-eabi-gcc7.3*

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

Listing 3.1: Adjustment to be made to the source code to use our plugin

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 custom functionality can be provided.

The changes to be made in the compiler flags are shown in Listing 3.2. In total, seven extra compiler flags have to be added, i.e. five 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 five plugin-related flags specify where to find the plugin (line 4), which

```
1  # 1) Define the plugin name
2  PLUGIN_NAME = CFEDplugin
3  # 2) Specify where to find the plugin
4  COMPILER_FLAGS += -fplugin=<pathToPlugin>/$(PLUGIN_NAME).so
5  # 2) Only protect the bit_count algorithm
6  COMPILER_FLAGS += -fplugin-arg-$(PLUGIN_NAME)-function=bit_count
7  # 3) Implement both inter-block and intra-block CFE detection
8  COMPILER_FLAGS += -fplugin-arg-$(PLUGIN_NAME)-techniqueType=
       fullCFED
9  # 4) Implement RACFED
10 COMPILER_FLAGS += -fplugin-arg-$(PLUGIN_NAME)-technique=RACFED
11 # 5) Do the full implementation, not selective
12 COMPILER_FLAGS += -fplugin-ar-$(PLUGIN_NAME)-selective=0
13 # 6) RACFED needs one register (r11)
14 COMPILER_FLAGS += -ffixed-r11
15 # 7) The run-time variable stack needs a stack pointer (r7)
16 COMPILER_FLAGS += -ffixed-r7
```

Listing 3.2: Adjustment to be made to the compiler flags to use our plugin

function to protect (line 6), what type of technique to apply (line 8), which technique to implement (line 10) and whether or not to implement it selectively (line 12). 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, selective=0` [84].

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. 3.12. Indicated in bold are the instructions added 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 BC 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 RACFED, 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

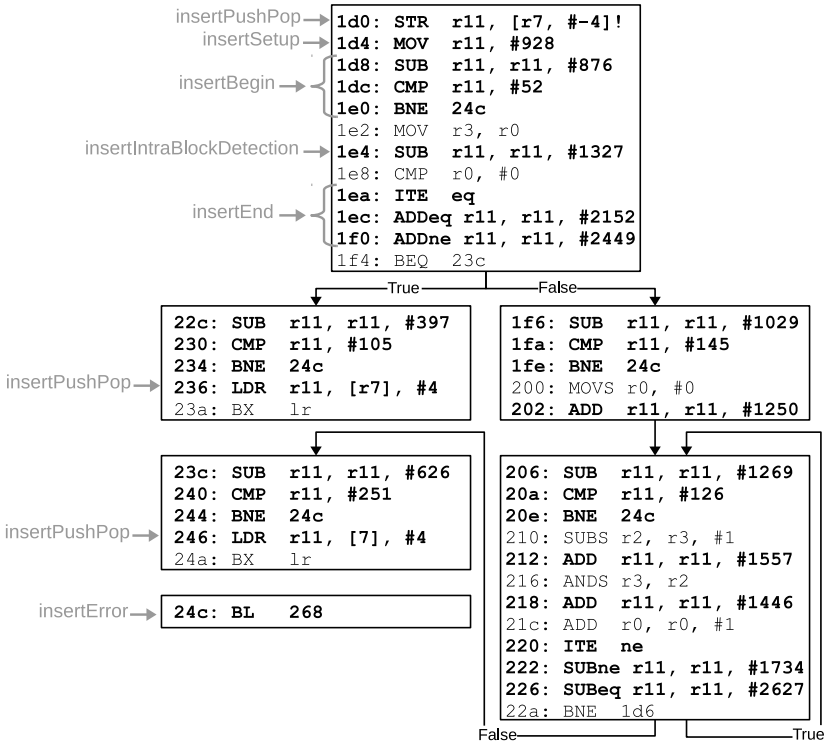


Figure 3.12: The CFG of the BC algorithm when compiled for an ARM Cortex-M3 using *arm-none-eabi-gcc7.3* and our plugin

updates inserted. As we will describe in Section 5.2.2, 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 is a final update of the run-time variable. To conclude the final instruction of the BC algorithm, before exiting, is now the POP of the run-time variable (`LDR`).

3.4.5 Experiments

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

Experiment Setup

To prove the validity of our plugin, we implemented our RACFED technique (cfr. Chapter 5) for the eight case studies introduced in Section 3.1 both manually in the high-level code and using the plugin in the low-level code. Next, we performed a fault injection campaign using our *Extended CFE* fault injection process presented in Section 3.3.2. 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 [48].

Results

The results of the fault injection campaign are shown in Fig. 3.13. 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 undesired 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. 3.13, 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

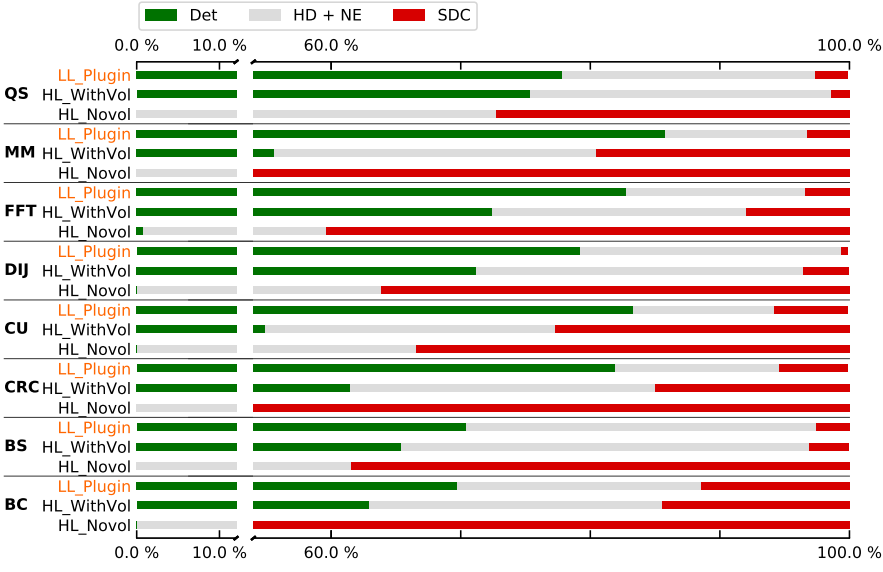


Figure 3.13: The results of the fault injection campaign.

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. 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 %. These numbers are also clearly indicated in Fig. 3.14

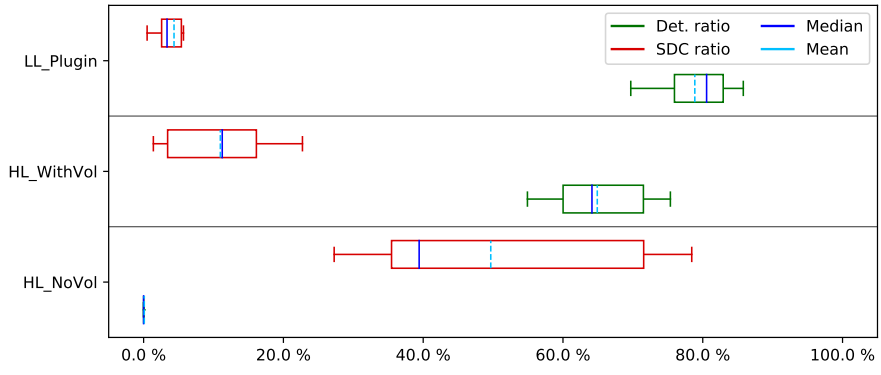


Figure 3.14: The error detection ratio and SDC ratio of the three configurations.

3.4.6 Regression Testing

This plugin is being developed incrementally, meaning that small portions, e.g., one technique, were added to the plugin per development cycle. The main advantage of this strategy development strategy is that a working plugin is delivered per development cycle, enabling experiments to be set up more quickly. Building software, in this case the plugin, incrementally means there is a need for regression testing. In our case, this test assures that previously supported techniques and case studies are still supported. This section discusses the developed regression test tool in more detail. First, we discuss the test criteria. Then, the test setup is presented. Next, the general execution flow of the tool is shown. Finally, an example test result is shown.

Test Criteria

The regression test should assure that the previously working functionality is still operational once a new development cycle has finished. In our case, this means that the regression test has to verify that:

1. All selected case studies can still be compiled for each supported technique. When adding your own compiler step, it is possible to corrupt the compilation process. Thus the compilation of the case studies must be verified. Furthermore, depending on the modifications made during the current development cycle, the implementation of multiple techniques

might have changed. Therefore, compilation with each supported technique has to be verified.

2. All selected case studies still execute correctly for each of the supported techniques and for each of the datasets, once compiled and deployed to the target. It is not because the case study compiles, that it executes correctly, so its functionality has to be explicitly verified. Since this can only be verified at run time, we have defined four run-time statuses for each case study:

    - **False CFE Detection:** This status means that the implemented CFE detection technique has falsely detected that a CFE has occurred. This means that the CFE detection technique was wrongly implemented and the plugin must be adapted. This is a negative run-time status.
    - **Hardware Fault:** This status means that a hardware fault detection mechanism present in the target was triggered. This type of fault is mostly triggered if the plugin has inserted extra instructions at illegal locations. This is again a negative run-time status.
    - **Data Corruption:** This third run-time status indicates that the case study was able to execute, but has produced a wrong output. Again, this is a negative run-time status.
    - **Correct Execution:** This fourth and final run-time status indicates that the case study has executed correctly. This is the desired status.

Test Setup

The four defined run-time statuses are the same statuses we used during fault injection experiments, so the source code of the case studies needs no modification to be used during the regression testing.

The regression testing tool itself is implemented as a Python script, using the same driver as the fault injection tool to access the PC of the target to detect the run-time statuses implemented as infinite loops. Furthermore, through Python it is possible to power-cycle the USB-ports of the USB hub in use. This power-cycle functionality is needed to give a hard reset to the target. Once the new case study has been uploaded, the target has to load it into its instruction memory and start executing it. For the NXP LPC 1768, this can only be done by giving it a hard reset, hence the need for the power-cycle of the USB port.

We use five targets because we have defined five datasets per case study. Each target is coupled to a certain data set, which gives our tool the ability to verify

the execution of the five data sets concurrently. Each target also has its own USB port on the USB hub, enabling to hard reset each target individually.

General Execution Flow

The general execution flow of the regression test is shown on the flowchart in Fig. 3.15. As shown, the same process is repeated for each supported technique. Then, for each technique, all case studies are tested. The first step executed per case study, is compiling the case study with the current CFE detection technique for all datasets. When the compilation fails, the error is recorded and the test restarts for another technique. Once a technique fails to generate buildable code, there is no need to further test this technique. If and only if it produces buildable code for all case studies, for all datasets, the technique should be submitted to the run-time tests.

If the compilation of all five data sets for the current case study succeeds, the run-time tests are executed. First, the compiled code is uploaded to the targets, with each target receiving the code for its assigned data set. Then, the Program Counter addresses of the four infinite loops, i.e. the run-time statuses, are determined by analyzing the compiled code. Once determined, the targets receive a hard reset, which makes sure the current case study is loaded into the instruction memory and that it is started.

Once the case study has executed, for all five data sets, the run-time statuses are determined and logged. Finally, this process repeats until no more CFE detection techniques are left.

Example Output

When executing the regression test, all results are logged into a .csv file. An excerpt of such a file is given in Table 3.2. As illustrated, a useful overview is given for each technique, per case study, per data set. The first rows show that for CFCSS, the BC algorithm did not compile for any of the data sets. Each time, the compiler crashed due to a segmentation fault. Due to these compilation errors, the other seven case studies were not considered for this technique.

The next few displayed rows show that for RACFED, the BC algorithm was correctly executed for all five case studies. For the BS algorithm, however, data set 2 resulted in the *False CFE Detection* run-time status and data set 4 resulted in the *Data Corruption* run-time status. This identifies that possibly some special control flow is executed for those two data sets and RACFED

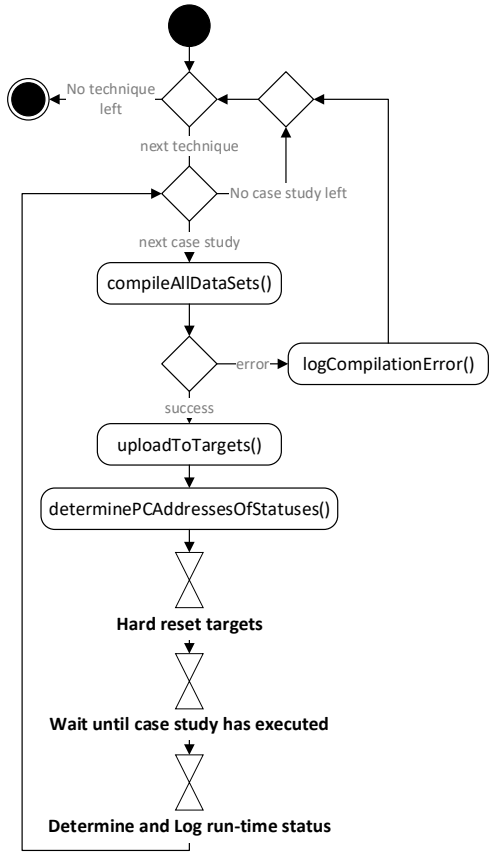


Figure 3.15: The execution flow of the regression test tool.

was not correctly implemented for those control flow cases. If RACFED is a technique that was previously supported, then this regression test has revealed that the changes made during the current GCC plugin development cycle have broken this technique.

If RACFED is a newly added technique, then the regression test has revealed that it was incorrectly implemented. Thus, although meant as a regression test tool, the Python script can also be used as a verification test for newly added CFE detection techniques.

Table 3.2: Excerpt of the created log file once the regression test has executed.

	...	
CFCSS		
BC	data set 0	Compilation Error: Segmentation Fault
	data set 1	Compilation Error: Segmentation Fault
	data set 2	Compilation Error: Segmentation Fault
	data set 3	Compilation Error: Segmentation Fault
	data set 4	Compilation Error: Segmentation Fault
RACFED		
BC	data set 0	Correct Execution
	data set 1	Correct Execution
	data set 2	Correct Execution
	data set 3	Correct Execution
	data set 4	Correct Execution
BS	data set 0	Correct Execution
	data set 1	Correct Execution
	data set 2	False CFE Detection
	data set 3	Correct Execution
	data set 4	Data Corruption
	...	

3.5 Conclusions

In this chapter we presented the developed tools, the selected case studies and the selected characterization criteria used throughout the various experiments discussed in this thesis. We selected eight case studies that cover a wide range of embedded systems domains. They also have varying CFGs, enabling a thorough validation of the CFE detection techniques.

To characterize the numerous discussed techniques, we selected three criteria: error detection ratio, execution time overhead in an error-free run and code size overhead. The error detection ratio is determined through fault injection, the execution time overhead is measured using a timer on the hardware target and the code size overhead is measured using the GNU *size* tool.

Because no fault injection tool was available to us to measure the error detection ratio, we developed our own. In this chapter we have presented the Python framework for hardware targets, the C++ framework for simulated targets and their supported injection processes. By defining three deterministic injection

processes, CFEs can be injected using many different approaches, enabling the validation of the CFE detection techniques.

This chapter concludes by discussing the implementation of our compiler extension, which has the form of a GCC plugin. The plugin works on a low-level intermediate language of GCC, called RTL, of the ARMv6-M and ARMv7-M ISAs and can implement the following methods: CFCSS, YACCA, YACCA_CMP, ECCA, RSCFC, SEDSR, SCFC, SIED, RASM and RACFED. Then, the internal working of the compiler extension was shown and discussed using the BC algorithm of the MiBench benchmark suite. Following, 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 case studies. Finally, we discussed the developed regression test tool to make sure that changes made during the development of the plugin did not break any working functionality.

Chapter 4

Experimental Study on Inter-block CFE Detection Techniques

The content and results of this chapter have been published in [82, 83]. Since their publication, the results presented in this chapter have been updated by using updated versions of the tools and our latest insights.

Using the various parts of Chapter 3, this chapter presents the experimental comparative study performed on the eight selected SOTA techniques discussed in Section 2.3. First, we present the experiment setup, followed by the results. To conclude, we derive five guidelines to build an optimal technique from the collected data.

4.1 Experiment Setup

To compare the eight selected SOTA techniques with one another, we implemented them for the eight case studies. In order to do a valid comparison, we only implemented the inter-block portion of the RSCFC and SIED techniques, since the remaining SOTA techniques can detect inter-block CFEs only.

Next, we performed a fault injection campaign and measured their code size and execution time in an error-free state. For the fault injection campaign, we injected 2000 inter-block CFEs for each combination of case study, technique

Figure 4.1: The five NXP LPC 1768 targets stacked on top of each other.

and dataset. This means that each technique is submitted to 10 000 CFEs (5 datasets × 2000 CFEs per dataset) per case study and, in total is submitted to 80 000 CFEs (8 case studies × 10 000 CFEs per case study). To inject inter-block CFEs only, we used our in-house built control-flow-aware injection process, discussed in Section 3.3.2.

As hardware target, the NXP LPC 1768 was selected, which is an ARM Cortex-M3 driven microcontroller running at 96 MHz, including 512 kB FLASH and 32 kB RAM. Because each case study has five data sets, we used five targets and assigned one target per case study. This setup is shown in Fig. 4.1 and enabled us to inject CFEs in all five data sets concurrently.

Some setup and validation code is required to launch the case studies, select the dataset and to verify their results. However, the focus of the protection techniques lies on the implementation of the proposed algorithms and CFEs are only injected while the algorithm is executing.

4.2 Results

Using the fault injection experiment results and the measured overhead, the eight techniques can be compared to one another. First, we compare them based on error detection ratio and SDC ratio. Next, we compare them based on imposed overhead.

4.2.1 Error Detection Ratio

The results of the inter-block CFE injection campaign are shown in Fig. 4.2. For each combination of technique and case study, we show the fault injection results based on the four categories of the error effects. Because the HD and NE categories are not in our control, these are shown as one light-gray bar. The desired result, the Det. category, is shown in green and the undesired result, the SDC category, is shown in red.

Analyzing the chart, it can be seen that for most case studies, either CFCSS or YACCA_CMP has the highest error detection ratio. In contrast, techniques such as RSCFC, SEDSR and SIED are unable to detect half of the injected CFEs for most case studies. When looking at the bigger picture, the techniques can be divided into two groups. CFCSS, YACCA, YACCA_CMP and ECCA yield an average error detection ratio of 70 %, 69 %, 75 % and 70 % respectively and form the first group. RSCFC, SEDSR, SCFC and SIED yield an average detection ratio of 45 %, 28 %, 54 % and 37 % respectively and constitute the second group. The difference in error detection ratios between both groups relates to how the techniques compute their run-time signature values. CFCSS, YACCA, YACCA_CMP and ECCA compute the run-time signatures gradually over multiple basic blocks. The run-time signatures hence contain information about previous basic blocks. This strategy leads to a high detection ratio, because once a signature update is skipped, the run-time signature can no longer hold the correct value. The next signature verification instruction will detect the incorrect value and report the CFE.

The other group of techniques compute the signature locally, which leaves room for undetected CFEs. SEDSR, SCFC and SIED do not even compute the new run-time signature value, they just assign it a new value using the `MOV` instruction. This means that any CFE that lands before the local signature update instruction, regardless of how many instructions have been skipped, will remain undetected.

An exception in this group is the RSCFC technique. While it uses gradual updates to compute the new run-time signature value, it has a low error detection

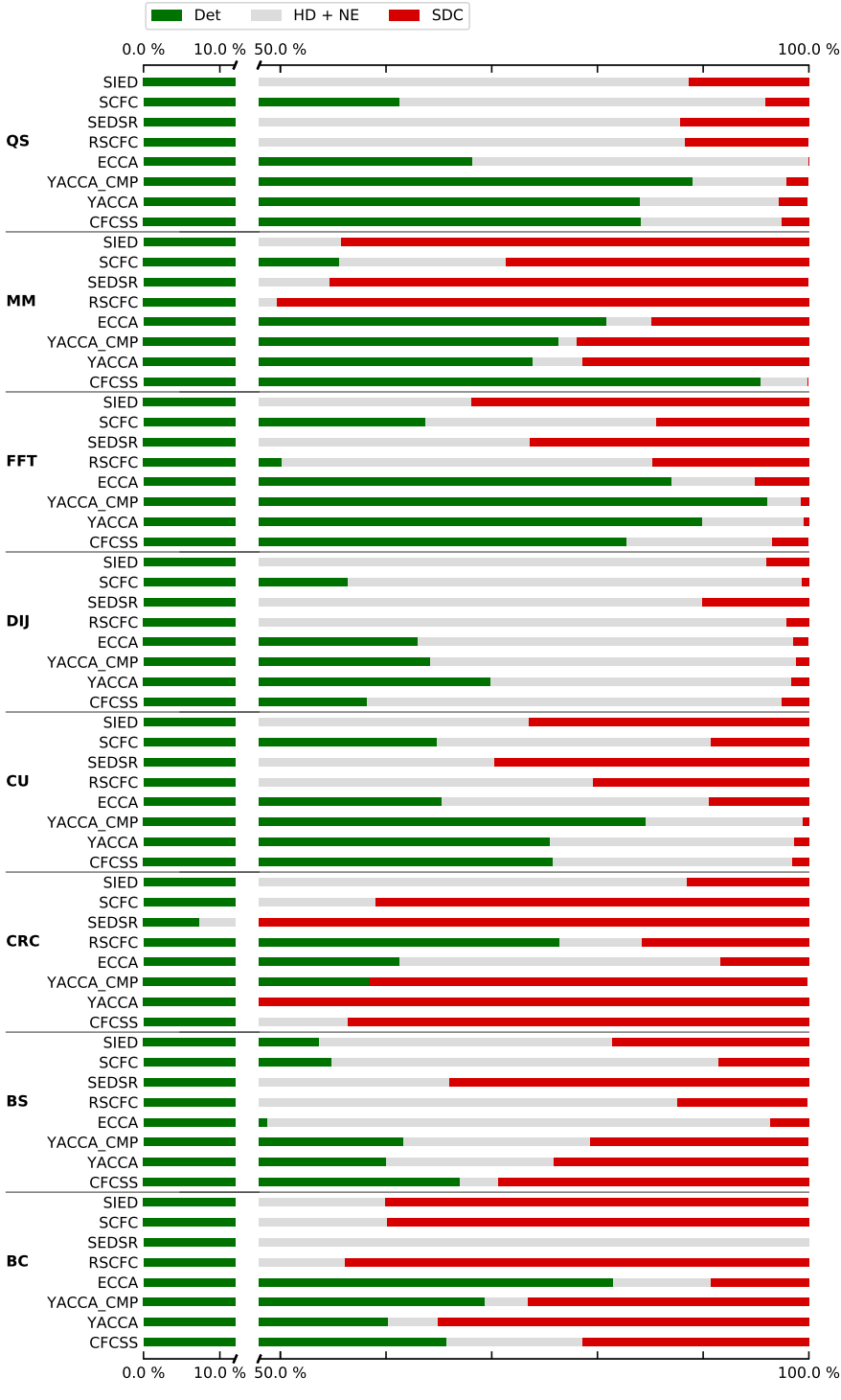


Figure 4.2: Bar chart showing the results of the performed inter-block CFE injection campaign.

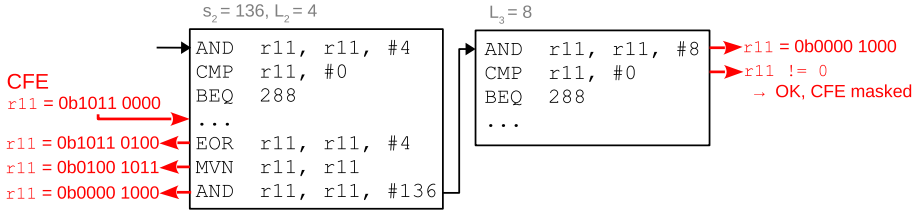


Figure 4.3: Partial CFG showing how a CFE is masked during the second run-time signature performed by RSCFC.

ratio. This is due to two CFE masking elements in its implementation. A first CFE masking occurs at the beginning of each basic block, when the signature is updated a first time and is verified. As shown Fig. 2.10, the update and compare instructions serve to verify whether or not a CFE has occurred by checking if one specific bit has been set in the signature. This means that any signature value that has the desired bit set, is deemed as correct which lowers the techniques error detection ratio.

The second CFE masking occurs at the end of the basic block, during the second run-time signature update. The purpose of this second update is to set the correct bits that indicate the valid successors of the current basic block. As described in Section 2.3.4, the update uses three bitwise operations, EXCLUSIVE OR operation, followed by a BITWISE INVERSE operation and finalized with an BITWISE AND operation. However, these three operations will always produce a value with the correct *successor bits* set when the run-time signature matches the CFG Locator bits on those positions. To clarify with an example, consider the partial CFG shown in Fig. 4.3. In this partial CFG, the most left basic block only has the right basic block as successor, so in an error free run the bit on position 3 is set by updating the run-time signature to a value of 136. However, when a CFE occurs which holds a wrong signature value but with the bit on position 3 holding the correct value, it can be masked. The update procedure in the most left basic block updates the signature to a value of 8, which is in fact wrong, but deemed correct in the run-time signature verification in its successor basic block.

4.2.2 Detection Cost

The code size cost of the SOTA techniques is shown in Fig. 4.4. This cost is determined by how many instructions are added per basic block. SEDSR shows to perform the best for all case studies, with CFCSS being a close second. More

general, we can divide the techniques into two groups, i.e. a low code size cost group and a high code size cost group. The low code size cost group consists of SEDSR, CFCSS, RSCFC, SIED and SCFC, which introduce between 4 to 9 extra instructions per basic block. The high cost group, which consists of YACCA, YACCA_CMP and ECCA, introduce between 3 to 24 extra instructions per basic block. This explains the difference between the two groups.

Overall, the code size cost is limited, because only a small part of the code base for each case study is protected. As discussed in Section 3.1, the code base of each case study consists of startup code, the actual algorithm and some result verification code, and only the actual algorithm is protected by the CFE detection technique. The algorithm is however a small part of the code base and thus results in less bespoke differences between most of the techniques.

When considering the execution time cost, presented in Fig. 4.5, both the number and the type of the added instructions are important. Again, the techniques can be divided into the two groups. SEDSR, CFCSS, RSCFC, SIED and SCFC add computationally inexpensive instructions, while YACCA and ECCA add computationally expensive instructions. To execute YACCA and ECCA, two unsigned integer divisions must be performed and although the Cortex-M3 has a hardware division unit, these divisions are far more expensive than simple additions and bit manipulations [38]. YACCA_CMP is an exception to this categorization. Although it inserts only computationally inexpensive instructions, it does belong to the high execution time cost group because it adds on average 15 instructions to each basic block.

Regarding the CU and FFT case studies, Fig. 4.5 shows some unexpected results. The protected code executes as fast or even faster than the unprotected code. This is because the compiler generates different assembly code for these two case studies when limiting the available registers. This different code executes faster than the code generated with all registers available and thus produces this unexpected result for the the CU and FFT case studies.

4.2.3 Experiments Conclusion

Based on the performed study, we can draw the following conclusions. When only considering error detection ratio, CFCSS, YACCA, YACCA_CMP and ECCA are the best techniques with an average error detection ratio of 70 %, 69 %, 75 % and 70 % respectively. The other considered techniques only have an average error detection ratio of 54 % or less. This difference in error detection ratio is clearly shown in Fig. 4.6 and is caused by the usage of gradual updates by CFCSS, YACCA, YACCA_CMP and ECCA.

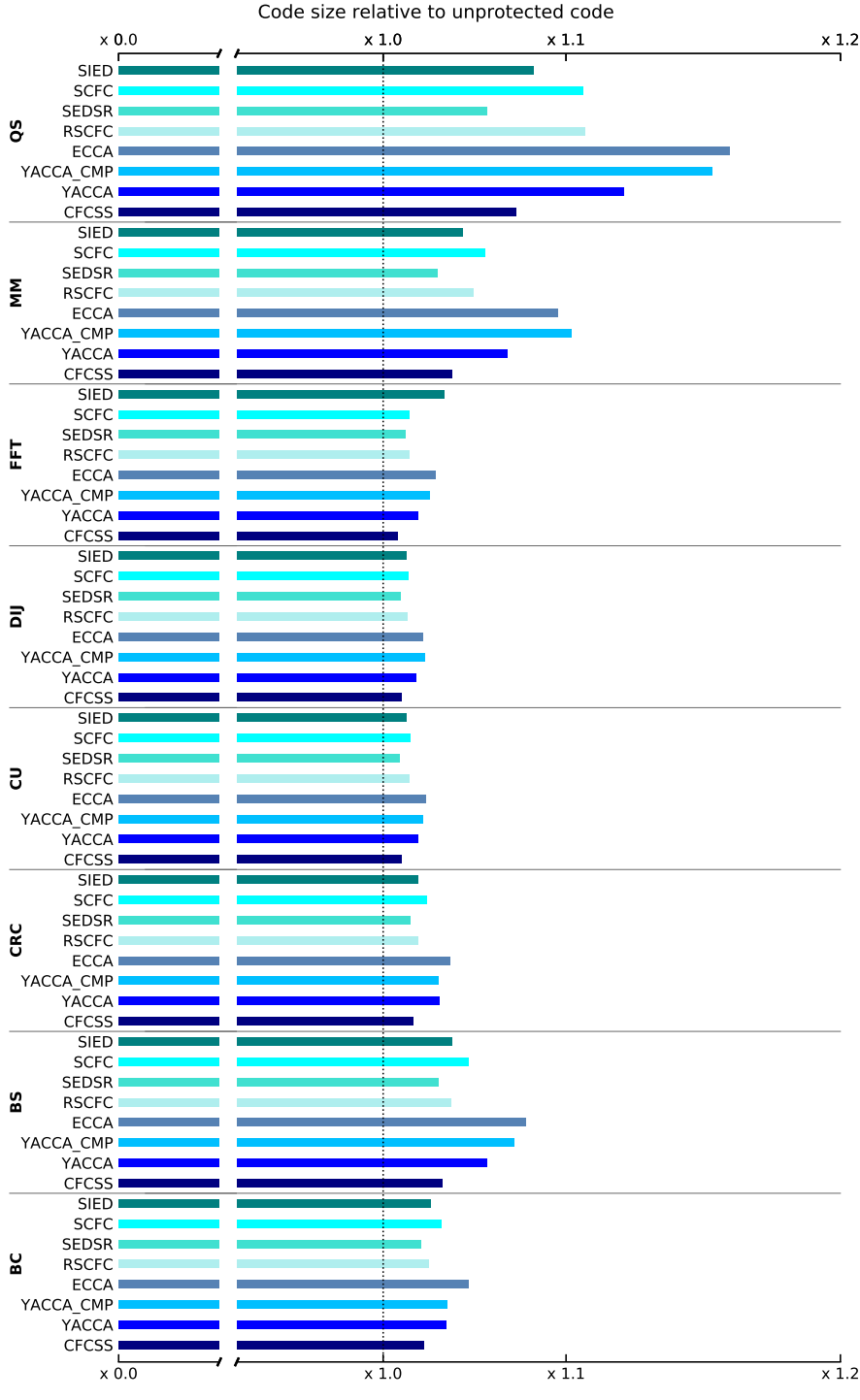


Figure 4.4: Code size cost of the SOTA techniques relative to the unprotected case studies.

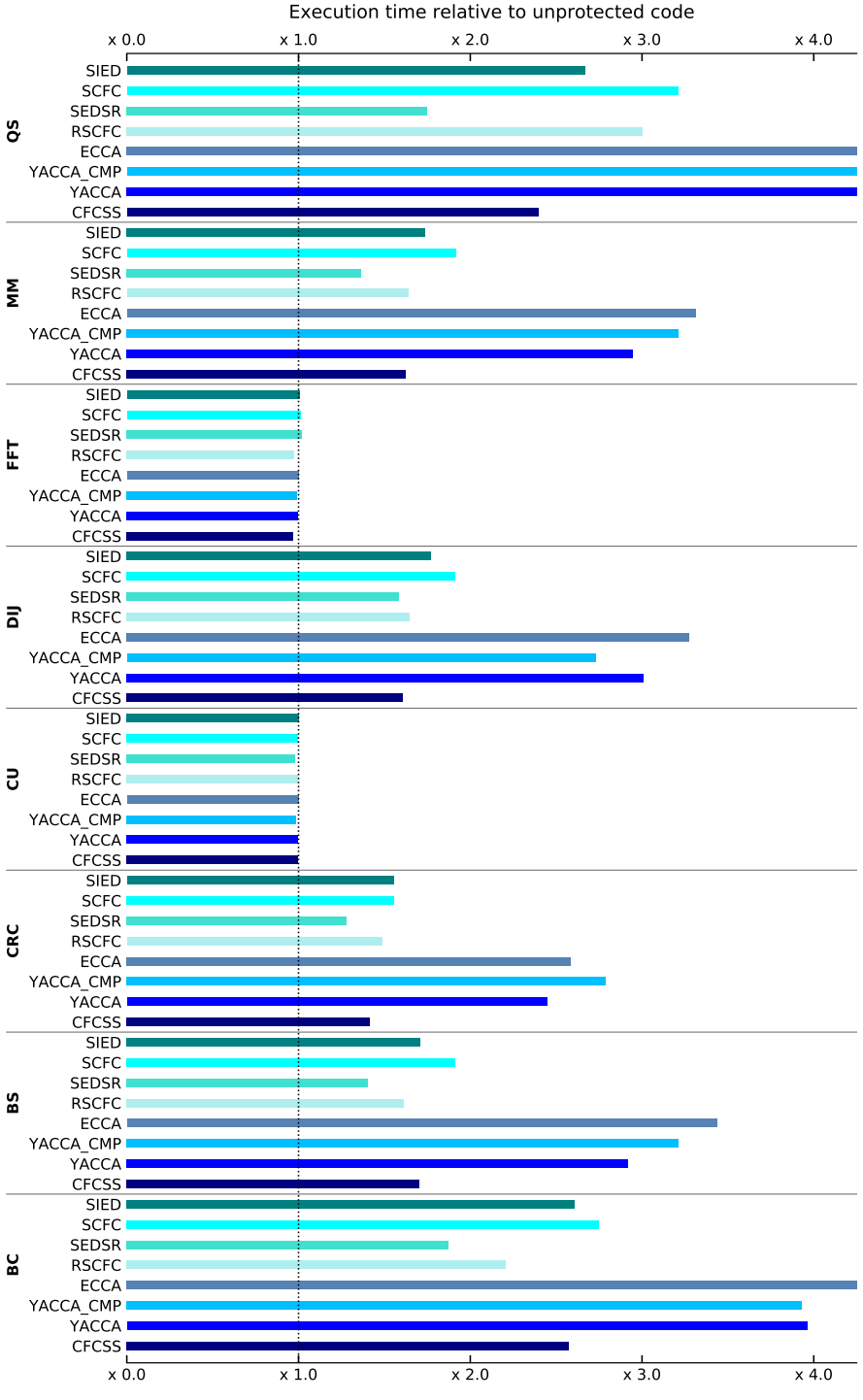


Figure 4.5: Execution time cost of the SOTA techniques relative to the unprotected case studies.

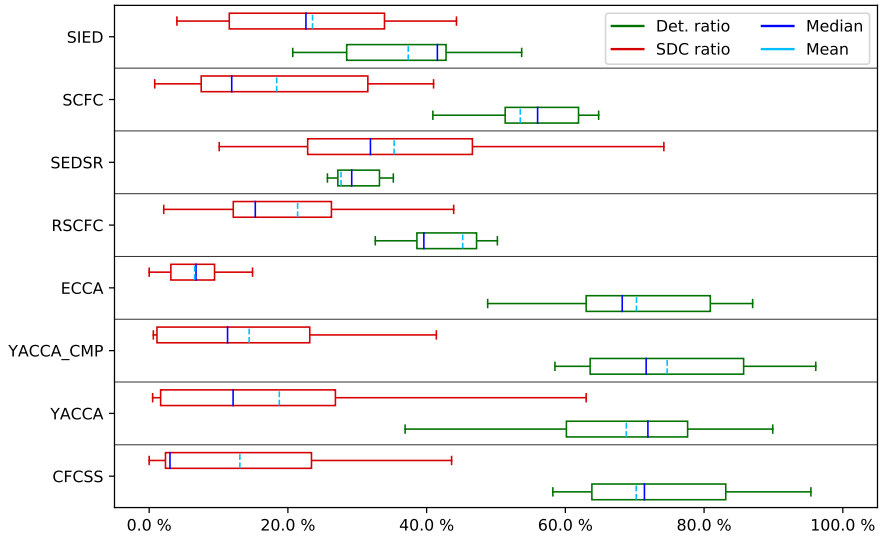


Figure 4.6: The error detection ratio and SDC ratio of the eight SOTA techniques

When only considering error detection cost, YACCA, YACCA_CMP and ECCA impose the highest cost, both in code size and execution time. This is not surprising as they need the most instructions and/or computationally expensive instructions to execute. The remaining techniques only insert computationally inexpensive and a relatively low number of extra instructions to be implemented. Furthermore, SEDSR and CFCSS impose the lowest error detection cost. Therefore, the study has shown that CFCSS makes the best trade-off between error detection ratio and error detection cost and is thus the best technique to implement for most case studies.

4.3 Room for Improvement

Although the study reveals CFCSS is the best technique to implement for most case studies, there is room for improvement. Using the gathered data, with a special interest to the types of CFEs that remained undetected, five guidelines can be derived as the basis to develop an optimal technique.

Guideline 1: Gradual Update

As described in Section 4.2.1, an optimal technique should use gradual updates. Local update techniques are too easily misled: erroneous jumps that land before update instructions remain undetected, regardless of how many instructions have been skipped. However, with a gradual technique, once a single update has been skipped, the erroneous jump will be detected.

Guideline 2: Avoid Shared State

The performed study demonstrated that a multitude of the considered techniques use a shared state for the signature throughout the basic blocks. This shared state is a value for the run-time signature that is correct for multiple basic blocks in the CFG. A prevailing choice for this value is the number zero, used in the following process:

1. Perform an operation on the run-time signature that updates it to zero in case of an error-free run.

2.a) Validate whether the run-time signature is zero; or

2.b) Perform a second operation on the run-time signature that depends on that value zero.

However, while the run-time signature is zero, or in other words has reached the shared state, erroneous jumps can occur and remain undetected. An indicative example is given in Fig. 4.7, where the run-time signature is represented as `r11`. Although the given example implements a fictitious technique, the problem is present in ECCA, RSCFC, SEDSR and SCFC. Fig. 4.7 shows that when the run-time signature reaches zero, an erroneous jump, skipping many instructions, can go undetected. Once the instruction `EOR r11, #2` has executed in the left basic block, the run-time signature is zero. As shown, a jump originating after that instruction and landing into another block between the `EOR` and `OR` instructions will not be detected. The run-time signature is expected to be zero at that point, so the update procedure (`OR r11, #32`) will assign the correct value to the run-time signature, thereby producing a false negative.

Guideline 3: Take Conditional Branches into account

A CFE can follow an intentional path through the program and still be an error. For example, a CFE can jump to the false branch of a basic block, while

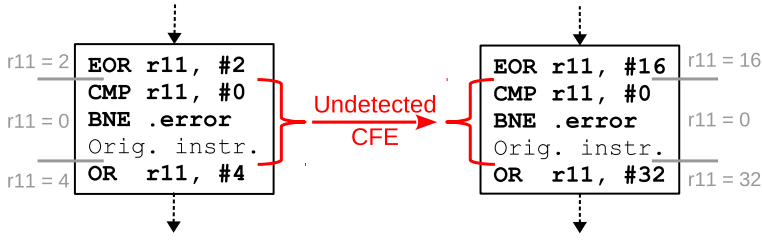


Figure 4.7: Example of an undetected CFE when the run-time signature (`r11`) reaches zero.

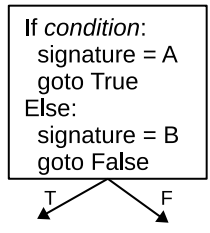


Figure 4.8: Pseudo-code describing how to comply with guideline 3.

in fact the true branch should have been taken. To increase the chance of detecting this type of error, the optimal signature monitoring technique should update the run-time signature conditionally, using the same condition as the conditional branch. Fig. 4.8 illustrates a possible implementation to comply with this guideline in pseudo-code.

Guideline 4: Take Exit Instructions into account

As aforementioned, the program executing on the target consists of setup code, the target algorithm and some validation code. This division of the code resulted in the algorithm having exit statements. Analyzing the experiment data, we noticed that some of the CFEs remained undetected because they caused an early exit out of the protected algorithm, by jumping into a basic block containing an exit statement. Given that such basic blocks have no protected successor, the erroneous exit was not detected by the implemented signature monitoring technique. To partially counter these errors, an extra verification should be inserted in front of all exit statements. If the CFE jumps right to the exit instruction itself, the technique will still fail to detect the erroneous exit. If it lands before that instruction, the error will be detected.

Guideline 5: Use as Few and as Inexpensive Instructions as possible

This guideline deals with both the error detection ratio and the imposed overhead. This guideline is related to error detection ratio because the more instructions that are used, the more likely a shared state is created. Considering imposed overhead, using as few instructions as possible ensures a low code size overhead. Using instructions that only require a few clock cycles to execute ensures a low execution time overhead. Moreover, this guideline also affects how signatures are chosen. RSCFC, SEDSR and SCFC define the width of their signature based on the amount of basic blocks in the target algorithm. Microcontrollers, however, are fixed to a specific register width, e.g. 32 bits, and the instruction set is based on that width. For the three mentioned techniques, this means that once the target algorithm has more than 32 basic blocks in a 32 bit architecture, either more instructions or more complex instructions are needed to perform the necessary calculations, increasing the imposed overhead.

4.4 Conclusion

Based on the performed comparative study of the SOTA techniques, we can draw the following conclusions. When only considering error detection ratio, CFCSS, YACCA, YACCA_CMP and ECCA are the best techniques with an average error detection ratio of 70 %, 69 %, 75 % and 70 % respectively. The other considered techniques only have an average error detection ratio of 54 % or less. This difference in error detection ratio is caused by the usage of gradual updates by CFCSS, YACCA, YACCA_CMP and ECCA.

When only considering error detection cost, YACCA, YACCA_CMP and ECCA impose the highest cost, both in code size and execution time. This is not surprising as they need the most instructions and/or computationally expensive instructions to execute. The remaining techniques only insert computationally inexpensive and a relatively low number of extra instructions to be implemented. Furthermore, SEDSR and CFCSS impose the lowest error detection cost. Therefore, the study has shown that CFCSS makes the best trade-off between error detection ratio and error detection cost and is thus the best technique to implement for most case studies.

However, the results showed there was room for improvement. Utilization of the gathered data, we proposed five guidelines to build an optimal inter-block CFE detection technique.

Chapter 5

Advancing the SOTA

The content and results of this chapter have been published in [79, 83, 84]. Since their publication, the results presented in this chapter have been updated by using updated versions of the tools and our latest insights.

In this chapter, we present our three CFE detection techniques that outperform the SOTA techniques. First, we discuss our RASM technique, which follows the five guidelines presented in the previous chapter. Next, we present our RACFED technique, which extends RASM with instruction monitoring to detect intra-block CFEs. This chapter concludes by discussing S-RACFED, a selective implementation of RACFED which aims to decrease overhead.

5.1 Random Additive Signature Monitoring

Using the five guidelines to build an optimal inter-block CFE detection technique, we developed a new technique called Random Additive Signature Monitoring (RASM). The technique uses additions and subtractions of random values to validate whether only valid paths through the CFG were taken.

5.1.1 High-Level Description

Before diving into the implementation details of RASM, the general idea behind the technique is discussed.

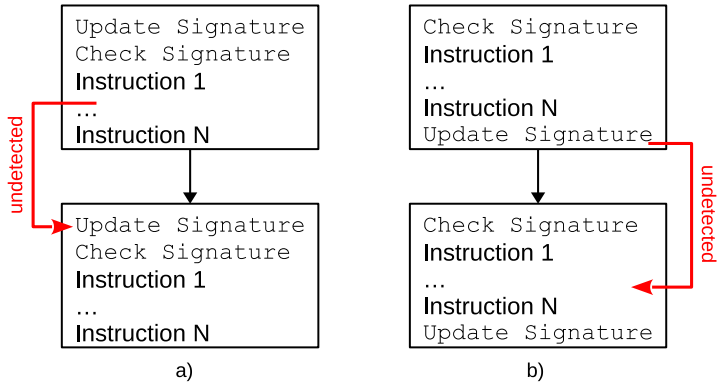


Figure 5.1: Examples of CFEs that remain undetected when using only one update and one verification instruction.

First of all, RASM has to comply with guideline 1 and thus has to use gradual updates to change the run-time value of the signature. In order to comply with guideline 5, the first idea was to only use one gradual update and one verification instruction. However when using only one gradual update instruction, there will always be an inter-block CFE that can occur that does not skip the gradual update. Since no gradual update is skipped, the error cannot be detected. Two examples are given in Fig. 5.1. For each example, an inter-block CFE is drawn that the inserted instructions cannot detect. Fig. 5.1 a) shows the implementation with both the update and verification instruction at the beginning of the basic block. As indicated, this combination fails to detect inter-block CFEs that originate after the verification instruction and land in front of the next update instruction. The combination shown in Fig. 5.1 b) separates the verification and update instructions in the basic block. The verification instruction is added to the beginning of the basic block and the update is inserted at the end. This configuration cannot detect an inter-block CFE that originates after the update instruction and lands in front of the next update instruction.

To solve this problem, RASM uses two gradual updates per basic block, as shown in the partial CFG of Fig. 5.2. In this CFG there is a valid path to go from point A to point B. RASM verifies whether this valid path has been taken, by

1. assigning signatures to each basic block and inserting checks that compare the run-time value to the compile-time value of the signature. This is indicated as `CheckSig1` and `CheckSig2` in Fig. 5.2; and

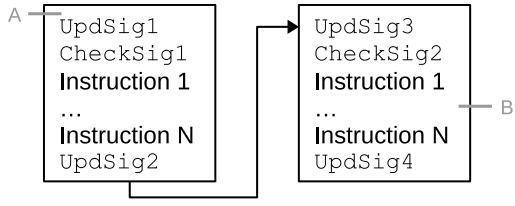


Figure 5.2: General principle of RASM.

2. inserting 2 signature updates per basic block,
    - one as first instruction to be executed, indicated as `UpdSig1` and `UpdSig3`; and
    - one as last instruction to execute, indicated as `UpdSig2` and `UpdSig4`.

Now, the valid path from A to B is `UpdSig1`, `CheckSig1`, `UpdSig2`, `UpdSig3` and `CheckSig2`. Because `UpdSig2` and `UpdSig3` are executed consecutively, their operation is determined with respect to one another. In other words, they have to act as one signature update, divided into two instructions. If both update operations would be determined without considering their relationship, the update result cannot match the expected result in `CheckSig2`. This mismatch will be reported as an error, even though no CFE has occurred. To summarize, `UpdSig2` and `UpdSig3` are determined with respect to each other in order to avoid false positives at `CheckSig2`.

5.1.2 Implementation Algorithm

The compile-time process to implement RASM, presented in Alg. 4, consists of three steps. The first step is a global step that assigns the needed variables to all basic blocks. The second step and third step are executed for each basic block and insert the two run-time signature updates and the signature verification. These run-time signature updates are mathematical operation, i.e. addition and subtraction. We selected mathematical operations as run-time signature updates, because the data from our comparative study (cfr. Chapter 4) showed that bitwise operations, such as EXCLUSIVE OR, can be too easily fooled.

Step 1: Assign the needed variables to all basic blocks.

The process starts by assigning two random values to each basic block. The first random value is the compile-time signature and is unique for each basic

Algorithm 4 Pseudo-code describing the compile-time process to implement RASM.

```
1:  for all Basic Block (BB) in CFG do
2:      repeat compileTimeSig ← random number
3:      until compileTimeSig is unique
4:      repeat subRanPrevVal ← random number
5:      until (compileTimeSig + subRanPrevVal) is unique
6:  for all BB in CFG insert at beginning
7:      signature ← signature − subRanPrevVal
8:      if signature ≠ compileTimeSig then ERROR()
9:  for all BB in CFG do
10:     if Last Instr. is return instr. and NrIntr_BB > 1 then
11:         Calculate needed variables
12:             returnVal ← random number
13:             adjustValue ← compileTimeSig_BB − returnVal
14:         Insert signature update before return instr.
15:             signature ← signature + adjustValue
16:             if signature ≠ compileTimeSig then ERROR()
17:     else
18:         for all Successor of BB do
19:             expectedValue ← compileTimeSig_succs + subRanPrevVal_succs
20:             adjustValue ← compileTimeSig_BB − expectedValue
21:             Insert signature update at BB end
22:                 signature ← signature + adjustValue
```

block. The second value is called `subRanPrevVal` and is used in the second step of this process to update the run-time signature. To uniquely identify each basic block, the sum of the compile-time signature and the `subRanPrevVal` has to be unique. A new `subRanPrevVal` is assigned until this is the case.

Step 2: Insert the first signature update and the signature verification.

Once the two random values are assigned to each basic block, the protective instructions are inserted in all basic blocks. The first instruction a protected basic block has to execute is an update of the run-time signature. This update is a subtraction between the signature and the `subRanPrevVal` of the current basic block. Once executed, the run-time signature should hold the compile-time signature of the current basic block. Next, a verification instruction is inserted to verify this result and report a CFE has occurred if the run-time signature holds any other value.

Step 3: Insert the last signature update.

The final step of the RASM implementation process is inserting the second update of the run-time signature at the end of each basic block. This last instruction assures all intentional paths in the CFG can still be taken in error-free runs, without causing a false positive CFE detection. In order to keep intentional paths valid, the signature is updated with an adjustment value. This value is calculated as the difference between the compile-time signature of the current basic block and the sum of the compile-time signature and `subRanPrevVal` of the successor block. This calculation is shown on lines 19 and 20 of Alg. 4. If the basic block ends with a conditional branch, this last update is executed conditionally, so the run-time signature has the correct value for the corresponding successor.

This last step changes if the basic block is an exit block. Depending on the number of instructions in such a basic block, either an extra verification is inserted or no instructions are inserted. If the basic block has more than one instruction, an extra verification is added in front of the return instruction. This extra verification checks whether or not the run-time signature matches the `returnVal`, which is a randomly chosen number. To avoid false positives, the last signature update is inserted in front of the exit verification. The adjustment value is then the difference between the compile-time signature of the current basic block and `returnVal`. This calculation and addition of the instructions is shown by lines 11-16 of Alg. 4. This extra verification enables to detect some CFEs that would cause a premature exit out of the program.

If the basic block only has a return instruction and no other instructions, this step inserts no extra instructions. This type of basic block only has the extra instructions inserted by step 2 of this process. Otherwise, such a basic block would have two signature verifications following each other. Not inserting this second run-time verification reduces the overhead imposed by RASM.

5.1.3 RASM Applied

Using the same example as in Ch. 2, Fig. 5.3 shows the result of the implementation process. The run-time signature is stored in register `r11` and the compile-time variables are shown for all basic blocks. The first instruction each basic block executes is the first update of the run-time signature (`SUB`). Next, each basic block verifies the result of the update, by comparing the run-time and compile-time value (`CMP`). When there is a mismatch between both values, control is transferred to the error handler located at address `240` (`BNE`). The last RASM-related instruction each basic block executes is the second update of

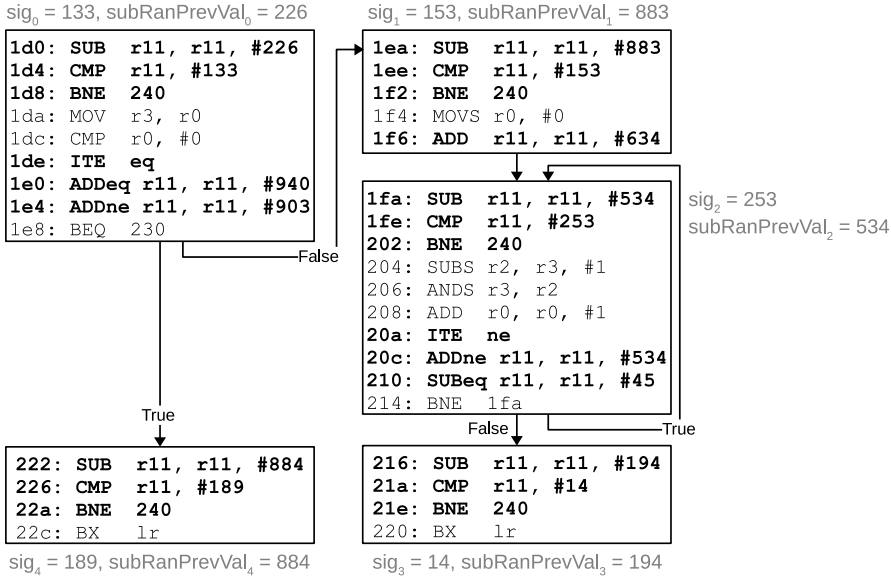


Figure 5.3: Our RASM technique applied to an example program in ARMv7M.

the run-time signature (`ADD` or `SUB`). The top-left and middle-right basic block show that this update is executed conditionally when the basic block ends with a conditional branch. Because the two bottom basic blocks are exit blocks with only the return instruction, no extra verification is inserted. This is the special case discussed in Step 3 of the previous section.

5.1.4 Theoretical Inter-Block CFE Detection

RASM uses gradual updates along intentional paths. This means that all inter-block CFEs that skip at least one update are detected. Skipping an update results in an unintended run-time signature value and results in the detection of the CFE.

The exceptions are CFEs that jump to a return instruction. Although these jumps skip at least one run-time update, they might not have a successor basic block with verification instructions. Such jumps can only be detected at the place where the exit statements jumps to. In ARMv7-M, a return instruction typically has one out of two forms: `BX lr` or `pop {..., pc}`. The `BX lr` means *branch to the address contained in register* `lr` and the `pop {..., pc}` means *branch to the address stored on the stack*. To detect erroneous jumps to this

instruction, verification instructions must be inserted at all locations where the register `lr` can point to, or at whatever address is stored on the stack.

5.1.5 Experiments

Next to a theoretical analysis, we validated RASM using fault injection experiments. We performed the same experiments, using the same case studies, the same hardware and the same fault injection process as for the comparative study of the SOTA techniques. To assess the error detection ratio and error detection cost of RASM, we compare the measured results to those of CFCSS and YACCA_CMP. We selected those techniques as references because the comparative study showed that CFCSS makes the best trade-off between error detection ratio and error detection cost and because YACCA_CMP achieves the highest error detection ratio.

Error Detection Ratio

Fig. 5.4 shows the results of the performed fault injection campaign. RASM achieves the highest error detection ratio for six of the eight case studies. Only for the BC and FFT case studies, YACCA_CMP achieves a higher error detection ratio. While the error detection ratio of RASM for the FFT case study is also high, i.e. 92 %, the technique behaves poorly when implemented for the BC case study. As shown, RASM is unable to detect at least 60 % of the injected errors and has an SDC ratio of 38 %. An analysis of the data showed that this low error detection ratio is due to CFEs being injected that jump to a return instruction. As discussed in Section 5.1.4, CFEs that land on a return instruction cannot be detected by RASM. Unfortunately, BC with RASM implemented leads to a higher possibility of such CFEs to be injected and thus remain undetected, causing a low error detection ratio and high SDC ratio.

When analyzing the average numbers, RASM does outperform both CFCSS and YACCA_CMP. Regarding error detection ratio, RASM achieves an average of 87 %, while CFCSS and YACCA_CMP achieve an average of 70 % and 75 % respectively. The same holds for the SDC ratio, with RASM achieving an average of only 6 %, while CFCSS has an average ratio of 13 % and YACCA_CMP has an average ratio of 14 %. These numbers show that RASM is a better technique than CFCSS and YACCA_CMP regarding the error detection ratio.

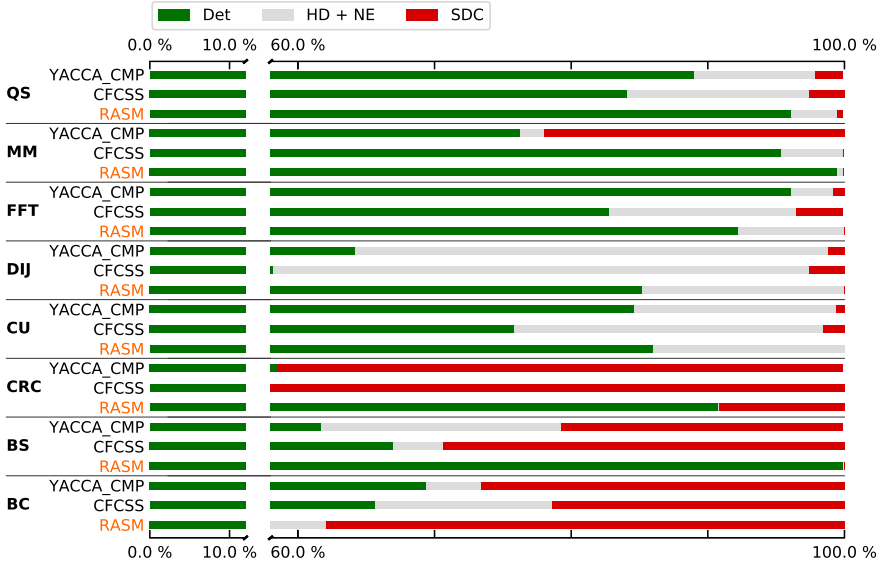


Figure 5.4: Error detection ratio of RASM compared to CFCSS and YACCA_CMP

Error Detection Cost

Next to the error detection ratio, the error detection cost, both execution time overhead and code size overhead, of RASM were measured. Again, we compared the results to those of CFCSS and YACCA_CMP. This comparison is presented in Fig. 5.5, with the execution time overhead shown in dark-blue and the code size overhead shown in light-blue.

When considering the execution time overhead, the chart shows that RASM imposes the lowest overhead for six of the eight case studies. Only the CU and FFT case studies produce different results, but as explained in Section 4.2.2 this is due to the generation of different code for these case studies when less registers are available to the compiler. When neglecting those two case studies, the average execution time overhead of RASM, relative to the unprotected code is $\times 1.77$. In contrast, the average execution time overhead of CFCSS and YACCA is $\times 1.89$ and $\times 3.39$ respectively.

Regarding the code size cost, only a limited overhead is measured as only a small part of the code base is protected. On average, RASM and CFCSS impose the same code size overhead, resulting in $\times 1.03$ the code size relative to the unprotected code. YACCA_CMP imposes a slightly higher code size overhead

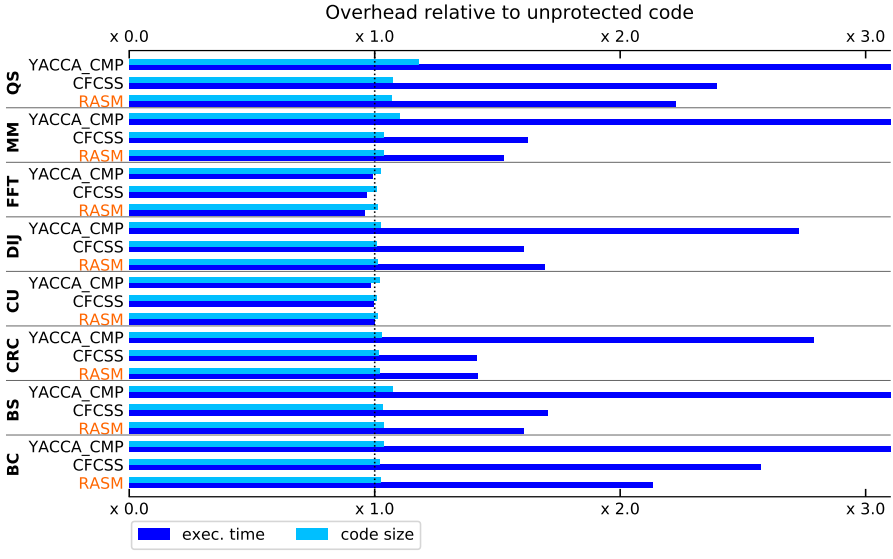


Figure 5.5: Error detection cost of RASM compared to CFCSS and YACCA_CMP

with an average of $\times 1.07$.

5.2 Random Additive Control Flow Error Detection

While RASM outperforms the SOTA techniques, it is only capable of detecting inter-block CFEs. To detect intra-block CFEs, RASM has to be extended with instruction monitoring. This extended version is called Random Additive Control Flow Error Detection (RACFED) and it uses additions and subtractions with random values to detect intra-block CFEs. Before discussing the changes to the implementation algorithm, we present the principle of RACFED.

5.2.1 High-Level Description

RACFED detects intra-block CFEs by inserting signature updates after each instruction, as shown in Fig. 5.6. The bold black instructions show the inter-block detection method, as proposed by our RASM technique, and the bold orange instructions show the added signature updates to create RACFED. As

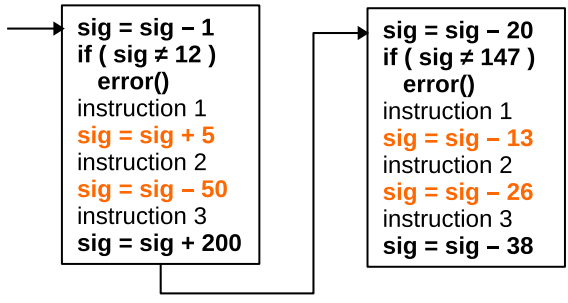


Figure 5.6: Principle of our RACFED technique.

can be seen, all updates modify the run-time signature *sig*. This implementation of instruction monitoring assures a high CFE detection ratio. By updating the run-time signature as part of the intra-block CFE detection method, the signature verification instruction already in place can be used. This principle is also used by the RSCFC technique and helps to lower the imposed overhead compared to using a new variable and extra verification instruction.

5.2.2 Implementation Algorithm

The compile-time process to implement RACFED is shown in Alg. 5. It is based on the RASM process and consists of four steps. The first step is a global step that assigns the needed variables to all basic blocks. The second step assures the implementation of the instruction monitoring. The third and fourth step implement the signature monitoring step of RACFED and are executed for each basic block. In essence, the first, third and fourth step are the same as for RASM and the second step extends RASM to RACFED.

Step 1: Assign the needed variables to all basic blocks.

As for RASM, the process starts by assigning two random values to each basic block. The first random value is the compile-time signature and is unique for each basic block. The second value is `subRanPrevVal` and is used in step 3 to update the signature. To uniquely identify each basic block, the sum of the compile-time signature and `subRanPrevVal` has to be unique for each basic block. A new `subRanPrevVal` is assigned until this is the case.

Algorithm 5 Pseudo-code describing the compile-time process to implement RACFED.

for all Basic Block (BB) in CFG **do**
    repeat compileTimeSig ← random number
    until $compileTimeSig$ is unique
    repeat subRanPrevVal ← random number
    until ($compileTimeSig$ + $subRanPrevVal$) is unique
for all BB in CFG **do**
    if $NrInstr_{BB} > 2$ **then**
        for all original instructions **insert after**
            signature ← $signature$ + $random\,number$
for all BB in CFG **insert at beginning**
    $signature \leftarrow signature - subRanPrevVal$
    if $signature \neq compileTimeSig$ then ERROR()
for all BB in CFG **do**
    if Last Instr. is return instr. **and** $NrIntr_{BB} > 1$ **then**
        Calculate needed variables
            returnVal ← random number
            currentValue ← $compileTimeSig_{BB} + \sum instMonUpdates_{BB}$
            adjustValue ← $expectedValue - returnVal$
        Insert signature update before return instr.
            $signature \leftarrow signature + adjustValue$
            if $signature \neq returnVal$ then ERROR()
    else
        currentValue ← $compileTimeSig_{BB} + \sum instMonUpdates_{BB}$
        for all Successor of BB **do**
            expectedValue ← $compileTimeSig_{succs} + subRanPrevVal_{succs}$
            adjustValue ← $currentValue - expectedValue$
            Insert signature update at BB end
                $signature \leftarrow signature + adjustValue$

Step 2: Implement the instruction monitoring.

Next, the instruction monitoring part of RACFED is implemented. The instruction monitoring consists of run-time signature updates with random values. After each original instruction an extra run-time signature update is inserted.

Not all basic blocks have this countermeasure implemented. Only basic blocks with more than two original instructions are processed in this step. In a basic block with only one instruction, an intra-block CFE cannot occur. All CFEs that occur in this type of basic block are inter-block jumps, which are detected via the signature monitoring instructions of RACFED. In a basic block with two instructions, an intra-block CFE can occur by prematurely stopping the execution of the first instruction and starting the execution of the second instruction. Such an intra-block CFE, however, cannot be detected via instruction monitoring because no update would be skipped. Therefore, the intra-block CFE detection countermeasure is only implemented in basic blocks with three or more instructions.

Step 3: Insert the first signature update and the signature verification.

The third and fourth step to implement RACFED are the same as for the RASM technique. The third step inserts the first update of the run-time signature and the only verification per basic block. The update is a subtraction between the signature and the `subRanPrevVal` of the current basic block. The update should result in the signature having the value of the compile-time signature. If not, a CFE has occurred and is detected.

Step 4: Insert the last signature update.

The last signature update that is inserted into each basic block, is the update that assures all intentional paths in the CFG can still be taken in error-free runs, without causing a false positive CFE detection. In order to keep intentional paths valid, the signature is updated with an adjustment value. This value is calculated as the difference between the signature updates of this block and the first update of the next basic block, as shown by lines 23-28 of Alg. 5. First, the current value of the signature is calculated by computing the total impact of the inserted intra-block updates. Next, the expected value per successor is calculated by summing up the `subRanPrevVal` and the compile-time signature of each successor. The adjustment value is the difference between the current run-time value and the expected value. The inserted update adds the adjustment

value to the run-time signature. If the basic block ends with a conditional branch, this last update is executed conditionally, so the run-time signature has the correct value for the corresponding successor.

This last step changes if the basic block ends with a return instruction. A return instruction is an instruction that exits the current function and returns to its caller. Depending on the number of instructions in such a basic block, either an extra verification is inserted or no instructions are inserted. If the basic block has more than one instruction, an extra verification is added in front of the return instructions. This instruction verifies whether or not the run-time signature matches a number chosen at random, called `returnVal`. To avoid false positives, the last signature update is inserted in front of the exit verification. The adjustment value is the difference between the signature updates of the current basic block and the `returnVal`, as described by lines 15-21 of Alg. 5. This extra verification enables to detect CFEs that would cause a premature exit out of the program.

If the basic block only has a return instruction and no other instructions, this step inserts no extra instructions. This kind of basic block only has the extra instructions inserted by step 3 of this process. Otherwise, such a basic block would have two signature verifications following one another. Not inserting this second run-time verification reduces the execution time overhead of RACFED.

5.2.3 RACFED Applied

Using the same example as in Ch. 2, Fig. 5.7 shows the result of the implementation process. The run-time signature is stored in register `r11` and the compile-time variables for each basic block are shown.

As with RASM, the first three instructions each basic block executes are a signature update and a verification of the run-time value against the compile-time value. If the two values do not match, control is transferred to the error handler located at address `0x24c`.

Next, this example shows that only a basic block with more than two instructions updates the run-time signature after each original instruction executed. As mentioned in the previous subsection, the intra-block updates are either an addition or a subtraction with a random value. For the given example, the intra-block updates are located at `0x1dc, 0x20a` and `0x210`. The final run-time signature update is inserted as in RASM, using an adjustment value and conditional execution when needed.

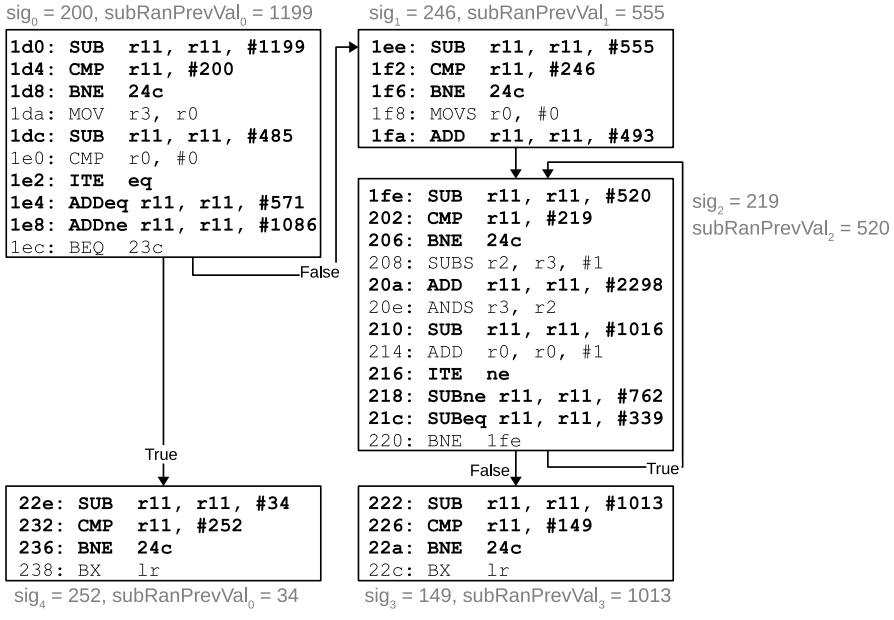


Figure 5.7: Our RACFED technique applied to an example program in ARMv7-M

5.2.4 Supporting ISAs without conditional execution

Up until know, we only considered the ARMv7-M ISA, which supports conditional execution. This conditional execution of instructions is used for the final signature update in each basic block, both in RASM and RACFED. However, some ISAs do not support conditional execution. Therefore, an adjustment has to be made to the compile-time process to be able to implement RASM and RACFED for such ISAs. To be able to implement RACFED for an ISA that does not support conditional execution, we opted to insert the signature update needed for the *True* path before the conditional branch and to insert an adjusted signature update for the *False* path after the conditional branch.

Applied to an example, the impact of this change to the implementation process can be seen in Fig. 5.8. On the left side, RACFED is implemented for an ISA that supports conditional and on the right side RACFED is implemented for an ISA that does not support it. As indicated in orange, one final update now occurs before the conditional branch and another after the conditional branch. In fact, the value to update the signature with before the conditional branch has

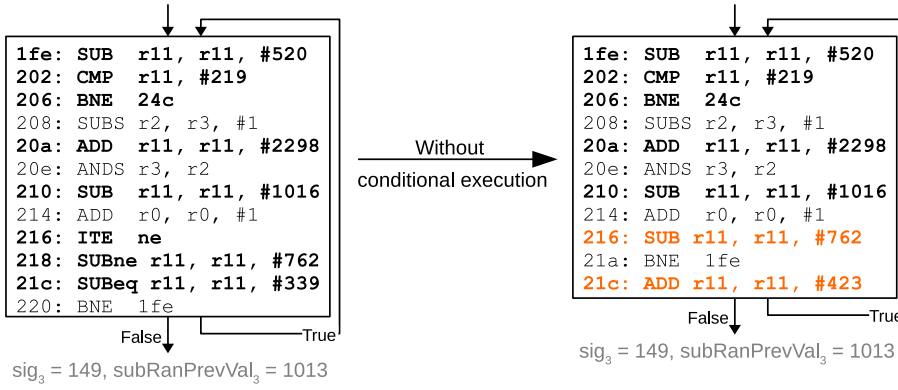


Figure 5.8: The impact on the implementation of RACFED without conditional execution support. The affected instructions are indicated in orange.

not changed. On the left side, the instruction at address `0x218` is the update to be executed for the *True* case and the corresponding update on the right side, located at address `0x216` matches perfectly. The value to update the signature for the *False* case has changed, as the update for the *True* case to be taken into account, and is calculated as follows. First, the current value of the signature is calculated: $219 + 2298 - 1016 - 762 = 739$. Next, the expected value of the *False* successor is determined: $149 + 1013 = 1162$. Then the difference between the two values is calculated: $1162 - 739 = 423$. Finally, the correct signature update instruction is inserted after the conditional branch, as shown at address `0x21c`.

This implementation of RACFED does have an impact on the overhead. As can be seen, the impact on code size overhead is rather small as the implementation when conditional execution is not supported does not need the `ITE` instruction. The impact on execution time can be big, as both update instructions indicated in orange have to be executed for the *False* case. The more *False* paths have to be taken throughout the CFG, the higher the execution time overhead will be.

5.2.5 Theoretical CFE Detection

RACFED uses gradual updates along intentional paths. This means that all CFEs that skip at least one update are detected. Skipping an update results in an unintended run-time signature value and results in the detection of the CFE.

A first exception is the vulnerability inherited from RASM: CFEs that jump to

a return instruction. Although these jumps skip at least one run-time update, they do not have a successor basic block with verification instructions. Such jumps can only be detected at the place where the exit statements jumps to. RACFED does not insert extra countermeasures against such CFEs and is thus vulnerable against them.

Another category of undetected CFEs are intra-block jumps that skip an original instruction but no update instruction. Considering the example of Fig. 5.7, an erroneous jump from address `0x20a` to `0x210` is not detected by RACFED, because no signature update is skipped. A possible solution to detect this kind of jump, is to insert a duplicate instruction of the original instruction. When this duplicate is inserted after the intra-block update, this assures that at least one version of the original instruction is executed [33].

The final category of CFEs that cannot be detected are shared-signature jumps. Because random values are used as intra-block updates, it is possible that one run-time signature value is valid at different points in the protected program. If a CFE jumps between two such points, it will not be detected. Since the landing point gets the expected value, the following updates will calculate the expected run-time signature value, thus masking the CFE.

5.2.6 Experiments

To validate RACFED, we performed a fault injection campaign using the same case studies as described in Section 3.1. Because both intra-block and inter-block CFEs have to be injected, we used the extended CFE injection process described in Section 3.3.2 on a simulated ARM Cortex-M3. Next, we measured the error detection cost in an error-free run. This was measured on a real hardware target, i.e. the NXP LPC1768, because the used simulator is an instruction-accurate simulator and not a cycle-accurate simulator. This makes the simulator not suited to measure the error detection cost. To place the measured results in perspective, we performed the same experiments with RSCFC and SIED and compare our results to their results.

Error Detection Ratio

The measured error detection ratio of all three techniques is presented in Fig. 5.9. To be complete, we also perfomed the experiments on the unprotected case studies. Again, the green bar indicates the percentage of CFEs that was detected by the implemented technique (Det), the red bar indicates the percentage of CFEs that was not detected and caused a wrong result (SDC), and gray bar

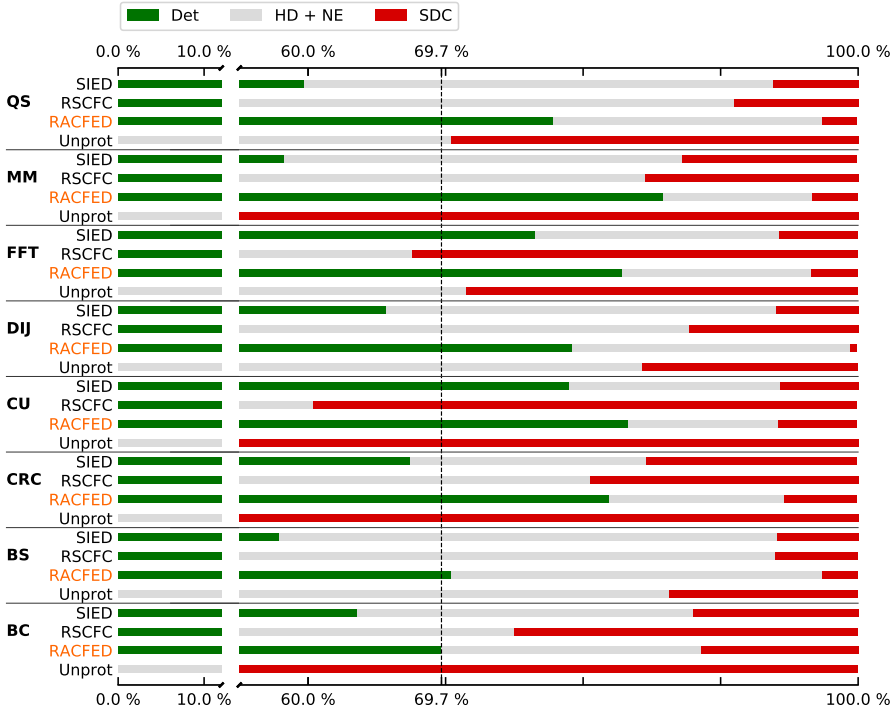


Figure 5.9: The error detection ratio of RACFED compared to RSCFC and SIED.

shows the percentage of CFEs that were either detected by the microcontroller itself or that were not detected and did not corrupt the result of the case study (HD + NE).

A first result that can be seen on the chart is that RACFED has the highest error detection ratio for all case studies, when compared to RSCFC and SIED. The lowest recorded error detection ratio for RACFED, i.e. 69.7 %, is even higher than all recorded ratios for RSCFC and six out of eight recorded ratios of SIED. A second result that can be deduced from the chart is that RACFED even has the lowest SDC ratio for seven out of eight case studies and is only marginally worse than SIED for the CU case study. Combined, these results show that RACFED better protects embedded systems against CFEs.

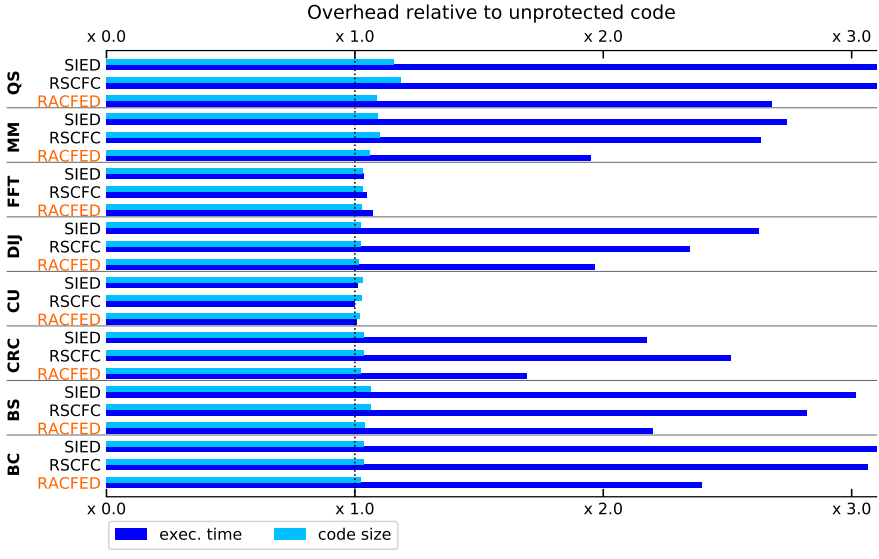


Figure 5.10: The error detection cost of RACFED compared to RSCFC and SIED.

Error Detection Cost

The error detection cost of the techniques is shown in Fig. 5.10. The dark-blue bars show the execution time of each technique compared to the unprotected code, while the light-blue bars present the code size of each technique.

As can be seen, RACFED imposes the lowest execution time overhead for six out of eight case studies, with an average execution time of 1.87 times that of the unprotected code. For the CU and FFT case studies, the compiler generates different code due to having less register available. This gives RSCFC the lowest overhead for the CU case study, and SIED for the FFT case study. For the other six case studies, the lower overhead of RACFED is due to two factors. A first factor is the usage of the signature for the intra-block updates, instead of an additional control variable. This eliminates the need to insert an instruction to set up a second variable and eliminates the need to insert extra verification instructions. Using the signature as control variable for the intra-block detection enables using the signature verification instruction to verify both the inter-block and intra-block updates. The second factor is that the intra-block updates are only inserted in basic blocks with more than two instructions.

Because RSCFC inserts intra-block updates in all basic blocks, it imposes a

higher cost than RACFED. On average, the execution time of a case study protected by RSCFC is 2.45 times that of the unprotected code. SIED imposes a higher cost than RACFED because it uses a second control variable to detect intra-block CFEs and thus inserts more instructions. The average execution time of a case study protected by SIED is 2.49 times higher than the unprotected code, which is similar to the cost of implementing RSCFC.

The same holds for the imposed code size overhead. RACFED imposes the lowest code size cost for the three considered techniques. The differences between unprotected code and between the three considered techniques is however smaller. This is because only a small part the code base for each case study is protected. As discussed aforementioned, the code base of each case study consists of startup code, the actual algorithm and some result verification code, and only the actual algorithm is protected by the CFE detection technique. The algorithm is however a small part of the code base and thus results in less bespoke differences between the three techniques.

5.3 Selective Implementation

While our proposed RACFED outperforms similar established techniques, both regarding error detection ratio and error detection cost, the imposed error detection cost is still relatively high. In an attempt to lower its overhead, we have experimented with a selective implementation for the technique. Before discussing the taken approach and presenting its effect on the imposed overhead, we provide an overview of different strategies to selectively implement control flow checking proposed in literature.

5.3.1 Different Approaches for Selective Implementation

Until now, all considered CFE detection techniques are full implementation techniques, i.e., they insert the signature update and verification instructions in each basic block in the CFG. In recent years, selective implementations have been proposed, where the extra instructions are only inserted in certain selected basic blocks. One approach to selectively apply CFE detection, is to simply ignore some basic blocks. This approach is taken by the S-SETA technique [52]. Within S-SETA, smaller basic blocks have no extra instructions inserted in order to reduce the imposed execution time overhead. While this approach drastically reduces the overhead, it also reduces the detection ratio, as CFEs that occur within the ignored basic blocks are possibly undetectable.

A second approach to selectively apply CFE detection, is to insert signature update instructions in each basic block, but to only insert signature verification instructions in selected basic blocks. Chielle et al. propose to only insert the verification instructions in larger basic blocks with their SETA-C technique [52]. Khudia et al. present another implementation of this approach with their Abstract Control Signatures (ACS) technique [53]. They divide the basic blocks into regions and assign a specific signature to each region. Each basic block in a region has a signature update instruction inserted, while signature verification instructions are only inserted in basic blocks where the control flow changes from one region to another. Khudia et al. define their regions as a collection of basic blocks which has a single entry point but multiple exit points. Ideally, the regions should possess certain properties that help in minimizing the number of signature updates and verifications.

Although SETA-C and ACS show promise, implementing them (automatically) can be difficult. SETA-C only inserts the verification instructions in larger basic blocks. For algorithms with little difference between the length in basic blocks, the decision for the threshold of *large* might be difficult to make. To implement ACS and gain the maximum decrease in the imposed overhead, the division of the CFG into regions is critical. However, Khudia et al. provide little guidance on how to find this optimal division, making this technique very difficult to implement.

Therefore, we propose an easier and more consistent implementation of this approach to selectively implement CFE detection techniques. We propose to only insert the signature verification instructions in return basic blocks. These are basic blocks which contain the return statement that exits out of the current function or program and returns control to the calling function or program. These basic blocks are the last basic block in each possible path through the target CFG and thus enables to detect all CFEs that occur within the CFG.

5.3.2 Selective RACFED

This section discusses how we applied the proposed selective implementation principle to our RACFED technique, which is called S-RACFED. First, we present the small change to be made to the compile-time process of RACFED to implement it selectively. Then, we show an example with S-RACFED applied.

Change in Compile-Time Process

There are four steps in the compile-time process to implement RACFED:

1. First, all needed compile-time variables are assigned to all basic blocks.
2. Then, the instruction monitoring is implemented. These update instructions help to detect intra-block CFEs.
3. Next, the first signature update and the signature verification instructions are inserted in each basic block.
4. Finally, the last signature update is inserted in each basic block.

To implement S-RACFED, the third step is slightly adjusted. The first signature update instruction is still inserted in each basic block, the signature verification, however, is only inserted in return basic blocks. Therefore, an extra check is inserted in the compile-time process that analyses whether or not the last instruction of the basic block is a return instruction. If so, only then the signature verification is inserted in the basic block.

S-RACFED Applied

Fig. 5.11 shows S-RACFED applied to an example program. To enhance readability, we used the same compile-time values for each basic block as in the example used to show RACFED. As illustrated, only the two bottom basic block contain a return statement. Therefore, only those basic blocks have the comparison instruction and the branch to the error handler inserted. For the other basic block, this means that two instructions less are inserted compared to the regular implementation of RACFED.

5.3.3 Experiments

We performed the same experiments as described in Section 5.2.6 for S-RACFED and compare the results to those of RACFED. As an extra criteria, we also measured the error detection latency of both techniques. Although often neglected, a downside of selective implementation is a higher error detection latency as the effective detection of the error, i.e. the comparison of run-time and compile-time values, is delayed. For these experiments, the error detection latency was measured as number of instructions that were executed between the error injection and the execution of the error handler.

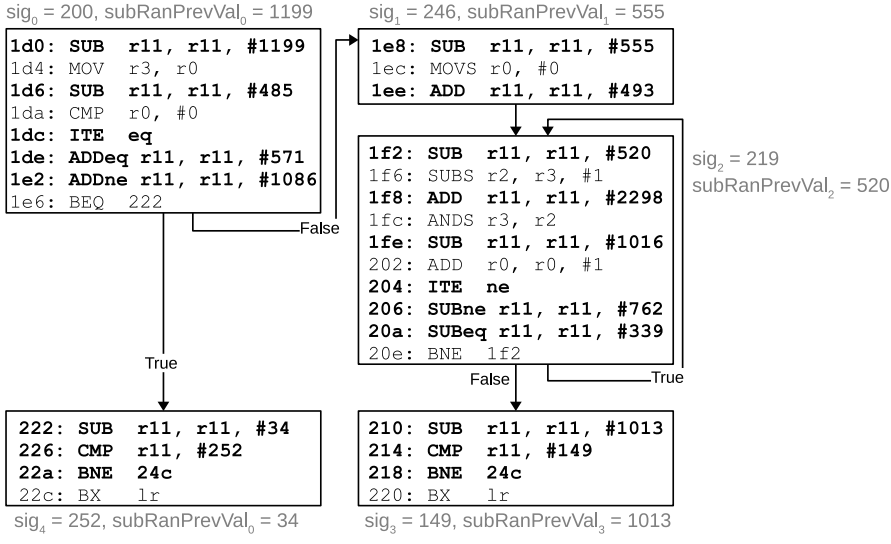


Figure 5.11: Our S-RACFED technique applied to an example program in ARMv7-M

Error Detection Ratio

The measured error detection ratio of S-RACFED is presented in Fig. 5.12. Again, the green bar indicates the percentage of CFEs that was detected by the implemented technique (Det), the red bar indicates the percentage of CFEs that was not detected and caused a wrong result (SDC), and gray bar shows the percentage of CFEs that were either detected by the microcontroller itself or that were not detected and did not corrupt the result of the case study (HD + NE).

As shown, the impact of implemented RACFED selectively changes from case study to case study. For the CU, DIJ, and QS case studies, the SDC ratio of S-RACFED is approximately the same as when RACFED is implemented. For the DIJ and QS case studies, S-RACFED even has a higher error detection ratio. A minor negative impact on the SDC ratio is measured for the BS and FFT case studies. For these two case studies, the SDC ratio increases with one or two percent. The FFT case study also reports a much lower error detection ratio with S-RACFED implemented, but the undetected errors are mainly covered by either the hardware or the application resilience. However, for the BC, CRC and MM case studies, the chart illustrates that the selective implementation has a major negative impact on both the error detection ratio and the SDC ratio.

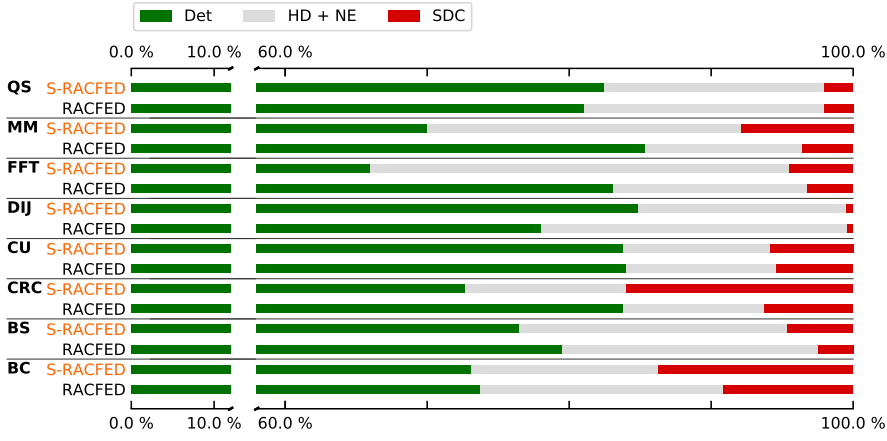


Figure 5.12: The error detection ratio of S-RACFED compared to that of RACFED.

For these case studies, the SDC ratio increases with five to ten percent, resulting in an intolerable SDC ratio between ten and eighteen percent. Regarding the CRC and MM case studies, this increase is coupled with a significant decrease in error detection ratio.

Although little to no difference between the fault injection results of RACFED and S-RACFED were expected, the results indicate that, depending on the case study, large differences are possible. Especially for the smaller case studies, i.e. BC and CRC, bigger differences were noted. The data shows that when S-RACFED is implemented, more single-bit CFEs resulted in premature undetected exits out of the algorithm, increasing the SDC ratio.

Error Detection Cost

The error detection cost of the techniques is shown in Fig. 5.13. The dark-blue bars show the execution time of each technique compared to the unprotected code, while the light-blue bars present the code size of each technique.

Here, the chart displays the expected results, with S-RACFED imposing less execution time overhead and code size overhead. Again, depending on the case study, the reduction in overhead can be larger. On average, the execution time with S-RACFED implemented is $\times 1.66$ that of the unprotected code. In contrast, the execution time with RACFED implemented is $\times 1.87$ that of the unprotected code, on average. The reduction in code size overhead is

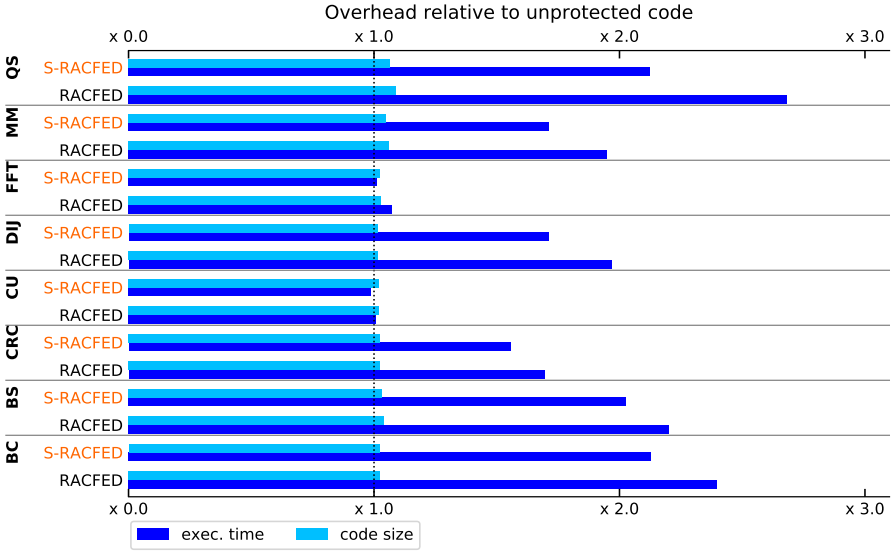


Figure 5.13: The error detection cost of S-RACFED compared to that of RACFED.

smaller, because the only a small part of the entire code base has been protected. On average, code size when S-RACFED is implemented is $\times 1.03$ that of the unprotected code, while the code size overhead of RACFED is $\times 1.04$.

Error Detection Latency

While a lower error detection cost is the advantage of selectively implementing a CFE detection technique, its disadvantage is a higher error detection latency. For the performed experiments, we measured the error detection latency as the number of executed instructions between the point of injection and the execution of the error handler. The results are shown in the box plots of Fig. 5.14, with the error detection latency data for RACFED shown in dark-blue and the data for S-RACFED depicted in light-blue. The median value for each box plot is shown with the orange line. To show the broad range of data as clear as possible, we chose a logarithmic scale on the x-axis.

As illustrated, the error detection latency for RACFED falls within the range from 1 executed instruction to 30 executed instructions for six out of the eight case studies. For the CU and FFT the upper limit of the range extends to 4000

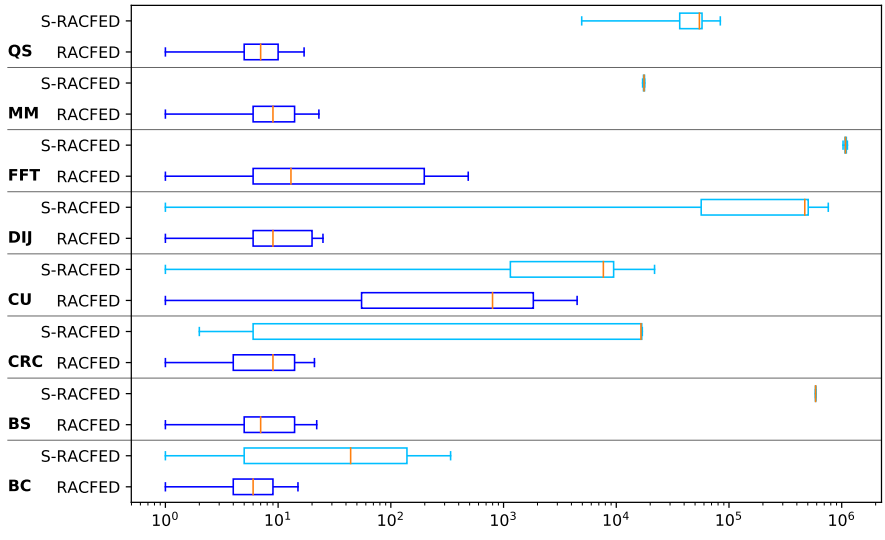


Figure 5.14: The error detection latency of S-RACFED compared to that of RACFED.

instructions and 500 instructions, respectively. The median value for most case studies is located around 10 instructions.

The error detection latency of S-RACFED, however, is much higher for all case studies. Here, we can divide the case studies in two groups. The first group consists of BC, CRC, CU and DIJ, and has an error detection latency range from 1 executed instruction to various upper limits, going to 1 million executed instructions before the CFE was detected for the DIJ case study. The second group consists of the remaining case studies, i.e. BS, FFT, MM and QS, and shows little spreading in its error detection latency. For these case studies, the CFE was only detected once the entire program has executed. The data also shows that little to no CFEs were injected that landed in exit basic blocks, since those would result in a low error detection latency.

These results indicate that S-RACFED is not suited for all types of applications. For example, an application that controls an actuator preferably has a low error detection latency as erroneously controlling an actuator can cause damage or injuries. Applications that are suitable for S-RACFED are applications that process data and return the result. Since only the result is of importance, a higher detection latency can often be tolerated.

5.4 Comparison of Our Techniques

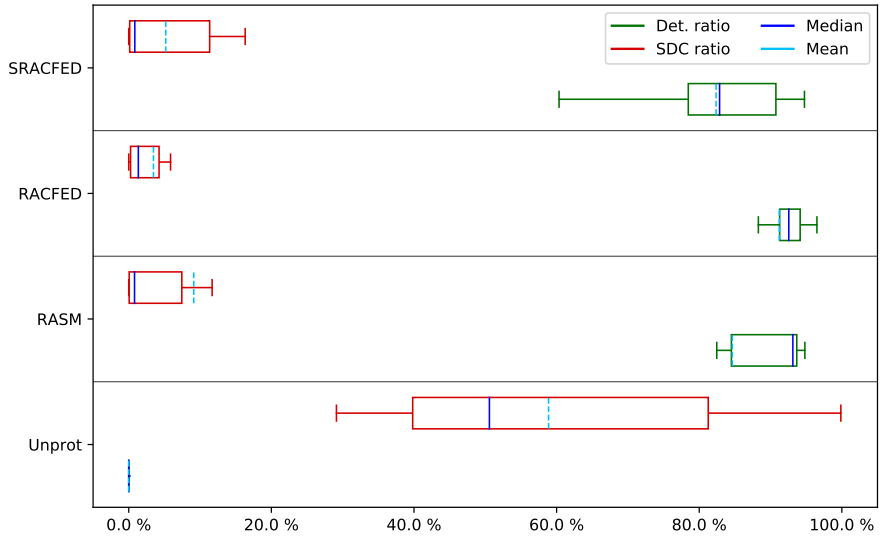
In this section, we compare our three CFE detection techniques to one another. This comparison is based on their inter- and intra-block CFE detection capabilities and on their overhead.

5.4.1 Inter-block and Intra-block CFE detection

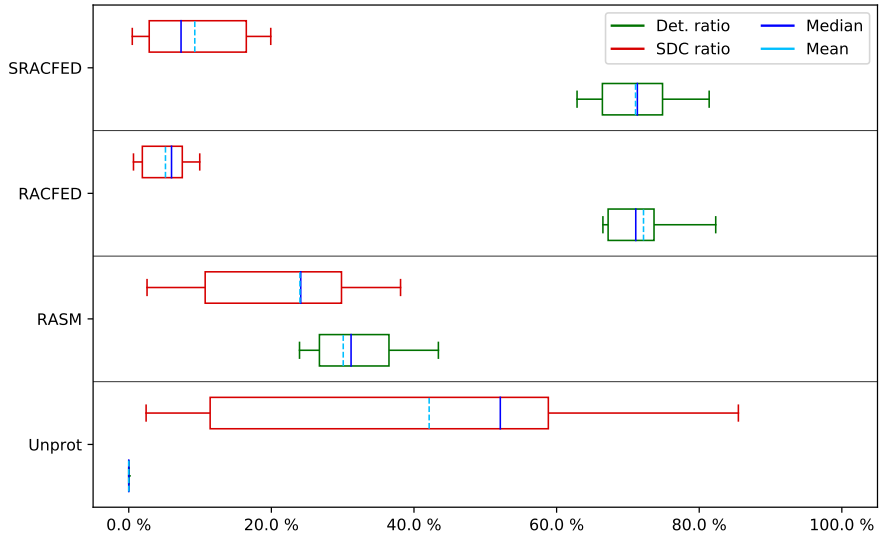
To compare our three techniques to one another, we implemented them for the eight case studies and submitted them to both an extended inter-block CFE and an extended intra-block CFE fault injection campaign. To enable such a fault injection campaign, we made a small modification to our extended CFE injection process (cfr. Section 3.3.2), so that it would either only inject inter-block CFEs or only intra-block CFEs. The results of these injection campaigns are shown in Fig. 5.15, with the inter-block CFE injection results shown in Fig. 5.15a and the results of the intra-block CFE campaign shown in Fig. 5.15b. To be complete, we also show the data for the unprotected case studies.

Regarding the inter-block CFE fault injection results, there are no surprises. As shown in Fig. 5.15a, on average 50 % of the injected inter-block CFEs cause SDC for the unprotected case studies. There is, however, a big range in the measured data as some case studies, such as DIJ and QS, are more inherently resilient than others, such as CRC and MM. Considering our techniques, it can be seen that all three techniques significantly lower the SDC ratio and that RACFED is the better of the three techniques. RASM and RACFED do share the same median value of 93 % for their error detection ratio and the same median value of 1 % for the SDC ratio, but it is clear that RACFED has a smaller range in measured data and a better average value for both criteria. This is mainly because RASM performed poorly for the smaller BC and CRC case studies, with detection ratios of 41 % and 82 % and SDC ratios of 53 % and 12 %, respectively. This poor performance for those case studies impacts the range of the measured data and its average results. Our S-RACFED technique performs worse than expected. Although it achieves the same low median value of 1 % for its SDC ratio, the median value for its detection ratio is only 83 %. For both criteria the technique also shows a bigger range of measured data, again confirming that our strategy for selective implementation can have a big impact on both criteria.

Regarding the intra-block CFE fault injection results, there are some surprises. As shown in Fig. 5.15b, the unprotected case studies have an average SDC ratio of 52 %. The measured data, however, spans a big range going from an SDC ratio of 2 % for the DIJ case study to a ratio of 85 % for the CRC case study.



(a) The results of the extended inter-block CFE fault injection campaign.



(b) The results of the extended intra-block CFE fault injection campaign.

Figure 5.15: The results of the extended inter-block and intra-block CFE fault injection campaigns.

A second surprise is that RASM lowers the SDC ratio for intra-block CFEs. Although not designed to detect intra-block CFEs, the technique achieves an average detection ratio of 30 % and an average SDC ratio of 24 %. While that performance is not good, it is better than expected since our RASM technique applies no form of *instruction monitoring* to detect intra-block CFEs. The measured data for both RACFED and S-RACFED is as expected. Both techniques detect a good amount of intra-block CFEs and reduce the SDC ratio. While both techniques share a median value of 71 % for their detection ratio and 6 % for their SDC ratio, we conclude that RACFED slightly outperforms S-RACFED because the technique has less range in the measured data. This makes the technique more consistent and therefore better.

5.4.2 Realistic Overhead Measurement

Up until now, we only considered the overhead for all techniques when applied to 1 application function. To get a more realistic view of the overhead our three techniques impose, we applied them to the entire code base that is compiled with GCC. Referring to Section 3.4.3, this means that for all eight case studies we left the *function* plugin argument empty and then measured both the code size overhead and the execution time overhead. This is more realistic, since in industrial applications such as those we describe in Chapter 6, multiple functions or even the entire code base is protected instead of just one function. The results of these measurements are shown in Fig. 5.16.

Regarding the code size overhead, Fig. 5.16 reveals no surprises. RASM clearly imposes less code size overhead than both RACFED and S-RACFED. The median value for RASM is $\times 1.13$ while those for RACFED and S-RACFED are $\times 1.21$ and $\times 1.17$, respectively. Furthermore, as expected S-RACFED imposes less code size overhead than RACFED but does display a bigger range than RASM.

The execution time overhead supports a similar conclusion. RACFED clearly imposes the highest execution time overhead, with a median value of $\times 2.18$ and a maximum value of $\times 3.75$. The selective implementation of RACFED imposes a lower execution time overhead, with a median value of $\times 1.87$ and a maximum value of $\times 3.09$. Our inter-block CFE detection technique RASM imposes the lowest execution time of all three techniques, with a median value of $\times 1.65$ and a maximum value of $\times 3.22$.

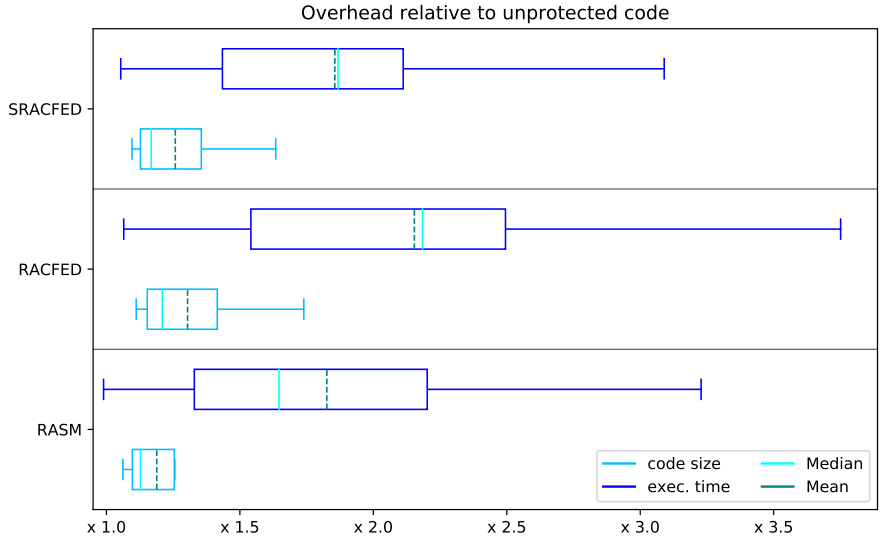


Figure 5.16: The more realistic code size overhead and execution time overhead of our CFE detection techniques.

5.5 Conclusion

To prove the validity of the five guidelines derived in Chapter 4, we developed the RASM technique which follows all five guidelines. To maximize its error detection ratio, RASM uses two gradual updates per basic block and inserts an extra run-time signature verification instruction in exit basic blocks. In order to minimize the imposed error detection cost, the used instructions to update the run-time signature are additions and subtractions, as these are computationally inexpensive instructions in most, if not all, instruction set architectures. Then, a theoretical analysis of the technique's CFE detection capabilities was performed. It showed that all inter-block CFEs that skip at least one signature update will be detected. The only exception are CFEs that jump directly to an exit statement. As such statements are not followed by a signature verification, the CFE remains undetected. Following the theoretical analysis, a fault injection campaign was performed using the same case studies, hardware and fault injection process as used for the comparative study. To put the measured results into perspective, they were compared to the results of CFCSS and YACCA_CMP. Regarding error detection ratio, RASM outperforms the reference techniques with an average error detection ratio of 87 % and an average SDC ratio of 6 %. In contrast, CFCSS and YACCA_CMP have an average

error detection ratio of 70 % and 75 % and an average SDC ratio of 13 % and 14 %, respectively. The same conclusion holds regarding the error detection cost of the techniques. Relative to the unprotected code, the average execution time and code size of RASM are $\times$1.77 and $\times$1.03 respectively. While CFCSS imposes the same code size overhead, its execution time overhead is larger with an average of $\times$1.89. Not surprisingly, the imposed error detection cost of YACCA_CMP is even higher, with an average execution time and code size of $\times$3.39 and $\times$1.07. These figures show that RASM makes the best trade-off between error detection ratio and error detection cost and is therefore the best technique to implement for most case studies.

In order to detect intra-block CFEs, we extended our RASM technique with intra-block updates. This extended technique, called RACFED uses additions and subtractions between the run-time signature and random values to detect intra-block CFEs. To lower the error detection cost, only basic blocks with more than two instructions have these intra-block updates inserted. This is based on the fact that no detectable intra-block CFEs can occur in basic blocks with only one or two instructions. In theory, this implementation enables RACFED to detect all CFE that skip an update. To validate RACFED, we implemented it for eight case studies and performed a fault injection campaign. Next, we measured the error detection cost in an error-free run. To place the measured results in perspective, we compare them to the results of RSCFC and SIED. The fault injection campaign showed that RACFED has the highest error detection ratio and the lowest SDC ratio, compared to RSCFC and SIED. Regarding error detection cost, i.e. execution time overhead and code size overhead, RACFED introduces the lowest cost for six of the eight case studies. For the remaining two case studies, RACFED is only marginally worse than RSCFC and SIED. Combined, these results show that RACFED performs better than RSCFC and SIED, since RACFED detects more errors at a lower cost.

To lower the imposed overhead, we propose a new approach to selectively implement CFE detection techniques, based on the SETA-C approach. In our approach, all basic block have all necessary signature update instructions and other run-time variable update instructions inserted, but the comparison instructions are only inserted in exit basic blocks. To show the validity of this approach, we applied it to our RACFED technique to create the selectively implemented S-RACFED technique. We then re-performed the fault injection experiments with S-RACFED implemented for the eight case studies. The obtained data shows that for some case studies, i.e. CU, DIJ and QS, there was no impact on the error detection ratio and SDC ratio. For the remaining case studies, however, there was a negative impact on both the error detection ratio and the SDC ratio. Next, we measured the imposed overhead of S-RACFED. The data showed that, when S-RACFED is implemented, the execution time

Table 5.1: Summary of the criteria of RASM, RACFED and S-RACFED.

	inter-block CFE error det. ratio	inter-block CFE SDC ratio	intra-block CFE error det. ratio	intra-block CFE SDC ratio	code size overhead*	exec. time overhead*
RASM	85 %	10 %	31 %	24 %	×1.19	×1.83
RACFED	91 %	3 %	72 %	5 %	×1.30	×2.15
S-RACFED	82 %	5 %	71 %	9 %	×1.28	×1.86

* These numbers were acquired when the entire code base was protected, i.e. the *function* plugin argument left empty.

of the protected code is ×1.66 that of the unprotected code and the code size overhead is ×1.03 that of the unprotected code. As expected, this is a lower imposed overhead than RACFED. Finally, we measured the error detection latency of both RACFED and S-RACFED. Where RACFED has a low error detection latency, as most CFEs are detected after each branch between basic blocks, the error detection latency of S-RACFED was significantly higher. This is expected, as for most CFEs the entire program has to execute before they can be detected. This high error detection latency makes S-RACFED less suited for actuator controlling applications, but due to its lower overhead the technique is suitable for data processing applications.

Table 5.1 summarizes this chapter as it provides an overview of our three CFE detection techniques. The table shows the average error detection ratio and SDC ratio for both types of CFE, and the average code size and execution time overhead. For the overhead, we selected the average values of the results measured when the techniques were applied to the entire code base of the case studies. This enables the table to show the most realistic values.

On the Genericity of our Developed Techniques

To finish up this conclusion, we want to clearly indicate that our three CFE detection techniques RASM, RACFED and S-RACFED are applicable to any processor architecture. Although the techniques have been used only on the ARMv7-M ISA throughout this chapter, the techniques do not require any special functionality of that instruction set. As shown in Alg. 4 and Alg. 5, our techniques only require additions, subtractions, comparisons and conditional branches to be implemented, which are supported by any ISA. While the techniques do use conditional execution for some updates, we described in Section 5.2.4 that conditional execution is in fact not mandatory to implement our techniques.

Moreover, we want to stipulate that although our developed techniques have been implemented in the different case studies using the *arm-none-eabi-gcc-7.3* compiler and our GCC plugin, this does not limit the applicability of our techniques. As aforementioned, our CFE detection techniques only use four common types of instructions and we opted to automate the implementation process via our GCC plugin, but this does not affect the techniques themselves. The developed plugin is merely a tool to aid in the implementation of the techniques, but it is not intertwined with the techniques themselves. The developed techniques can be implemented manually or by any other tool for any program for which the low-level code is available.

Chapter 6

Industrial Case Studies

The content and results of the small scale industry 4.0 factory have been published in [85]. The other discussed case study has not been published due to an NDA with Televic Healthcare NV.

This chapter presents two industrial case studies on which we have applied our research. The first industrial case study is a small scale factory and was mainly used to analyze which effects CFEs would have on such type of setup and to determine if RACFED would increase the reliability. The second case study consists of two modules communicating over a bus system. This case study uses hardware and software of Televic Healthcare NV. This case study had as purpose to analyze how CFEs would affect the bus communication.

6.1 Small Scale Industry 4.0 Factory

In order to further validate the research, i.e. the developed techniques, the GCC plugin and the fault injection tool, a small scale Industry 4.0 case study was developed. This case study consists of three stations from the Festo-Didactic MPS® series: a distribution station, a testing station and a sorting station [54]. 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. As their names give away, each station performs a part of that process. The setup is shown in Fig. 6.1, with the distribution station on the left, the testing station in the middle and the sorting station on the right.

Figure 6.1: The small scale Industry 4.0 setup. 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.

First, the small scale setup is presented. Next, the effects of CFEs in the unprotected setup are demonstrated, both theoretical and through fault injection, to clearly indicate what can go wrong. Following, a software-based CFE detection mechanism is implemented and the fault injection campaign repeated. Finally, a basic but functional recovery method is presented, which enables the setup to recover from CFEs automatically and resume its normal operation.

6.1.1 Case Study Setup

In this section, we describe each station in more detail, then we discuss the structure of the programs controlling each station.

Distribution Station

The distribution station is the start of the process. It contains a stacked magazine, an ejection cylinder, a swivel drive and a vacuum gripper. Its execution process is represented by a state machine in Fig. 6.2. The stacked magazine contains the workpieces, which are pushed out one by one using the ejection cylinder. Once pushed out, the swivel drive moves the vacuum gripper

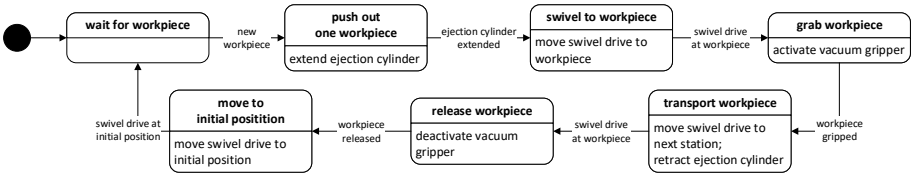


Figure 6.2: An abstract state machine of the process of the distribution station.

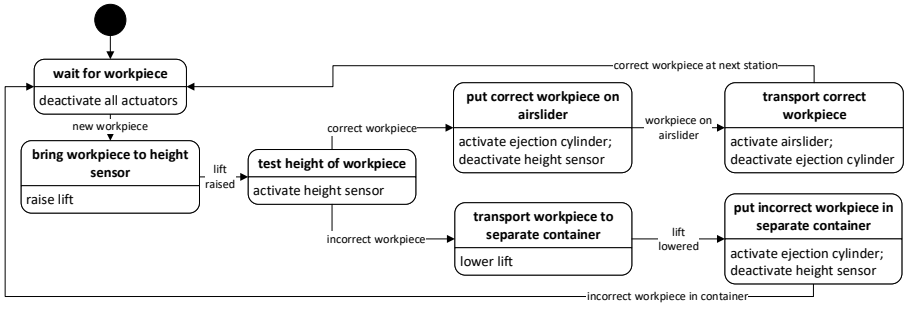


Figure 6.3: An abstract state machine of the process of the testing station.

to the workpiece, enabling it to be grabbed. Finally, the gripped workpiece is moved to the next station and released.

Testing Station

The testing station is the second station of the process. It tests the height of the received workpiece. Only workpieces with a correct height are then transported to the next station via an airslider. Incorrect workpieces are taken out of circulation and put aside. In essence, this station performs a quality check and discards workpieces that do not meet the desired quality. This execution flow is shown in Fig. 6.3.

Sorting Station

The sorting station is the final station of the process and its execution flow is shown in Fig. 6.4. It receives the correct workpieces from the testing station and sorts them according to color and material. This distinction is made based on the input from two sensors. The first sensor detects whether or not the workpiece is made out of magnetic material. The second sensor determines whether or not a workpiece is black. We used three types of correct workpieces:

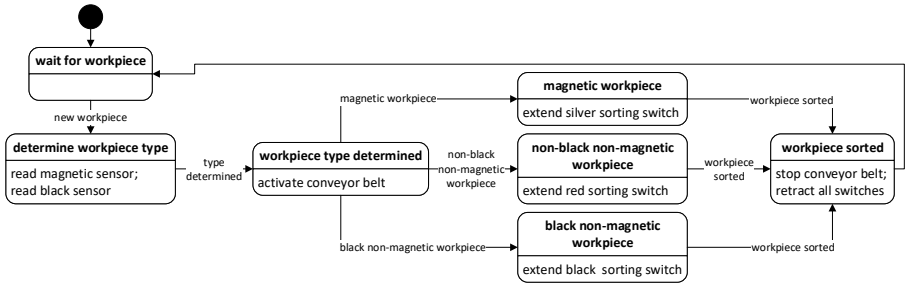


Figure 6.4: An abstract state machine of the process of the sorting station.

silver magnetic workpieces, non-magnetic red workpieces and non-magnetic black workpieces. Once the type of the workpiece is determined, a conveyor belt moves it forward, until a sorting switch pushes the workpiece into the correct container.

Control Program Structure

In order for each station to perform its task, the actuators have to be controlled for the right amount of time. To determine whether or not an actuator has reached its desired position, sensors are used. These sensors are mounted at the positions of interest and send a signal, a logical *1*, once the actuator has reached that position.

This actuator-sensor combination and the fact that each station has to perform its task sequentially, has led us to implement the control program of each station as indicated in Alg. 6. The shown pseudo-code describes the control program of the distribution station, but the structure is similar for the testing and the sorting stations. A logical *1* is shown as ON in the pseudo-code and a logical *0* as OFF.

As can be seen, the program uses *busy waiting* to correctly drive each actuator. This means that the processor repeatedly checks whether or not a certain resource is available, as shown on line 3 of Alg. 6. An alternative implementation is to put the processor in a sleep mode and wake it up once the resource is available. This sleep strategy is most commonly used to preserve power in a battery-powered system, but this is of no concern for our small-scale setup.

In total, this setup uses three microcontrollers to perform its function as we selected an NXP LPC 1768, which is an ARM Cortex-M3 driven microcontroller, to drive each station and thus to execute the control program.

Algorithm 6 Pseudo-code describing the control program of the distribution station.

```
1:  function MAIN()
2:      extendCylinder ← ON                          ▷ push out workpiece
3:      while cylinderIsExtended ≠ ON wait
4:      rotateSwivelLeft ← ON
            ▷ move gripper to workpiece
5:      while swivelIsLeft ≠ ON wait
6:      rotateSwivelLeft ← OFF
7:      vacuumOn ← ON                                ▷ grab workpiece
8:      while WorkpieceIsGripped ≠ ON wait
9:      extendCylinder ← OFF
10:     rotateSwivelRight ← ON
            ▷ move workpiece to next station
11:     while swivelIsRight ≠ ON wait
12:     rotateSwivelRight ← OFF
13:     vacuumOn ← OFF                               ▷ release workpiece
```

6.1.2 Effects of Control Flow Errors

This section first presents a theoretical analysis of the three categories of possible CFE effects on the control program of each station. Next, we describe the experimental analysis which was performed through a fault injection campaign.

Theoretical Analysis

To analyze which high impact effect a CFE can have on our setup, consider Fig. 6.5, which is a high-level representation of the control program of each station. Each control program is the sequential execution of changing the outputs of the microcontroller and then waiting for a specific sensor to indicate the mechanical part or parts have moved to the desired location. Based on that structure, three categories of possible CFE effects can be derived: a CFE skipping one or more wait instructions, a CFE landing on a wrong wait instruction and a CFE causing the re-execution of a previous part of the control program.

The first CFE effect that can be derived is caused by a CFE skipping one or more wait instructions. This type of CFE forces the microcontroller to update its outputs to drive the mechanical parts, while it is not yet necessary. In turn, this can lead to dangerous situations and/or cause damage. An

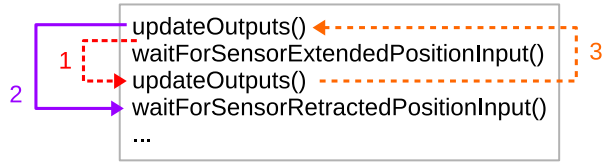


Figure 6.5: High-level representation of the control program of each station and the two categories of possible CFEs.

example is indicated by the red jump (arrow 1) in Fig. 6.5. This CFE skips the *waitForSensorExtendedPositionInput* instruction and causes the microcontroller to update its outputs. Such CFEs disrupt the execution flow of the station and cause the incorrect treatment of the current workpiece.

A second CFE effect that is possible is a CFE landing on a wrong wait instruction. An example of such CFE is indicated by the purple jump (arrow 2) in Fig. 6.5. It causes an infinite wait of the microcontroller for a sensor input that will never arrive, as its corresponding actuator has not been driven. Moreover, the actuators that were being controlled, are driven indefinitely which might cause premature defects on those mechanical parts.

The final and third CFE effect that can be derived is the re-execution of a program part due to a CFE. Such CFE jumps to a previous part of the control program and hence forces the re-execution from that point onwards, as shown by the orange jump (arrow 3) in Fig. 6.5. The effect of such a CFE is really dependent on its origin and landing points. In some cases, it might have no effect on the outcome of the setup or in other words, such a CFE can trigger an idempotent operation. For example, such a CFE can cause the testing station to remeasure the height of the current workpiece. While this results in a small time delay in the workpiece processing, the station will be able to make the correct decision and either pass the workpiece to the sorting station or put it aside.

Of course, the effect of a re-execution CFE might be more serious. For instance, when it occurs in the distribution station while it is transporting a workpiece to the testing station. Due to the re-execution, that station might discard the current workpiece and start processing the next workpiece in its magazine. This results in workpiece loss. Or, regarding the sorting station, such CFE might cause the current workpiece to be sorted on the wrong container.

Experimental Analysis

To inject CFEs, we used our in-house developed Python fault injection framework presented in Section 3.3. While this framework is meant to inject many CFEs automatically, we extended it to allow a more customized CFE injection. We created a process to define a start address for the CFE and the address to jump to. Both these addresses are obtained from the disassembly file of the control program of the station under test. The address to jump to is created by flipping one bit of the start address. We used the one-bit bit flip model as this is the most common used fault model in this area of research.

For this preliminary study, approximately 30 CFEs were injected in each station. Of those injected CFEs, 60 % caused a corruption in the execution of the station, 10 % were detected by the fault handlers present in the Cortex-M3 and the remaining 30 % had no effect on the station. The most common outcome of the injected CFEs was that an actuator was not controlled, resulting in the target program waiting indefinitely for the sensors to indicate that the actuator had finished its task.

Regarding the distribution station, this either meant that a workpiece was not pushed out of the magazine, that it was not gripped by the vacuum gripper or that it was not released once transported to the next station.

For the testing station, this resulted in workpieces not being ejected, or the airslider remaining on after a workpiece has been pushed onto it or the airslider shutting of too soon preventing the workpiece from reaching the next station.

The most common CFE effects for the sorting station were either the conveyor belt that was left on and the stopper, that keeps the workpiece in front of the color and material sensor, that was not retracted once the color and material were determined. In some occasions, a CFE caused the conveyor belt to stop and caused the sorting switch to retract too soon, keeping the workpiece on the conveyor belt. This meant that the current workpiece was sorted as the type of the next workpiece, thus corrupting the entire sorting order of the successive workpieces.

6.1.3 Adding CFE Detection

The above section makes it clear that Industry 4.0 control programs have to be made reliable. Therefore, we used our GCC plugin to apply RACFED to the control program [80, 84]. Because we used an open source library to control the digital Input/Output lines of the NXP LPC1768 and the functionality of each station is programmed using different functions, we left the *function*

plugin argument empty. From Section 3.4, this means that all functions must be protected unless the `noProtection` attribute is set. Due to this setup, approximately 480 functions are protected per station.

Fault Injection Campaign

To analyze the effect of RACFED, we repeated the fault injection campaign. As implementation for the `CFED_Handler` function, i. e. the error handler, we chose to light an LED followed by an infinite loop. The LED gave us a visual confirmation and the infinite loop makes it easy for the fault injection tool to determine whether or not the CFE was detected.

As expected, most of the injected CFEs are detected by RACFED. Furthermore, for this study, the CFEs that RACFED did not catch, were caught by the hardware fault handlers of the microcontroller. An exception to this were some undetected CFEs within the testing station. These made the station restart its operation, which then correctly processed the given workpiece, resulting in a small delay in the whole process. To conclude, after adding our RACFED technique, none of the injected CFEs resulted in an incorrectly processed workpiece in this study.

However, we know from previous fault injection studies that there are CFEs which remain undetected for RACFED. A more extensive study is needed to discover such CFEs, in order to analyze their effect on the process and to determine a way for RACFED to detect them.

Automated Recovery Method

Since most of the CFEs are now detected, we enhanced the error handler to automatically recover from the CFE, as shown in Alg 7. As shown in Section 3.4.3, the action to perform when a CFE is detected is application specific. For our Industry 4.0 setup, we deemed automatic recovery to be the appropriate measure. To enable automatic recovery, we divided the functionality of each station in several phases and added a variable to indicate the current phase. When a CFE is detected, the error handler reads this variable to know which phase of the process was active and restarts the process from that point onward. Once the current workpiece has been processed, a software reset is issued to make sure there are no lingering effects of the detected CFE.

While crude, this recovery method proved effective for all of the injected CFEs. This means that none of the injected CFEs during this preliminary study was able to corrupt the *phase* variable. However, such CFEs are possible and

Algorithm 7 Pseudo-code describing the recovery method applied to the control program of the distribution station.

```
1:  global phase
2:  function CFED_Handler()                          ▷ CFE error handler
3:      switch phase do
4:          case WORKPIECE_AVAILABLE :
5:              pushOutWp()
6:              transportWp()
7:              vacuumOn ← OFF                       ▷ release workpiece
8:              break
9:          case TRANSPORT_WORKPIECE :
10:             transportWp()
11:             vacuumOn ← OFF
12:             break
13:         case RELEASE_WORKPIECE :
14:             vacuumOn ← OFF
15:             break
16:     systemReset()
17: function pushOutWp()                             ▷ push out workpiece
18:     extendCylinder ← ON
19:     while cylinderIsExtended ≠ 1 wait
20: function transportWp()                           ▷ transport workpiece
21:     rotateSwivelLeft ← ON
22:     while swivelIsLeft ≠ ON wait
23:     rotateSwivelLeft ← OFF
24:     vacuumOn ← ON
25:     while WorkpieceIsGripped ≠ ON wait
26:     extendCylinder ← OFF
27:     rotateSwivelRight ← ON
28:     while swivelIsRight ≠ ON wait
29:     rotateSwivelRight ← OFF
30: function main()                                  ▷ main loop
31:     phase ← WORKPIECE_AVAILABLE
32:     pushOutWp()
33:     phase ← TRANSPORT_WORKPIECE
34:     transportWp()
35:     phase ← RELEASE_WORKPIECE
36:     vacuumOn ← OFF                               ▷ release workpiece
```

Table 6.1: The measured code size overhead and execution time overhead of RACFED when applied to the small scale Industry 4.0 setup, shown per station.

	Distribution Station	Testing Station	Sorting Station
Code Size overhead	×1.66	×1.66	×1.67
Exec. Time overhead	×1.00	×1.00	×1.00

further study is needed to implement a more fine-tuned recovery method that is effective for all possible CFEs. Furthermore, this recovery method only recovers the software from the CFE. If due to the CFE the workpiece has been sorted in a wrong container, it will remain in that container, despite the CFE being detected and being recovered.

Overhead of the CFE Detection

As for the previous case studies, we also measured the code size overhead and the execution time overhead of RACFED when applied to the small scale Industry 4.0 case study. To measure the code size overhead, we again used the *text* output of the *arm-none-eabi-size* tool, when used on the compiled *.elf* file. To measure the execution time overhead, we used an on-board hardware timer of the NXP LPC 1768 to measure the processing time of one workpiece. We measured this time for all workpieces on all three stations, both for the unprotected and the protected code. The measured results are shown in Table 6.1.

As can be seen, applying RACFED to the code base of each station resulted in the code size of each station to be 1.66 times larger than that of the unprotected code base. The execution time of the protected code is, however, the same as that of the unprotected code. While this might be surprising at first, when further analyzing the code base and how it executes, this result is more logical. As described in Section 6.1.1, the control program for each station consists of the sequential execution of changing the output to move mechanical parts of the station, then waiting for the sensors to indicate that the mechanical part has reached the desired location, then changing the outputs again, and so on. This shows that for the main part, no code is executing and the microcontroller is waiting for the desired sensor input. Therefore, the impact of RACFED on these control programs is minimal and in fact negligible. Table 6.1 thus shows that applying RACFED always imposes code size overhead, but that its impact on the execution time overhead depends on the protected program.

6.1.4 Conclusion

This section presented a preliminary CFE study on a small scale Industry 4.0 setup. As Industry 4.0 case study we used three Festo-Didactic MPS® stations that make up a small factory. The function of the three stations is to process workpieces from the stacked magazine and to sort workpieces of correct height according to color. Each of the stations is controlled by an ARM Cortex-M3, a highly used industrial microprocessor.

First, we used our fault injection tool to inject CFEs into the unprotected code of each station. Even the small number of injected faults clearly showed that CFEs are a real danger for Industry 4.0 setups, as they can lead to hanging programs or actuators being controlled erroneously.

Next, we implemented the RACFED technique for the control programs of each station using the GCC plugin, to analyze its effect on the control programs. Once implemented, RACFED was able to detect almost all injected CFEs, increasing the reliability of the program. The technique also enabled to implement a crude CFE recovery method, which enables the control program to automatically recover from the CFE and continue processing the next workpieces. This increases the availability of our factory.

Finally, we discussed the overhead of applying RACFED to the control programs. While a code size overhead of $\times 1.66$ was measured for each station, RACFED had no impact on the execution time of the stations.

6.2 Bus Communication

This section discusses an industrial case study performed in collaboration with Televic Healthcare NV. Due to a non-disclosure agreement, details have been omitted and proprietary systems have been renamed.

This case study has focused on two devices, herein named Producer and Consumer, connected through a communication bus, herein named the bus. The Producer sends out one message of the type MSG1 per second and does not react to the message of type MSG2 sent by the Consumer. In contrast, the Consumer sends out one message of the type MSG2 per second and reacts to the message of type MSG1 by toggling an LED each time such a message is received. This is depicted in Fig. 6.6. The codebase of both devices is largely the same, as it mostly contains the software stack to communicate on the bus.

The purpose of this use case was to determine which effects CFEs have on the

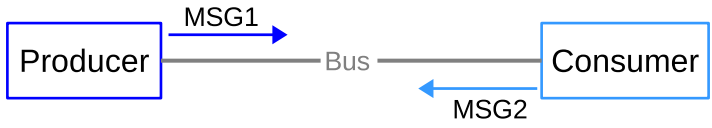


Figure 6.6: The use case setup. A Producer and a Consumer each sending a message on the bus.

bus communication. Since almost no application layer is present in the codebase of the two devices, fault injection experiments can determine the resilience of the communication software stack against CFEs. By applying RACFED and repeating the fault injection experiments, a rise in resilience should be seen.

6.2.1 Experiments

We performed fault injection experiments on the two devices, both on the unprotected and the protected code. The data collected during the fault injection on the unprotected code shows its inherent strength or weakness against CFEs, while the data collected during the fault injection on the protected code is used to determine the impact of RACFED on the resilience to CFEs of the communicating devices. First, we'll discuss the used fault injection process, then show the used plugin arguments to apply RACFED and conclude with the results.

Fault Injection Process

The fault injection experiments performed on this setup consist of the following steps:

1. List all C functions. For both devices this is a list containing approximately 120 functions.
2. For each function in that list
   (a) Inject 100 CFEs using our *Random In Range* injection process described in Section 3.3.2 in which the range is the entire list of functions created in step 1. This means that each CFE starts within the function, but ends in any of the other functions contained in the list. This will be indicated as RIR (random in range).

(b) Inject 100 CFEs using our *Random In Range* injection process in which the range consists of the current function only. This means that each CFE starts and ends within the current function. These results will be indicated as RIF (random in function).

Using the GCC plugin to Implement RACFED

To implement RACFED using our GCC plugin, we added the `CFED_Handler` method to the codebase of the two devices and added the necessary compiler flags as follows:

- The `CFED_Handler` method was implemented as an infinite loop. This enables easy detection by the SWIFI-tool, as it just needs to read the PC register of the target to determine if this function has been called or not.
- The plugin argument `function` was left empty, so the entire codebase compiled with the plugin is protected. As listed in the previous section, this resulted in approximately 120 functions being protected.
- The `techniqueType`, `technique` and `selective` arguments are specified to define RACFED, using `fullCFED`, `RACFED` and `0` respectively.
- Finally, two compiler flags were added to make sure registers `r6` and `r7` are available, using the GCC option `-ffixed-r<number>`. As explained in Section 3.4.4, register `r7` is used as stack pointer for the run-time variable stack and register `r6` is used to contain the run-time variable of RACFED.

Results

The results of the fault injection campaign are again categorized as introduced in Section 3.2.1, and are shown in Fig. 6.7. Again, the green bar indicates the CFEs detected by RACFED, the red bar indicates the non-detected errors that were able to corrupt the bus communication and the gray bar indicates the errors that where either detected by a hardware-based mechanism present in the device, or errors that were not detected and that did not corrupt the bus communication.

Regarding the unprotected code, we see that the SDC ratio ranges between 10 % and 12 %. In other words, the bus communication was corrupted by 10 % to 12 % of the injected CFEs on average. Bus communication corruption either means that too many messages were sent, too few messages were sent or that wrong messages were sent. Once RACFED is implemented, this ratio goes

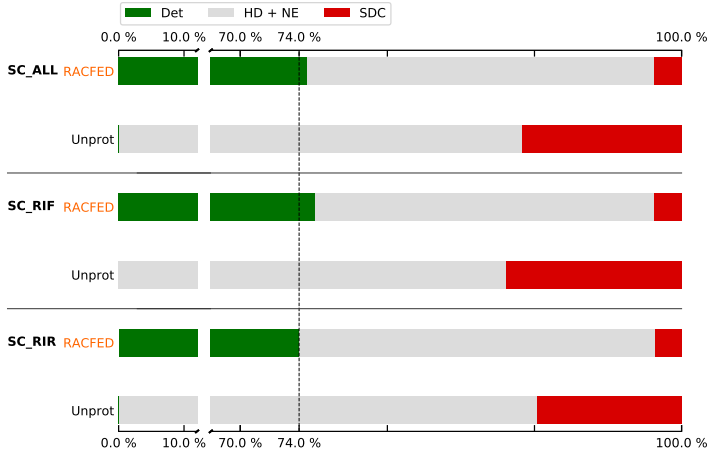


Figure 6.7: The results of the fault injection campaign on the Consumer device. The two lower bars present the results of the RIR injection, the middle two bars show the results of the RIF injection and the top two bars show the overall results.

down to approximately 2 %. As discussed in Section 5.2.5 this is inherent to RACFED as it is unable to detect all CFEs. In addition, RACFED detects on average 74 % of the injected CFEs.

However, this error detection and SDC reduction comes at a cost, i.e. an increase in execution time and code size. Using an on-board timer, we measured the execution time for both the unprotected and the protected code and using the *arm-none-eabi-size* tool we measured the code size of the produced *.elf* file for both versions. With RACFED implemented, the code needs $\times 1.7$ the execution time compared to the unprotected code and the code size has increased to $\times 2.3$ the code size of the unprotected code.

6.2.2 Conclusion

We performed a fault injection campaign on both the unprotected and protected code of the devices. The results of the fault injection campaign performed on the unprotected code, revealed that on average 10 % of the injected errors resulted in bus corruption. When protecting all C functions using RACFED, an average error detection ratio of 74 % and an average SDC ratio of 2 % are measured. Implementing RACFED thus results in a significant reduction of the SDC ratio. This reduction of the SDC ratio, and the error detection ratio

of 74 % do come at a cost. With RACFED implemented, the execution time increased to ×1.7 that of the unprotected code and the code size even increased to ×2.3 that of the unprotected code.

However, protecting the full codebase might not be an option for the final release of the code. As aforementioned, little to no application layer was present in the current codebase. An elaborate application layer and possibly other functionalities will be present in the final codebase, which could mean that a fully protected codebase no longer fits in the memory of the selected microcontroller. To solve this issue, there are three possible solutions. A first solution would be to change the microcontroller to a more expensive one with more on-board memory and a second solution would be to include an extra memory chip on the devices. While these two solutions would be effective, they would also incur an extra cost per product. This would decrease the profit and lead to losing customers. Therefore, a more feasible third solution is to protect less C functions. Protecting less C functions results in less overhead, making it possible to fit the protected code in the current microcontrollers. A first attempt to identify C functions that would be safe to leave unprotected, i.e. have no impact on the SDC ratio, proved futile. Every assessed configuration of protected and unprotected functions increased the SDC ratio of the devices, making them unusable. This is because there is no application layer present in current codebase. In the final release of the code, the application layer will implement certain error detection mechanisms, such as a watchdog timer and performing CRC checks on the data to be put on the bus, that will also detect a certain percentage of CFEs making it safe to keep certain C functions unprotected.

6.3 Industrial Applicability of RACFED

To conclude this chapter, we will discuss the industrial applicability of our RACFED technique and other research results.

As shown in the previous chapter and in this chapter, using our GCC plugin it is easy to apply a CFE detection technique to the codebase of an application. As shown in Table 6.2, once applied our technique detects most CFEs and always reduces the SDC ratio. In the table, the academic case studies described in Section 3.1 are depicted as *Acad. case studies* and no results for the small scale Industry 4.0 setup are shown, as this was a preliminary study only and is not feasible to draw general conclusions. These results show that RACFED increases the resilience to CFEs of the target application, making it a valid choice to be used in industrial applications.

Table 6.2: Overview of the measured error detection ratio and SDC ratio for RACFED.

	Error Det. ratio	SDC ratio
Acad. case studies (Avg.)	78.86 %	4.35 %
Industry 4.0 setup (Avg.)	—*	—*
Bus communication (Avg.)	74.00 %	2.00 %
Measured range of ratio	[69.70 %, 85.37 %]	[0.41 %, 11.50 %]

* No results shown as only data from a preliminary study is available.

Table 6.3: Overview of the measured code size overhead and execution time overhead, when RACFED is applied to the entire codebase.

	Code Size Overhead	Exec. Time. Overhead
Acad. case studies (Avg.)	×1.30	×2.15
Industry 4.0 setup (Avg.)	×1.66	×1.00
Bus communication (Avg.)	×2.30	×1.70
Measured range of overhead	[×1.11, ×2.30]	[×1.00, ×3.75]

Next to the error detection ratio, the overhead imposed by RACFED also determines whether or not is should be applied to an application. Table 6.3 provides an overview of the overhead measured throughout this thesis, when RACFED was applied to the entire codebase of the target application.

To no surprise, the code size overhead grows the more protected code is present in the compiled *.elf* file. For the Bus communication case study, all protected code ended up in the *.elf* file, producing a high code size overhead of ×2.30. As shown, the measured code size overhead varies between the minimum value of ×1.11 and the maximum value of ×2.30. On the other hand, the execution time overhead highly depends on the application. As presented in Table 6.3, applying RACFED had no impact on the execution time of the small scale Industry 4.0 setup, had some impact on the execution time of the Bus communication case study and had major impact on the execution time of the academic case studies. Throughout the thesis, the measured values range between the minimum value of ×1.00 and the maximum value of ×3.75, which was measured for the BC algorithm.

Although experiments (cfr. Section 5.2.6) have shown that the overhead introduced by RACFED is lower than that of established techniques, it is

still rather large. Whether this is acceptable or not depends on the target application and target hardware. For our academic case studies, the overhead was of no importance, as there is still plenty of memory to store the code available and no real-time behavior is needed. For the small scale Industry 4.0 setup, the code size overhead was again of no importance and the measurements even revealed there is no difference in execution time between the unprotected and the protected versions of the program. As described, this is because most time is spent on waiting for sensor input generated by the mechanical parts. During this waiting, no code is executed resulting in the same execution time for both versions. This leads us to believe that our RACFED technique can be used for industrial applications, as its impact can be minimal. When considering the Bus communication case study, we know that the current code size overhead is too large. The current code base did not contain any application logic and other needed logic. This extra needed logic also has to be protected by RACFED, meaning an even higher code size overhead will be measured. This will result in the code no longer fitting within the current memory. Then the options are 1) to use a more expensive microcontroller which has larger memory to store code, or 2) to lower the code size overhead by e.g. applying RACFED and S-RACFED in a more intelligent manner. While option 1 is the easiest solution, it does mean a higher cost for the final product and thus a lower profit margin, which is not desired. Therefore, option 2 is the way to go, but as we describe in the Future Work section of the next chapter, this does require some refactoring of the GCC plugin. For this case study, the imposed execution time did not result in a late delivery of the messages sent over the bus. It could however have an impact on the general application, once the extra logic is implemented, but this remains to be seen.

To summarize, this section has shown that RACFED always increases the resilience to CFEs of the system, but whether or not the imposed overhead is of concern depends on the application and the system hardware. This makes the current combination of RACFED and the GCC plugin already a viable option to apply to some industrial applications, but with an adjustment in the working of our plugin, multiple techniques, e.g. both RACFED and S-RACFED, could be applied to the same code base. This could lower the introduced overhead, making our research a viable option for more industrial applications.

Chapter 7

Conclusions

This chapter concludes this thesis by providing some possible directions for future work and by providing a summary.

7.1 Future Work

This section describes possible extensions on this research, such as using the research on an application processor, eliminating the undetectable shared state CFEs from RACFED, performing an more extensive fault injection campaign on the small scale Industry 4.0 setup, further enhancing the usability of the GCC plugin, researching the impact of processor architecture on the CFE detection techniques and including DFE detection techniques in the GCC plugin.

Moving to an Application Processor

Near the end of the PhD, the idea was launched to move to an application processor, i.e. an ARM Cortex-A7, and apply the research results. The goal would be to setup an architecture in which a reliable world and an unreliable world would execute separately. The unreliable world would execute infotainment and user-interaction applications on top of the Linux operating system. The reliable world would execute the core functionality of the system using bare-metal code and would be the application domain for the GCC plugin and CFE detection techniques.

In preliminary work, we attempted to use the TrustZone technology to setup this architecture, as this technology provides two execution worlds on the application processor [55]. We were able to setup two worlds and use our GCC plugin to implement RACFED on a bare-metal application in one of the worlds. However, this preliminary work did not result in setting up the complete envisioned architecture and therefore another option was briefly explored. More detailed information on the work performed for this research track can be read in Appendix A.

Eliminating Undetectable Shared State CFEs from RACFED

In Section 5.2.5, we discussed three types of CFEs that RACFED cannot detect: 1) CFEs that jump to a return instruction; 2) intra-block CFEs that skip an original instruction, but no update instruction; and 3) shared-signature CFEs. While the first two types of undetectable CFEs cannot be eliminated from RACFED, the third type could be eliminated from RACFED by enhancing the compile-time process. One option to remove these undetectable CFEs is executing an extra check while adding the intra-block updates that verifies whether or not the resulting value of the run-time signature is unique. A second option is to stop using random numbers during the implementation of RACFED and use fixed numbers.

We chose not the implement the first option, the extra check, because in some cases, e.g., programs with a lot of basic blocks or programs with large basic blocks, this could lead to RACFED being not usable. Because we opted to implement RACFED as a technique that only uses one register, the values that can be used during the run-time signature updates are limited. As run-time signature updates, we use an instruction in the form of `ADD r11, r11, #value`. The allowed values for the `#value` used in the run-time update are limited and depend on the used ISA. To conclude, for the ARMv7-M ISA this means that the run-time signature must remain within the interval $[-2000, 4000]$, giving a range of 6000 unique values. For the ARMv6-M ISA, the run-time signature value must remain within $[0, 255]$, giving a range of only 255 values. While the range for the ARMv7-M ISA would suffice in most cases and the extra check to verify the uniqueness of the run-time signature could be implemented, this check would make it practically impossible to use RACFED for the ARMv6-M ISA due to its limited range of values. Therefore, this check was not implemented.

Converting the implementation of RACFED to use two registers, would greatly expand the allowed values for the run-time signature and would enable this uniqueness check to be implemented. However, further study is needed to

determine the best way to convert RACFED to a two register implementation without having an impact on its error detection ratio.

As aforementioned, the second option to eliminate the shared state from RACFED is to discard the use of random numbers and instead use fixed numbers. One option to discard random numbers and to use fixed numbers is to use a Sidon sequence to determine the compile-time signature and the `subRanPrevVal` for each basic block in the first step of the implementation algorithms of both RASM and RACFED. A Sidon sequence is a set of unique natural numbers in which all pairwise sums are different [56]. Since those are the two properties needed for the compile-time signature and the `subRanPrevVal` value, such a set could be used to select the desired values. Whether or not a Sidon sequence exists that can be used for all possible programs, requires further research. Another option to discard random numbers and to use fixed numbers is to develop an algorithm that determines up front which numbers to use as compile-time signature and the `subRanPrevVal` value for each basic block, which operations to use for the intra-block updates and with what values to guarantee the uniqueness of each update result. How to develop such an algorithm is subject to further study.

Extensive Fault Injection on Small Scale Factory

A first extension for this research is to perform a more extensive fault injection campaign on the small scale factory presented in Section 6.1. Up until now, only a preliminary study has been performed on this setup, which enabled us to draw preliminary conclusions about both RACFED and the proposed recovery mechanism. A more extensive fault injection campaign would serve to:

- better understand all possible effects of CFEs on a more industrial setup. In this thesis, we defined three possible CFE types based on the data. The more extensive study could reveal that more types are possible or that more causes for the derived CFE types are possible.
- Better evaluate if RACFED does indeed prevent all dangerous situation to occur. Although RACFED detect almost all errors injected during the preliminary study, we know from other experiments that RACFED is unable to detect all CFEs, cfr. Section 5.2.6. A more extensive fault injection campaign that covers more code, will reveal whether or not dangerous situations can still occur in the small scale factory.
- Finally, a more extensive study will uncover whether or not the recovery mechanism is equipped to deal with all possible CFEs. For example, the

performed preliminary fault injection did not inject errors in the recovery mechanism. However, since the recovery mechanism itself is now quite large, it can also fall victim to CFEs. So by also injecting faults while the recovery mechanism is executing, the ability of the mechanism to deal with those will be known.

In order to perform such an extensive fault injection campaign and eliminate the possibility to break the equipment, or eliminate the possibility to put the equipment under too much stress, a Hardware-in-the-Loop (HIL) setup must be created. In a HIL setup, the input from sensors can be simulated and the output meant for the actuators can be captured. This means that the microcontroller does not need to be connected to the setup in order to execute its functionality. Since the microcontroller would not be connected to the physical setup, the possibility of damage is eliminated. At this moment, further research is needed to determine the best approach to create a HIL setup for each of the stations of the small scale factory, either by using the dSPACE SCALEXIO present at the research group or by creating a new system, and how to incorporate that setup into the developed fault injection tool [57].

Enhancing the usability of our GCC plugin

With the current implementation of our GCC plugin, only one technique can be implemented for the target functions that must be protected. This approach limits the usability of the plugin. For example, enabling both RACFED and S-RACFED to be implemented for different target functions could have lowered the imposed overhead on the case study described in Section 6.2 with a limited impact on the error detection ratio and SDC ratio. Not all functions in the code base require a low error detection latency and thus could be protected by a selective technique such as S-RACFED. Because for the current implementation of the plugin, the CFE detection technique to implement is selected using a compiler flag, such mixture of techniques is not supported.

A solution to enable such behavior is to create a new function attribute, e.g., `CFEDtechnique`, in which the user can specify which technique to apply for this function. With such an attribute, the user can specify which technique to implement for each individual target function and thus tailor the best detection strategy. Further study is needed to find out how to build such a function attribute that also allows to specify whether or not the selected technique must detect intra-block CFEs and to specify whether or not the selected technique must be implemented selectively, as these criteria are now also passed to our plugin through compiler flags.

Researching the Impact of the Processor Architecture

This thesis has focused on ARMv7-M and the ARMv6-M ISAs. Important questions to raise are: what is the impact of the ISA and what is the impact of the processor architecture on the CFE detection techniques? Other microprocessors, both ARM-produced and others such as the TI MSP430, the Microchip PIC32 or the MIPS32 processors, have other ISAs, use different hardware, have different hardware fault handlers, etc. All these factors might impact how these microprocessors react to CFEs and how CFE detection techniques perform. To answer these questions, both the SWIFI tool and the GCC plugin need to be adapted. The SWIFI tool needs to be expanded, to support these different targets either using real hardware or using simulated hardware. The GCC plugin needs to be adapted to support these extra ISAs. Once adapted, the tools can be used to perform experiments that can determine the impact of a different ISA or a different processor architecture on CFE detection techniques.

Including Data Flow Errors

In a recent paper, which builds on previous work [58], Rhisheekesan et al claim that control flow checking (CFC) techniques actually increase the vulnerability of a system to soft errors:

> Our studies contradict claims by various CFC schemes that they provide significant protection for program execution in processors against soft errors and proves that these schemes make matters worse as they make the programs more susceptible to soft errors. Using our gemV-CFC framework, we estimate the architectural vulnerability of the program "without CFC" and "with CFC." Our experimental results show that the vulnerability of the program "with CFC" is higher than "without CFC" for software-only and hybrid CFC techniques. This can be attributed to the additional vulnerability due to the extra instructions that implement the CFC scheme offsets the small reduction in vulnerability achieved by the CFC technique. Even though the vulnerability remains almost the same for hardware-only CFC techniques, the overheads incurred in design cost, area, and power due to the hardware modifications required for their implementation cannot be hidden. Also the contribution of reduction in vulnerability by the existing CFC techniques is small, since the total number of bits protected by these CFC schemes is only a small fraction of the total number of vulnerable bits in the processor. (Rhisheekesan et al., p. 11:23, [59])

Rhisheekesan et al. base this claim on the fact that CFE detection techniques only protect a limited number of bits, e.g. PC and the instruction register, present in the system and leave the rest of the bits in the system unprotected. Combined with the fact that CFC schemes increase the execution time of the program and therefore increase the probability for erroneous bit-flips to be induced, they conclude that CFE detection techniques actually increase the vulnerability of a system to soft errors. While we agree that CFE detection techniques increase the execution time of the program and thus the probability for an erroneous bit-flip, we do not think it is fair to treat CFC schemes as techniques that were designed to provide protection for all bits present in the system. In fact, we do believe that CFE detection techniques, both established techniques as our own developed techniques, greatly increase the resilience of systems against CFEs as we have shown with our experimental results. As discussed in Chapter 1 and Section 2.1, erroneous bit-flips can manifest themselves as CFEs and DFEs. The work of Rhisheekesan et al. demonstrates that only implementing a CFE detection technique to increase the resilience against all erroneous bit-flips is not enough and that a DFE detection technique is also needed towards that regard.

DFE detection techniques have been studied within our research group, oncurrently to this research. That research track has primarily focused on comparing existing techniques and determining the best locations to insert comparison instructions. With the work of Rhisheekesan et al. in mind, a future direction for this research is to include the researched DFE techniques, both established and developed in-house, in the GCC plugin presented in Section 3.4. Since the researched DFE techniques also have to be implemented in the low-level code of the target program, adding them to the GCC plugin would ease their implementation.

Closely related to that track, an extra research track is to use the gained knowledge to develop a technique that can detect both CFEs and DFEs. A preliminary step taken within this research track was the merger of RACFED and an in-house developed DFE detection technique [99]. While the performed experiments show that it's effective at detecting errors, the merged technique also imposed a lot of overhead. And as described above, overhead actually increases the probability of erroneous bit-flips, therefore further research is needed to determine if more effective implementations are possible to decrease the imposed overhead.

7.2 Summary

This thesis has focused on software-implemented CFE detection techniques. CFEs are unwanted jumps through an executing program caused by erroneous bit-flips at the hardware level, introduced by external disturbances such as high-energy particles or EMI. Software-implemented CFE detection techniques insert extra control variables and control variable update instructions in the target program at compile time. At run time, those update instructions recalculate the control variables and compare the run-time values of those control variables with their expected value determined at compile time. A mismatch between the run-time and compile-time values indicates a CFE has been detected. Depending on what type of CFE the technique is designed to detect, inter-block and/or intra-block CFEs, the control variables are less frequently or more frequently updated.

Software-implemented CFE detection techniques have been around for quite some time, leading to a variety of techniques. In Chapter 2, we explained the working of eight techniques, six designed to detect inter-block CFEs only and two designed to detect both inter-block and intra-block CFEs. Next to explaining their internal operation using high-level instructions, we have also provided a low-level implementation using the ARMv7-M ISA. We provide this low-level implementation because the techniques only achieve their maximum error detection ratio when implemented in the low-level code. However, most literature describing the techniques neglect to provide such a low-level implementation, leaving it up to the possible user of the technique to make the translation himself/herself. By providing an example implementation using a specific low-level language, this thesis should make it easier for others to implement the selected techniques in other low-level languages.

To be able to perform the experiments on all techniques, two supporting tools have been developed: 1) a compiler extension and 2) a fault injection tool which are described in Chapter 3. The compiler extension, which has the form of a GCC plugin, enables to automatically implement the techniques in the low-level code of the target program or function. As mentioned before, most literature describing a CFE detection technique use high-level instructions to explain the working of the technique. However, implementing the technique in a high-level language reduces their error detection ratio. This is because high-level language is not mapped one-to-one onto low-level code, which is executed by the target. This leads to many CFEs remaining undetected. Therefore, the solution is to implement the techniques directly in the low-level code of the target program. To do this successfully, both knowledge about the low-level code of the target embedded system and about the technique are needed. Even then, implementing the technique manually into the low-level code is error-prone. Our

GCC plugin presented in Section 3.4 resolves these issues, as it automatically implements the desired technique in the low-level code of the target program. Currently, the plugin supports all techniques discussed in this thesis, but adding a new technique is simple since it only requires implementing an interface of six functions.

The second developed support tool is our SWIFI-tool, discussed in Section 3.3. This tool contains several CFE injection processes and supports different targets. During the course of this research, three fault injection process were developed: 1) control flow aware injection, 2) random in range injection and 3) extended CFE injection. Each injection process serves a different purpose. The control flow aware injection enables the user to inject inter-block and intra-block CFEs deterministically for one target program function. By building the CFG of that function in memory, the process knows which bit-flip would result in either an inter-block CFE or an intra-block CFE. The random in range injection process injects random CFEs into a part of the program that is specified by the user. This includes both inter-block and intra-block CFEs, but also CFEs that jump between functions if multiple program functions are included within the user-defined range. The third process, extended CFE injection, gradually steps through a user-defined range of the program, and injects all possible CFEs for that range. This increases the code coverage of the injection process. While the first two processes are intended to be executed on a real hardware target, the last process is intended to be executed on a simulated target. The extended CFE injection process requires a lot of time when executed on a real hardware target, so by using it on an instruction set simulator such as the Imperas simulator, the injection can be performed faster.

While a variety of techniques exists, there is no guideline to help select the best technique to use. Therefore, we performed an experimental comparative study between the eight selected techniques. To perform this study, we implemented the techniques for the same eight case studies, executed them on the same hardware and subjected them to the same fault injection campaign. These measurements enabled us to analyze three criteria of the techniques: error detection ratio, execution time overhead and code size overhead. As described in Chapter 4, the study revealed that the techniques could be divided into two groups based on their error detection ratio. One group that performed well, achieving an average error detection ratio of 70 % or more, and a second group that performed poorly, achieving an average error detection ratio of only 54 % or less. The first group performed better because they used gradual updates, using the previous value of the run-time variables to calculate the new values, instead of local updates, using a simple assignment, as in the second group. When considering the imposed overhead, the technique called SEDSR performed the best [31]. It imposes both the lowest execution time overhead and

code size overhead, because it only uses a few and computationally inexpensive instructions. Overall, the technique called CFCSS performed best when both error detection ratio and error detection cost are considered [27]. It achieves a high error detection ratio and imposes a low overhead.

There was, however, room for improvement. Using the data of the study, we derived five guidelines in order to develop an optimal inter-block CFE detection technique:

1. Use gradual updates to assign a new value to the run-time variables. This increases the error detection ratio.
2. Avoid a shared state of the run-time variables. A shared state is a value for the run-time variable that is valid at multiple program points. CFEs that jump between such program points cannot be detected, thus avoiding such a shared state increases the error detection ratio.
3. Take conditional branches into account when updating the run-time variable. When updating the control variables in a basic block that ends with a conditional branch, the value of the control variable should be different for both paths. This enables to detect whether or not the correct path, i.e. the *True* path or the *False* path, was taken.
4. Take exit statements into account by inserting an extra comparison between the run-time and compile-time values. Exit statements are statements that exit out of the current function and return control to the calling function. Since the calling function might not be protected with a CFE detection technique, inserting an extra check for the run-time variables before the exit statement increases the changes to detect such CFEs.
5. Use as few and as computationally inexpensive instructions to update the control variables. This requirement helps to reduce the imposed overhead.

Based on those guidelines, we developed a new inter-block CFE detection technique called RASM. In Chapter 5, we present the compile-time process to implement the technique and provide a low-level example with RASM implemented. By repeating the performed fault injection campaign and measuring the imposed overhead, we compared RASM with CFCSS. The results of that study show that RASM outperforms CFCSS for all three considered criteria. The fault injection campaign shows that RASM has an average error detection ratio of 86 % and an SDC ratio of 6 %. In contrast, CFCSS has an average error detection ratio of 70 % and an SDC ratio of 13 %. Regarding the imposed overhead, RASM imposes the same code size overhead as CFCSS but

imposes a lower execution time overhead. Combined, these results show that following the five guidelines yields a more optimal technique.

Although RASM outperforms the other established techniques, its limitation is that it only detects inter-block CFEs. To detect intra-block CFEs, more frequent updates of the control-variable are needed. Chapter 5 presents how we expanded the RASM technique into the RACFED technique that is able to detect intra-block CFEs, by adding more additions and subtractions of the control-variable. Next, we performed an extensive fault injection campaign and measured the imposed overhead. To place the results in perspective, we compared them to the results of two similar techniques, RSCFC and SIED. Regarding error detection ratio, the proposed RACFED technique outperforms the two similar techniques. On average, RACFED has an error detection ratio of 79 % and an SDC ratio of 4 %. In contrast, RSCFC only achieves an error detection ratio of 39 % and has a high SDC ratio of 20 %, and SIED achieves an error detection ratio of 66 % with an SDC ratio of 9 %. Considering the error detection cost, i.e. imposed execution time overhead and code size overhead, RACFED again outperforms the two similar techniques. When RACFED is implemented, the execution time rises to 1.87 times that of the unprotected code and the code size rises to 1.04 times that of the unprotected code. RSCFC and SIED impose a much higher execution time overhead, as the execution time rises to 2.45 times that of the unprotected code. They also impose a slightly higher code size overhead, as the code size rises to 1.06 times that of the unprotected code. These numbers show that RACFED is a more useful CFE detection technique, as it achieves a higher error detection ratio and imposes a lower error detection cost.

In an effort to further reduce the imposed overhead of RACFED, we propose a selective implementation in Chapter 5. Our proposed approach is to keep all control variable update instructions in all basic blocks, but to only insert the necessary comparison instructions in the exit basic blocks of the program or function. These blocks are included in all paths through the target program or function, making them the ideal location to perform the comparison. The aim of this approach is to keep the error detection ratio as high as possible, while having an impact on the imposed overhead. Once applied to RACFED, an extensive fault injection campaign was conducted and next to imposed overhead, we also measured the error detection latency. When comparing regular RACFED and the selective S-RACFED, small changes in the fault injection results are noted. Where RACFED achieves an error detection ratio of 81 %, S-RACFED achieves a slightly lower error detection ratio of 76 %. This decrease in error detection ratio is coupled with an increase in SDC ratio, as S-RACFED reports a ratio of 7 % compared to the 4 % of RACFED. The data revealed that, when S-RACFED is implemented, more CFEs resulted in an

undetectable premature exit from the program. Regarding imposed overhead, the results were as expected. The average execution time of S-RACFED is 1.66 times that of the unprotected code, while that of RACFED is 1.87 times that of the unprotected code. Only a limited reduction in code size overhead is measured, because only a small part of the code base of the case studies was protected with the technique. Finally, we measured the error detection latency of both techniques. We measured this as the number of instructions that have executed between the point of CFE injection and the point of execution of the error handler. These results clearly show that when RACFED is implemented, the error detection latency is the length of one basic block, ranging between 1 and 30 executed instructions. However, when S-RACFED is implemented the error detection latency rises to the length of the target program for most case studies. This clearly indicates the downside of the selective implementation and makes S-RACFED not suited for a number of applications, e.g., actuator control programs.

This thesis concludes with a chapter on valorization of the research. In Chapter 6, we discuss how we applied this research to two industrial case studies. The first case study being a small scale factory, that processes workpieces. Using three stations, workpieces are pushed out of a stacked magazine and transported to the testing area from which only the correct workpieces are moved to the final station where they are sorted by color. First we performed a manual fault injection campaign on the unprotected code base of the three different stations to determine the effect of CFEs on each of the stations. The two most common types of CFE effects were the program either waiting indefinitely for a non-controlled actuator to reach a new position, or just skipping a part of the program resulting in actions occurring at the wrong time causing the workpiece to be handled incorrectly. Then, we applied RACFED to the code base of each station, using our GCC plugin, and repeated the manual fault injection campaign. During this second fault injection campaign, most CFEs were either detected by RACFED or the hardware fault handlers of the microcontroller. Implementing RACFED thus significantly increased the reliability of each station. As a final step for this small scale factory, we implemented a CFE recovery mechanism that restarts the execution of the station from a program point before the CFE was injected. While crude, this mechanism proved effective for all injected CFEs of the manual fault injection campaign.

The second industrial case study is based on hardware and software from Televic Healthcare NV and focused on two modules communicating with each other over a bus. The case study was used to analyze how CFEs would affect to chosen bus communication software stack. By performing a fault injection campaign on the unprotected code, we now know that approximately 10 % of the injected errors is able to corrupt the bus communication. By applying RACFED, that number

is reduced to 2 %. This reduction in communication error comes at a price of execution time overhead and code size overhead. The execution time increased to $\times 1.7$ that of the unprotected code and the code size even increased to $\times 2.3$ that of the unprotected. These numbers lead us to believe that protecting the entire code base of the final release of the code, which will include an elaborate application layer, is not feasible for the current microcontroller. To that end, a strategy to lower the imposed overhead, by leaving some functions unprotected, was explored. However no real results could be derived, because the current code base is too restricted.

7.2.1 Research Objectives

In Chapter 1, we defined three research objectives to be achieved in this thesis:

1. Conduct a comparative study of SOTA techniques to determine the best technique.
2. Lower the required effort to implement CFE detection techniques.
3. Advance the SOTA by proposing new techniques.

The first objective was met in Chapter 4, in which we present a comparative study of eight SOTA techniques. By implementing the techniques for the same case studies, executing them on the same hardware and subjecting them to the same fault injection campaign, an objective study was performed. This study revealed that the technique called CFCSS is the most suitable SOTA technique to implement, as it has a high error detection ratio and imposes a low overhead.

The second objective is fulfilled throughout this thesis. First, we provided a low-level implementation for the investigated SOTA techniques in Section 2.3. Although our low-level implementation uses the ARMv7-M ISA, it should enable others to easily map it to another ISA. Secondly, we proposed a compiler extension to automatically implement all discussed techniques into the low-level code of the target program, in the form of a GCC plugin in Section 3.4. By making a small adjustment to the source code and the compiler flags of the target program, the selected technique can be implemented. This eliminates the need to know the inner working of a technique and completely eliminates the effort needed to effectively implement the technique in the low-level code. Finally, this objective was met by providing the compile-time processes and low-level implementation of our own techniques in Chapter 5. In this chapter, we present our own CFE techniques focusing on the low-level implementation.

The third and final objective was accomplished by proposing three new CFE detection techniques that advance the SOTA in Chapter 5. First, we proposed our RASM technique which is an inter-block CFE detection technique that outperforms CFCSS. RASM achieves both a higher error detection ratio and a lower imposed overhead than CFCSS. Next, we proposed RACFED, which extends RASM with intra-block CFE detection capabilities. When compared to similar techniques, i.e. RSCFC and SIED, RACFED shows to have a higher error detection ratio and to impose a lower overhead, making it a more suitable technique to implement. Finally, we presented S-RACFED which is a selective implementation of our RACFED technique. With this selective implementation, the imposed overhead is lowered while having a minimal impact on the error detection ratio.

To conclude, we have met all three research objectives and this thesis, therefore, presents a practical approach to software-implemented CFE detection.

Appendix A

Moving to an Application Processor

This section discusses an industrial case study performed in collaboration with Televic Healthcare NV. Due to a non-disclosure agreement, details have been omitted and proprietary systems have been renamed.

The goal of this case study was to use an application processor and to set up an architecture in which a reliable and and unreliable world could execute separately. The unreliable world would execute infotainment and user-interaction applications on top of the Linux operating system. The reliable world would execute the core functionality of the system using bare-metal code and would be the application domain for the GCC plugin and CFE detection techniques. First the envisioned architecture is discussed in more detail, then attempts to create this architecture using TrustZone is presented, to end with other options to develop the envisioned architecture.

A.1 Architecture

As shown in Fig. A.1, the main idea is to have two distinct worlds executing on an ARM Cortex-A7. One of those worlds is called the reliable world and executes bare-metal code, while the other is called the unreliable world and will execute application on top of the Linux operating system. The reliable world is meant to execute the core functionality of the system, in contrast to the unreliable world which will execute applications related to infotainment and

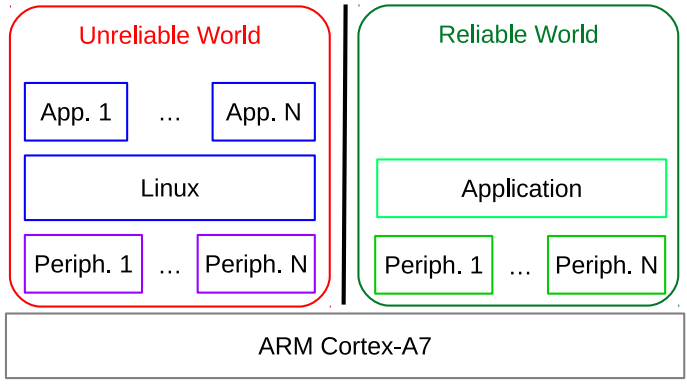


Figure A.1: The envisioned architecture with a separate reliable and unreliable world.

user-interaction. The two worlds are expected to execute separately, whereby the unreliable world can hang or crash without impacting the reliable world. This also means that both worlds have their own assigned peripherals, which are not accessible from another world. If the same peripheral is needed by both worlds, the peripheral will be assigned to the reliable world and the unreliable world will work with a virtual clone of the peripheral which will communicate with the reliable world to access the actual peripheral.

A.2 Using TrustZone

The first idea to build the envisioned architecture was to use the TrustZone capabilities of the Cortex-A7. As explained on the ARM website, TrustZone is a hardware isolation mechanism that creates an isolated secure world which can be used to provide confidentiality and integrity to the system [55]. Using this hardware isolation mechanism, the reliable world could be realized in the secure world and the unreliable world could be realized in the non-secure world. An extra reason to develop the architecture using TrustZone is that microcontrollers, e.g. the Cortex-M33, are now also outfitted with the technology, which could enable to setup the architecture on both application processors and microcontrollers.

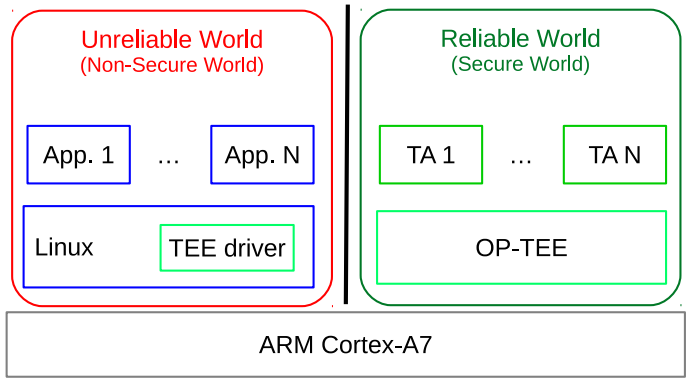


Figure A.2: The envisioned architecture using OP-TEE (simplified view of the OP-TEE components).

A.2.1 Building the Architecture with OPTEE

An attempt to create the desired architecture was performed using OP-TEE as Trusted Execution Environment (TEE) [55, 60]. OP-TEE was selected as it is one of the most used TEEs and enables to build the envisioned architecture as shown in Fig. A.2. Once set up, we built a Trusted Application (TA), i.e. an application executing in the TrustZone, and extended the GCC plugin and added support for the ISA of the Cortex-A7, the ARMv7-A instruction set [61]. Then, we modified the compiler flags of the TA, so the plugin would be used when compiling the principal function of the TA. After compiling, executing the TA and manually verifying the disassembly, we concluded that this first attempt to add support for the ARMv7-A instruction set was successful. This was, however, a very minor verification and further tests using other algorithms and by protecting multiple functions are needed to fully claim that the instruction set is supported. We know from previous experiences that recompiling the used libraries and adjusting linker files to fully support the second stack, to save and restore the run-time variables of the CFE protection technique, is necessary.

The disadvantage of the created setup is that OP-TEE expects the Linux environment, the non-secure world, to be in charge of scheduling the TAs. This in contrast with the envisioned architecture in which both worlds can operate separately. Different approaches were developed to solve this issue, but none of them proved successful, leaving us to believe OP-TEE cannot be made to comply with the desired architecture and other options should be evaluated.

A.2.2 Building the Architecture with LTZvisor

One of the other options to build the architecture is using TrustZone-assisted virtualization, in which the TrustZone is used to create a reliable wold virtual machine and an unreliable world virtual machine. Although not developed for it, research shows that TrustZone can be used for virtualization [62–64]. One framework that supports such virtualization is LTZVisor, a Lightweight TrustZone-assisted hyperVisor:

> LTZVisor is a lightweight TrustZone-assisted hypervisor. It allows the consolidation of two virtual machines (VMs), running each of them in an independent virtual world (secure and non-secure) supported by TrustZone-enabled processors. The secure world is typically used for running a small TCB [Trusted Computing Base] size VM, while the non-secure world is used for running a rich environment.
>
> For the Cortex-A series a typical configuration encompasses running a RTOS as secure VM and a GPOS as non-secure VM. For the new generation Cortex-M microcontrollers it can be used for running a bare metal application or a lightweight RTOS as secure VM side by side with a RTOS or an embedded OS for IoT as non-secure VM. (S. Pinto, Introduction [65])

As shown in Fig. A.3, LTZVisor could be used to setup the envisioned architecture. The Non-Secure VM can be used to create the unreliable world and the Secure VM can be used to develop the reliable world. However, we were not successful to set up the architecture. As the selected platform for this research track is not listed as a supported platform, we had to port LTZVisor to our target. Our attempt to port the framework proved inadequate and due to time constraints, no further research or development was spent on this research track.

A.3 Other Possibilities to build the Desired Architecture

Because none of our attempts to build the desired architecture succeeded while using the TrustZone technology, other options were explored. One option is to use a microprocessor with separate cores, such as an i.MX7. The i.MX7 both has an Cortex-A7 and a Cortex-M4. Both cores can claim there own peripherals and

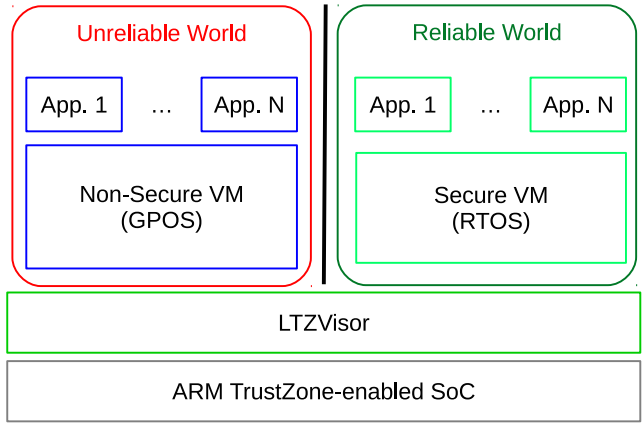


Figure A.3: The envisioned architecture using LTZVisor. Based on Fig. 1 found on p. 4:5 of [63].

can communicate with each other, so the reliable world could be executed on the Cortex-M4 while the unreliable world could be executed on the Cortex-A7. A second option is to use another TEE, which does support the separate execution of the secure and non-secure world. While other TEEs exist, such as Prove&Run ProvenCore, Sequitur Labs CoreTee and Trustonic Kinibi that should support such an architecture, these are not open-source [66–68]. These closed source solutions make it more difficult to quickly setup a proof-of-concept and cannot be protected with the GCC plugin, which could be needed to further enhance the reliability of the system. An open-source alternative is Sierraware SierraTEE, but the selected platform is not listed as supported platform, which makes this a unusable alternative [69]. A third option is to use a hypervisor instead of a TEE and execute the reliable world and unreliable in their own virtual machine. Open-source hypervisors that support the envisioned architecture are jailhouse, Xen ARM and Sierraware SierraVisor [70–72]. Unfortunately, none of these lists the selected platform as a supported platform. Moreover, for Jailhouse and Xen ARM it is not clear what would happen if the unreliable world would crash and reboot, as they both depend on them for their configuration. As the envisioned architecture is an architecture often needed in safety-critical mixed-criticality systems, most closed-source hypervisors are related to that marked with well-known examples such as Sysgo PikeOS, QNX Hypervisor, Green Hills Software Integrity Multivisor and HighIntegritySystems SafeRTOS [73–76]. As these are designed for safety-critical applications, they are often expensive and offer too many features for our use case and therefore were not used. A final option is to develop our own TEE or hypervisor to build the envisioned architecture. This option was not executed due to time constraints.

ARMv7-M Instructions

<instr.>S	An S at the end of an instruction indicates that the instruction updates the system flags
ADD	Addition
AND	Bitwise And
B	Unconditional Branch
BEQ	Branch Equal
BNE	Branch Not Equal
BX	Branch and Exchange
CBZ	Compare and Branch if Zero
CMP	Compare
EOR	Bitwise Exclusive Or
IT <condition>	If <condition> true, Then ...
ITE <condidtion>	If <condition> true, Then ..., or Else ...
ITTEE <condition>	If <condition> true, Then ... and Then ..., or Else ... and Else ...
MOV	Move
MUL	Multiplication
MVN	Bitwise Inverse
SUB	Subtraction

UDIV Unsigned Integer Division

Bibliography

[1] i-SCOOP. Industry 4.0: the fourth industrial revolution - guide to industry 4.0. [Online]. Available: https://www.i-scoop.eu/industry-4-0/

[2] F. Ayatolahi, B. Sangchoolie, R. Johansson, and J. Karlsson, "A study of the impact of single bit-flip and double bit-flip errors on program execution," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2013, pp. 265–276.

[3] R. Baumann, "Soft errors in advanced computer systems," *IEEE Design Test of Computers*, vol. 22, no. 3, pp. 258–266, May 2005.

[4] M. White and Y. Chen, "Scaled cmos technology reliability users guide," National Aeronotics and Space Administration (NASA), Tech. Rep. 20100014217, January 2010. [Online]. Available: https://nepp.nasa.gov/files/16361/08_102_4%20new%20del_White.pdf

[5] J. Abella, F. J. Cazorla, E. Quiñones, A. Grasset, S. Yehia, P. Bonnot, D. Gizopoulos, R. Mariani, and G. Bernat, "Towards improved survivability in safety-critical systems," in *2011 IEEE 17th International On-Line Testing Symposium*, July 2011, pp. 240–245.

[6] N. Kanekawa, E. H. Ibe, T. Suga, and Y. Uematsu, *Dependability in Electronic Systems: Mitigation of Hardware Failures, Soft Errors, and Electro-Magnetic Disturbances*. Springer, 2011. [Online]. Available: https://www.springer.com/gp/book/9781441967145

[7] 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*, April 2011, pp. 3C.3.1–3C.3.6.

[8] 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*. Springer Japan, 2019, ch. Radiation-Induced Soft Errors.

[9] 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)*, 2002, pp. I031–1–I031–5.

[10] K. Kim and A. A. Iliadis, "Critical bit errors in cmos digital inverters due to pulsed electromagnetic interference," in *2007 International Conference on Electromagnetics in Advanced Applications*, Sept 2007, pp. 217–220.

[11] N. A. Estep, J. C. Petrosky, J. W. McClory, Y. Kim, and A. J. Terzuoli, "Electromagnetic interference and ionizing radiation effects on cmos devices," *IEEE Transactions on Plasma Science*, vol. 40, no. 6, pp. 1495–1501, June 2012.

[12] 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)*, April 2012, pp. SE.2.1–SE.2.6.

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

[14] A. Tang, S. Sethumadhavan, and S. Stolfo, "CLKSCREW: Exposing the perils of security-oblivious energy management," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 1057–1074.

[15] M. M. Team, "International roaroad for devices and systems - 2018 update: More moore," IEEE IRDS, Tech. Rep., 2018. [Online]. Available: https://irds.ieee.org/images/files/pdf/2018/2018IRDS_MM.pdf

[16] *Regulation (EU) 2017/745 of the European Parliament and of the council of 5 April 2017 on medical devices*, European Parliament and the Council Std., 2017.

[17] *IEC 62304: Medical device software - Software life cycle processes*, International Electrotechnical Commission (IEC) Std., 2006.

[18] *ISO 26262: Road Vehicles - Functional safety*, International Organisation for Standardization (ISO) Std., 2011.

[19] *EN 50128: Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems*, European Committee for Eletroctechnical Standardization (CENELEC) Std., 2011.

[20] *IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems*, International Electrotechnical Commission (IEC) Std., 2010.

[21] A. Armoush, "Design patterns for safety-critical embedded systems." Ph.D. dissertation, Department of Mathematics, Computer Science and Natural Sciences, RWTH Aachen University, 2010.

[22] T. Claeys, J. Catrysse, D. Pissoort, and Y. Arien, "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)*, Aug 2018, pp. 725–729.

[23] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, *Software-implemented hardware fault tolerance*. Springer Science & Business Media, 2006.

[24] A. Abdi, S. A. Asghari, S. Pourmozaffari, H. Taheri, and H. Pedram, "An effective software implemented data error detection method in real time systems," in *Advances in Computer Science, Engineering & Applications*. Springer, 2012, pp. 919–926.

[25] P. Lakhani, "A review: software fault tolerance and their techniques," *IRJMST*, vol. 5, no. 12, pp. 8–15, 2014.

[26] G. Nazarian, D. G. Rodrigues, A. Moreira, L. Carro, and G. N. Gaydadjiev, "Bit-flip aware control-flow error detection," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 2015, pp. 215–221.

[27] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE transactions on Reliability*, vol. 51, no. 1, pp. 111–122, 2002.

[28] 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*. IEEE, 2003, pp. 581–588.

[29] 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*, vol. 10, no. 6, pp. 627–641, 1999.

[30] A. LI and B. Hong, "Software implemented transient fault detection in space computer," *Aerospace science and technology*, vol. 11, no. 2, pp. 245–252, 2007.

[31] 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*, vol. 5, no. 09, pp. 664–670, 2012.

[32] 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*, vol. 10, no. 1, pp. 481–490, 2014.

[33] 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.* IEEE, 2003, pp. 589–596.

[34] (2017, April) Embedded markets study: Integrating iot and advanced technology design, application development and processing environments. AspenCore. [Online]. Available: https://m.eet.com/media/1246048/2017-embedded-market-study.pdf

[35] ARM cortex-m3 introduction. ARM University Relations. Accessed on 18/07/2019. [Online]. Available: https://www.arm.com/files/pdf/CortexM3_Uni_Intro.pdf

[36] J. Van Waes, J. Lannoo, A. Degraeve, D. Vanoost, D. Pissoort, and J. Boydens, "Effectiveness of cyclic redundancy checks under harsh electromagnetic disturbances," in *Proc. Int. Symp. Electromagnetic Compatibility - EMC EUROPE*, Sep. 2017.

[37] (2006, February) The simple CPU project: Architecture description. disk91. Accessed on 18/07/2019. [Online]. Available: https://www.simple-cpu.com/cpu-instruction-set-architecture-en.php

[38] *ARMv7-M Architecture Reference Manual*, Ddi 0403e.b ed., ARM, 110 Fulbourn Road Cambridge, England CB1 9NJs, Dec. 2014.

[39] X. Guang and Z. Zhang, *Linear Network Error Correction Coding*, 1st ed., ser. SpringerBriefs in Computer Science. Springer-Verlag New York, 2014.

[40] R. Zhu and Y. Ma, *Information Engineering and Applications*, 1st ed., ser. Lecture Notes in Electrical Engineering. Spinger-Verlag London, 2012, vol. 154.

[41] H. König, *Protocol Engineering.* Springer-Verlag Berlin Heidelberg, 2012.

[42] R. Dorf, *Systems, Controls, Embedded Systems, Energy, and Machines*, ser. The Electrical Engineering Handbook. CRC Press, 2016.

[43] S. C. Satapathy, V. Bhateja, and A. Joshi, *Proceedings of the International Conference on Data Engineering and Communication Technology*, 1st ed., ser. Advances in Intelligent Systems and Computing. Springer Singapore, 2017, vol. 469.

[44] W. Fischer, *Digital Television: a Practical Guide for Engineers.* Springer-Verlag Berlin Heidelberg, 2004.

[45] R. C. Gonzalez and R. E. Woods, *Digital Image Processing.* Pearson, 2017.

[46] 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.* IEEE, 2001, pp. 3–14.

[47] H. Schirmeier, C. Borchert, and O. Spinczyk, "Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2015, pp. 319–330.

[48] Imperas. (2018) Revolutionizing embedded software development. Online. [Online]. Available: http://www.imperas.com/

[49] (2014) Embedded market study: Then, now, what's next? UBM Tech Electronics. [Online]. Available: https://cms.edn.com/ContentEETimes/Documents/Embedded.com/MarketStudy/2014-embedded-market-study-then-now-whats-next.pdf

[50] (2018) Open source Python library for programming and debugging ARM Cortex-M microcontrollers using CMSIS-DAP. Mbedmicro. [Online]. Available: https://github.com/mbedmicro/pyOCD

[51] D. Rath, "Design and implementation of an on-chip debug solution for embedded target systems based on the ARM7and ARM9 family." Master's thesis, University of Applied Sciences Augsburg, 2005. [Online]. Available: http://openocd.org/files/thesis.pdf

[52] E. Chielle, G. S. Rodrigues, F. L. Kastensmidt, S. Cuenca-Asensi, L. A. Tambara, P. Rech, and H. Quinn, "S-SETA: Selective software-only error-detection technique using assertions," *IEEE Transactions on Nuclear Science*, vol. 62, no. 6, pp. 3088–3095, Dec 2015.

[53] D. S. Khudia and S. Mahlke, "Low cost control flow protection using abstract control signatures," *SIGPLAN Not.*, vol. 48, no. 5, pp. 3–12, Jun. 2013.

[54] Festo-Didactic. Mps the modular production system. [Online]. Available: http://www.festo-didactic.com/int-en/learning-systems/mps-the-modular-production-system/stations/

[55] (2019, May) Introducing arm trustzone. ARM. [Online]. Available: https://developer.arm.com/ip-products/security-ip/trustzone

[56] J. Cilleruelo, "Infinite sidon sequences," *Advances in Mathematics*, vol. 255, pp. 474 – 486, 2014.

[57] SCALEXIO: Modular real-time system. dSPACE. Accessed on 18 July 2019. [Online]. Available: https://www.dspace.com/en/inc/home/products/hw/simulator_hardware/scalexio.cfm

[58] A. Shrivastava, A. Rhisheekesan, R. Jeyapaul, and C.-J. Wu, "Quantitative analysis of control flow checking mechanisms for soft errors," in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14. New York, NY, USA: ACM, 2014, pp. 13:1–13:6.

[59] A. Rhisheekesan, R. Jeyapaul, and A. Shrivastava, "Control flow checking or not? (for soft errors)," *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 1, pp. 11:1–11:25, Feb. 2019.

[60] (2019) Op-tee documentation. Linaro. [Online]. Available: https://optee.readthedocs.io/index.html

[61] *ARMv7-A and ARMv7-R Architecture Reference Manual*, Ddi 0406c.d ed., ARM, 110 Fulbourn Road Cambridge, England CB1 9NJs, March 2018. [Online]. Available: https://static.docs.arm.com/ddi0406/cd/DDI0406C_d_armv7ar_arm.pdf

[62] S. Pinto, D. Oliveira, J. Pereira, N. Cardoso, M. Ekpanyapong, J. Cabral, and A. Tavares, "Towards a lightweight embedded virtualization architecture exploiting arm trustzone," in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, Sep. 2014, pp. 1–4.

[63] S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral, "LTZVisor: TrustZone is the Key," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), M. Bertogna, Ed., vol. 76. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 4:1–4:22.

[64] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM Comput. Surv.*, vol. 51, no. 6, pp. 130:1–130:36, Jan. 2019.

[65] S. Pinto. (2017, October) Ltzvisor: a lightweight trustzone-assisted hypervisor. [Online]. Available: https://github.com/tzvisor/ltzvisor

[66] (2019, May) Provencore. Preve&Run. [Online]. Available: https://www.provenrun.com/products/provencore/

[67] (2019, May) CoreTEE - foundational security for IoT systems. Sequitur Labs. [Online]. Available: https://www.sequiturlabs.com/coretee/

[68] (2019, May) Mobile device security is hard - trustonic makes it easy. Trustonic. [Online]. Available: https://www.trustonic.com/solutions/trustonic-secured-platforms-tsp/

[69] (2019, May) Sierratee trusted execution environment. Sierraware. [Online]. Available: https://www.sierraware.com/open-source-ARM-TrustZone.html

[70] (2019, May) Jailhouse. Siemens. [Online]. Available: https://github.com/siemens/jailhouse

[71] (2019, May) Xen ARM with virtualization exensions whitepaper. XenProject. [Online]. Available: https://wiki.xenproject.org/wiki/Xen_ARM_with_Virtualization_Extensions_whitepaper

[72] (2019, May) Sierravisor hypervisor development toolkit. Sierraware. [Online]. Available: https://www.sierraware.com/arm_hypervisor.html

[73] (2019, May) Pikeos hypervisor. Sysgo. [Online]. Available: https://www.sysgo.com/products/pikeos-hypervisor/

[74] (2019, May) Qnx hypervisor. QNX. [Online]. Available: https://blackberry.qnx.com/en/products/hypervisor/index

[75] (2019, May) Integrity multivisor secure virtualization. Green Hills Software. [Online]. Available: https://www.ghs.com/products/rtos/integrity_virtualization.html

[76] (2019, May) Pre-certified safety rtos. HighIntegritySystems. [Online]. Available: https://www.highintegritysystems.com/safertos/

My Publications

[77] J. Vankeirsbilck, V. B. Thati, J. Van Waes, H. Hallez, and J. Boydens, "Control flow aware software-implemented fault injection for embedded CPUs," in *IEEE XXVI International Scientific Conference Electronics (ET)*, Sep. 2017, pp. 1–4.

[78] J. Vankeirsbilck, T. Cauwelier, J. Van Waes, H. Hallez, and J. Boydens, "Software-implemented fault injection for physical and simulated embedded CPUs," in *IEEE XXVII International Scientific Conference Electronics (ET)*, Sep. 2018, pp. 1–4.

[79] J. Vankeirsbilck, J. Van Waes, H. Hallez, and J. Boydens, "A new approach to selectively implement control flow error detection techniques," in *Advances on P2P, Parallel, Grid, Cloud and Internet Computing*, L. Barolli, P. Hellinckx, and J. Natwichai, Eds. Cham: Springer International Publishing, Nov. 2019, pp. 704–715.

[80] J. Vankeirsbilck, H. Hallez, and J. Boydens, "Automatic implementation of control flow error detection techniques," in *accepted at IASED International Conference on Wireless Networks and Embedded Systems (ICWNES)*, July 2019.

[81] J. Vankeirsbilck, J. Van Waes, H. Hallez, and J. Boydens, "Automated regresstion testing of a GCC toolchain used on embedded CPU programs," in *IEEE XXVIII International Scientific Conference Electronics (ET)*, Sep. 2019, pp. 1–4.

[82] J. Vankeirsbilck, V. B. Thati, H. Hallez, and J. Boydens, "Inter-block jump detection techniques: a study," in *IEEE XXV International Scientific Conference Electronics (ET)*, Sep. 2016, pp. 286–289.

[83] J. Vankeirsbilck, N. Penneman, H. Hallez, and J. Boydens, "Random additive signature monitoring for control flow error detection," *IEEE Transactions on Reliability*, vol. 66, no. 4, pp. 1178–1192, Dec 2017.

[84] J. Vankeirsbilck, N. Penneman, H. Hallez, and J. Boydens, "Random additive control flow error detection," in *Computer Safety, Reliability, and Security*, B. Gallina, A. Skavhaug, and F. Bitsch, Eds. Cham: Springer International Publishing, 2018, pp. 220–234.

[85] J. Vankeirsbilck, J. Van Waes, H. Hallez, D. Pissoort, and J. Boydens, "Control flow errors in an industry 4.0 setup: a preliminary study," in *IEEE International Conference on Systems, Man and Cybernetics (SMC)*, Oct 2019, pp. 2305–2310.

[86] J. Vankeirsbilck, H. Hallez, and J. Boydens, "Soft error protection in safety critical embedded applications: an overview," in *2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*. IEEE, 2015, pp. 605–610.

[87] J. Vankeirsbilck, H. Hallez, and J. Boydens, "Integration of soft errors in functional safety: a conceptual study," in *Annual Journal of Electronics*, 2015, pp. 108–111.

Co-Authored Publications

[88] J. Van Waes, **J. Vankeirsbilck**, J. Lannoo, D. Pissoort, and J. Boydens, "Effectiveness of data triplication in harsh electromagnetic environments," in *IEEE International Symposium on Electromagnetic Compatibility*, August 2018, pp. 266–270.

[89] J. Van Waes, J. Lannoo, **J. Vankeirsbilck**, A. Degraeve, J. Peuteman, D. Vanoost, D. Pissoort, and J. Boydens, "Effectiveness of hamming single error correction codes under harsh electromagnetic disturbances," in *IEEE International Symposium on Electromagnetic Compatibility*, August 2018, pp. 271–276.

[90] J. Van Waes, J. Lannoo, **J. Vankeirsbilck**, D. Vanoost, D. Pissoort, and J. Boydens, "Resilience of error correction codes against harsh electromagnetic disturbances: Fault mechanisms," *IEEE Transactions on Electromagnetic Compatibility*, 2019.

[91] J. Van Waes, J. Lannoo, **J. Vankeirsbilck**, D. Vanoost, D. Pissoort, and J. Boydens, "Resilience of error correction codes against harsh electromagnetic disturbances: Fault elimination for triplication-based error correction codes," *IEEE Transactions on Electromagnetic Compatibility*, 2019, accepted Oct 17, 2019.

[92] J. Van Waes, **J. Vankeirsbilck**, J. Lannoo, D. Vanoost, D. Pissoort, and J. Boydens, "Complementary fault models for assessing the effectiveness of hamming codes," in *IEEE XXVI International Scientific Conference Electronics (ET)*, Sep. 2019.

[93] J. Van Waes, D. Vanoost, **J. Vankeirsbilck**, J. Lannoo, D. Pissoort, and J. Boydens, "Resilience of duplication error detection codes against harsh electromagnetic disturbances," *IEEE Transactions on Electromagnetic Compatibility*, 2019, submitted.

[94] V. B. Thati, **J. Vankeirsbilck**, and J. Boydens, "Comparative study on data error detection techniques in embedded systems," in *IEEE XXV International Scientific Conference Electronics (ET)*, Sep. 2016, pp. 282–285.

[95] V. B. Thati, **J. Vankeirsbilck**, J. Boydens, and D. Pissoort, "Data error detection and recovery in embedded systems: a literature review," *Advances in Science, Technology and Engineering Systems Journal*, vol. 2, no. 3, pp. 623–633, 2017.

[96] V. B. Thati, **J. Vankeirsbilck**, J. Boydens, D. Pissoort, and N. Penneman, "An improved data error detection technique for dependable embedded software," in *IEEE 23RD Pacific Rim International Symposium on Dependable Computing (PRDC)*, vol. 2018-December. IEEE, 2018, pp. 213–220.

[97] V. B. Thati, **J. Vankeirsbilck**, N. Penneman, D. Pissoort, and J. Boydens, "CDFEDT—comparison of data flow error detection techniques in embedded systems: an empirical study." ACM; New York, NY, USA, 2018, pp. 23:1–23:9.

[98] V. B. Thati, **J. Vankeirsbilck**, D. Pissoort, and J. Boydens, "Instruction level duplication and comparison for data error detection: a first experiment," in *IEEE XXVII International Scientific Conference Electronics (ET)*, Sep. 2018, pp. 1–4.

[99] V. B. Thati, **J. Vankeirsbilck**, D. Pissoort, and J. Boydens, "Hybrid technique for soft error detection in dependable embedded software: a first experiment," in *accepted at IEEE XXVIII International Scientific Conference Electronics (ET)*, Sep. 2019.

[100] P. Cordemans, N. De Witte, **J. Vankeirsbilck**, W. Melis, and J. Boydens, "Isolating real-time from processor-intensive processes in embedded multi-core systems," in *Annual Journal of Electronics*, vol. 9. Technical University of Sofia, 2015, pp. 104–107.

[101] K. T'Jonck, H. Strauven, **J. Vankeirsbilck**, H. Hallez, and J. Boydens, "Design and implementation of an unobtrusive sensor system to support incontinence care of elderly in nursing homes," in *accepted at IEEE XXVI International Scientific Conference Electronics (ET)*, Sep. 2019.

[102] J. Van Waes, **J. Vankeirsbilck**, J. Lannoo, D. Pissoort, and J. Boydens, "Effectiveness of hamming SEC/DED code in harsh electromagnetic environments," in *presented as poster at International Conference on Computer safety, Reliability and Security (SAFECOMP)*, Sep. 2018. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01878607/document

[103] J. Van Waes, **J. Vankeirsbilck**, D. Pissoort, and J. Boydens, "Electromagnetic interference in the internet of things: an automotive insight," in *IEEE XXVI International Scientific Conference Electronics (ET)*, Sep. 2017, pp. 89–92.

[104] J. Van Waes, **J. Vankeirsbilck**, D. Pissoort, and J. Boydens, "Functional safety standard's techniques and measures in light of electromagnetic interference," in *IEEE XXVI International Scientific Conference Electronics (ET)*, Sep. 2017, pp. 85–88.

Curriculum

Personal Information

- **name:** Jens Vankeirsbilck
- **address:** Nieuwe Veldstraat 9, 8810 Lichtervelde, Belgium
- **mobile:** +32 487 613 503
- **email:** jens.vankeirsbilck@kuleuven.be | jens.vankeirsbilck@hotmail.com
- **date of birth:** October 21st 1992

Education

- 10/2014 - current: PhD in Engineering Technology, Computer Science, KU Leuven, Bruges Campus, Belgium
- 09/2013 - 07/2014: Master of Science in Engineering Technology, Electronics-ICT, option ICT, KU Leuven, Technology Campus Ostend, Belgium
- 09/2010 - 07/2013: Bachelor of Science in Engineering Technology, Electronics-ICT, option ICT, Koninklijke Hogeschool Brugge Oostende, Belgium
- 09/2004 - 07/2010: Secondary Education, Latin-Science, Sint-Jozefsinstituut College, Lichtervelde - Torhout, Belgium

Experience

- 10/2014 - current: PhD researcher @ KU Leuven Bruges Campus, Bruges

Training

- Knowledge transfer: capturing and transferring tangible and intangible knowledge in your organization
- Secure your hardware and application
- Software Reliability
- Summer Schools on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems, with different training sessions on:
  - Intellectual Property Strategy
  - Operating Systems for Secure and Safe Embedded Systems
  - Compilation
  - Memory Systems Resilience

FACULTY OF ENGINEERING TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE
M-GROUP
Spoorwegstraat 12, bus 7923
B-8200 Brugge
jens.vankeirsbilck@kuleuven.be
https://iiw.kuleuven.be/brugge/m-group/