DIPLOMARBEIT

Safety-Betrachtungen unter Verwendung kommerzieller Hardware

eingereicht bei der Fakultät für Elektrotechnik der
Technischen Universität Wien,
ausgeführt zur Erlangung des akademischen Grades
eines Diplom-Ingenieurs unter der Leitung von

O.Univ.Prof. Dipl.-Ing. Dr.techn. Dietmar Dietrich
Univ.Ass. Dipl.-Ing.(FH) Dr.techn. Heimo Zeilinger

am

Institut für Computertechnik (E384)

durch

Bernd Thiemann, BSc.
Matr.Nr. 0627763
Müllnergasse 31/14, 1090 Wien

Wien, am 12.05.2014 _____

Kurzfassung

Die Arbeit beschäftigt sich mit neuen Lösungen von Sicherheitssystemen für industrielle Anwendungen. Durch die immer umfangreicher werdende Automatisierung von industriellen Prozessen werden Regelungsaufgaben durch Mikrocontroller oder Industrie-PCs ersetzt. Es gilt zu garantieren, dass der PC durch eine Fehlfunktion keine Gefährdung für anwesende Menschen oder Material darstellt. Aktuell am Markt befindliche Sicherungssysteme garantieren diese Fehlerbeherrschung durch speziell entwickelte Software und Hardware. Die Hardware muss auf Fehlerfreiheit, bzw. Verhalten bei einem Fehler analysiert werden. Bei State of the Art CPUs ist dies aufgrund der Komplexität nicht mehr möglich. Der hier verfolge Ansatz beruht auf der Idee, die Software dahingehend zu erweitern, dass Hardwarefehler zuverlässig erkannt werden können. Durch die Fehleraufdeckung in Software soll zusätzlich eine größtmögliche Hardwareunabhängigkeit erreicht werden. Das geplante Konzept sieht Informationsredundanz vor. In der Software werden während der Ausführung statische Informationen mit den dynamischen des zu regelnden Prozesses kombiniert. Dieses Konstrukt zur Fehleraufdeckung in Software wird „Coded Processing“ genannt und wird im Eisenbahnbereich und in der industriellen Automatisierung erst spärlich eingesetzt. Eine Evaluierung wird durchgeführt, um zu prüfen, ob eine hardwareunabhängige Lösung anhand von Coded Processing möglich ist und diese den Standards für funktionelle Sicherheit genügt. Die theoretische Untersuchung wird mittels Fehlermodellen durchgeführt. Für die praktische Analyse wird eine Codierung inklusive Beispielapplikationen implementiert und Fehler während der Ausführung injiziert, anhand derer die Fehleraufdeckungswahrscheinlichkeit bestimmt wird. Basierend auf diesen Daten wird eine Abschätzung getroffen werden, ob eine Zertifizierung nach Safety Integrity Level (SIL) 2/3 laut International Electrotechnical Commission (IEC) 61508 möglich ist.

Abstract

In this thesis new solutions for safety controllers for industrial applications will be analyzed. Due to the increasing use of automation in industrial processes microcontrollers will be replaced by industrial PCs. In this context it is important, that an error in the computer does not cause any risk to people or material nearby. The error behavior of state-of-the art safety controllers is guaranteed through specialized hardware and software. The hardware has to be analyzed in detail to evaluate the absence of errors respectively the erroneous behavior. With state-of-the art processors this is not possible any more. The complexity of the internal structures has become overwhelming. The approach in this thesis is based on the idea to extend the software with special routines, so hardware errors can be detected with a determined probability. Due to the error detection in software, hardware independence should be achieved. The intended design is based on information redundancy in which the dynamic data of the industrial process are combined with static, predetermined information. This concept is called "coded processing" and is sparely used in railway applications and industrial automation. In the end an evaluation is made, if a hardware independent safety controller including coded processing is possible. The theoretical analysis is done by different fault models. For the practical analysis an example application will be implemented, which uses coded processing. During the execution of this application typical hardware faults are be injected to observe the fault reaction. Based on this data an estimation is done, if a certification according to SIL 2/3 laut IEC 61508 would be possible.

Inhaltsverzeichnis

Abkürzungen

ABS Antiblockiersystem.

ADC Analog-Digital Converter.

ALU Arithmetical Logical Unit.

BB Basic Blocks.

CCF Common Cause Failure.

COTS Commercial off-the-shelf.

CPU Central Processing Unit.

CRC Cyclic Redundancy Check.

DC Diagnostic Coverage.

DMA Direct Memory Access.

DRAM Dynamic Random Access Memory.

ECC Error Correcting Code.

EN Europäische Norm.

FIT Failure in Time.

FMEA Failure Mode and Effects Analysis.

FPGA Field Programmable Gate Array.

GCC GNU Compiler Collection.

GPOS General Purpose Operating System.

HFT Hardwarefehlertoleranz.

I^2C Inter-Integrated Circuit.

IEC International Electrotechnical Commission.

IP Intellectual Property.

LLVM Low Level Virtual Machine.

MMU Memory Management Unit.

PC Personal Computer.

PCIe Peripheral Component Interconnect Express.

PFH Propability of Failure per Hour.

PL Performance Level.

RAM Random Access Memory.

SDC Silent Data Corruption.

SEU Single Event Upset.

SFF Safe Failure Fraction.

SIL Safety Integrity Level.

SIMD Single Instruction, Multiple Data.

SMBus System Management Bus.

SPS Speicher Programmierbare Steuerung.

SRAM Static Random-Access Memory.

SSE Streaming Single Instruction, Multiple Data (SIMD) Extensions.

TÜV Technischer Überwachungsverein.

TMR Triple Modular Redundancy.

1 Einleitung

Die Arbeit beschäftigt sich mit der Analyse neuer Ansätze zur Realisierung sicherheitstechnischer Systeme im Bereich der industriellen Automation. In den letzten Jahren ging der Trend bei Sicherheitssystemen für funktionale Sicherheit weg von der hart verdrahteten Not-Aus-Relais Technik hin zu Softwarelösungen. Diese sind deutlich einfacher bei der Installation und flexibler im Betrieb. Das Verhalten des Systems im Notfall wird durch ein sicherheitskritisches Programm festgelegt. Eine Umstrukturierung des Sicherheitskonzepts bedarf keiner Neuverkabelung mehr, sondern eines Software-Updates. Durch die verringerte Anpassungszeit und das feingranulare Reagieren auf Sicherheitsrisiken ist es möglich geworden industrielle Maschinen benutzerfreundlicher zu gestalten.

Die Aufgabe eines Sicherheitssystems ist es, einen industriellen Prozess, unabhängig von der Art der Gefahr, die davon ausgeht, zu sichern. Die Gefahr kann ein chemischer Prozess, wie zum Beispiel einen Kessel, der bei Überdruck bersten kann, oder eine bewegte, mechanische Einheit, wie einem Roboter, darstellen. Unabhängig von der Anwendung muss das Sicherheitssystem die Prozess- und Umgebungsparameter überwachen und im Falle einer Abweichung vom Normbetrieb einen sicheren Zustand anfahren. Um dies garantieren zu können, muss das Sicherheitssystem selbst sicher sein, um nicht durch eine Fehlfunktion eine Gefahr auszulösen oder eine drohende Gefahr zu übersehen.

Denkt man an industrielle Fertigungen, ist ein Arbeiten zwischen Mensch und Roboter Hand in Hand derzeit trotz der ausgeklügelter Sicherheitssysteme nicht möglich. Für den sicheren Betrieb von Robotern sind diese durch diverse Schutzmechanismen, wie Lichtgitter oder Käfige, von der Umgebung getrennt. Soll ein sicherer Betrieb gewährleistet sein, muss die Reaktionszeit der Sicherheitssysteme weiter gesenkt werden. Eine Möglichkeit wäre, Steuerung und Sicherheitssystem, welche bisher getrennt voneinander arbeiteten, zu verschmelzen. Da die Steuerung vor allem bei dynamischen Roboterbewegungen sehr rechenintensiv ist, muss diese auf rechenstarken Computersystemen durchgeführt werden. Bei Sicherungssystemen hingegen liegt der Schwerpunkt vermehrt auf Eigensicherheit als auf Performance, ein Fehler in Hardware oder Software darf idealerweise keinesfalls zu einem Sicherheitsrisiko führen. Dies wird erreicht, indem ältere, betriebsbewährte Hardware verwendet wird, welche durch penible Analyse auf weitgehende Fehlerfreiheit geprüft wird.

Die somit entstehenden Anforderungen an die Steuerung - hohe Performance - und an das Sicherheitssystem - nachweisbare Fehlerfreiheit - scheinen sich auf den ersten Blick auszuschließen.

In dieser Arbeit wird ein Konzept entwickelt, welches diesen, auf den ersten Blick unvereinbaren Ausschluss auflöst. Da eine Immigration des rechenintensiven Steuerungsprogramms nicht auf die vergleichsweise langsame, sichere Hardware verlagert werden kann, muss die umgekehrte Lösung in Betracht gezogen werden. Das Sicherheitsprogramm muss auf die rechenstarke Hardware portiert werden und trotzdem den gewünschten Sicherheitsstandards genügen. Dies stellt einen neuen Schritt in der Entwicklung sicherer Systeme dar, welcher nur durch eine neue Art von Sicherungsmethoden bewerkstelligt werden kann. Eine Umsetzung davon ist das sogenannte "Coded Processing". Obwohl das mathematische Prinzip dahinter seit den 1960iger Jahren bekannt ist, wurde es in diesem Umfeld und vor allem Umfang noch nie eingesetzt. Damit bestünde die Möglichkeit fehlererkennende Software zu schreiben, die unabhängig von der verwendeten Hardware arbeitet. Folglich muss es möglich sein, auch bei moderner, ungeprüfter Hardware eine Fehlererkennungsrate, welche gleich oder besser als die geprüfte Hardware ist, zu erreichen. Ist die Hardware von Sicherheits- und Steuerungsfunktion erst verschmolzen, kann auch die Software der beiden Aufgaben vereint werden. Gegenüber dem bisherigen System, bestehend aus einer aktiven Steuerung und einer davon unabhängigen Überwachungseinrichtung, kann die Reaktionszeit des Gesamtsystems verbessert werden. Zusätzlich muss nicht wie üblich im Fehlerfall ein Nothalt ausgeführt werden, sondern es kann eine gezielte fehler- bzw. risikomindernde Aktion ausgeführt werden.
Damit das System auch verwendet werden darf, muss es gewissen Normen für die funktionale Sicherheit erfüllen. Die wichtigste Norm in diesem Gebiet ist die IEC 61508 [IEC10a] und sie stellt die Basis für weitere Normen dar. In diesem Zusammenhang soll untersucht werden, welche Voraussetzungen ein Sicherheitssystem nach der Norm erfüllen muss und welches SIL erreicht werden kann.

Ziel dieser Arbeit ist das Entwickeln und Evaluieren eines Konzeptes, welches formal den Anforderungen von SIL 2/3 entspricht. Die Sicherheit soll rein von der Software abhängen, wodurch eine einfache Austauschbarkeit der Hardware erreicht wird. Als Hardware können aktuelle Commercial off-the-shelf (COTS) Komponenten angenommen werden, welche über hohe Rechenleistung und ausreichend Speicher verfügen. COTS Hardware besteht aus komplexen Elementen, welche zusätzlich noch Intellectual Property sind, wodurch die inneren Strukturen als nicht bekannt angenommen werden müssen. Systematische Fehler innerhalb der Hardware müssen als vorhanden, aber unbekannt behandelt werden. Die Softwarelösung muss diese und weitere Fehler erkennen und im Falle des Auftretens einen sicheren Zustand einnehmen. Für die Anwendbarkeit in industriellen Prozessen muss die Reaktionszeit beziehungsweise die Fehleraufdeckungszeit im Millisekundenbereich liegen.
Zur Evaluierung dieser Forderungen wird eine Mathematikbibliothek nach den Gestaltungsgrundsätzen des „Coded Processing“ implementiert. Dem Anwender soll es möglich sein mit wenigen Anpassungen am Applikationsprogramm die Berechnung codiert durchzuführen. Anhand dieser Mathematikbibliothek werden Laufzeitvergleiche zwischen codierter und nativer Ausführung durchgeführt, um Fehleraufdeckungszeit im Millisekundenbereich nachzuweisen. Für die Überprüfung, ob die in den Normen gestellten Anforderungen erfüllt werden, wird eine quantitative und qualitative Analyse durchgeführt. Einerseits wird das entwickelte Konzept den Gestaltungsleitsätzen der Normen gegenübergestellt, andererseits soll eine Simulation inklusive Fehlerinjektion nachweisen, dass das Konzept den quantitativen Forderungen genügt.

Am Beginn der Arbeit steht eine steht eine Technologieanalyse. Diese beinhaltet einen Überblick über den Aufbau aktueller sicherheitstechnischer Komponenten und deren Argumentation um das nötige Sicherheitslevel zu erreichen. Des weiteren beinhaltet sie eine Beschreibung von Coded

Processing und deren Codierungsverfahren. Es werden die Eigenschaften verschiedener Codierungen untersucht und gegeneinander verglichen. Ebenso ist eine Recherche über die zu beachtenden Normen, mit Schwerpunkt auf der IEC 61508 [IEC10a], welche die Vorschriften für die Safety Integrity Levels enthält, durchzuführen. Zusätzlich muss die Normen EN 13849 [EN08] behandelt werden, da Konformität zu dieser Norm oft gefordert wird. Die Erkenntnisse dieses Arbeitsschritts sind in Kapitel 2 zusammengefasst. Darauffolgend wird anhand dieser Informationen ein Konzept erstellt, welches die Anforderungen nach SIL 2/3 bei größtmöglicher Hardwareunabhängigkeit erreichen soll. Eine ausführliche Beschreibung des erarbeiteten Konzepts ist in Kapitel 3 angeführt. Eine Analyse des Konzepts wird auf der Basis von Fehlermodellen aus der Wissenschaft und der IEC 61508 durchgeführt und wird in Kapitel 4 diskutiert. Für die praktische Untersuchung wird eine Mathematik-Bibliothek erstellt, welche die Berechnung von zwei Beispielapplikationen codiert durchführt. Während der Ausführung werden Fehler durch eine spezielle Software injiziert, wie sie typisch bei Computersystemen auftreten. Details zu der Bibliothek, der Applikationen und der Fehlerinjektion ist dem Kapitel 5 zu entnehmen. Die entstandenen Daten der codierten Berechnung und der Fehlerinjektion werden unter Berücksichtigung des entwickelten Konzepts betrachtet und somit eine Abschätzung der Umsetzbarkeit durchgeführt. Dies ist in Kapitel 6 nachzulesen.

2 Technologieanalyse

Dieses Kapitel gibt einen Überblick über die technischen Grundlagen von aktuellen Sicherheitssystemen. Bevor jedoch auf die technischen Details aktueller Sicherheitslösungen eingegangen wird, werden die in diesem Segment wichtigen Normen IEC 61508 [IEC10a] und Europäische Norm (EN) 13849 [EN08] diskutiert. In diesem Zusammenhang werden die wichtigsten Termini, welche für die weitere Arbeit wichtig sind, erläutert. Darauf folgt eine Vorstellung der häufigsten High-Level Architekturen von Sicherheitssystemen und die Grundlagen zu „Coded Processing“.

2.1 Normen und Begriffe

Die Normen stellen ein Regelwerk zusammen, mit welchem der Hersteller der Produkte beweisen kann, dass sie Stand der Technik sind. Produkte, welche nicht dem Stand der Technik entsprechen, können nicht zertifiziert werden [Rei12, S. 253]. Die Haftung im Falle eines Versagens läge somit beim Hersteller. Bei zertifizierten Produkten trifft den Hersteller aufgrund der strengen Regeln beim Entwurf und der Fertigung keine Schuld. Hersteller sicherheitskritischer Elemente sind somit auf eine Zertifizierung ihrer Produkte durch den Technischer Überwachungsverein (TÜV) angewiesen.

Dieser Abschnitt gibt einen Überblick über die relevanten Standards und deren Anforderung um die funktionale Sicherheit zu garantieren.

2.1.1 Begriffe und Definitionen

Das Sicherheitssystem, oft auch Sicherheitssteuerung genannt, ist einer Speicher Programmierbaren Steuerung (SPS) nicht unähnlich. Doch im Vergleich zu einer SPS ist der Aufgabenbereich eingeschränkt. Es wird nichts aktiv gesteuert, es wird überwacht und im Fehlerfalle die Gefahrenquelle abgeschaltet. Der Schwerpunkt einer Sicherheitssteuerung liegt auf der Fehleraufdeckung. Tritt im Betrieb ein Fehler im Sicherheitssystem auf, muss dieser erkannt werden, bevor dadurch eine Gefährdung entstehen könnte. In der deutschen Sprache ist der Begriff „Fehler“ weitläufiger als im englischen. Im Englischen unterscheidet man zwischen „Fault“, „Error“ und „Failure“, welche im Deutschen mit „Fehler“, „Abweichung“ und „Ausfall“ übersetzt werden [IEC10a, Teil. 4]. Der Fehler (Fault) ist als „Verlust der Fähigkeit die geforderte Funktion auszuführen“ definiert [Wra10, S. 295]. Ausgelöst durch einen Fehler (Fault) kann eine Abweichung (Error) auftreten,

welche einer Nichtübereinstimmung aus erwartetem und tatsächlichem Wert entspricht [Wra10, S. 291]. Auf diesen Fehlzustand kann der Ausfall (Failure) folgen, welcher als „Beendigung der Fähigkeit einer Funktionseinheit eine geforderte Funktion auszuführen“ [Wra10, S. 291] festgelegt ist. Um einen Fehler während des Betriebs aufzudecken werden Diagnosetests durchgeführt. Die Wahrscheinlichkeit damit einen Fehler zu entdecken wird durch den „Diagnosedeckungsgrad“ bestimmt [Wra10, S. 292].
Tritt ein Fehler auf, muss das System in einen Sicherheit bietenden Zustand, den „sicheren Zustand“ überführt werden, dessen Sicherheit als „Freiheit von nicht akzeptierbarem Risiko“ [Wra10, S. 299] verstanden wird. Durch einen hohen Diagnosedeckungsgrad können Fehler detektiert werden, welche möglicherweise nicht zu einer Abweichung, gefolgt von einem Ausfall geführt hätten, wodurch sich die Verfügbarkeit reduziert. Diese ist definiert, mit welcher Wahrscheinlichkeit man das System zu einem zufälligen Beobachtungszeitpunkt im korrekt arbeitenden Zustand vorfindet [Bör04, S. 5]. Im Weiteren werden die Begriffe „sicherheitskritisch“, „sicherheitsrelevant“ und „sicherheitsbezogen“ Synonym verwendet.

2.1.2 IEC 61508 - Funktionale Sicherheit sicherheitsbezogener elektrischer/ elektronischer/ programmierbarer elektronischer Systeme

Die IEC 61508 [IEC10a] besteht aus insgesamt 7 Teilen. Bei Teil 1 bis 4 handelt es sich um den normativen, bei Teil 5 bis 7 um den informativen Teil. Der informative Teil ist nicht zwingend zu befolgen, gibt aber eine Vielzahl von Richtlinien für die Entwicklung an. Der informative Teil ist eine Hilfestellung an den Entwickler und gibt Beispiele für Realisierungen an. Ziel ist es die Anzahl der systematischen und zufälligen Fehler auf ein vertretbares Mindestmaß zu reduzieren. Abgeleitet von dieser Norm existieren spezifischere Auslegungen für die Bereiche Prozessindustrie IEC 61511 [IEC03], Maschinenbau IEC 62061 [IEC10b] und Kraftfahrzeuge ISO 26262 [ISO12].

Mit der IEC 61508 wird der Begriff des *Safety Integrity Levels* SIL eingeführt, deren Skala von 1 bis 4 reicht. SIL 1 stellt die geringste Sicherheitsstufe dar und SIL 4 die höchste. SIL 1 muss von Einrichtungen erfüllt werden, bei welchen im Fehlerfalle Verletzungen von Einzelpersonen zu erwarten sind. Bei SIL 4 ist im Fehlerfalle mit Verletzung und Tod mehrerer Personen zu rechnen. Die Einteilung, welches SIL ein Produkt erreicht, errechnet sich aus der Propability of Failure per Hour (PFH) und der Safe Failure Fraction (SFF). Tabelle 2.1 und Tabelle 2.2 geben die vorgeschriebenen Werte an. Die PFH gibt an, wie wahrscheinlich ein gefährlicher Ausfall pro Stunde auftritt. Mit der SFF wird angegeben, mit welcher Wahrscheinlichkeit ein Systemausfall ein „sicherer Ausfall“ ist. Sichere Ausfälle können in drei Gruppen eingeteilt werden: sichere entdeckte Ausfälle, sichere unentdeckte Ausfälle und unsichere entdeckte Ausfälle. Es wird davon ausgegeangen, dass nicht alle Fehler aufgedeckt werden (können). Dies stellt aber kein Problem dar, wenn von dem Fehler und dessen Auswirkungen keine Gefahr ausgeht. Tritt ein gefährlicher, jedoch entdecker Fehler auf, können immer noch Maßnahmen zur Beherrschung oder Schadensbegrenzung durchgeführt werden. Einzig die gefährlichen unentdeckten Fehler sind mit allen Mitteln zu verhindern. Die SFF berechnet sich wie in 2.1 dargestellt. Die Rate der sicheren (safe) Ausfälle wird durch λ_S repräsentiert, λ_{DD} für die Rate der gefährlichen, entdeckten (dangerous detected) Ausfälle und λ_D für die Rate der gefährlichen Ausfälle. Ob ein Fehler entdeckt wird, bevor Gefahr besteht, hängt von der SFF ab. Um eine hohe SFF zu erreichen, werden Selbsttestfunktionen eingebaut. Diese können von einem einfachen Watchdog-Timer bis zu einer vollständigen Ablaufkontolle der sicherheitstechnischen Anwendung reichen. Mit Hilfe der Selbstestfunktionen ist in begrenzem Maßstab möglich Fehler gemeinsamer Ursache aufzuspüren. Fehler gemeinsamer Ursache (Common Cause Failure (CCF)) stellen in jedem System eine große Gefahr dar.

Hardwaremäßig redundant oder diversitär aufgebaute Systeme könnten zum Beispiel von einer Betriebspannungsunterbrechung gleichzeitig betroffen sein.

$$SFF = \frac{\sum \lambda_S + \sum \lambda_{DD}}{\sum \lambda_S + \sum \lambda_D} \tag{2.1}$$

Im Folgenden dieser Arbeit wird vor allem auf Teil 2 der IEC 61508 Rücksicht genommen, denn diese beschäftigt sich mit den Anforderungen an die Hardware sicherheitskritischer Steuerungen. Trotz des hier gewählten Lösungsansatzes die zertifizierte Hardware ablösen zu wollen und dessen hohe Fehleraufdeckung in Software nachzubilden, muss bei dem Softwareentwurf speziell auf die Fehler der Hardware eingegangen werden. Fehler, welche bei bisherigen Sicherheitssteuerungen bereits in Hardware abgefangen wurden, müssen nun in Software behandelt werden. Somit ist es wichtig sich mit den Hardwarefehlern auseinanderzusetzen und wie sich diese in höheren Softwareschichten auswirken.

In der IEC 61508 werden zwei Typen von Bauelementen unterschieden. Für ein Typ-A Element müssen alle möglichen Ausfallsarten eindeutig definiert und das Verhalten im Fehlerfalle bekannt sein. Somit können nur einfach aufgebaute Elemente wie Widerstände, Kondensatoren, etc. als Typ-A bezeichnet werden. Ein Element ist Typ-B, wenn die Ausfallsarten und das Verhalten im Fehlerfalle nicht definiert ist oder aufgrund der Komplexität nicht definiert werden kann, beispielsweise Mikroprozessoren, integrierte Schaltkreise, etc. Wird ein sicherheitskritisches Produkt aus Typ-A und Typ-B Elementen aufgebaut, muss davon ausgegangen werden, dass das ganze Produkt Typ-B wird. Ein Industrie-Personal Computer (PC) ist ein sehr komplexes Produkt und somit als Typ-B (Tabelle 2.2) zu behandeln.

Tabelle 2.1: Ausfallsgrenzwerte bei kontinuierlichem Betrieb oder hoher Anforderungsrate [IEC10a, Teil 1, S. 37]

SIL	PFH
4	$\geq 10^{-9}$ bis $< 10^{-8}$
3	$\geq 10^{-8}$ bis $< 10^{-7}$
2	$\geq 10^{-7}$ bis $< 10^{-6}$
1	$\geq 10^{-6}$ bis $< 10^{-5}$

Tabelle 2.2: Maximal zulässiger SIL für ein aus Typ-B Elementen aufgebautes (Teil-) System [IEC10a, Teil 2, S. 27]

SFF	**Hardwarefehlertoleranz**		
	0	1	2
$< 60\,\%$	nicht erlaubt	SIL 1	SIL 2
$60\,\% - < 90\,\%$	SIL 1	SIL 2	SIL 3
$90\,\% - < 99\,\%$	SIL 2	SIL 3	SIL 4
$\geq 99\,\%$	SIL 3	SIL 4	SIL 4

Die Anforderungen an die SFF, bezogen auf das SIL, zeigt die Tabelle 2.2. Die Hardwarefehlertoleranz (HFT) von N sagt aus, wie viele Fehler ein System tolerieren kann um die sichere

Ausführung nicht zu gefährden. Treten nun $N + 1$ Hardwarefehler auf, kann die Sicherheitsfunktion beeinträchtigt werden. In dieser Arbeit muss von einer HFT von Null ausgegangen werden. COTS-Hardware ist nicht nach den Regeln der Norm entworfen und bietet keinerlei Rüstzeug um selbst bei Multi-Core-Systemen als zweikanalig argumentiert zu werden. Daraus folgend ist eine SFF von mindest 99 % gefordert. Dieser Wert ist sehr hoch und erlaubt bei der Entwicklung des sicherheitskritischen Systems keine Unachtsamkeiten.

2.1.3 EN 13849 - Sicherheit von Maschinen, Sicherheitsbezogene Teile von Steuerungen

Die EN 13849 [EN08] besteht aus zwei Teilen. Teil eins behandelt allgemeine Gestaltungsleitsätze und Teil zwei die Validierung und löst die EN 954 ab. In der EN 13849 werden die Sicherheitslevel nach Kategorien 1 bis 4 und Performance Level (PL) a bis e angegeben. Entsprechend der Forderung nach SIL 3 der IEC 61508 wäre hier ein PL von e gefordert. Die EN 13849 schreibt den strukturellen Hardwareaufbau entsprechend Abb. 2.1 für die einzelnen PL vor. Die Querverbindung c zwischen den Logikeinheiten L1 und L2 steht für die gegenseitige Überwachung der Systeme. Jede Logikeinheit ist Watchdog für die andere und kann im Fehlerfall den sicheren Zustand einleiten. Der in der Norm gezeigte Aufbau ist jedoch nicht die zwangsläufig für die Hardware anzuwenden, es ist als logisches Schaltbild anzusehen und kann jederzeit verändert werden, sofern nachgewiesen werden kann, dass Sicherheit nicht eingeschränkt wird.

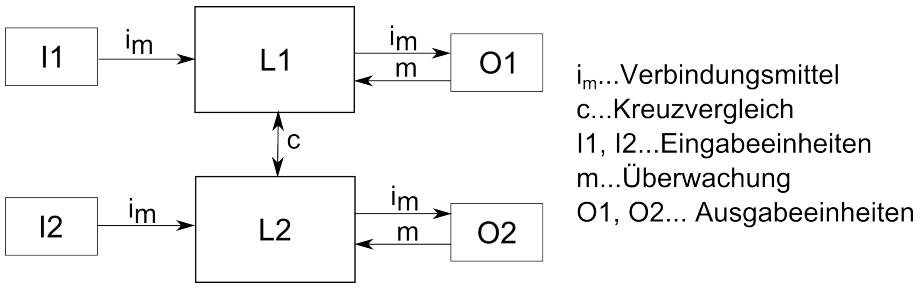

Abbildung 2.1: Logischer Aufbau eines Kategorie 4 Systems entsprechend EN 13849 [S. 46]

Weiters ist gefordert, dass ein einzelner Fehler nicht zum Verlust der Sicherheitsfunktion führt. Bei einer Anhäufung unerkannter Fehler darf das System nicht die Sicherheitsfunktion verlieren. Die CCF sowie die MTTF_d muss „hoch“ sein, dies wird als $> 90\,\%$ definiert. Trotz der vorgeschriebenen, zweikanaligen Architektur stellen CCFs den wunden Punkt dieses Aufbaus dar. Die besten Gegenmaßnahmen werden im Anhang F aufgezählt:

- physikalische Trennung der Signalpfade um Kriechströme zu verhindern
- Diversität zwischen den Kanälen
- Entwurf und Erfahrung mit den Bauteilen
- Analyse der Ausfallsarten durch Effektanalyse
- Ausbildung der Monteure
- Umgebungsschutz vor EMV, Temperatur, Feuchtigkeit

Diese Aufzählung ist natürlich keinesfalls vollständig, sie soll nur einen Überblick über die zu berücksichtigenden Fehlerquellen ermöglichen. Die Anforderungen sind nur durch zertifizierte Hardware erreichbar. Da dies hier keine Option ist, müssen andere Mittel gefunden werden, wie sie in Kapitel 3 vorgestellt werden.

Es zeigt sich, dass kein Sicherheitssystem fehlerfrei sein kann, jedoch muss idealerweise jeder aufgetretene Fehler detektiert werden und darf nicht zu einem Sicherheitsrisiko führen. Eine Möglichkeit mit Fehlern umzugehen ist diese zu tolerierten. Ein fehlertolerantes System kann einen oder mehrere detektiere Fehler tolerieren und die Sicherheitsfunktion weiter ungehindert ausführen. Ein fehlertolerantes System muss eine oder mehrere Formen der Redundanz beinhalten.

2.1.4 Redundanz

Ein redundantes System verfügt immer über zusätzliche Funktionalität, welche im fehlerfreien Betrieb nicht benötigt wird. Allgemein existierten vier Formen von Redundanz [Bör06, S. 65]

- Hardware-Redundanz,
- Software-Redundanz,
- Zeitredundanz,
- Informationsredundanz.

Ein hardwareredundates System besitzt zusätzliche Hardwarekomponenten. Fällt eine Komponente aus, kann eine weitere die Funktion übernehmen. In diesem Zusammenhang existiert der Begriff der HFT. Eine HFT von n bedeutet, dass n Fehler toleriert werden können, beziehungsweise bei n+1 Fehlern muss mit dem Verlust der Sicherheitsfunktion gerechnet werden.
Bei Software-Redundanz ist das gleiche Programm mehrere Male vorhanden. Bleibt eines aufgrund eines Fehlers stehen, kann das zweite die Berechnung zu Ende führen. Bei Software bietet sich zusätzlich heterogene Redundanz, auch Diversität genannt, an. Hierbei wird das Programm mit gleicher Funktionalität, aber unterschiedlichem Quellcode mehrfach implementiert. Günstige und dennoch effektive Diversität kann durch die Verwendung unterschiedlicher Compiler oder Compilerparameter [GG09] erreicht werden.
Bei Zeitredundanz besteht noch zusätzliche Zeit bis die Daten ausgegeben werden müssen. So kann das Programm erneut gestartet werden und noch vor der Deadline die Ergebnisse liefern. Liegen die Daten mehrfach vor, spricht man von Informationsredundanz. Redundanz kann neben der Fehlertolerierierung auch zur Fehleraufdeckung verwendet werden. Die häufigsten Redundanz-Architekturen werden nun diskutiert.

Redundanzarchitekturen

Bei den hier vorgestellten Architekturen wird nur auf die Berechnungslogik genauer eingegangen. Das gezeigte ist ebenso auf die Eingangs- und Ausgangsmodule anwendbar, der Schwerpunkt dieser Arbeit liegt jedoch auf der Berechnungslogik. Um eine ausreichende Fehlererkennung zu erreichen, wird oft ein oder mehrere Elemente mehrfach ausgeführt. Die parallel arbeitenden Komponenten können zur Fehleraufdeckung oder Fehlertoleranz verwendet werden. Bei der Fehleraufdeckung wird das Ergebnis der mehrfachen Berechnung verglichen und im Falle einer Ungleichheit

die Fehlerreaktion ausgeführt. Die Fehlererkennung ist Voraussetzung für Fehlertoleranz. Nur bei einem erkannten Fehler kann mittels Fehlertoleranz entgegengewirkt werden.

Die einfachste Form eines Eingangs-Ausgangs-Systems, welches keine Redundanz besitzt, jedoch in weiterer Folge der Arbeit noch wichtig wird, ist das 1oo1-System in Abb. 2.2.

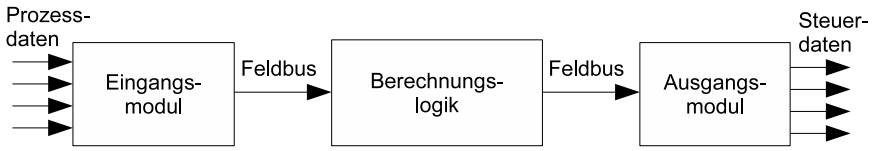

Abbildung 2.2: 1oo1 System [Bör06, p 158]

Das Eingangsmodul besitzt die physikalischen Schnittstellen zur Umgebung. Im Modul können z. B. Analog-Digital-Wandler verbaut sein. Die Übertragung der Daten von Eingang zur verarbeitenden Logik wird typisch über ein Bus-System durchgeführt. Da es sich hierbei um sicherheitskritische Daten handelt, wird ein entsprechendes Sicherheitsprotokoll verwendet. In der Verarbeitungslogik wird aus den Eingangsdaten ein Ausgangsabbild berechnet und wiederum über ein sicherheitskritisches Protokoll an das Ausgangsmodul übergeben. Aufgrund der Abhängigkeit jedes Blocks zum vorhergehenden kann in dieser sogenannten 1-out-of-1 (kurz 1oo1)-Architektur kein Fehler erkannt und somit toleriert werden.

Ist die Hardware für die Berechnung doppelt ausgeführt, aber nur eine Berechnung für eine korrekte Funktion notwendig, spricht man von einem 1oo2-System Abb. 2.3. Dieser Aufbau ist häufig in sicherheitskritischen Systeme zu finden.

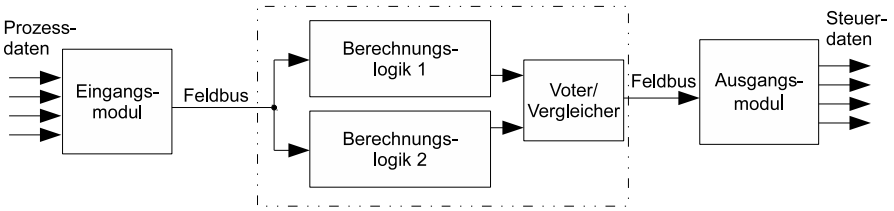

Abbildung 2.3: 1oo2 System

Die zweite Berechnungseinheit kann dem Aufbau der ersten entsprechen. Diesen Fall nennt man homogene Redundanz. Ist der interne Aufbau unterschiedlich, das Verhalten jedoch gleich, wird es heterogene Redundanz, oder diversitäre Redundanz genannt. Beide Berechnungseinheiten generieren ein vollständiges Ergebnis und leiten dieses an den Voter/Vergleicher weiter. Hier wird entschieden, wie mit den Ergebnissen weiter vorgegangen wird. Im Falle einer Übereinstimmung wird das Ergebnis an den Ausgang weitergeleitet. Falls die berechneten Ergebnisse voneinander abweichen, stehen zwei Optionen zur Auswahl. Einerseits kann der Voter seine Stimme für eines der beiden Ergebnisse abgeben und dieses an den Ausgang weiterleiten. Die Entscheidung wird anhand von Vorwissen getroffen, um zu erkennen, welches Ergebnis das Falsche ist. Dieser Aufbau erhöht die Verfügbarkeit, als auch die Komplexität des Voters, da er mehr als nur vergleichen muss. In 1oo2-System wird ein Voter nur selten eingesetzt, ein 1oo3-System eigenet sich dafür besser, da hier eine Mehrheitsentscheidung getroffen werden kann. Werden die Ergebnisse einem Vergleicher zugeführt, vergleicht dieser die Übereinstimmung und löst im Falle einer Abweichung

eine Fehlerbehandlung aus. In industriellen Anwendungen wird meist ein Not-Stop ausgeführt, um den Prozess in einen sicheren und gefahrlosen Zustand überzuführen. Die somit gesteigerte Sicherheit und reduzierte Verfügbarkeit ist ein grundsätzliches Problem von Sicherheitssteuerungen. Der Aufbau besitzt jedoch einen Schwachpunkt: die korrekte Funktion des Vergleichers/Voters kann nicht überprüft werden. Trifft er die falsche Entscheidung oder verfälscht die richtigen Ergebnisse, sind keine Sicherungsmaßnahmen mehr vorhanden um den Fehler abzufangen. Hinzu kommt noch, dass nur Endergebnisse auf deren Richtigkeit überprüft werden. Für eine erhöhte Fehleraufdeckung empfiehlt es sich, ebenso während der Berechnung Zwischenergebnisse zwischen den Berechnungseinheiten auszutauschen und so etwaige Fehler aufzudecken, bevor ein falsches Ergebnis an die Auswertung geliefert wird.
Die Erweiterung um die Diagnose wird im Architekturnamen durch ein angehängtes „D“ symbolisiert. In einem 1oo2D System wie in Abb. 2.4 werden von den Berechnungseinheiten Zwischenergebnisse über den Diagnose-Kanal ausgetauscht. Es ist zu berücksichtigen, dass trotz der Querverbindung keine negative Beeinflussung der Einheiten stattfinden darf. Darum wird hier meist einfache, auf Polling basierende, Kommunikation angewendet. Bei Interrupt-Mechanismen sind für einen „Babbling Idiot“ anfällig. In diesem Fall würde eine Einheit ununterbrochen senden und unterbricht aufgrund eines Interrupts immer die andere Einheit, wodurch diese an der korrekten Ausführung gehindert wird.

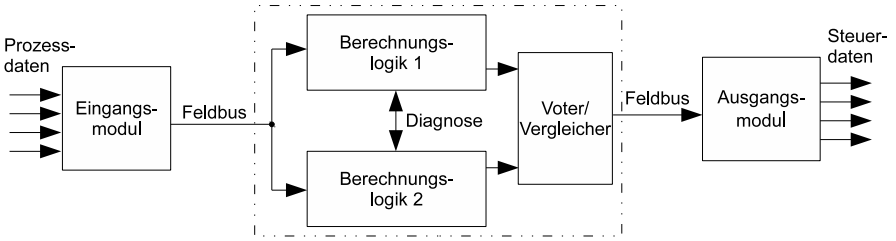

Abbildung 2.4: 1oo2D System

Die Fehleraufdeckung ist höher als ohne Quervergleiche, der Schwachpunkt des Systems, der Voter, ist in dieser Architektur ebenso vorhanden und kann durch einen Einzelfehler zu letalen Systemausfällen führen.

Ein in der Industrie sehr häufig zu findender Aufbau ist die 2oo2-Architektur, optional als 2oo2D ausgeführt (Abb. 2.5). Ein großer Vorteil ist, dass der Vergleicher/Voter nicht benötigt wird. Die Berechnungseinheiten arbeiten weiterhin parallel und unabhängig voneinander. Doch anstatt das volle Ergebnis an den Ausgang zu liefern, gibt jede Berechnungseinheit nur eine Hälfte des Ergebnisses aus. Beispielsweise liefert Berechnungslogik 1 den ersten Teil und Berechnungslogik 2 den zweiten. Vor der Übertragung werden diese Teile in ein Datenpaket kopiert und über den Feldbus übertragen. Verlief die Berechnung korrekt, findet das Ausgangsmodul zwei Teilergebnisse, welche sich gegenseitig vervollständigen. Ist bei der Berechnung jedoch ein Fehler aufgetreten und die Teilergebnisse passen nicht zueinander, kann der Ausgangsknoten den Fail-Safe-Zustand einleiten. Diese Methode setzt voraus, dass das sicherheitskritische Feldbussystem Übertragung doppelter Daten unterstützt. Dies stellt aber kein Problem dar, da die Norm IEC 61508 für höchste Sicherheitsanforderungen über Black-Channel[1] indirekt doppelte Datenübertragung fordert [EN11].

Das Hinzufügen eines dritten Hardwarekanals wie in Abb. 2.6 ist nur bei Systemen mit höchsten Sicherheits- und Verfügbarkeitsanforderungen gefordert. Beispielsweise kann bei Flugzeugen im

[1] Verfahren zur Übertragung sicherheitskritischer Daten über nicht gesicherte Kommunikationskänale

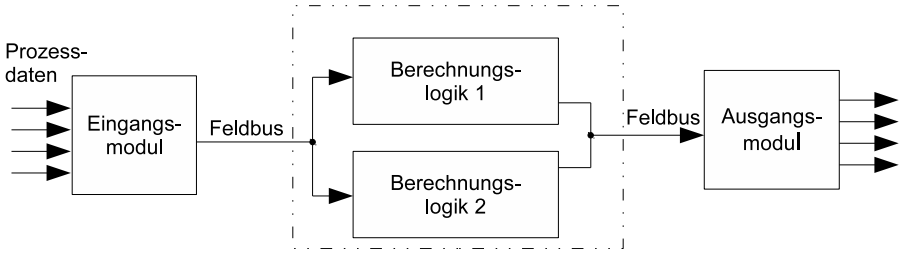

Abbildung 2.5: 2oo2 System

Flug nicht ohne weiteres ein sicherer Zustand angefahren werden. Es muss selbst in Fehlerfall eine Restfunktionalität vorhanden bleiben um sicher landen zu können. Das dabei verwendete 2oo3-System wird auch Triple Modular Redundancy (TMR) genannt und besitzt eine ausgeklügelte Verschaltung der Voter, um im Fehlerfall den defekten Kanal vollständig zu isolieren und eine Beeinflussung der noch korrekt arbeitenden Einheiten zu verhindern. In typischen industriellen Anwendungen sind drei Hardwarekanäle jedoch nicht notwendig, da so gut wie alle Prozesse einen sicheren Zustand besitzten, welcher im Fehlerfall angefahren werden kann. Eine Vervielfachung der Hardware schützt jedoch nicht vor CCFs.

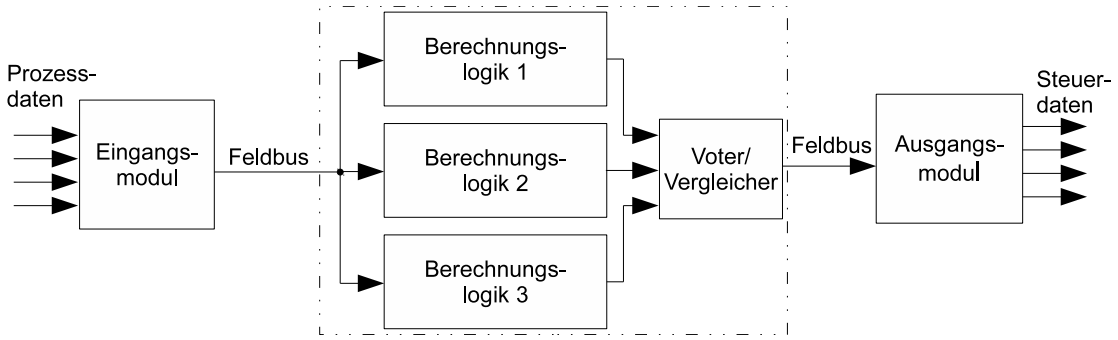

Abbildung 2.6: 2oo3 System [Bör06, p 161]

2.1.5 Fehlerarten

Die Fehlerarten eines Sicherheitssystems lassen sich nach ihrer Ursache und ihrem zeitlichen Auftreten klassifizieren. Das Auftreten der folgenden vier Punkte muss durch entsprechende Maßnahmen verhindert werden [Sch11, S. 7].

- Transienter Fehler: ein Fehler ist transient, wenn er nach einmaligem Auftreten nicht mehr vorhanden ist. Transiente Fehler sind immer zufällig auch wenn sich durch schlechtes Design, wie zum Beispiel systematischen Fehlern, die Wahrscheinlichkeit für transiente Fehler erhöht. Transiente Fehler zu detektieren ist eine der Herausforderungen, auf welche in dieser Arbeit besonderes Augenmerk gelegt werden muss. Ein Fehler dieser Kategorie wäre zum Beispiel ein Bit-Flip im Speicher.
- Permanenter Fehler: ein permanenter Fehler ist dauerhaft im System vorhanden. Ursache dafür kann einerseits ein systematischer Fehler im Design sein, andererseits ein zufällig im

Betrieb aufgetretener Defekt, welcher, im Gegensatz zu einem transienten Fehler, von Dauer ist. Permanente Fehler sind leichter zu detektieren als transiente, da durch das dauerhafte Auftreten die Wahrscheinlichkeit mehrere Daten zu korrumpieren größer ist. Stuck-at-Fehler sind eine häufige Form von permanenten Fehlern.

- Systematischer Fehler: systematische Fehler wirken sich immer permanent im System aus. Sie entstehen während der Entwicklungsphase und sind dauerhaft vorhanden. Durch strukturierte Entwicklung und ausführliche Tests der Einzelkomponenten können diese Fehler reduziert werden. Ein Fehler im Sourcecode fällt in diese Kategorie.

- Zufälliger Fehler: die zufälligen Fehlern stellen eine große Gefahr dar und müssen deswegen besonders intensiv behandelt werden. Im Gegensatz zu systematischen Fehlern können diese Fehler nicht durch ausführliche Tests in der Designphase aufgedeckt werden, obwohl ein schlechtes Design zufällige Fehler begünstigen kann.

Die größte Gefahr geht von den zufälligen, transienten Fehlern aus. Auswirkungen können nicht nur verfälschte Daten sein, sondern auch ein veränderter Programmablauf. Ein Bit-Flip im Speicher kann durch die heute weit verbreitete von-Neumann Architektur Daten wie auch Instruktionen verändern. Aufwendige Hardware- und Softwarelösungen sind notwendig, um diese Fehler beherrschen zu können.

2.1.6 Systemtests

Die vollständige Funktionsfähigkeit der Sicherheitssteuerung muss idealerweise zu jedem Zeitpunkt im Betrieb garantiert werden. Um dies zu ermöglichen, werden Selbsttests eingebaut um die Einzelelemente zu überprüfen und einen Fehler zu finden, bevor er sich in einer sicherheitskritischen Berechnung auswirken kann. Die Ausführung dieser Tests kann entweder online durchgeführt werden, das heißt während des Normalbetriebs und es muss keine gesteuerte Maschine gestoppt werden oder offline, wobei die Sicherheitsapplikation während des Selbsttests angehalten wird. Die muss jedoch nicht zwingend von außen erkennbar sein. Bisher bestehende Systeme besitzen meist offline-Tests. Dazu wird die Sicherheitsapplikation angehalten oder eine Ruhephase des Systems ausgenutzt, in welcher einzelne Module nicht benötigt werden. Sie finden offline statt, da nicht benötigte Teile getestet werden; die Applikation muss jedoch nicht angehalten werden, somit gehen Selbsttests nicht mit dem Stillstand der Maschine einher. Beispielsweise werden nicht benötigte Teile im Random Access Memory (RAM) mittels Galpat oder Walking-Bit auf Stuck-at-Fehler oder Übersprechen von Daten- und Adressleitungen untersucht. Für den Test der Arithmetical Logical Unit (ALU) können Berechnungen mit bereits vorher bekannten Ergebnissen durchgeführt werden. Ein Vergleich zwischen den vorher und aktuell berechneten Daten zeigt die Fehlerfreiheit der ALU.
Diese Tests zeigen meist nur systematische, permanente Fehler auf. In seltenen Fällen können zufällige, während des Selbsttests auftretende transiente Fehler detektiert werden die zu einem Sicherheitshalt des Systems führen. Ein Löschen des Speichers bereinigt das Problem, da es die Eigenschaft der transienten Fehler ist, nur für einen Zeitraum vorhanden zu sein. Der Schwerpunkt von offline-Selbsttests ist das Aufdecken von permanenten Fehlern. Darum könnte das Detektieren eines transienten Fehlers als störend interpretiert werden. Dem ist jedoch nur bedingt so, denn wäre der transiente Fehler in der Applikation aufgetreten, hätte er zu Sicherheitsrisiken führen können. Dies zeigt die Schwäche von offline-Tests: permanente Fehler können unabhängig von der Applikation aufgedeckt werden, transiente Fehler können nur durch Zufall entdeckt werden und

deren Aufdeckung ist ohne weiteren Nutzen.
Um transiente Fehler während der Sicherheitsapplikation aufzudecken, sind online-Tests notwendig. Einer der wichtigsten online-Tests ist der Vergleich der Ausgangsdaten zwischen redundanten Berechnungspfaden, wie in Abschnitt 2.1.4 ausführlich erklärt. Besser wäre es, Fehler früher aufzudecken noch bevor ein Endergebnis ausgegeben wird. Für den RAM des Systems kann dies im einfachsten Fall z. B. mittels Parity-Bit oder speziellen Error Correcting Code (ECC)-Speichermodulen erreicht werden, wie sie im Serverbereich weit verbreitet sind. Die ALU im Betrieb zu testen deutlich komplexer. Eine häufig verwendete Technik ist das Mehrfachausführen von Berechnungen. Dies führt zur Fehleraufdeckung in der Annahme, dass der transiente Fehler nur einmal auftritt. Das Beste aus der Welt der Online- und Offline-Tests zu kombinieren ist der Ansatz des Coded Processing und wird in 2.3 näher erläutert.

2.1.7 Fehler realer Systeme

Das System besteht aus mehreren Teilen, welche unterschiedlichen Fehlerarten und -raten unterworfen sind. Dementsprechend müssen auch die Fehleraufdeckungsmaßnahmen je nach Element angepasst werden, wobei es sich bei Central Processing Unit (CPU) und Speicher um die zwei wichtigsten Elemente eines Computersystems handelt.

In der Publikation [RV07] von Valzco und Faure werden die logischen Elemente einer CPU analysiert und die Fehlermöglichkeiten der einzelnen Module quantitativ untersucht. Die Grundbausteine, aus welchen eine CPU besteht, lauten Register, Adress- und Datenbusse , Integer-/Floating-Point Unit, Instruktions-Cache, Daten Cache, Steuerwerk und Debug Unit. Jedes dieser Elemente hat aufgrund seines unterschiedlichen Aufbaus und Funktion bestimmte Schwächen für Bitfehler.
Register: Register sind bei so gut wie jeder Berechnung oder Datenmanipulation involviert, damit verbunden ist eine starke Auswirkung von Single Event Upset (SEU). Am gefährlichsten ist es, wenn gegen Ende der Berechnung, kurz vor der Ausgabe ein Bitfehler passiert. Es ist schwierig den Fehler zu entdecken und eine Fehlerbehandlung einzuleiten. Die Fehlerauswirkung kann ebenso nicht eingegrenzt werden, da die Register Daten, Instruktionen und Adressen bearbeiten und somit das zukünftige Verhalten bei einem Bitfehler unvorhersehbar ist.
Adress- und Datenbusse: dieser Teil beinhaltet meist nur wenige Register, nur ein paar Latches um Adressen zu speichern. SEUs in diesen Latches führen zu falschen Daten/Adressen Schreib-/Lesevorgängen. Adressfehler können durch eine Memory-Protection erkannt werden, jedoch nur, wenn der erlaubte Speicherbereich verlassen wird. Fehlerhafte Zugriffe im zugeteilten Speicherbereich bleiben unbemerkt.
Integer Unit, Floating-Point Unit: Fehler in diesen Elementen sind schwer, oder gar nicht von selbst zu erkennen. Weder Hardware oder Betriebssystem bieten eine Möglichkeit hier Abweichungen zu erkennen. Eine SEU Erkennungswahrscheinlichkeit von 0 % [Joã09] zeigt die Wichtigkeit von zusätzlichen Schutzmaßnahmen.
Instruction Cache: dieser besteht aus zwei Teilen, einem großen Static Random-Access Memory (SRAM) für die geladenen Daten und einem Tag-Array, welches die Gültigkeit der Daten im SRAM festlegt. Veränderungen durch SEUs in den Tag-Arrays können zu zwei Fehlerarten führen.

- Eine Instruktion wird fälschlicherweise als ungültig bezeichnet. Daraus folgt ein Neuladen der Instruktion aus dem externen Speicher und eine Verzögerung im Programmablauf.
- Wird ein Tag von ungültig auf gültig gesetzt, wird ein veralteter Befehl ausgeführt und der Programmfluss fehlerhaft.

Für die Daten im Instruktion-Teil des Caches entstehen vier Fehlermöglichkeiten:

- Veränderung der Instruktion bei ungültig gesetztem Tag führt zu keiner Veränderung im Programmablauf,
- verfälschte Instruktion ist nicht mehr Teil des Instruktion-Sets führt zu einer Exception/Trap,
- der SEU tauscht die Instruktion und ein falscher Befehl wird ausgeführt und
- der SEU verändert die Operanden der Instruktion.

Zusammengefasst führen die Fehler wieder zu falschen Outputs oder Verlust der Sequenz (veränderter Programmfluss).
Data Cache: ähnlicher Aufbau wie bei dem Instruktion Cache in Daten- und Tag-Array. Fehler im Tag-Array können Daten fälschlicherweise gültig bzw. ungültig machen. Fehlermöglichkeiten im Allgemeinen gleich dem Instruktion Cache.
Steuerwerk: Diese beinhaltet möglicherweise komplexe Algorithmen zur Steuerung des Prozessors (pre-fetching, out-of-order execution,...). Durch den komplexen Aufbau können Fehler sich vielseitig auswirken: falscher Programmablauf, Trap/Exception, etc.
Debug Unit: SEUs können spezielle Prozessormodi ausführen, welche möglicherweise nicht dokumentiert sind. Daraus entstehen unvorhersehbare Outputs und Brüche im Programmfluss.

Aufgrund der hohen Packungsdichte am Wafer stellt der Speicher einen besonders kritischen Teil eins Computers dar, weshalb er hier genauer analysiert wird. Der Festwertspeicher stellt ein geringes Risiko für unentdeckte dar, da er nur einmal geschrieben und ab dann nur noch gelesen wird. Um Fehler in den Daten festzustellen bieten sich Prüfsummen wie z. B. die zyklische Redundanzprüfung an, wodurch Fehler mit nur geringem zusätzlichen Speicher- und Rechenaufwand entdeckt werden können.

Deutlich kritischer ist der Arbeitsspeicher zu betrachten. Hier wird die Information in Form von Anwesenheit bzw. Abwesenheit von Elektronen dargestellt. Moderne Arbeitsspeicher sind als Dynamic Random Access Memory (DRAM) aufgebaut, in welchen die gespeicherten Elektronen in Kondensatoren geladen werden. Diese Ladung kann nur wenige Millisekunden gehalten werden, wodurch eine Auffrischung notwendig wird [HJL02, S. 455]. Diese stetige Auffrischungen gepaart mit schnellen Lese - und Schreibvorgängen sind besonders fehleranfällig. Wird nur ein Kondensator falsch geladen, oder verliert seine Ladung zu schnell, werden die Nutzdaten verfälscht. Ebenso sind Schreib- und Lesefehler nicht unbeachtet zu lassen: es besteht die Möglichkeit, dass aufgrund eines fehlerhaften Transistors in der Adressleitung bei bestimmten Schreib-/Lesekombinationen eine falsche Stelle bearbeitet wird. Google Inc. [Bia09] berichtet von einer Abhängigkeit der Fehler von Temperatur und Alter. Wie zu erwarten ist, führt eine erhöhte Temperatur zu höheren Fehlerraten. Google konnte hier eine Verschlechterung um Faktor drei feststellen. Das Alter der Speicherelemente wirkt sich ebenso auf die Bit-Fehler-Rate negativ aus. Nach bereits 10 Monaten kann eine erhöhte Fehlerrate festgestellt werden. Nach nur 20 Monaten wird bereits ein 6-fach erhöhter Wert (im Vergleich zum Ausgangswert) festgestellt.
Sieht man von den irdischen Fehlerquellen ab, existiert noch eine weitere Gefahr, welche die Nutzdaten im Speicher verfälschen kann. Ionisierende Strahlung, wie sie bei radioaktiven Zerfall oder auch Sonnenstürmen vorkommt stellt eine vollkommen unberechenbare Gefahr dar, welche zu transienten Speicherfehlern führen kann. Trifft ionisierende Strahlung auf eine Speicherstelle, kann diese den gespeicherten Wert verändern [Sch07]. Wie stark diese Fehlerquelle zu

berücksichtigen ist hängt von einer Vielzahl von Faktoren hab. Einerseits ist die Strukturbreite wichtig, denn bei kleineren Strukturen werden weniger Ladungsträger zur Darstellung eines Bits benötigt, welche dann leichter verfälscht werden können. Mit sinkender Strukturbreite sinkt auch die Betriebsspannung, welche die Problematik mit ionisierender Strahlung weiter verschärft. Tezzaron Semiconductor [Sem05] berichtet in für DRAM eine Soft Error Rate von $2,3 \cdot 10^{-12}$ pro Bit pro Stunde (entspricht 2.300 Failure in Time (FIT)[2]/Mbit). Daraus ergibt sich bei 1 Gbit Speicher ein Fehler alle 435 Stunden, oder 20 Bitfehler im Jahr. Google Inc. [Bia09] beobachtete in deren Serverfarmen durchschnittlich 25.000 FIT bis 75.000 FIT/Mbit entgegen der bisherigen Annahme von 200 FIT bis 5.000 FIT.

Auf den Speicher muss daher besonderes Augenmerk gelegt werden, da zukünftige Speichermodule noch geringere Strukturbreiten aufweisen werden und somit die Fehleranfälligkeit weiter steigt. Zusätzlich werden zukünftige Sicherungsprogramme ausführlicher, wodurch der benötigte Speicher steigt und dadurch die Wahrscheinlichkeit einen transienten Bitfehler im sicherheitskritischen Speicherbereich anzutreffen. Abhilfe kann ein ECC-RAM liefern, welcher Speicherfehler erkennt und selbstständig korrigieren kann. Dieser RAM ist im Serverbereich bereits weit verbreitet da hier eine Anhäufung von Speicherfehlern über die Zeit zur sogenannten „Datenbankerosion“ führen kann. In der Sicherheitstechnik konnte dieser Speicher nicht Fuß fassen, da die Fehlererkennung rein Bitfehler aufdeckt. Falsche Adressierung oder systematische Fehler sind nicht erkennbar, wodurch kein Weg an redundanten Speichermodulen vorbeiführte. Das Gleiche gilt natürlich ebenso für Parity gesicherten Speicher, wobei hier die Hamming-Distanz im Vergleich zu ECC stark verringert ist.

2.1.8 Beispiel einer Sicherheitssteuerung

In diesem Punkt soll eine typische am Markt befindliche Sicherheitssteuerung analysiert und deren Komponenten für hohe Fehleraufdeckung aufgezeigt werden. Es zeigt in welchen Punkten sich das in Kapitel 3 entwickelte System von am Markt befindlichen abhebt.

Entsprechend der IEC 61508 Teil 2 lassen sich die Eckdaten einer standardisierten Sicherheitssteuerung skizzieren. Zum einen besteht sie aus zwei hardwaremäßig unabhängigen Kanälen, woraus eine HFT von eins folgt. Laut Tabelle 2.2 ist somit für SIL 3 eine SFF von $90\,\% < 99\,\%$ gefordert. Ob die beiden Kanäle als 1oo2D- oder 2oo2D-Architektur geschaltet sind, bleibt Entscheidung des Herstellers. Als Prozessoren eignen sich leistungsstarke Mikrocontroller z. B. der Firma ARM [5]. Diese sind weit verbreitet und die gefundenen Fehler im Prozessorkern (Silicon Errata) sind gut dokumentiert. Zusätzlich zu den digitalen Bauelementen sind analoge nötig um die Umwelteinflüsse messen zu können und im Falle einer Verletzung der geforderten Grenzwerte einen Nothalt einzuleiten. Analog-Digital Converter (ADC) werden einerseits eingesetzt, um die Betriebspannung zu überwachen, andererseits um die Temperatur des Chips, des Gehäuses und der Umgebung zu messen. Die Richtigkeit dieser Messung ist sicherheitskritisch, weshalb zusätzlich Referenzspannungsquellen eingesetzt werden, um periodisch die ADCs zu testen. Ein möglicher Aufbau dieser Testschaltung ist in Abb. 2.7 dargestellt. Der Messeingang des ADCs kann über den CTRL-Ausgang entweder an die Versorgungsspannung V_{cc} angeschlossen werden, oder an das Widerstandsnetzwerk. Im Normalbetrieb wird die Betriebsspannung zum Ground (GND) gemessen, für Selbsttests wird auf einen der beiden Spannungsteiler umgeschaltet. Die Widerstandsverhältnisse R_1 zu R_2 und R_3 zu R_4 sollten so gewählt werden, dass der Spannungsbereich des ADCs abgedeckt wird.

[2] $1Fit = 10^{-9}\frac{1}{h}$

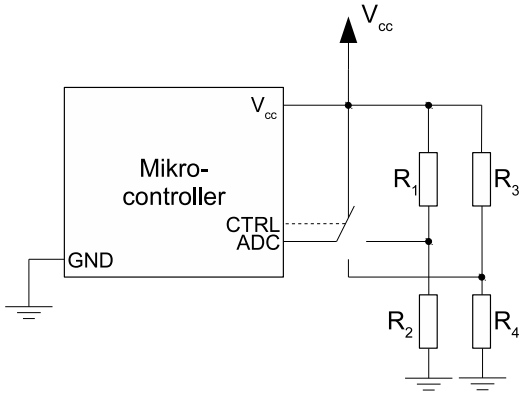

Abbildung 2.7: ADC Testschaltung

Mit der Umgebung wird meist über ein Feldbussystem interagiert, welches für sicherheitskritische Daten zugelassen ist. Die Daten werden dann von einem sicheren Eingangsmodul aufgenommen und an die Sicherheitssteuerung geschickt. Die Ausgangsdaten werden von einem sicheren Ausgangsmodul empfangen und auf den Prozess angewendet. In seltenen Fällen befinden sich sichere Eingangs-/Ausgangsklemmen direkt an der Sicherheitssteuerung. Diese Lösung kommt jedoch nur bei kleineren Sicherheitsanwendungen zum Einsatz, bei größeren kann durch die Dezentralisierung der Verkabelungsaufwand reduziert werden. Der Übergang von interner Logik zu Ausgangsklemmen wird über Optokoppler geführt, um Störungen auf der Leitung keine Chance für Interferenzen mit der Berechnungseinheit zu geben.
Die Software auf den beiden Hardwareplattformen ist diversitär ausgeführt. Das Verhalten nach außen ist bei beiden gleich, die interne Verarbeitung jedoch nicht. Dazu werden unterschiedliche Programmierer-Teams angestellt, oder in einer günstigeren Variante nur die Compiler-Optionen variiert [GG09]. Somit wird die Wahrscheinlichkeit erhöht, dass sich Hardwarefehler in den unterschiedlichen Kanälen unterschiedlich auswirken, wodurch ein Fehler leichter aufgedeckt werden kann.
Neben der Steuerungsanwendung werden noch Diagnose-Programme ausgeführt, welche die Hardware testen, um permanente Fehler, welche im Betrieb auftreten, aufdecken zu können. Die Norm empfiehlt zum Beispiel für den RAM den Galpat-, oder Abraham Test [IEC10a, Teil 2, Tabelle A.6]. Dieser Test muss neben der Anwendung laufen und darf sie nicht beeinflussen.
Tritt ein interner Fehler auf, wird in den „Fail-Silent“-Modus gewechselt, dies bedeutet das Gerät verhält sich passiv und generiert keine Ausgangsdaten mehr. Das Ausgangsmodul schaltet nun nach einer gewissen Totzeit alle Ausgänge sicher ab.

2.2 Fehlermodelle

Um die Auswirkungen von Hardwarefehlern zu untersuchen, muss ein Modell des PCs erstellt werden. Diesem Modell werden Fehler unterstellt und die Auswirkung dieser untersucht. Ein gutes Fehlermodell sollte einerseits der Hardware so weit wie möglich entsprechen, doch andererseits die Fehleranalyse so einfach wie möglich machen. Ein Fehlermodell ist unbrauchbar, wenn es zwar in der Fehleruntersuchung gut abschneidet, jedoch der Realität nicht ausreichend entspricht. Das in dieser Arbeit benötigte Fehlermodell muss einerseits sehr präzise sein, da ein

hoher Diagnosedeckungsgrad gefordert ist, andererseits muss es generisch sein, um für verschiedene COTS-Hardware zu gelten.

Das grundlegendste Fehlermodell auf Hardware-Ebene berücksichtigt nur einzelne *Stuck-at* Fehler und *Bit-Flips* [Gol06, S. 12]. Unter einem Stuck-at Fehler versteht man das permanente festhängen eines Bits. Eine Stelle in der Hardware, unabhängig ob es sich dabei um Register, Speicher, ALU, Bus oder ähnliches handelt, kann nur mehr einen Binärwert annehmen. Ein beschreiben oder löschen der fehlerhaften Stelle ändert nichts am Wert. Ein Bit-Flip stellt den unerwünschten Wechsel einer Bit-Stelle in der Hardware dar. Das Auftreten ist transient und wird beim nächsten Beschreiben korrigiert. Wird jedoch gelesen, kann sich der Fehler fortpflanzen und das System gefährden. Dieses Modell ist für die digitalen Sichtweise jeder Hardware anwendbar, jedoch nur selten praktikabel. Der interne Aufbau von COTS-Hardware ist meistens nur dem Hersteller bekannt, wodurch die Auswirkungen eines einzelnen Bitfehlers nicht nachvollziehbar sind. Selbst wenn die Hardware bis auf Transistorebene bekannt wäre, wäre die Komplexität dennoch undurchschaubar.

Einen Abstraktionsschritt höher über der Hardware existieren die System-Ebenen Fehlermodelle, sie stelle die in Abschnitt 2.1.7 erwähnten Fehler in verallgemeinerter Form dar. Das Modell besteht im Entwurf von [Gol06, S. 13] aus zwei Vertretern, einerseits den Datenfehlern und den Codefehlern (oder Instruktionsfehler). Bei den Datenfehlern wird von einem Fehler in einer Speicherstelle ausgegangen, unabhängig vom Ort wo die Daten gespeichert sind. Ein Codefehler verfälscht eine Instruktion im Programmcode. Auch hier ist nicht festgelegt wo der Fehler genau auftritt. Die Codefehler werden in zwei weitere Unterkategorien eingeteilt. Typ 1 verfälscht Instruktionen in dem ein Befehl durch einen anderen ersetzt wird. Der Kontrollfluss wird nicht verändert, das Rechenergebnis hingegen schon. Eine Ausnahme dieser Definition muss jedoch gemacht werden, falls das Rechenergebnis für den weiteren Kontrollfluss wichtig ist. Der Typ 2 behandelt Kontrollflussfehler, welche sich bei Sprungbefehlen auswirken. Es könnte beispielsweise die Zieladresse eines Sprunges verfälscht werden, oder die Sprungbedingung selbst wird manipuliert.

Ein sehr ähnliches Fehlermodell wird von Forin [For89] verwendet. Es besitzt drei Gruppen von Fehlern, Operationsfehler (Operation Error), Operatorfehler (Operator Errors) und die Operandenfehler (Operand Error). Die Operandenfehler sind den Datenfehlern, die Operationsfehlern den Codefehlern Typ 1 ähnlich sind. Die Operandenfehler von Forin können, im Gegensatz von den Datenfehlern, alle Instruktionen verfälschen, egal ob sie für den Kontrollfluss wichtig sind, oder nicht. Zusätzlich behandelt das Modell noch Spezialfälle wie verlorene Updates.

Die Wahrscheinlichkeit, dass sich ein Fehler auf den Kontrollfluss auswirkt, kann bis zu 78 % betragen [Mir92], weshalb für diesen Fall ein spezielles Fehlermodell entwickelt wurde. Es basiert auf der Annahme, dass sich ein Programm in sogenannte Basic Blocks (BB) einteilen lässt [Gol06, S. 63]. Ein BB besteht aus einer Ansammlung von Befehlen, welche bei der Ausführung nicht unterbrochen werden dürfen. Wird ein Block ausgeführt, wird immer bei der ersten Instruktion gestartet. Eine Sprungbedingung oder ein Unterprogrammaufruf darf sich nur am Ende des BB befinden. Der BB ohne den Sprungbefehl am Ende wird auch „Body“ genannt, woraus sich zwei Sonderfälle ergeben. Befindet sich am Ende eines Block kein Sprung, stimmen BB und Body überein. Besteht der BB nur aus einem Sprung, existiert kein Body. Anhand dieses Soll-Verhaltens können fünf mögliche Fehlverhalten abgeleitet werden, dargestellt in Abb. 2.8.

Fehler in diesem Modell können entweder zu einem inter-Blockfehler oder intra-Blockfehler führen. Ein intra-Blockfehler führt dazu, dass Sprungquelle und Ziel sich in unterschiedlichen BB befinden (Typ 1 bis 4). Sprünge bei inter-Blockfehlern verlassen den gerade ausgeführten BB nicht (Typ

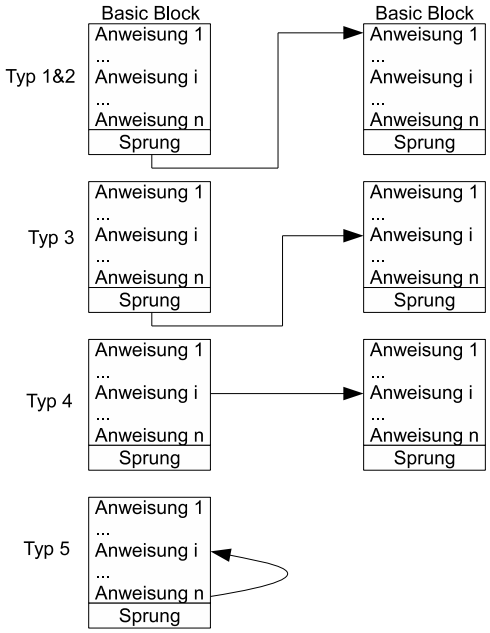

Abbildung 2.8: Fehlermodell für Basic Blocks [Gol06, S. 67]

5).
Typ 1 und 2 sind sich sehr ähnlich, bei beiden wird vom Ende eines BB zu dem Anfang eines anderen BBs gesprungen. Bei Typ 1 handelt es sich um einen verbotenen Sprung und bei Typ 2 um einen erlaubten, jedoch vom Soll-Programmfluss abweichenden Sprung. Ein Sprung vom Typ 3 führt von Endes des Quellblocks in die Mitte eines anderen BB und nicht an den Anfang. Geht der Sprung nicht vom Ende eines Blocks aus, sondern von dem Body, so handelt es sich laut Definition um einen Fehler Typ 4. Wird innerhalb des Bodys ein Sprung ausgeführt, welcher wieder in den Body des Eigenen führt, spricht man von einem Fehler Typ 5.

Mit den bereits vorgestellten Fehlermodellen lässt sich das Verhalten im Fehlerfall auf Systemebene gut darstellen. Gänzlich ignoriert werden jedoch Phänomene welche sich der digitalen Betrachtungsweise entziehen. Dazu Zählen die analogen Vorkommnisse, wie zum Beispiel Spannungsschwankungen, Jitter, Latch-up-Effekt[3] und viele mehr. Diese Effekte werden in der IEC 61508 [IEC10a, Teil 2, Tabelle A.1] aufgezählt und im Folgenden behandelt. Ziel ist es alle Anforderungen für eine Diagnostic Coverage (DC) von „hoch“ zu erfüllen, wie diese für ein einkanaliges System gefordert ist.
Die Einträge in der Tabelle A.1 in der IEC 61508 [IEC10a] sind sehr generisch formuliert, da sie für eine Vielzahl von Sicherungssystemen gelten. Für die einzelnen Anwendungsfälle müssen die Angaben spezialisiert werden. Die Anforderungen dürfen nicht als Checkliste gesehen werden, viel wichtiger ist die Auseinandersetzung mit den Risiken und gezielte Gegenmaßnahmen zu Verbesserung des Gesamtsystems.

[3] fehlerhafte Spannung am Halbleitersubstrat führt zur Zerstörung des Bauteils

2.3 Coded Processing

In der IEC 61508 [IEC10a, Teil 2, Tabelle A.4] [IEC10a, Teil 7, Clause A.3.4] wird die „codierte Verarbeitung“ von sicherheitskritischen Daten mit dem Prädikat „hoher Diagnosedeckungsgrad“ erwähnt, eine Erläuterung befindet sich im Folgenden.

Bei codierter Verarbeitung (englisch „Coded Processing“) handelt es sich um Codes, welche den Nutzdaten hinzugefügt werden um eine Fehlererkennung zu ermöglichen. Optional kann anschließend eine Fehlerkorrektur durchgeführt werden. Ziel ist es, eine möglichst hohe Fehlererkennungswahrscheinlichkeit, unabhängig von der Hardware, zu erhalten. Zu allen Variablen im Benutzerprogramm wird redundante Information hinzugefügt, in welcher Form dies geschieht hängt von der Codierungsart ab. Alle Codierungen dieser Art beruhen auf der Idee den möglichen Zahlenraum einzuschränken. Es wird Information hinzugefügt, welche nur bestimmte Bitkombinationen erlaubt. Tritt im Speicher oder nach der Berechnung eine ungültige Kombination auf, kann von einem Bitfehler ausgegangen werden.
Der Unterschied von arithmetischen Codes, wie sie in dieser Arbeit vertieft behandelt werden, zu den anderen fehleraufdeckenden Codes besteht in der Speicherung und Verarbeitung der Prüfsumme. Bei den häufig verwendeten Prüfalgorithmen wie Cyclic Redundancy Check (CRC) oder Parity-Bit ist die Prüfsumme von den Nutzdaten getrennt im Speicher und wird ebenso getrennt, in einem zusätzlichen Rechenschritt, erzeugt. Man nennt diese Codes auch systematisch und separierbar [Gar66]. Bei separierbaren Codes werden die Nutzdaten getrennt von den Prüfdaten berechnet. Ein Code ist systematisch, wenn bei einer Datenbreite von n Bits k (wobei $k < n$) für die Prüfdaten verwendet werden und $(n - k)$ Bitstellen für die Nutzdaten verbleiben [Rao72]. Bei arithmetischen Codes handelt es sich um einen nicht separierbaren, nicht systematischen Code, das heißt Nutzdaten und Prüfdaten werden in dem gleichen Rechenschritt erstellt. Die Prüfsumme wird mit den Nutzdaten verflochten und gleichzeitig bei jeder Weiterverarbeitung aktualisiert.

Relevante Forschungs- und Entwicklungsarbeiten

Das Gebiet der mathematischen Codierung für erhöhte Fehleraufdeckung bei Berechnungen wurde in den Anfängen des Computerzeitalters entwickelt, jedoch das volle Potential bisher nicht ausgeschöpft. David Brown legt den ersten Stein für codierte Verarbeitung im Jahr 1960 [Bro60]. In der Luftfahrt wurde das Konzept der codierten Verarbeitung ebenfalls untersucht [Liu72]. Die Gewichtsersparnis durch Fehlererkennung in Software anstatt redundanter Hardware ist im Flugverkehr ein großer Vorteil. In der weiteren Entwicklung wurde kodierte Verarbeitung für die Signaltechnik in Eisenbahnsystemen eingesetzt [For89]. Die hohe Fehleraufdeckung ermöglicht einen Einsatz in höchst sicherheitskritischen Systemen, wie sie Züge darstellen [Cla09].

Ein ausführliches Werk zu Coded Processing für Systeme der Gegenwart publizierte Schiffel [Sch11] von der Technischen Universität Dresden. In der Arbeit wird der Schwerpunkt auf AN-Codes und deren Erweiterungen gelegt. Es wurden zwei Software-Tools entwickelt, welche dem Programmierer die Arbeit des händischen Codierens abnehmen. Einmal wird das Programm in einem Interpreter ausgeführt, welcher zur Laufzeit die Codierung vornimmt. In einem zweiten Tool wird ein Compiler modifiziert, damit dieser zur Compilezeit die Codierung vornimmt. Mit diesen Tool werden mehre Beispielprogramme erstellt und die Fehleraufdeckung durch Fehlerinjektion sowie die Performance gemessen. Während in Schiffels Arbeit theoretisch das Gebiet der

Codierungen erarbeitet wird, wird in dieser Arbeit ein vollständiges Sicherheitskonzept im Einklang mit der IEC 61508 [IEC10a] entwickelt, welches nur als einen Baustein von vielen codierte Verarbeitung zur Sicherung enthält.

In der industriellen Sicherheitstechnik wurde Coded Processing bisher nur wenig Aufmerksamkeit geschenkt. Eine Ursache dafür ist der gesteigerte Ressourcenverbrauch verglichen mit uncodierter Berechnung [Sch11, S. 166]. Mit fortschreitender Rechengeschwindigkeit der CPUs werden Lösungen, basierend auf Coded Processing, attraktiver.

Arithmetische Codes

Arithmetische Codes besitzen die Eigenschaft trotz der Anwendung einer arithmetischen Operation erhalten zu bleiben. Mit dieser Eigenschaft können Rechenfehler, welche während einer Operation aufgetreten sind, detektiert werden. Man sagt die Codierung bleibt erhalten, wenn für die codierte Operation $\oplus$ mit den codierten Variablen X', Y' und die uncodierte Operation $+$ mit den uncodierten Variablen X, Y Eq. 2.2 gilt [Kor07, S. 74].

$$X' \oplus Y' = (X + Y)' \tag{2.2}$$

Die einfachste Form der arithmetische Codes, kurz „AN-Codes“, basieren im ersten Schritt auf dem Prinzip, jede Variable x_f der Nutzdaten mit der Konstante A zu multiplizieren. Für A kann eine Ganzzahl frei gewählt werden (siehe Abschnitt 2.3), jedoch ist zu beachten, dass die Wahl die Fehleraufdeckungseigenschaften beeinflusst. Die Multiplikation von A mit der Variable x_f führt zu der codierten Variable 2.3.

$$x_c = A \cdot x_f. \tag{2.3}$$

Diesen Vorgang nennt man *Codieren*, wird aus x_c wiederum x_f extrahiert, spricht man vom *Decodieren*. Die Variable wird vor der ersten Berechnung codiert und erst am Ende des Steuerungszyklus dekodiert. Somit müssen die Ein- und Ausgangseinheiten des Sicherungssystems nicht über die Codierung informiert sein.

Die Variable x_f wird auf den ganzen verfügbaren Zahlenbereich verteilt, dies ist in Abb. 2.9 ersichtlich. Somit ist nicht mehr jede Zahl erlaubt, sondern nur mehr Vielfache von A. Bei der Wahl eines passenden As ist zu beachten, dass der verfügbare Zahlenbereich in der CPU nicht überschritten wird. Andererseits verlangt eine hohe Fehlererkennung ein großes A. Näheres zu einer optimalen Wahl von A wird in Teil 2.3 diskutiert. Es steht somit für die Variable x_f nicht der ganze vorhandene Zahlenbereich für die Nutzdaten zur Verfügung, sondern $x_{fmax} = \frac{2^n}{A}$, wobei n die Bitbreite der CPU angibt. Um die Auswirkung der Reduktion der nutzbaren Bitbreite gering zu halten, empfiehlt sich mindest eine 64 Bit CPU-Architektur, bzw. der Einsatz der SSE-Einheit, wie sie in allen aktuellen CPUs vorhanden ist. Diese kann Variablen mit ein Bitbreite von bis zu 128 Bit verarbeiten. Zusätzlich verfügen Intel CPUs mit Sandy Bridge Mikroarchitektur über eine 256 Bit breite „Vector Extension“ [Int13, S. 2-14].

Die Rechnung mit AN-codierten Variablen ist trivial. Die Addition zweier codierter Variablen x_c und y_c berechnet sich wie folgt:

$$z_c = x_c + y_c = A \cdot x_f + A \cdot y_f = A \cdot (x_f + y_f) \tag{2.4}$$

Die Probe, ob diese Berechnung korrekt war, lässt sich mittels einer Module-Operation durchführen. Wenn die Bedingung 2.5 erfüllt ist, handelt es sich bei dem Ergebnis, wie auch bei den Ausgangsoperanden, um ein Vielfaches von A.

$$mod\left(A \cdot (x_f + y_f), A\right) = 0 \tag{2.5}$$

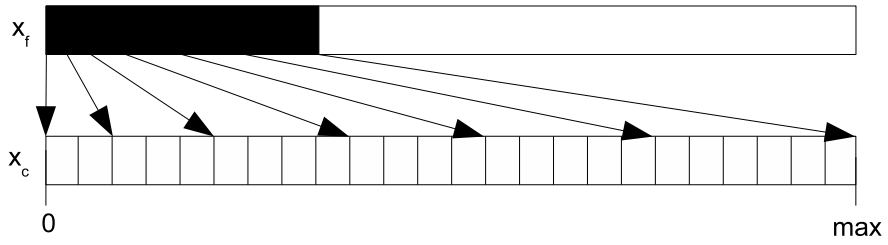

Abbildung 2.9: Prinzip der AN-Codierung

Die Wahrscheinlichkeit bei AN-codierten Daten einen Fehler aufzudecken ergibt sich als $1/A$ [Oze92]. Dies ist die Wahrscheinlichkeit, dass bei einem zufälligen Fehler ein anderes gültiges Codewort generiert wird. Bei der Addition und Subtraktion handelt es sich um die einfachsten Rechenoperationen mit codierten Variablen. Bei einer Multiplikation muss das Ergebnis vor der weiteren Verwendung korrigiert werden, da ein A^2 Term entsteht, dieser muss mit einer Division durch A entfernt werden. Soll eine Division von codierten Zahlen durchgeführt werden, muss die Anpassung einer der Variablen vor der eigentlichen Division durchgeführt werden. Würden zwei AN-codierte Zahlen dividiert, wäre das Ergebnis uncodiert. Dieser Zustand muss während der ganzen Berechnung vermieden werden, denn er öffnet die Möglichkeit für unentdeckbare Fehler. Abhilfe kann geschaffen werden, wenn der Dividend vor der Division mit A multipliziert wird. Die Multiplikation führt jedoch zu einem unerwünschten Nebeneffekt: da die codierte Verarbeitung nur mit ganzzahligen Variablen durchgeführt werden kann, werden bei der Division, sofern Kommastellen entstünden, diese abgeschnitten. Somit kann ein Fall eintreten, in dem Eq. 2.6 gilt.

$$A \cdot Int\left(\frac{A \cdot x_c}{A \cdot y_c}\right) \neq Int\left(\frac{A \cdot A \cdot x_c}{A \cdot y_c}\right) \tag{2.6}$$

Wobei links in Eq. 2.6 die verbotene Division steht, da sich die As in der Klammer kürzen und für die Variable vorübergehend uncodiert vorhanden ist. Rechts ist die zu verwendende Division hinzugefügt, in welcher die Nutzdaten zu keinem Zeitpunkt uncodiert vorhanden sind. Wie sich aber zeigt, führt die rechte, erlaubte Division zu falschen Ergebnissen.
Um dies zu veranschaulichen, ein Beispiel:
Angenommen $A = 5$, $x_f = 11$ und $y_f = 7$ daraus ergibt sich aus dem linken Teil von 2.6 mit $x_c = 55$, $y_c = 35$ die Division $5 \cdot \left(\frac{55}{35}\right) = 5 \cdot (1) = 5$. In der Klammer steht 1, da die Division $55/35$ in Integer 1 ergibt. Wird der rechte Teil von Eq. 2.6 berechnet, ergibt sich $\frac{5 \cdot 55}{35} = 7$, wenn die Nachkommastellen wieder verworfen werden. Man erkennt durch die unterschiedliche Stelle im Rechengang, an welcher die Nachkommastellen verworfen wurden, dass zwei unterschiedliche Ergebnisse entstehen. Würde man die Division uncodiert durchführen, wäre $\frac{11}{7} = 1$ das richtige Ergebnis. Wie sich zeigt, wäre bei der Probe die Division links $mod(5, 5) = 0$ korrekt und rechts mit der für Coded Processing optimierten Division $mod(7, 5) = 2$ falsch. Es ist somit erneut eine Korrektur notwendig. Dazu muss vom Dividend vor der Division um $A \cdot mod(x_c, y_c$ subtrahiert werden, wie in 2.7 ersichtlich.

$$\frac{x_f}{y_f} = Int\left(\frac{A^2 \cdot x_c - A \cdot mod(x_c, y_c)}{A \cdot y_c}\right) \tag{2.7}$$

Vorteil der AN-codierten Berechnung ist die vergleichsweise einfache Implementierbarkeit sowie die Vielzahl an möglichen Rechenoperationen. Ein Schwachpunkt hingegen ist die Erkennung falscher Operanden oder Operatoren. Wird beispielsweise der Operator vertauscht und anstatt

einer Addition eine Subtraktion durchgeführt, ist das Ergebnis weiterhin ein gültiges Codewort und die Probe durch Modulorechnen erkennt den Fehler nicht. Ebenso ist es durch die Probe nicht ersichtlich, ob ein falscher, AN-codierter Wert aus dem Speicher geladen und für die Berechnung verwendet wurde. Um dies zu erkennen muss die Formel 2.3 um eine weitere Konstante erweitert werden.

Wird der AN-Code um eine Konstante erweitert, spricht man von dem ANB-Code, welcher sich auf David Brown [Bro60] zurückführen lässt [Gar66]. Das B wird zu der AN-codierten Variable hinzuaddiert und wird *Signatur* genannt.

$$x_c = A \cdot x_f + B_x \tag{2.8}$$

$$1 < B < A \tag{2.9}$$

Für die Signatur gilt die Bedingung 2.9. Entgegen der Festlegung der unteren Grenze auf Null [Sch11, S. 33] zeigt sich Eins als bessere Wahl, da Eins das neutrale Element der Multiplikation darstellt (weiteres siehe in Anschnitt 4.1).

Das hinzugefügte B ist als Offset zu sehen, welcher die codierte Variable im Bereich zwischen $A \cdot x_f$ und $(A+1) \cdot x_f$ verschiebt. Veranschaulicht für $x_f = 3$ wird dieses Konzept in Abb. 2.10. Durch die Verschiebung um B_x wird jede Variable einmalig. Werden zwei gleiche x_f codiert,

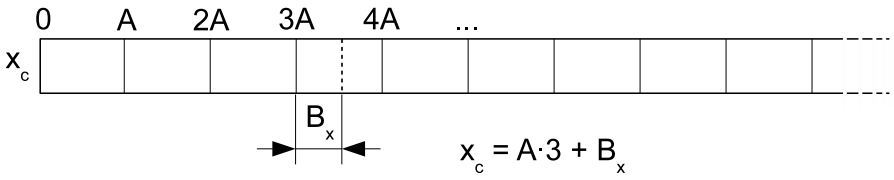

Abbildung 2.10: Prinzip der ANB-Codierung

müssen unterschiedliche Bs verwendet werden. Somit schränkt ein zu kleines A die zur Verfügung stehenden Signaturen ein, woraus folgt, dass die Berechnung niemals mehr als $A - 2$ Variablen verwenden darf.
Diese Codierung ermöglicht es, in der Probe zusätzlich zu kontrollieren, ob die richtigen Variablen verarbeitet wurden. Eine Addition ergibt sich wie in 2.11 aufgeführt.

$$z_c = x_c + y_c = A \cdot x_f + B_x + A \cdot y_f + B_y \tag{2.10}$$

$$= A \cdot (x_f + y_f) + (B_x + B_y) \tag{2.11}$$

Die Probe zur Kontrolle der Berechnung ist wieder durch eine Modulooperation durchzuführen. Verlief die Berechnung korrekt, muss 2.12 erfüllt sein.

$$mod(A \cdot (x_f + y_f) + B_x + B_y, A) = B_x + B_y \tag{2.12}$$

Man erkennt, dass die fälschliche Vertauschung eines Operanden durch eine andere codierte Variable an dieser Stelle auffallen muss, da für jede Variable mit einer anderen Signatur beaufschlagt wird. Das Plus der Addition spiegelt sich ebenfalls in der Probe wider. So kann in einem Rechenschritt überprüft werden, ob auch der richtige Operator verwendet wurde. Bei einer Subtraktion $x_c - y_c$ muss die Modulooperation gleich $B_x - B_y$ sein. Neben dem Vorteil der Erkennung von

falschen Operanden und Operatoren entstehen bei der Durchführung von Multiplikationen und Divisionen ANB-codierter Variablen Probleme.

Werden zwei codierte Variablen multipliziert, sind im Ergebnis die A und B Terme vermischt vorhanden.

$$z_c = x_c \cdot y_c = (A \cdot x_f + B_x) \cdot (A \cdot y_f + B_y)$$
$$z_c = A^2 \cdot x_f \cdot y_f + A \cdot x_f \cdot B_y + A \cdot y_f \cdot B_x + B_x \cdot B_y \tag{2.13}$$

Anstatt des gewünschten Ergebnisses 2.14, mit welchem eine Probe einfach durchgeführt und durch Decodierung auf die Variablen zugegriffen werden könnte, ergibt sich nach der Multiplikation Eq. 2.13, in welcher Codierungsparameter unterschiedlicher Variablem verknüpft sind.

$$z_c = A \cdot (x_f \cdot y_f) + (B_x \cdot B_y) \tag{2.14}$$

Die Korrektur, mit welcher 2.13 auf die Form 2.14 gebracht wird benötigt einen umfassenden Eingriff in das codierte Ergebnis. Die Terme $A \cdot x_f \cdot B_y$, $A \cdot y_f \cdot B_x$ und $B_x \cdot B_y$ müssen durch eine Subtraktion entfernt und $A \cdot B_x \cdot B_y$ hinzuaddiert werden. Eine abschießende Division durch A führt zu der gewünschten Form. Die Division ANB-codierter Variablen ist nicht möglich, diese muss entweder durch eine Subtraktion in Software nachgebildet werden, oder die Variablen werden vor der Division von ANB zu AN-codiert. Dieses Entfernen der Signatur erhöht die Wahrscheinlichkeit für unentdeckte Fehler und muss für höchste Fehleraufdeckung zusätzlich abgesichert werden.

Während der Berechnung verändern sich die Bs stetig, da die Generierung der Signaturen für die Ergebnisse einerseits von den Operanden abhängt und andererseits von der Rechenoperation. Während all dieser Berechnungen ist zu beachten, dass die Bedingung $B < A$ nicht verletzt wird. Bei Addition und vor allem bei Multiplikationen kann dies nicht garantiert werden. Überschreitet die Signatur A, schlägt die Probe fehl. Die Ursache ist in 2.11 anhand einer Addition dargestellt. Man erkennt das Problem an der mit $mod(z_c, A)$ beschrifteten Stelle. Durch das Überschreiten von B über A ist das Ergebnis der Modulo-Probe nicht mehr $B_x + B_y$ sondern $A - B_x + B_y$. Folglich löst die Fehlererkennung aus, obwohl die Berechnung korrekt verlief. In diesem Fall ist eine Signaturkorrektur notwendig bevor vor die Probe durchgeführt werden kann. Wird die Variable dennoch decodiert, ist das Ergebnis x_f nach dem Entfernen der Nachkommastellen um 1 zu groß.
Im Fall einer Addition sieht diese so aus, dass vor der Probe überprüft wird, ob das entstandene B größer A ist. Ist das der Fall, ist die neue soll-Signatur $mod(B_x + B_y, A)$, welche zu z_c hinzuaddiert und von der $B_x + B_y$ abgezogen wird.
Bei der Multiplikation wird die Bedingung $B < A$ noch früher verletzt, somit ist zusätzlich zur Korrektur von 2.13 noch eine Signaturkorrektur von plus $mod(B_x \cdot B_y, A)$ und minus $B_x \cdot B_y$ notwendig.
Die Subtraktion ist von einem ähnlichen Signaturproblem betroffen, jedoch ist nicht $B < A$ Gefahr, sondern dass $B_x - B_y < 1$. Dies führt zu dem Umstand, dass wieder die Probe einen Fehler auslösen würde und die Decodierung ein um 1 zu geringes Ergebnis liefern würde.

Die Vorgestellte ANB-Codierung schützt vor unentdeckten, vertauschten Operanden und Operatoren. Es existiert jedoch ein Szenario, in welchem selbst die ANB-Codierung nicht helfen kann. Ein sogenanntes verlorenes Update ist nur durch ANBD-Codierung aufzudecken.

Man stelle sich vor, es wird ein Ergebnis immer auf die gleiche Speicherstelle geschrieben und im Folgezyklus wieder gelesen. Wenn jetzt während des Speichervorgangs ein Adressfehler auftritt und das Ergebnis an eine falsche Stelle geschrieben wird, jedoch wieder von der ursprünglichen,

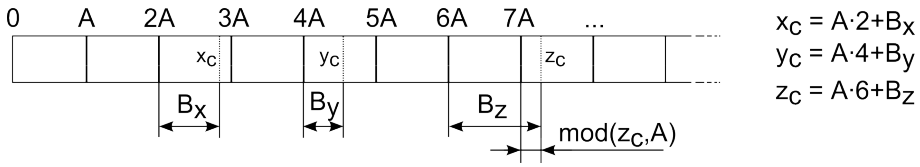

Abbildung 2.11: Fehler durch Signaturüberlauf

an sich richtigen, Stelle gelesen. Das Ergebnis des Vorzyklus ist somit verloren, da mit einem veralteten Wert weiter-gerechnet wird. Dieser Vorfall wird *verlorenes update* genannt und kann nur durch ANBD-Codes aufgedeckt werden. ANBD-Codes gehen zurück auf Forin [For89], wobei das „D“ für *Date* steht. Es handelt sich dabei um einen Zeitstempel, welcher angibt, in welchem Rechenzyklus die Variable das letzte Mal verwendet wurde. Die Codierung erfolgt wie in 2.15 dargestellt. Für B und D gilt die Bedingung $B + D < A$. Dies ist notwendig, da es sonst zu einem Überlauf kommt und Signatur plus Zeitstempel das Ergebnis verfälschen würden. Die Begrenzung durch A ist bei dieser Codierung noch kritischer als bei der ANB-Codierung, da hier pro Variable mehrere Zeitstempel während der Verarbeitung benötigt werden. Ansonsten ist die Codierung der ANB-Version sehr ähnlich, das zeigt sich auch bei den Signaturkorrekturen, nur muss nun zusätzlich der D-Anteil berücksichtigt werden.

$$x_c = A \cdot x_f + B_x + D \tag{2.15}$$

Weitere Codierungsverfahren

Neben den arithmetischen AN-Codes existieren noch mehrere Vertreter, mit welchen eine codierte Verarbeitung durchgeführt werden kann. Bei den beiden vorgestellten Codierungen handelt es sich um systematische, separierbare Codes. Dies bedeutet laut Definition in Abschnitt 2.3 einerseits, dass die Nutzdaten und Prüfdaten getrennt voneinander im Speicher sind und andererseits, dass die Prüfdaten über andere Algorithmen erstellt werden als die Berechnung der Variablen.

Bei dem Berger Code beinhaltet die Prüfsumme die Anzahl der binären Nullen der Nutzdaten [Kor07, S. 66]. Somit können Bit-Flips in die gleiche Richtung (immer von 0 zu 1 oder von 1 zu 0) garantiert erkannt werden. Das Verfahren eignet sich für asymmetrische Sicherung, bei welcher davon ausgegangen werden kann, dass Bit-Flips nur in eine Richtung vorkommen können. Ist die Anzahl der Bitfehler von 0 zu 1 gleich der von 1 zu 0 (oder umgekehrt), so kann dies mit dem Berger Code nicht erkannt werden. Bitfehler in eine Richtung können immer erkannt werden, da zum Beispiel eine beliebige Anzahl an umfallenden logischen Eins die Anzahl der logischen Nullen erhöht. Wenn jetzt zusätzlich die Bits in der Prüfsumme in gleicher Weise von 1 auf 0 wechseln, wird die gespeicherte Anzahl an Nullen geringer, obwohl sie in den Nutzdaten größer wurde. Diese Verfälschung ist somit immer erkennbar.
Die Fehleraufdeckungsmöglichkeiten wie sie bei AN-, ANB-, ANBD-Codes möglich sind, können mit dem Berger Code jedoch bei weitem nicht erreicht werden.

Eine weitere Codierung, ebenfalls mit dem Ziel Rechenfehler aufzudecken, ist der Residue Code. Wie der Name schon sagt, bezieht sich der Residue Code auf den Rest und in diesem Falle der Division. Die Prüfsumme wird durch eine Modulo-Operation generiert und ergibt sich aus

$$x_A = mod(x_f, A)$$

[Sch11, S. 23], [Kor07, S. 76]. Der Parameter A ist nicht mit dem A der AN-Codes aus Abschnitt 2.3 zu verwechseln. Die Wahl von A unterliegt hier anderen Kriterien, obwohl auch hier ein großer Wert die Wahrscheinlichkeit für eine Fehleraufdeckung erhöht.

Wahl von A und B

Für die Auswahl von A und B müssen diverse Randbedingungen berücksichtigt werden, wie zum Beispiel, dass der verfügbare Variablenraum nicht zu stark eingeschränkt wird und die Hamming-Distanz zwischen den codierten Variablen maximiert wird.
Einerseits sollte A so groß wie möglich gewählt werden, um den Abstand zwischen zwei aneinandergrenzenden codierten Variablen so groß wie möglich zu machen, da die Wahrscheinlichkeit einen Fehler nicht zu entdecken $\frac{1}{A}$ lautet [Oze92, S. 185]. Zusätzlich erlaubt ein großes A eine große Anzahl von Signaturen, da die Bedingungen gelten $B < A$ für ANB-Code, beziehungsweise $B + D < A$ für ANBD-Code. Andererseits schränkt ein großes A den verfügbaren Zahlenraum ein, da schnell ein Überlauf der Bitbreite des Prozessors erreicht wird.

Für höchste Fehleraufdeckung ist ein großes A Voraussetzung. Wie aus Tabelle 2.1 ist für SIL 3 eine PFH von 10^{-7} oder kleiner gefordert. Daraus errechnet sich die geforderte Bitbreite als $ld(10^7) \approx 23,25$, wodurch A mindest 24 Bit breit sein muss. Auf einem 64 Bit Prozessor können somit 40 Bit für Nutzdaten verwendet werden. Die starke Einschränkung des Zahlenraumes kann umgangen werden, wenn der Prozessor über eine Streaming SIMD Extensions (SSE)-Einheit verfügt. Mit den 128 Bit breiten Registern ist es möglich 104 Bit für Nutzdaten zur Verfügung zu stellen.
Zusätzlich zur Bitbreite von A gilt es zu beachten, dass nicht nur zwei aneinander grenzende Dezimalzahlen weit auseinander liegen sollen, ebenso sollte die Binärdarstellung aller codierten Zahlen so weit wie möglich von den anderen abweichen. Dieser Bitabstand, auch Hamming-Distanz genannt, welche angibt, in wie vielen Stellen sich zwei Codewörter unterscheiden und wie viele Bitfehler noch detektiert werden können. Die schlechteste Wahl für A wäre eine Potenz von 2. Aufgrund der binären Darstellung, deren Basis 2 ist, wäre eine Multiplikation einer Verschiebung nach links gleichzusetzen [Sch11, S. 94]. Im Allgemeinen wird für A eine Primzahl empfohlen [Bra12, S. 165], da ein A bestehend aus den Faktoren $i \cdot f$ zu einem unentdeckbaren Fehler führt, wenn i mal ein fehlerhafter Vorgang (z. B. eine Schleife) eine Verfälschung in der Höhe f verursacht. Koren [Kor07, S. 76] empfiehlt für $A = 2^a - 1$, wobei es sich bei a um einen Integer handelt und a maximal die vorhandene Bitbreite für die Prüfsumme annehmen darf. Durch diese Wahl von A wird eine vereinfachte Dekodierungsdivision ermöglicht.

Neben der passenden Wahl von A bleibt zu erarbeiten, wie B angewendet werden soll. An sich ist die Auswahlmöglichkeit von B durch A begrenzt, wodurch kein spezielles Auswahlverfahren sinnvoll wäre, welches die möglichen Signaturen weiter einschränkt. Theoretisch könnte den Variablen je nach chronologisch erstem Auftreten immer eine um eins erhöhte Signatur zugeteilt werden als dem vorhergehenden, denn für die Hamming- und arithmetische Distanz ist A verantwortlich. Keine Möglichkeit der Sicherung durch A besteht jedoch, wenn $x_f = 0$ ist. In diesem Fall ist die Wahl von A bedeutungslos und es obliegt der Signatur dafür zu sorgen, dass die codierte Null nicht gleich der uncodierten ist. Siemens [Ric02] sieht dafür eine konstante Signatur von -1 vor, welche alle Binärstellen umdreht. Diese Lösung mag die Null sehr gut absichern, führt jedoch zu gravierenden Einbußen bei der Fehleraufdeckung. Mehr dazu in Abschnitt 3.4.

Codierte Fließkommaberechnung

Die Fließkommaberechnung von codierten Daten ist mit AN-, ANB-, und ANBD-Code nicht möglich. Mit anderen Codierungsverfahren ist eine Fließkommaberechnung im Prinzip möglich. Die Publikation von Jien-Chung Lo beinhaltet einen Vorschlag zur Fließkommacodierung mit Hilfe des Berger Codes [Lo92]. Eine zweite Publikation vom gleichen Autor erweitert das Portfolio der codierten Fließkommaberechnung um Residue Codes erweitert [Lo94]. Der mathematische Aufwand dieser Codierung ist jedoch sehr groß. Der Grund, dass diese Codierungen in dieser Arbeit nicht weiter verfolgt werden, liegt in der Tatsache, dass für die Grundrechenarten in Hardware modifizierte Berechnungseinheiten benötigt werden. Dieser Punkt ist mit der Forderung nach COTS-Hardware nicht vereinbar.

Vergleich der Codierungen: mögliche Rechenarten, Schwächen

Vergleicht man die Codierungen miteinander, erkennt man, dass diese sich in vielen Eigenschaften ähnlich sind. Die Tabelle 2.3 gibt an, ob die jeweilige Berechnung direkt durchgeführt werden kann. Ist ein Feld mit „nein“ versehen, besteht meist die Möglichkeit die Berechnung durch eine andere zu ersetzen. Eine Division kann zum Beispiel durch eine Subtraktion ersetzt werden, jedoch benötigt dieser Ersatz deutlich mehr Rechenzeit.

Tabelle 2.3: Vergleich der Codierungen (Auszug aus [Sch11, S. 36])

	Berger	Residue	AN	ANB	ANBD
Addition	j	j	j	j	j
Subtraktion	j	j	j	j	j
Multiplikation	j	j	j	j	j
Division	j	n	j	n	n
logisches AND	j	j	j	j	j
logisches OR	j	j	j	j	j
logisches NOT	j	j	j	j	j
Ablaufkontrolle	n	n	n	j	j
j...ja, n...nein					

Ein großer Vorteil der ANB und ANBD-Codierung ist die Möglichkeit der Ablaufkontrolle mittels Signaturen. wodurch die Fehleraufdeckung stark erhöht werden kann. Sollen Divisionen durchgeführt werden, sind die zur Verfügung stehenden Codes stark eingeschränkt. Die logischen Operationen AND, OR, NOT können als arithmetische Berechnungen und If-Abfragen implementiert werden.

Error Detection by Data Diversity and Duplicated Instructions

Diese Methode, kurz ED^{4}I genannt, ist eine Softwarelösung zur Erhöhung der Fehlertoleranz gegen Hardwarefehler [Nah02]. Es ist möglich transiente und permanente Fehler aufzudecken, indem die Software mehrfach ausgeführt wird, wobei die mehrfache Ausführung mit unterschiedlichen Zahlen erfolgt. Einmal wird mit den Originalprozessdaten gerechnet und ein weiteres Mal mit

einem Vielfachen derer. Die Prozessdaten x werden, wie in 2.16 gezeigt, mit einer Konstante k zu x'.

$$x' = k \cdot x \tag{2.16}$$

Auf diese Art und Weise wird das gleiche Programm mit gleichem Verhalten mehrfach und intern unterschiedlich auf der gleichen Hardware ausgeführt. Transienten Fehlern wird durch die Mehrfachausführung entgegengewirkt, permanente sollen durch die Verschiebung der Nutzdaten um Faktor k erkannt werden. Nach der Berechnung wird kontrolliert, ob das Ergebnis der zwei Berechnungen um k verschoben ist. Wenn ja, verlief die Berechnung korrekt, wenn nicht, muss ein Fehlerzustand eingenommen werden.
Das Konzept ED^{4}I hat Ähnlichkeit mit der AN-Codierung, denn auch dort wird mit einem Vielfachen der Ursprungsvariablen gerechnet. Der Unterschied liegt in der Argumentation der Fehleraufdeckung. AN-Codierung benötigt ein möglichst großes A für beste Fehleraufdeckung (siehe 2.3). ED^{4}I hingegen basiert auf der diversen Ausführung des Programmes und fordert daher keine große Verschiebungskonstante k. Die Aufgabe von k liegt darin, verglichen mit den Originaldaten, möglichst viele Bitstellen zu invertieren. Das Verfahren ED^{4}I hat Ähnlichkeit mit Recomputing with Shifted Operands (RESO) [PF82], denn hier wird eine zweite Berechnung mit Variablen durchgeführt, welche um k verschoben wurden.

Die Wahl von k ist essentiell für eine hohe Fehleraufdeckungsrate. Um keinen Überlauf zu generieren wurden nur die Zahlen von -5 bis 5 untersucht [Nah02, S. 14]. Es zeigt sich, dass für die unterschiedlichen Hardwarekomponenten unterschiedliche ks das Optimum bilden. Bei Datenübertragungen über den Bus ist $k = -1$ die beste Wahl, bei Matrixmultiplikationen zeigt sich 4 oder -4 am effektivsten. Besteht das Programm hingegen zum größten Teil aus Additionen, wird -2 empfohlen.

Im Gegensatz zu den AN-Codierungen ermöglicht der Aufbau von ED^{4}I eine Sicherung von Fließkommazahlen. Diese muss jedoch anders erfolgen als bei Integer-Zahlen. In Prozessoren werden Fließkommazahlen nach dem ANSI 754 Standard [1] gespeichert und verarbeitet. Diese sieht vor, eine 32 Bit Zahl in ein Vorzeichenbit, 8 Bit Exponent und 23 Bit Mantisse zu zerlegen. Wird hier die Prozessvariable mit k multipliziert, wird möglicherweise nur eine Bitstelle verändert. Somit muss eine andere Lösung gefunden werden, um mantisse- und exponentunabhängig voneinander zu verändern, ohne die Repräsentation der Originalvariable zu zerstören. Vorgeschlagen wird $k = \frac{3}{2}$ für die Mantisse. Dieser Wert besitzt die Eigenschaft, dass kein Underflow entstehen kann, da $k > 1$. Die Wahrscheinlichkeit für einen Overflow ist durch $k < 2$ reduziert. Die Multiplikation mit einem k kleiner 2 führt maximal zu einer Erhöhung des Exponenten um 1. Um zusätzlich den Exponenten zu schützen muss dieser ebenfalls verändert werden. Vorgeschlagen wird eine Addition von 10101010_2 oder 01010101_2 zu dem Exponenten der Originalvariable. Äquivalent kann eine Multiplikation mit $k = 2^{10101010_2}$ oder $k = 2^{01010101_2}$ erfolgen. Verknüpft man die Vorgaben für Mantisse und Exponent, entsteht $k_1 = -\frac{3}{2} \cdot 2^{10101010_2}$ und $k_2 = -\frac{3}{2} \cdot 2^{01010101_2}$, wobei das Minus an der Spitze eingeführt wird um das Sign-Bit zu invertieren.

Messungen mit k_1 und k_2 für Floatingpoint-Operationen haben ergeben, dass die Fehleraufdeckungswahrscheinlichkeit bei mindest 81,7 % liegt [Nah02, S. 31]. Der Bus ist am besten gesichert mit 100%. Bei dem Addierer ergibt sich zusätzlich das Problem, dass mit einer Wahrscheinlichkeit von 2,9% die Datenintegrität verletzt wird.

2.4 Softwarekomponenten

Für das Konzept im Kapitel 3 und die Umsetzung im Kapitel 5 werden spezielle Softwarekomponenten benötigt, welche im Folgenden aufgeführt sind.

Hypervisor

Der Hypervisor ist ein Programm, welches es ermöglicht mehrere Betriebssysteme (Gäste) auf der gleichen Hardware zu betreiben [Fox12]. Dazu werden sogenannte virtuelle Maschinen oder Partitionen erzeugt, in welchen die Betriebssysteme laufen. Diese wissen nichts von der Existenz der anderen. Somit liegt die Aufgabe des Hypervisors darin, virtuelle Maschinen zu emulieren, sodass diese nicht bemerken, dass die Hardware mit anderen Betriebssystemen geteilt wird. Es scheint ihnen als seien sie alleine und hätten vollständigen Zugriff auf die Hardware. Da dem nicht so ist, muss der Hypervisor zwischen den Partitionen isolieren. Dazu ist Isolation der Hardwarekomponenten notwendig, damit eine Partition nicht auf Teile zugreift, welche gerade eine andere Partition benützt. Speicher kann statisch getrennt werden, bei CPU-Zugriffen ist eine dynamische Einteilung notwendig, damit jede Partition einen ausreichenden Anteil an Rechenzeit zugesprochen bekommt. Handelt es ich um ein Single-Core-System, muss der Hypervisor dafür sorgen, dass jedem Gast genügend CPU-Zeit zur Verfügung gestellt wird. Bei Multi-Core-Systemen kann die Aufteilung der Rechenzeit symmetrisch oder asymmetrisch erfolgen. Bei der symmetrischen Verteilung wird jedem Gast ein Kern zugewiesen, welchen er voll auslasten darf. Werden die Kerne nach Bedarf den Partitionen zugewiesen, wird dies asymmetrische Verteilung genannt. Ein Hypervisor für sicherheitskritische Aufgaben besitzt einen Echtzeit-Scheduler, welcher die Rechenzeit so aufteilt, sodass alle Deadlines eingehalten werden können. Ein Hypervisor lässt sich in Typ 1 und Typ 2 einteilen [Man13, S. 298]. Ein Hypervisor vom Typ 1 läuft direkt auf der Hardware und ist als „Minimalbetriebssystem“ ausgelegt. Alle Instruktionen des Typ 1 Hypervisors werden privilegiert ausgeführt und benötigen eine entsprechende Hardwareunterstützung wie Intel-VT [8] oder AMD-V [4]. Ein Hypervisor vom Typ 2 wird innerhalb des Wirt-Betriebssystems betrieben. Vorteile dieser Lösung sind einerseits die vereinfachte Konfiguration des Hypervisors und andererseits, dass die Hardware keine Virtualisierungserweiterungen besitzen muss. Nachteil ist die reduzierte Performance in den Partitionen, da die Hardware nicht nur mit anderen Partitionen geteilt werden muss, sondern auch mit dem vollständigen Gastgeberbetriebssystem. Für Echtzeitanwendungen wird deshalb häufig ein Typ 1 Hypervisor verwendet.

Fehlerinjektionssoftware

Ein gutes fehlertolerantes System sollte bereits anhand des Entwurfs eine nachweisbare Betriebssicherheit besitzen. Um dies zu untermauern bietet sich eine Untersuchung während des Betriebs an. Da Hardwarefehler nur sehr sporadisch auftreten und die Lokalisierung sehr schwer ist, muss ein Mittel gefunden werden um die Effizienz der Tests zu steigern. Häufig werden dazu gezielt Fehler in Hardware oder Software injiziert. Um dies zu bewerkstelligen muss entweder die verwendete Hardware oder Software modifiziert werden. Die ideale Fehlerinjektion gibt genaue Auskunft über die DC des Systems. Ein genaues Modell ist jedoch meist nicht möglich, da der Aufwand zu groß wäre. Zusätzlich darf nicht vergessen werden, dass jede Adaption von Software oder Hardware zu einer Veränderung des Fehlerverhaltens führen kann, wodurch die Tests nicht mehr repräsentativ werden. Die Injektion der Fehler in Hardware und Software wird in folgende Kategorien eingeteilt.

Die mit Abstand effizienteste Methode wäre es, die Fehler in die Hardware einzubauen und die zu testende Software damit zu betreiben [Ben03, S. 28]. Die Aussagekraft dieser Untersuchung ist sehr hoch, leider auch der Aufwand und die Kosten. Für jeden Fehler müsste eine speziell adaptierte Hardware vorliegen. Dies ist bei COTS-Hardware nicht möglich und selbst bei Eigenentwicklungen sehr kostenintensiv. Dem kann teilweise entgegengewirkt werden, indem man auf Debug-Schnittstellen, sofern vorhanden, zurückgreift. Für das Debuggen und Testen der Hardware bauen Hardwareherstellen oft zusätzliche Schnittstellen ein, welche in Normalbetrieb nicht verwendet werden dürfen. Im schlechtesten Fall können diese Debug-Schnittstellen die Systemsicherheit zusätzlich gefährden. Für Untersuchungen bieten sie jedoch Möglichkeit direkt auf Hardwarekomponenten zuzugreifen, diese auszulesen und zu verändern. Dazu muss die Abarbeitung des Programms angehalten, Registerinhalte verändert und weiter ausgeführt werden. Diese hardwarenahe und effiziente Fehlerinjektion setzt jedoch die Zusammenarbeit mit dem Chiphersteller voraus, da die Debug-Schnittstellen oft proprietär sind. Neben diesem existieren noch weitere Nachteile, wie z. B. der Bedarf von spezieller Hardware für die Debug-Schnittstellen und geringe Portabilität. Vorteile dieser Technik wäre (abgesehen von den Injektionsvorgängen) die Echtzeitbetriebsgeschwindigkeit der Tests und dass die Software nicht angepasst werden muss, wodurch eine feldnahe Untersuchung möglich wird. Softwaretools zum Durchführen dieser Injektionen RIFLE [Hen94], FOCUS [CG92], MARS [Fuc96] sind nicht mehr für aktuelle Hardware verfügbar [Ben03, S. 31], weshalb heutzutage von dieser Methode abgesehen wird.
Sofern es möglich ist, bietet es sich an die Hardware in einem Field Programmable Gate Array (FPGA) nachzubauen, somit können an wichtigen Stellen Manipulationselemente eingebaut werden. Für die Untersuchung wird die Software auf dem Prozessor im FPGA ausgeführt und an bestimmten Stellen im Programm werden die Manipulationselemente aktiviert. Im Prinzip können alle Elemente des CPUs nach belieben verändert werden. Hardware, welche Intellectual Property (IP) einer Firma ist kann jedoch nicht zur Untersuchung herangezogen werden, da die internen Strukturen bekannt und veränderbar sein müssen.

Zum anderen können Fehler in die Software eingebaut werden. Dies wäre beispielsweise eine Addition, welche anstatt einer Multiplikation durchgeführt wird. Diese Injektionsstrategie ist kostengünstig und einfach anzuwenden und kann für jede Software angewandt werden. Die Vielseitigkeit zeigt sich auch bei der Abstraktionsebene von der Hardware. Fehler in einzelnen Variablen können vorgespiegelt werden, bis hin zu unzuverlässigen Netzwerkverbindungen. Der Nachteil ist die geringe Aussagekraft der Untersuchung. Fehler können nur in einer Abstraktion von der Hardware durchgeführt werden. Das Fehlerverhalten wie es z. B. ein Bit-Flip auslösen würde lässt sich nicht eruieren. Fehlersimulationen in Software benötigen im Allgemeinen eine längere Ausführungszeit, da während des Tests die Injektionssoftware ausgeführt werden muss. Bei zeitkritischen Untersuchungen kann dies zu einem veränderten Verhalten führen, verglichen mit der fehlerfreien Ausführung. Allgemein ist darauf zu achten, dass die Injektion nicht Ursache für unerwünschte Fehler und Programmverhalten ist. Beinhaltet z. B. die Injektionssoftware einen Programmierfehler, welcher zum Absturz des Programms führt, kann dies leicht auf eine unzureichende Zuverlässigkeit des Hauptprogramms interpretiert werden. Eine weitere Herausforderung ist die Generierung permanenter, virtueller Hardwarefehler. Transiente Fehler lassen sich hingegen effizient erzeugen (siehe Kapitel 5). Fertige Softwaretools wie DOCTOR [Seu95], EXFI [Ben98] für diese Injektionen sind ebenfalls bereits in die Jahre gekommen und für aktuelle Hardware nicht mehr verfügbar.
Eine der wenigen aktuellen Fehlerinjektionssoftwares ist XCEPTION [Ben03, S. 125] [11], diese ist aber aktuell nur für 32 Bit Prozessoren verfügbar, der Entwurf in Kapitel 3 benötigt jedoch 64 Bit.

Aus Ermangelung an verfügbarer und aktueller Fehlerinjektionssoftware muss in Zuge dieser Arbeit eine Eigenentwicklung durchgeführt werden. Aufgrund des Kostenfaktors wird eine Fehlerinjektion auf Softwareebene zurückgegriffen. Die Injektion der Fehler geschieht auf Assembler-Level, welche die hardwarenächste Möglichkeit darstellt Softwarefehler zu simulieren. Die genaue Entwicklung der Software sowie ihre Funktionen sind in Abschnitt 5.3 ausgeführt.

3 Konzept

In diesem Kapitel wird das Konzept der Sicherheitssteuerung vorgestellt, welche die Anforderung nach COTS-Hardware erfüllen und dennoch Zertifizierbarkeit nach IEC 61508 ermöglichen soll.

3.1 Basisannahmen für weitere Arbeit

Für das folgende Konzept müssen Annahmen gemacht werden, diese betreffen Hardware und Software, die Performance, sowie die Umgebung des Systems.

Performance

An die Hardware werden keine speziellen Anforderungen gestellt, da dies die Hardwareunabhängigkeit beeinflussen oder ausschließen würde. Jedoch wird davon ausgegangen, dass die Rechenleistung der Hardware ausreichend ist. Dies bedeutet, sie kann die Berechnungen innerhalb der vom Prozess benötigten Reaktionszeit durchführen. Ist diese Performance durch aktuelle Hardware nicht unmittelbar zu erreichen, stellt dies kein Problem dar. Da die Computer immer schneller werden, kann im Zweifelsfall von zukünftiger, leistungsfähigerer Hardware ausgegangen werden.

Real-Time Anforderungen

Es wird davon ausgegangen, dass die Hardware real-time fähig ist. Bei simplen Prozessoren stellt die Echtzeitanforderung keine Einschränkung dar. Aufgrund des eher einfachen internen Aufbaus können die Ausführungszeiten für Instruktionen genau festgelegt und im Datenblatt nachgeschlagen werden. Bei modernen Desktop-CPUs ist dieses nicht mehr möglich. Der komplexe Aufbau bestehend aus Sprungvorhersage, Pipeline und Caches und anderen nicht deterministischen Elementen kann zu unvorhersehbaren Verzögerungen führen. Ein Beispiel liefert die Quality Assurance Farm von *OSADL* [13]. Trotz weitgehend konstanter Ausführungszeit finden sich bei manchen Hardwareplattformen immer wieder Ausreißer. Diese dürfen die Sicherheit des Systems nicht beeinflussen. Die vorgesehene Lösung sieht in diesem Fall das Ausführen eines sicheren Halt vor. Die Auswirkung von zeitlicher Ungenauigkeit führt somit zu einer verringerten Verfügbarkeit. Wählt der Hersteller eine ungeeignete Hardware, wird dies durch die Software zwar erkannt, kann aber nicht bereinigt werden.

Sicherer Feldbus

Bei dieser Anwendung wird davon ausgegangen, dass die neue Sicherheitssteuerung über keine direkten Schnittstellen zum Prozess verfügt. Die Daten welche die Steuerung generiert, werden über einen sicheren Feldbus an die Ausgangsmodule weitergeleitet. Die Wahl des Feldbusses bleibt dem Hersteller überlassen, eine Zulassung für das benötigte SIL ist jedoch Voraussetzung. Durch das nicht Vorhandensein von direkten Schnittstellen zum Prozess kann der PC, auf welchen die Sicherheitssteuerung betrieben wird, von nur einer Spannungsquelle versorgt werden. Diese Eigenschaft ist Voraussetzung für die Verwendung von COTS-Hardware, denn diese Bauelemente sind nicht für redundante Versorgung ausgelegt.

Für die Wahl des sicheren Feldbusses stehen mehrere am Markt verfügbare Systeme zur Auswahl. Bei den weiteren Betrachtungen wird openSafety [7] verwendet, da es gegenüber anderen Feldbussen für sicherheitskritische Aufgaben unter der BSD-Lizenz [12] frei verfügbar ist und mit anderen (nicht sicheren) Feldbussen kombiniert werden kann.

Sichere Eingangs-/Ausgangsklemmen

Es wird davon ausgegangen, dass sichere, zertifizierte Eingangsklemmen verwendet werden. Das Klemmmodul ist verantwortlich Leitungsunterbrechungen und Kurzschlüsse zu Masse/Versorgungsspannung zu erkennen und die Sicherheitssteuerung über die Anomalie zu informieren. Die Ausgangsklemme verfügt im einfachsten Fall über einen Schalter der die Spannungsversorgung von dem zu sichernden Gerät trennt. Wie dies im Detail geschieht, ist nicht weiter von Relevanz, üblich sind jedoch zwei Relais in Serie geschaltet, so kann bei einem defekten Relais dennoch abgeschaltet werden.

Fehlerfreie Software

Es wird vorausgesetzt, dass die zertifizierte Software keinerlei Fehler beinhaltet, wobei keine Anforderungen an den Entwicklungsprozess gestellt werden. Es ist aber davon auszugehen, dass die Software nach IEC 61508 Teil 3 [IEC10a] entwickelt werden muss, wodurch systemische Fehler zu einem großen Teil verhindert werden sollen. An die nicht zertifizierte Software können keine Anforderungen gestellt werden. Die unzertifizierte Software ist genauso zu behandeln wie die unzertifizierte Hardware, es muss jederzeit und überall mit einem Fehler gerechnet werden, welcher sich aufgrund des Gesamtkonzeptes nicht auf die Sicherheit auswirken darf.

Verhalten im Fehlerfall

Tritt ein Fehler im sicheren Steuerungssystem auf, wird der *Fail Silent* Zustand eingenommen. Dieser zeichnet sich dadurch aus, dass das Gerät im Fehlerfall keine Ausgangsdaten generiert und einen passiven Zustand einnimmt. Es ist nun Aufgabe des sicheren Ausgangsknoten die Energieversorgung von dem zu sichernden Element zu trennen. Dieser Vorgang, auch *Safe Torque Off* genannt, wird beim Ausgangsknoten eingeleitet, wenn nach einer vorher definierten Zeitspanne keine Daten mehr eintreffen.

3.2 Architektur

Soll unzertifizierte Hardware für eine Sicherheitssteuerung verwendet werden, muss von einer HFT Null, bzw. von einem einkanaligen System ausgegangen werden. Dies ist selbst der Fall, wenn eine Multi-Core CPU verwendet wird. Soll diese als mehrkanalig argumentiert werden, muss die Unabhängigkeit zwischen den Kanälen nachgewiesen werden. Dies ist theoretisch möglich, da die IEC 61508 [IEC10a, Teil 2, Annex E] strenge Regeln beinhaltet um von sogenannter On-Chip-Redundanz auszugehen. Diese Anforderungen sind jedoch für COTS-CPUs aus mehrerlei Gründen nicht einhaltbar. Erstens handelt es sich bei den Prozessor-Interna um IP wodurch die Einhaltung der Vorschriften in der Norm nicht nachgewiesen werden kann. Zweitens wird ein COTS-CPU aus wirtschaftlichen Gründen nicht nach den Regeln der Norm entwickelt und produziert und drittens würde eine Argumentation von On-Chip-Redundanz zu einer Hardwareabhängigkeit führen. Erschwerend kommt hinzu, dass die restlichen Hardwarekomponenten neben der CPU ebenso nur ein Mal vorhanden sind, wodurch ein redundantes System nicht realisierbar ist. Ausgangspunkt für den Konzeptentwurf ist somit das Design in [Wra10, S. 96], welches einen Vorschlag für ein einkanaliges Sicherheitssystem beinhaltet. Es besteht aus einer Hardware, welche das gleiche Sicherheitsprogramm in diversitärer Form mehrfach abarbeitet. Für eine gesteigerte Fehleraufdeckung ist ein zusätzliches Vergleicher-Element hinzugefügt.
Um für dieses Konzept die geforderte SFF von $\geq 99\,\%$ (siehe Tabelle 2.2) zu erreichen, wurde auf das Prinzip des Coded Processing zurückgegriffen, welches laut IEC 61508 [IEC10a, Teil 2, Tabelle 4.A] selbst bei einkanaliger Verarbeitung einen hohen Diagnosedeckungsgrad ermöglicht. Der logische Aufbau entspricht dem in Abb. 2.1. Dennoch besitzt diese softwaremäßige Überwachung nicht tolerierbare Unzulänglichkeiten bei der Kontrolle der externen Parameter. Dazu ist es nötig, dass zusätzliche Komponenten zur Überwachung der PC-Umgebungsparameter hinzugefügt werden.

Die daraus entwickelte Architektur besteht zum größten Teil aus Software und einer möglichst generisch gehaltenen, zertifizierten Hardwareerweiterung zur Überwachung. Das High-Level Konzept ist in Abb. 3.1 dargestellt. Es basiert auf einer COTS-Hardware, auf welcher ein Hypervisor betrieben wird. Dieser stellt die Isolation für die beiden Sicherheitsanwendungen zur Verfügung und ermöglicht eine doppelte Ausführung der Sicherheitsapplikationen. Optional kann ein General Purpose Operating System (GPOS) in einer weiteren Partition des Hypervisors betrieben werden. Die Sicherheitsanwendungen sind divers implementiert, wobei eine Partition codierte Daten (siehe Abschnitt 2.3) verarbeitet. Die Hardwareerweiterung, genannt *Watchdog*, ist über eine Peripheral Component Interconnect Express (PCIe)-Schnittstelle [14] an den PC angebunden. Dieser ist zur Überwachung der Hardware und der Software eingesetzt (siehe Abschnitt 3.3). Die Anbindung an die Eingangs-Ausgangsmodule erfolgt über einen sicheren Feldbus.
Auf die Details der Komponenten wird nun eingegangen.

Die Isolation des Hypervisors sorgt dafür, dass die Sicherheitspartitionen getrennt voneinander betrieben werden können. Somit ist eine doppelte Ausführung von Programmen auf der selben Hardware möglich. Das entstehende System ist virtuell zweikanalig. In diesem Fall wird ein Typ-1 Hypervisor verwendet. Durch diesen Aufbau wird eine Informationsredundanz erreicht, da die zwei Sicherheitsprogramme getrennt voneinander arbeiten und jede Partition ihren zugeteilten Speicherbereich besitzt. Somit sind die Daten doppelt, in jeder Partition ein Mal, vorhanden. Des weiteren wissen sie nichts von der Anwesenheit der anderen oder des Hypervisors. Da in diesem Aufbau der Hypervisor nur einmal vorhanden ist und beide Sicherheitsprogramme bedient, können sich Fehler darin kritisch auf die Systemsicherheit auswirken. Die Verwendung eines zertifizierten Hypervisors hat den Vorteil, dass diese Software bereits IEC 61508 SIL 3 zertifiziert

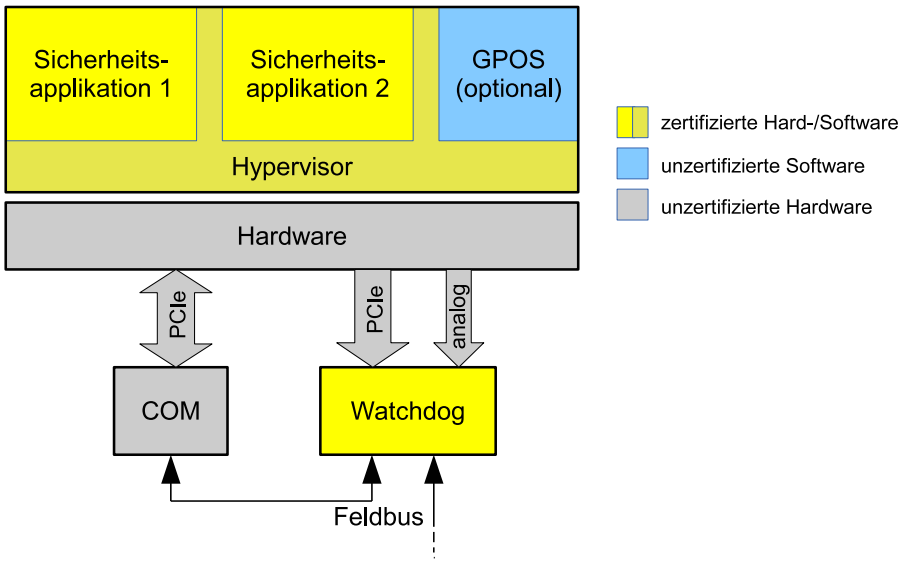

Abbildung 3.1: High Level Architektur der Sicherheitssteuerung

am Markt erhältlich ist [18, 20].

Die Sicherheitsanwendungen beinhalten die Steuerungsregeln für den zu sichernden Prozess. Zusätzlich sind hier die Mechanismen eingebaut, welche Fehler in der Hardware oder darunter liegenden Software erkennen muss. In beiden Programmen werden die sicherheitskritischen Daten unabhängig voneinander verarbeitet. Das Konzept sieht vor, dass eine der Anwendungen mit codierten Daten rechnet und die andere mit uncodierten. Auf diese Art wird Diversität erreicht, wodurch es sehr unwahrscheinlich ist, dass die Programme die gleichen Befehle zur gleichen Zeit ausführen. Dieser Fall darf nicht auftreten, da ein Fehler in beiden Kanälen die gleiche Veränderung hervorrufen kann und dies in einem Vergleich der Ergebnisse nicht erkennbar ist. Der uncodierte Kanal rechnet zusätzlich zu den Steuerungsdaten noch die Signaturen des codierten Kanals mit. Bei ANB- und ANBD-Codes ist es möglich, die Signaturen unabhängig von den Nutzdaten zu berechnen, wodurch eine sehr feinmaschige Ablaufkontrolle machbar ist.

Der Vergleich der Signaturen wird zusätzlich zum internen Vergleich im Watchdog ausgeführt. Aufgrund seiner autonomen Hardware und Software kann davon ausgegangen werden, dass sich kein Mehrfachfehler einschleicht, welcher Hauptprogramm und Watchdog in gleicher Weise beeinflusst und ein Rechenfehler nicht erkannt wird. Tritt dieser äußerst unwahrscheinliche Fall doch ein, müssen die Daten noch den sicheren Ausgangsknoten passieren, welcher ebenso die Daten vergleicht. Diese Argumentation basiert auf einer Eigenschaft des openSafety-Protokolls [7], welche besagt, dass die Daten doppelt übertragen werden müssen. Es werden die Daten jeder der zwei Sicherheitsanwendungen in einem openSafety-Paket an den Empfänger geschickt. Dieser überprüft die Gleichheit; ist diese nicht gegeben, wird das Paket als fehlerhaft gewertet und wird ignoriert. Kommt über eine bestimmte Zeitspanne kein gültiges Paket, wird ein Sicherheitsstopp ausgeführt.

Die Kommunikationsschnittstelle (COM-Knoten) ist für die Datenverbindung zwischen Sicherheitssteuerung und Eingangs-/Ausgangsmodulen zuständig. Dieser Knoten verpackt die von der Sicherheitsanwendung generierten sicherheitskritischen-Daten in ein Feldbus-Paket.
Der COM-Knoten ist nicht zertifiziert, denn Fehler, die hier entstehen, führen entweder zu einem

Time-out bei der Übertragung oder einem Bruch des sicherheitskritischen-Protokolls, welche beim Empfänger detektiert werden können.

Die Daten werden, wie in jedem steuerungstechnischen System, in der Reihenfolge Eingangsabbild empfangen, Steuergrößen berechnen, Stellgröße ausgeben, verarbeitet.
Im Detail geschieht dies für sicherheitskritische Daten folgendermaßen: Ein sicheres Eingangsmodul sendet die Daten in einem sicherheitskritischen-Paket an den PC. Das Datenpaket passiert den Watchdog ohne betrachtet zu werden. Im COM-Knoten werden die Daten entpackt und an die Sicherheitsanwendung 1&2 geschickt. Diese berechnen die Stellgrößen unabhängig voneinander. Die Ergebnisse werden nun per PCIe an den Watchdog gesendet, dieser überprüft die Gleichheit. Nun werden die Daten als sicherheitskritisches Datenpaket an den COM-Knoten gesendet, welcher sie in das Feldbus-Datenpaket einfügt. Dieses Datenpaket wird über den Feldbus gesendet und muss dabei den Watchdog passieren. Ist während der Berechnung eine Anomalie aufgetreten oder sind die diversitär berechneten Ergebnisse nicht gleich, wird das Paket und alle folgenden Safety-Pakete gesperrt. Genaueres zur Überwachung durch den Watchdog in Teil 3.3.

Für nicht sicherheitskritische Daten ist der Pfad ähnlich, jedoch werden sie an das GPOS weitergeleitet und zu jedem Zeitpunkt vom Watchdog ignoriert.

3.3 Watchdog

Bei dem Watchdog handelt es sich um eine zertifizierte Hardware, welche zusätzlich zur codierten Datenverarbeitung die Fehleraufdeckung erhöhen soll. Die Einführung einer zertifizierten Hardware widerspricht auf den ersten Blick der in Kapitel 1 beschriebenen Aufgabenstellung. Dies trifft jedoch nicht zu, da es das Ziel ist unzertifizierte Hardware für sicherheitskritische Aufgaben verwenden zu dürfen. Durch die Ausführung sicherheitskritischer Programme auf nicht zertifizierter Hardware ist es möglich tagesaktuelle und rechenstarke PC-Komponenten verwenden zu können. Diese Chance wird durch den Einsatz eines zertifizierten Watchdogs nicht eingeschränkt. Es kann weiterhin unzertifizierte Hardware für den PC verwendet werden, sofern sie die benötigten Anschlüsse für die benötigten Diagnosefunktionen besitzt. Die im Folgenden an den Watchdog gestellten Anforderungen sind nicht sehr rechenintensiv, wodurch ein zertifizierter Watchdog für eine Vielzahl von gegenwärtigen und zukünftigen Industrie-PCs eingesetzt werden kann.

Externe Diagnoseeinheit

In einer ersten Untersuchung wurde eine externe Diagnoseeinheit untersucht, welche nur am sicheren Feldbus angeschlossen ist. Diese wäre somit ein Busteilnehmer wie alle anderen sicheren Eingangs- und Ausgangsknoten auch. Vorteil ist die leichte Erweiterbarkeit des sicherheitskritischen Systems, da der Industrie-PC nicht verändert werden muss.
In dieser Position könnte der Watchdog nur Datenpakete überprüfen, welche bereits gesendet wurden und für alle Busteilnehmer zugänglich sind. Folglich ergibt sich das Problem, wie der Watchdog im Falle eines erkannten Fehlers reagieren soll. Würden die sicheren Eingangs- und Ausgangsknoten auf eine Bestätigung der Richtigkeit der Daten vom Watchdog warten, wäre einerseits die Reaktionszeit erhöht, wie auch das Datenaufkommen am Feldbus. Die Ursache dieser beiden Probleme liegt darin, dass der Watchdog spezielle Zustimmungs- oder Fehlerpakete bei jedem im PC erzeugten sicherheitskritischen Datenpaket schicken muss. Ein weiterer Nachteil dieses Entwurfs ist es, dass die Software in den Empfängern angepasst werden muss, denn diese

müssten nun mit der Verarbeitung der Daten warten bis der Watchdog zugestimmt hat. Der Anforderung des Fail-Silent Zustands kann ebenso nicht nachgekommen werden. Sendet der PC aufgrund eines Fehler dauerhaft falsche Daten, müsste der Watchdog auf jedes Datenpaket mit einer Fehlernachricht reagieren, wodurch der Feldbus weiter belastet wird.
Verfolgt man die Idee des Watchdogs am Feldbus von diesem Standpunkt aus weiter, könnte man vorschlagen, dass gleich die Datenempfänger die empfangenen Daten auf Richtigkeit prüfen. Dies ist naheliegend und würde zu keiner zusätzlichen Netzwerklast führen. Trotz des Vorteils des eingesparten Watchdogs muss auch hier die Software jedes Datenempfängers angepasst werden. Da diese üblicherweise ebenso zertifiziert ist dies eine sehr kostenintensive Aufgabe.
Sieht man von diesen –wenn auch schwer– lösbaren Problemen ab zeigen sich in der IEC 61508 Teil 2 Annex A eine Vielzahl von Fehlermöglichkeiten und vorgeschriebenen Diagnosefunktionen, welche bisher nicht beachtet wurden. Beispielsweise ist es nicht möglich Fehler am Taktgeber, der Spannungsversorgung oder unzulässige Temperaturen im Gehäuse über den Feldbus zu diagnostizieren. Aus diesen Gründen wurde von der Verwendung eines externen Watchdogs abgesehen und die im Gehäuse des Industrie-PCs implementierte Lösung weiter verfolgt.

Interner Watchdog

Um die Unzulänglichkeiten einer externen Diagnoseeinheit zu beseitigen, wurde eine im PC-Gehäuse verbaute Lösung gefunden. Der Anschluss an den PC erfolgt über die PCIe-Schnittstelle. Diese ist bei vielen aktuellen PCs vorhanden, wodurch eine einfache Portierbarkeit der Hardware erreicht wird. Die Funktionen des Watchdogs gehen weit über die eines Timers, wie man ihn von Mikrocontrollern kennt, hinaus. Seine Aufgabe hier ist vielfältig und er ist zur Überwachung der analogen und digitalen Domains eingesetzt. Für die digitale Überwachung besitzt er die PCIe-Anbindung, für die analoge Überwachung sind weitere Anschlüsse vorgesehen.

Der Watchdog arbeitet passiv, er generiert keine Daten. Das Prinzip beruht darauf, dass der Watchdog nur Daten empfängt und unabhängig vom restlichen PC agiert. Wird eine Abweichung jeglicher Art erkannt wird niemand (Sicherheitsanwendungen, COM-Knoten,..) informiert, es werden nur die ausgehenden, sicherheitskritischen Pakete gesperrt. Voraussetzung dafür ist, dass die Diagnosefunktionen schneller durchgeführt werden können, als das Erstellen des sicherheitskritischen Pakets. Optional kann im Fehlerfall der ganze Netzwerkverkehr gesperrt werden. Dies erschwert die Wartbarkeit des Systems, da im Fehlerfall keine Diagnose aus der Ferne (über das Netzwerk) durchgeführt werden kann. Es müsste eine Techniker vor Ort die Wartungsarbeiten durchführen. Vorteil hingegen wäre, dass der Watchdog noch einfacher aufgebaut werden kann, wenn er nicht zwischen sicherheitskritischen und nicht kritischen Paketen unterscheiden muss. Die Unterscheidung impliziert, dass der Watchdog jedes Paket bis zu einem gewissen Grad interpretieren muss, um festzustellen, ob es sicherheitskritische Daten beinhaltet. Die Komplexität des Watchdog wird durch die Paketfilterung gesteigert, als wenn einfach der ganze Netzwerkverkehr unterbrochen werden würde.

Die einzelnen Aufgaben des Watchdogs sind den Anforderungen der IEC 61508 [IEC10a, Appendix A, Tabelle A.1 bis A.18] angepasst und ergänzen die Fehleraufdeckung durch Coded Processing:

- Ausgangsspannung des Netzteils messen
- CPU-Kernspannung messen
- Temperatur messen

- Signaturen der Berechnung überprüfen, inklusive Flusskontrolle der Sicherheitsanwendungen
- Vergleichen der Ergebnisse zwischen den Sicherheitsanwendungen
- sicherheitskritische Pakete freigeben oder sperren

Um diese Aufgaben zu erfüllen, müssen zusätzliche zum PCIe noch weitere Anschlüsse hinzugefügt werden. Eine Auflistung, inklusive kurzer Beschreibung, sowie deren Informationsflussrichtung ist in Tabelle 3.1 dargestellt.

Tabelle 3.1: Schnittstellen des Watchdogs

Beschreibung	analog/digital	Richtung
2x Feldbus für Buswächteraufgabe	digital	Input/Output
PCIe für Datenanbindung	digital	Input
Anschlüsse für Temperaturmessung	analog/digital	Input
Anschlüsse für Betriebsspannungsmessung	analog	Input
Anschlüsse für CPU-Spannungsmessung	analog	Input
Anschlüsse für CPU-Spannungsmessung über SMBus	digital	Input

Betriebsspannung messen

Watchdog überwacht die Sekundärspannung des Netzteils, indem es direkt mit dem Watchdog verbunden wird. Der auf dem Watchdog verbaute ADC wird zyklisch auf korrekte Funktion überprüft. Dazu kann ein bekannter Spannungsteiler gemessen werden und die Ergebnisse mit den erwarteten verglichen, wie bereits in Abschnitt 2.1.8 und Abb. 2.7 erklärt.

CPU-Kernspannung messen

Watchdog überwacht die Kernspannung über die V_{cc}- und V_{ss}-Sense-Anschlüsse der CPU [Int10, S. 69]. Auch hier wird der ADC zyklisch auf korrekte Funktion überprüft. Anschluss des Watchdog an die entsprechenden Pins der CPU direkt über Drähte und Steckverbindungen. Leitungen werden nicht über das Mainboard geroutet. Somit wird eine mögliche Failure Mode and Effects Analysis (FMEA) vereinfacht, außerdem handelt es sich hierbei um analoge Leitungen, welche nicht über den PCIe zum Watchdog geleitet werden könnten.

Temperaturmessung

Der Watchdog überwacht die Temperatur im Gehäuse, im CPU-Kern und an weiteren relevanten Punkten. Anschluss der Temperatursensoren entweder analog (z. B. temperaturabhängige Widerstände) oder digital (z. B. Temperatursensoren mit Inter-Integrated Circuit (I^2C)-Anschluss). Bei analoger Beschaltung kann der ADC getestet werden, bei I^2C steht diese Möglichkeit nicht zu Verfügung, da der Sensor und Digitalisierung in einem fertigen Baustein integriert sind.
Die Kerntemperatur der CPU wird über den System Management Bus (SMBus) ausgelesen. Der Watchdog kann diese Messung nicht auf Gültigkeit überprüfen, oder die Funktionstüchtigkeit nachweisen, da sie auf unzugänglichen Bauteilen innerhalb des CPU-Cores basiert.

Ablaufkontrolle

Der Watchdog bekommt in fix vorgegebenen Zeitabständen Daten von den beiden Sicherheitsanwendungen zugeschickt, bei welchen die Gleichheit überprüft wird. Dazu muss er die uncodierten Daten aus einem Pfad codieren und mit dem codierten Zwischenergebnis vergleichen. Auf diese Art lässt sich eine sehr genaue Ablaufkontrolle für die Sicherheitsanwendungen realisieren. Wird ein Ergebnis zu früh, zu spät oder falsch übermittelt, kann von einem Fehler ausgegangen werden. Da der Watchdog einen eigene Zeitbasis (eigenen Quarz) besitzt, kann eine schleichende Veränderung des CPU-Quarzes festgestellt werden. Rückmeldung an die CPU und die Software über die Diagnose- und Vergleichsergebnisse wird nicht benötigt.

Feldbus-Pakete freigeben

Jedes Paket, welches gesendet oder empfangen wird, muss den Watchdog passieren. Eingehende Pakete werden ohne Kontrolle durch den Watchdog an den COM-Knoten weitergeleitet, genauso wie ausgehende, nicht sicherheitskritische Pakete immer durchgeschleift werden. Ausgehende, sicherheitskritische Pakete müssen gezielt vom Watchdog freigegeben werden. Dies geschieht nur, wenn während der Berechnung keine Auffälligkeiten (z. B. Spannungsschwankungen, falsche Signaturen) erkannt wurden. Wurde ein Problem erkannt, wird kein sicherheitskritisches-Paket mehr freigegeben. Die nicht sicherheitskritische Kommunikation wird nicht beeinflusst. Dies ermöglicht weiterhin Diagnosefunktionen zum Rückverfolgen des Ausfalls. Außerdem soll es möglich sein, dass das GPOS durch den Hypervisor getrennt von der Sicherheitsanwendung weiterhin arbeitet.

3.4 Codierung

Eine der beiden Sicherheitsanwendungen arbeitet den Programmcode mit codierten Daten ab. Die Ursache liegt darin, da trotz der Quellcode-Diversität in den Kanälen die Nutzdaten in gleicher Form in beiden virtuellen Berechnungseinheiten vorliegen. Dies kann verhindert werden, wenn in einer Partition Daten, repräsentiert durch ein anderes Bitmuster, verarbeitet werden. Dazu wurde hier eine Codierung in einem der Kanäle gewählt.

Im Zuge des Konzeptentwurfs wurden ANB-Codierungen genauer untersucht (siehe Abschnitt 2.3). Diese zeichnet sich durch eine hohe Fehleraufdeckung und dennoch handhabbarem Mehraufwand aus. Die Codes mit Signaturen (ANB und ANBD) haben gegenüber den anderen Codierungen (AN, Berger-, Residue-Code) den großen Vorteil, dass der Programmfluss anhand der Signaturen von außen überwacht werden kann, ohne dass die Nutzdaten dem Überwacher bekannt sein müssen. Somit kann die überwachende Einheit über deutlich weniger Rechenleistung verfügen als für die Hauptberechnungen notwendig sind.

Die Entscheidung zwischen ANB und ANBD-Codes wurde anhand der Untersuchung in [Sch11, S. 160] getroffen und ist in Tabelle 3.2 ersichtlich. Die Tabelle zeigt die Fehleraufdeckungsrate, im Gegensatz zu der Silent Data Corruption (SDC) wie sie in der Datenquelle [Sch11, S. 160] vorhanden sind. Ein kritischer Fehler tritt ein, wenn das Endergebnis falsch ist, jedoch keine der Fehleraufdeckungsmaßnahmen angeschlagen haben. In diesem Falle bestehen die Aufdeckungsmaßnahmen aus einer Probe der jeweiligen Berechnung, wie sie in Abschnitt 2.3 beschrieben wird. Zusätzlich wird die Laufzeit des Programms gemessen; dauert die Berechnung überproportional lange, wird von einem Fehler im Kontrollfluss ausgegangen und das Programm durch eine kontrollierende Instanz beendet.

Bei den in Tabelle 3.2 eingetragenen Berechnungen handelt es sich um

- topK: suche die K häufigsten Elemente in einem Datenstrom,
- abs: Berechnung eines Antiblockiersystems,
- tcas: „Traffic Alert and Collision Avoidance System“, Kollisionswarnsystem für die Luftfahrt,
- primes: Primzahlsuche nach dem Sieb des Eratosthenes,
- pid: Implementierung eines Proportional-Integral-Differential-Reglers,
- md5: Hashsummenberechnung nach dem „Message-Digest Algorithm 5“.

Tabelle 3.2: Vergleich der Codierungen, Angabe der Fehleraufdeckung in Prozent [Sch11, S. 160]

Berechnung	Fehlerart	uncodiert	AN	ANB	ANBD
topK	deterministisch	99,349	99,954	99,996	100
	probabilistisch	98,827	100	100	100
	permanent	81,01	100	100	100
abs	deterministisch	65,544	93,084	99,657	99,588
	probabilistisch	69,226	99,783	100	100
	permanent	80,754	100	100	100
tcas	deterministisch	92,585	98,279	99,557	99,966
	probabilistisch	92,514	99,867	99,996	100
	permanent	71,988	99,954	99,996	100
primes	deterministisch	69,616	99,986	99,994	100
	probabilistisch	64,769	99,95	99,983	100
	permanent	61,932	99,166	100	100
pid	deterministisch	78,789	99,262	99,811	99,881
	probabilistisch	74,076	99,983	99,983	100
	permanent	34,856	100	100	100
md5	deterministisch	56,428	97,196	99,871	99,92
	probabilistisch	61,408	100	100	100
	permanent	98,404	100	100	100

Zur besseren Übersicht und Vergleichbarkeit werden die Daten aus Tabelle 3.2 in der Abb. 3.2 graphisch dargestellt.
Die uncodierten Daten weisen eine stark schwankende Fehleraufdeckung auf. Permanente Fehler beim Proportional-Integral-Differential-Regler weisen nur eine Aufdeckung von ca. 34 % auf, wobei hingegen deterministische Fehler im topK-Programm mit einer Wahrscheinlichkeit von über 99 % erkannt werden. Wird eine AN-Codierung verwendet, steigt die Aufdeckung bei allen Anwendungen stark an, wobei die geringste Fehleraufdeckung bei deterministischen Fehlern in der Antiblockiersystem (ABS)-Anwendung sich mit ca. 93 % zu Buche schlägt. Dieser Wert wäre nach Tabelle 2.2 für SIL 2 ausreichend, für SIL 3 jedoch zu gering. Die Erweiterung der Codierung um eine Signatur birgt eine signifikante Verbesserung im Vergleich zur AN-Codierung. Der vorherige, schlechte Wert von ABS mit deterministischem Fehler wird auf über 99,6 % gesteigert. Die

ANB-Codierung erfüllt in jedem der Beispiele die Vorgabe der Fehleraufdeckung von über 99 %. Für höchste Sicherheit könnte nun auf ANBD-Codierung gewechselt werden. Dies ist jedoch nicht rentabel, da der deutlich erhöhte Implementierungs- und Performanceaufwand in keiner praktikablen Relation zu der gesteigerten Fehleraufdeckung steht. Um diesen marginalen Unterschied zwischen ANB- und ANBD-Codierung darzustellen ist in Abb. 3.3 der Bereich von 99 % bis 100 % vergrößert dargestellt.

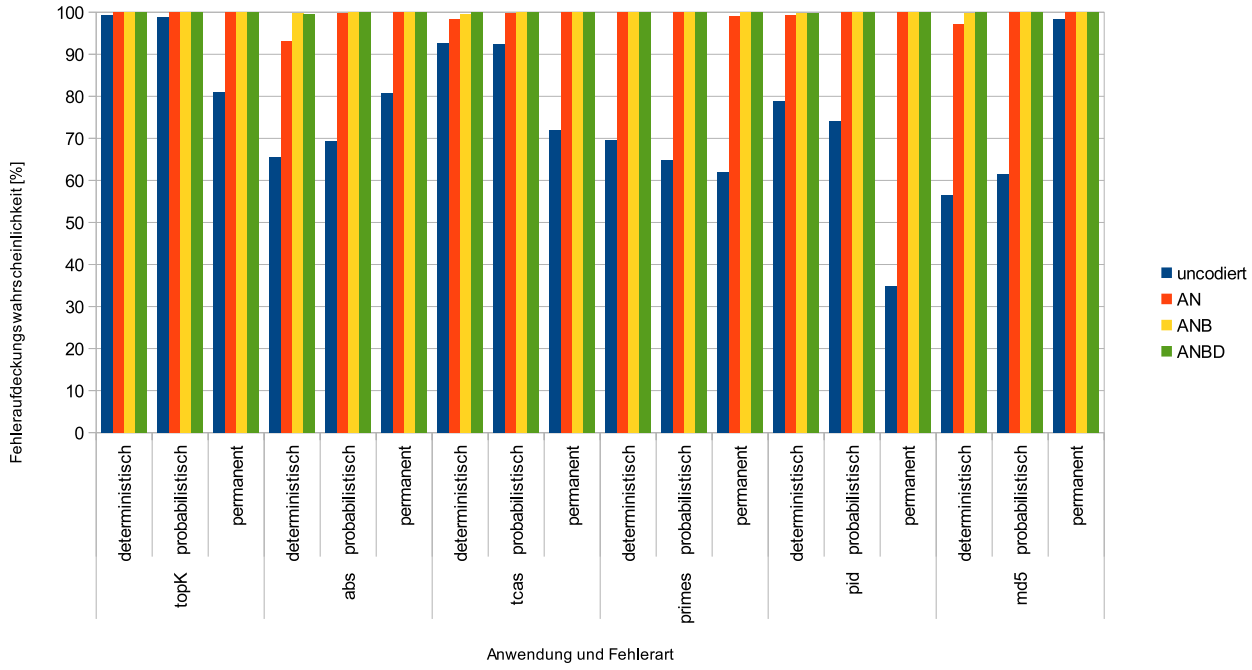

Abbildung 3.2: Vergleich der Fehleraufdeckung verschiedener Codierungen

Die Wahl der Parameter A und B geschieht entsprechend den Vorschriften in Teil 2.3. Beim Anlegen einer codierten Variable wird die Signatur mittels eines Pseudozufallsgenerator erstellt. Diese Pseudozufälligkeit ist notwendig, da die parallel rechnende Partition die gleichen Zufallsentscheidungen treffen muss, um eine Ablaufkontrolle anhand der Signaturen zu ermöglichen. Zusätzlich wird das Debuggen der Anwendung vereinfacht, da die Zufallsreihenfolge nachverfolgt werden kann. Das Alter einer Signatur kann bestimmt werden, wodurch in gewisser Weise eine ANBD-Codierung entsteht, jedoch ohne zusätzlichen Rechen- oder Programmieraufwand.
Von einer konstanten Signatur, wie sie in dem Patent DE10219501 [17] vorgeschlagen wird, wurde Abstand genommen. Die Implementierung und Ausführung des Programms wird zwar vereinfacht, jedoch entstehen viele Verwundbarkeiten des Codes. Einerseits besteht nicht die Möglichkeit das Alter einer Variable zu eruieren, wie es bei pseudozufälligen Signaturen möglich ist, andererseits heben sich zwei gleiche Signaturen bei einer ANB-codierten Subtraktion auf, wodurch das Ergebnis nur mehr AN-Codiert ist.

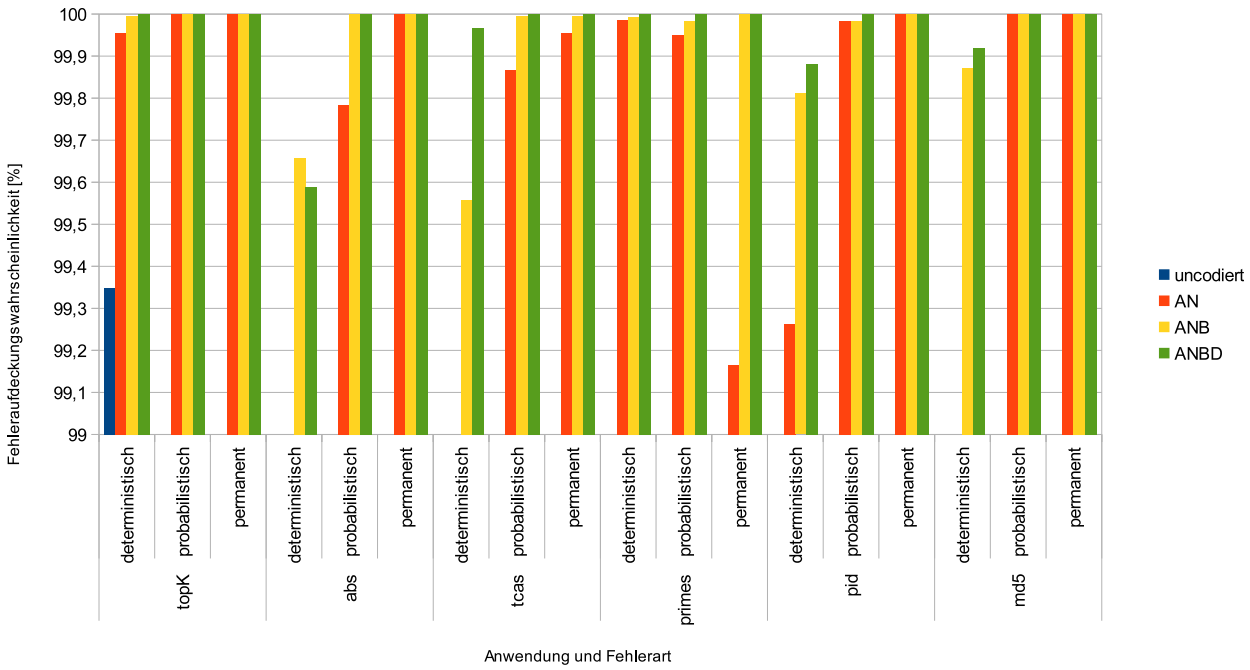

Abbildung 3.3: Vergleich der Codierungen mit einer Fehleraufdeckung größer 99 %

4 Konzeptanalyse

Das zuvor vorgestellte Konzept muss den Anforderungen der Normen genügen. Dazu wurden Fehlermodelle entwickelt, welche in diesem Kapitel vorgestellt und untersucht werden. Die folgende Analyse ist keine vollständige Argumentation um eine Zertifizierung durchzusetzen. Dies würde den Rahmen der Arbeit sprengen, dennoch gibt dieses Kapitel Argumente wieder, welche für eine Zertifizierung ausgebaut werden können.

Die Sicherheitsgrundpfeiler des Konzepts sind die Datenredundanz und die Zeitredundanz. Datenredundanz ist durch zwei Mittel vorhanden, einerseits Coded Processing, andererseits durch die doppelte Verarbeitung. Coded Processing fügt jeder Variablen redundante Informationen hinzu, um die Wahrscheinlichkeit eine Verfälschung aufzudecken zu erhöhen. Zusätzlich sind alle Daten doppelt vorhanden, da der uncodierte Kanal ebenfalls über die vollständigen Prozessdaten verfügt. Die doppelte Verarbeitung durch Kanal 1 und 2 führt zu einer Zeitredundanz. Es kann ausgeschlossen werden, dass im gleichen Taktzyklus die gleichen Daten verarbeitet werden. Selbst bei einer Mehrkern-CPU kann durch die andere Repräsentation der Daten und unterschiedliche Rechenverfahren ausgeschlossen werden, dass die gleichen Daten im gleichen Taktzyklus verarbeitet werden. Folglich ist es nicht möglich, dass ein gleicher Fehler in beiden Kanälen zur gleichen Zeit dazu führt, dass die Daten in gleicher Weise verfälscht werden. Dies wäre fatal, da der Vergleicher in diesem Fall den Fehler nicht bemerken könnte.

4.1 Fehlermodelle auf Systemebene

Die Fehlermodelle, wie sie in Abschnitt 2.2 vorgestellt wurden, werden im Folgenden auf das Konzept in Kapitel 3 angewendet, um mögliche Schwachpunkte aufzudecken.

Goloubevas Fehlermodell

Diese Fehlermodell, wie es in [Gol06, S. 13] diskutiert wird, besteht aus drei Gruppen von Fehlern, Datenfehler und Codefehler Typ 1 oder Typ 2. Datenfehler modifizieren gespeicherte Informationen transient oder permanent und haben (laut Definition) keinen Einfluss auf den Kontrollfluss des Programms.
Bei Coded Processing werden alle Nutzdaten in codierter Form und optional in decodierter Form abgelegt. Die Rechnungen geschehen alle mit den codierten Daten. Wird eine der Variablen durch einen Datenfehler verändert, besteht die Wahrscheinlichkeit $\frac{1}{A}$ [Oze92], dass ein reguläres Codewort entsteht. Werden, wie in 2.3, errechnet für A mindest 24 Bit verwendet, sind Datenfehler

mit der geforderten Wahrscheinlichkeit aufdeckbar. Kritisch in diesem Zusammenhang ist die Korrekturfunktion der Multiplikation. Im Gegensatz zu der in Abschnitt 2.3 Eq. 2.13 [Sch11, S. 62] gezeigten Korrektur ist es nicht notwendig uncodierte Variablen zu verwenden, es kann eine Lösung über die AN-Codierung gefunden werden. Diese bietet zwar nicht die Sicherheit der ANB-Codierung, jedoch ist die Fehleraufdeckung deutlich besser als würde mit uncodierten Variablen hantiert werden (siehe Abb. 3.2).

Instruktionsfehler Typ 1 führen zu einem Vertauschen der Instruktion. Der somit entstehende Befehl ist gültig und kann vom Prozessor ausgeführt werden. Für die Aufdeckung dieser Art von Fehler wurde die Signatur B eingeführt. Sie sichert durch ihr einmaliges Verhalten bei mathematischen Operationen gegen vertauschte Operatoren ab.

Instruktionsfehler Typ 2 (Kontrollflussfehler) sind durch Coded Processing nicht geschützt. Das mathematische Konstrukt der codierten Verarbeitung ist nicht in der Lage einen fehlerhaften Programmfluss aufzudecken. Dies ist nur möglich, wenn Informationen über den Soll-Programmablauf vorhanden sind. Bei statischen Programmen mit fixen Abläufen reicht es, eine Liste mit den zu entstehenden Signaturen zur Compilezeit zu erstellen und zur Laufzeit das korrekte Auftreten dieser zu überwachen. Programme mit dynamischem Ablauf lassen sich auf diese Weise nicht sichern, da sich der Kontrollfluss bei jedem Programmdurchlauf verändern kann. Eine dynamische Generierung der Prüfsignaturen kann nur von einer Einheit errechnet werden, welche über die vollen Prozessdaten verfügt. Zu diesem Zweck wurde in das Konzept der zweite, uncodierte Kanal hinzugefügt. Da dieser Kanal ebenso alle Prozessdaten empfängt und verarbeitet, kennt er den Programmablauf des codierten Kanals.

Forins Fehlermodell

Das Fehlermodell von Forin besteht aus Operationsfehler, Operandenfehler und Operatorenfehler. Trotz der Ähnlichkeit zu dem Fehlermodell in [Gol06, S. 13] können die Fehlerauswirkungen unterschiedlich ausfallen. Die Fehler laut Forin sind weitläufiger definiert, es dürfen bei jeder der drei Kategorien Berechnungen wie auch der Kontrollfluss verändert werden.
Operatorenfehler, wie das Vertauschen von einer gültigen Operation zur anderen, führen zu einer unerwarteten Verknüpfung der angegebenen Operanden. Die Auswirkungen auf arithmetische Operationen sind durch Coded Processing geschützt. Hier hilft die Vorhersehbarkeit der Signaturen B um das Vertauschen sicher zu erkennen. Handelt es sich bei der veränderten Instruktion um einen Sprungbefehl ist eine Veränderung des Kontrollflusses möglich. Der Sprung zum falschen Zeitpunkt z. B. durch die Verfälschung des Befehls jc[1] zu js[2] [Int13, S. 7-16] kann zu einem unvorhersehbaren Programmverhalten führen. Die Sicherung geschieht durch die redundante Ausführung im zweiten Kanal.

Operandenfehler und Operationsfehler gilt ebenso das bisher Beschriebene. Ein Operandenfehler in einer arithmetischen Operation kann durch die codierte Berechnung erkannt werden. Tritt der Operandenfehler bei einem Befehl zur Kontrollflusssteuerung auf, kann der Programmablauf verändert werden. Aufgrund der Abweichung der entstehenden Signaturen im Vergleich zum uncodierten Kanal kann der Fehler aufgedeckt werden. Operationsfehler stellen eine Verallgemeinerung von Operanden- und Operatorenfehler dar. Die Ausführung eines Befehls verläuft nicht wie erwartet, wird dessen Ursache nicht weiter auf Operand oder Operator eingegrenzt.

[1] Jump Carry
[2] Jump Sign

Zu beachten ist in diesem Zusammenhang die Konstellation des verlorenen Updates. Werden Daten zur Weiterverarbeitung aus dem Speicher geladen und nach der Abarbeitung statt an die gleiche Speicherstelle fälschlicherweise an eine andere geschrieben, werden beim nächsten Lesevorgang veraltete Daten verwendet, dargestellt in Abb. 4.1. Da die alten (ebenso wie die neuen) Daten mit gültiger Codierung vorliegen, kann der Fehler durch die Modulo-Probe nicht erkannt werden.

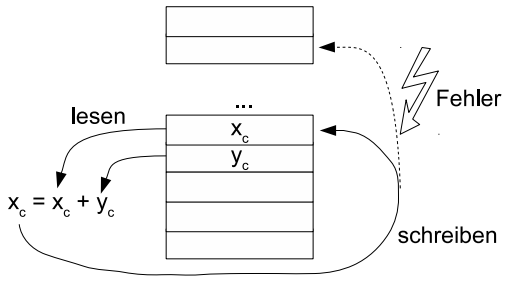

Abbildung 4.1: Konstellation für ein verlorenes Update

Um dies zu verhindern wurde der D-Anteil in Forins Arbeit [For89] hinzugefügt. Dieser D-Anteil wurde hier bewusst nicht implementiert. Dennoch muss nachgewiesen werden, dass ein verlorenes Update kein Risiko darstellt.
Da sich die Signatur B mit der Berechnung ändert, kann man sagen, dass B ist abhängig von der Zeit ist. Mit dieser Sichtweise entsteht ein $B(t)$, welches Eigenschaften des Zeitstempels D besitzt. Die Signatur ändert sich mit der Zeit und altert. Das zusätzliche Hinzufügen von D ist somit überflüssig. Eine Ausnahme muss jedoch gemacht werden: der Fehler ist nicht aufdeckbar, wenn die Signatur des alten und des neuen Wertes gleich ist. Dieser Sonderfall kann auf zwei Arten eintreten. Einerseits, wenn die Signatur B des nicht überschriebenen Operanden Null ist, andererseits wenn das neu entstandene B gleich dem alten B ist.

Zur genaueren Analyse wird zuerst die Addition untersucht, in welcher sich die Signatur des Ergebnisses laut 4.1 errechnet. Es wird angenommen, dass zu x_c mit der Signatur B_x, y_c mit der Signatur B_y addiert wird (siehe 2.11) und auf die Speicherstelle von x_c geschrieben wird.

$$B_x = B_x + B_y \tag{4.1}$$

Bedingung 4.1 ist erfüllt, wenn $B_y = 0$ ist, dies widerspricht jedoch der Definition der ANB-Codierung. Mit $B_y = 0$ würde es sich um eine AN-Codierung handeln. Eine weitere Möglichkeit 4.1 fälschlicherweise zu erfüllen, kann durch die Signaturkorrektur entstehen, wenn $B_x + B_y > A$ wird. In diesem Fall wird die Signatur durch die Operation $mod(B_x+B_y, A)$ wieder in den Bereich von Eq. 2.9 verschoben. Um ein verlorenes Update nicht erkennen zu können, muss der Fall von Eq. 4.2 eintreten. Da für $1 < B < A$ gilt, folgt $B_x + B_y < 2 \cdot A$. Die Modulooperation 4.2 lässt sich somit umschreiben auf Eq. 4.3, woraus abgelesen werden kann, dass die Bedingung 4.1 nur erfüllt ist, wenn $B_y = A$ gilt, welches $1 < B < A$ widerspricht. Doch zeigt sich genau hier ein anderes Problem, nämlich wenn $B_x + B_y = A$, ist die Signatur des Ergebnisses gleich A. Die Modulodivision durch A erzeugt eine Signatur von Null und das Ergebnis ist statt ANB- nur mehr AN-codiert. Das Eintreten dieses Sonderfalls muss abgefragt und korrigiert werden.

$$mod(B_x + B_y, A) = B_x \tag{4.2}$$

$$B_x = B_x + B_y - A \tag{4.3}$$

Bei der Multiplikation besteht die Signatur des Ergebnisses in Eq. 2.14 aus $B_x \cdot B_y$, wobei auch hier der Fall Eq. 4.4 nicht eintreffen darf. Die ist nur möglich, wenn $B_y = 1$, welches der Bedingung 2.9 widerspricht. Die Möglichkeit der Signaturkorrektur muss auch hier untersucht werden. Übersteigt die Signatur des Ergebnisses den Wert A, wird ähnlich der Addition eine Korrektur durchgeführt und die neue Signatur aus $mod(B_x \cdot B_y, A)$ errechnet und wieder in den erlaubten Zahlenbereich $1 < B < A$ geschoben. Der Fall Eq. 4.5 darf auch hier keinesfalls eintreten.

$$B_x = B_x \cdot B_y \tag{4.4}$$

$$mod(B_x \cdot B_y, A) = B_x \tag{4.5}$$

Dies ist möglich, wenn $B_x \cdot B_y = n \cdot A + B_x$, wobei $1 \leq n < A$, $n \in N$, oder anders ausgedrückt die Bedingung 4.6 erfüllt ist. Es zeigt sich ein weiteres Mal, dass A als Primzahl gewählt werden muss: vorausgesetzt A ist eine Primzahl, kann Formel 4.6 kann nur erfüllt sein, wenn B_x oder B_y gleich A ist, dies ist jedoch durch die Bedingung $1 < B < A$ verboten. Ist A keine Primzahl und kann in Faktoren zerlegt werden, ergeben sich Kombinationen B_x und B_y, welche Bedingung 4.5 erfüllen und dazu führen, dass B_x der Datenquelle und des Ergebnisses übereinstimmen. Folglich wäre ein verlorenes Update nicht erkennbar.

$$B_1 = \frac{A \cdot n}{B_2 - 1} \tag{4.6}$$

Betrachtet man die Subtraktion von $x_c - y_c$, so muss das Ergebnis die Signatur $B_x - B_y$ besitzen. Für den Fall, dass $B_x < B_y$ wäre die Signatur des Ergebnisses kleiner Eins und widerspricht der Bedingung $1 < B < A$, weshalb eine Korrektur durchgeführt werden muss. Betrachtet man zuerst den Fall ohne Korrektur, so kann nur die Kombination $B_x = B_y$ zu einem Problem führen, da das resultierende $B = 0$ und keine ANB-Codierung mehr vorhanden wäre. Dies ist jedoch nicht erlaubt, da eine Signatur nicht doppelt vergeben werden darf. Bei der Signaturkorrektur wird zu $B_x - B_y$ die Konstante A addiert um die Signatur in den Bereich $1 < B < A$ zu schieben. Folglich ist ein verlorenes Update nicht aufdeckbar, wenn Eq. 4.7 erfüllt ist. Man erkennt, dass dies nur möglich ist, wenn $B_y = A$ und diese Konstellation ist verboten.

$$B_1 = B_1 - B_2 + A \tag{4.7}$$

4.2 Fehlermodel aus IEC 61508

Die Tabelle A.1 der IEC 61508 [IEC10a, Teil 2] stellt Ausfälle dar, welche durch Sicherungsmaßnahmen erkannt werden müssen.

Der erste Eintrag „Elektromechanische Bauteile“ kann unbehandelt bleiben, da der PC über keine mechanischen Bauteile wie Relais verfügt. Relais sind nur in den Ausgangsknoten vorhanden und entziehen sich somit der Betrachtung der Sicherheitssteuerung.

Der nächste Eintrag „Diskrete Hardware“ bezieht sich auf digitale und analoge Ein-/Ausgänge und die Energieversorgung. Digitale Eingänge sind in Form einer sicherheitskritischen Feldbus-Schnittstelle vorhanden. Stuck-at Fehler oder unerlaubtes Verhalten an diesen Pins sind durch das sicherheitskritische Protokoll abgedeckt. Der Datentransport innerhalb des Computers von

der Netzwerk-Schnittstelle zur CPU bedarf einer Sicherung. Dazu werden die Daten von der Kommunikationsschnittstelle noch als sicherheitskritische Datenpakete an die Berechnungskanäle gesendet, somit wird der Black Channel[3] über den PCIe-Bus verlängert. Der Transport der Daten von den Berechnungskanälen zu der Kommunikationsschnittstelle geschieht ebenfalls in der Form von sicherheitskritischen Paketen.
Analoge Ein- und Ausgänge sind an der Sicherheitssteuerung nicht vorhanden, für diese Aufgaben sind sichere Eingangs-/Ausgangsknoten verantwortlich und an diesem Punkt unerheblich.
Die Energieversorgung der Sicherheitssteuerung stellt einen CCF für beide Berechnungskanäle dar. Ein Stromausfall führt zum Versagen aller Komponenten inklusive des Watchdogs. Die Fehlerreaktion muss in diesem Fall der sichere Ausgangsknoten übernehmen. Wenn dieser innerhalb eines definierten Zeitraums keine Nachricht von der Sicherheitssteuerung empfängt werden die Ausgänge abgeschaltet. Dies setzt voraus, dass entweder der Ausgangsknoten eine eigenständige, noch nicht ausgefallene Energieversorgung besitzt, oder dass zwangsöffnende Schalter die Ausgänge bedienen. Möglichkeit Zwei ist der typische Fall, jedoch ist die genaue Umsetzung nicht relevant für die Zertifizierung der Sicherheitssteuerung.

Ein besonders kritischer Punkt des Fehlermodells stellt der PC interne Bus und seine angeschlossenen Module dar. Die Norm schreibt allgemein eine Behandlung möglicher Zeitüberschreitungen und spezielle Untersuchungen der Speichermanagementeinheit, des direkten Speicherzugriffs und der Bus-Arbitrierung vor.
Zeitüberschreitungen sind durch die Deadlines des Watchdogs und des Ausgangsknotens überwacht. Kommt es zu überproportional langen Zugriffszeiten, verzögert sich die weitere Berechnung. Folglich sind die benötigten Ergebnisse nicht zum geforderten Zeitpunkt vorhanden. Dies bemerkt der Watchdog, der über einen eigenen Quarz und somit eine eigenen Zeitbasis verfügt. In weiterer Instanz bemerkt der Ausgangsknoten, welcher auch eine eigene Zeitbasis besitzt das Ausbleiben der Datenpakete und führt einen sicheren Halt aus.
Das Speichermanagement wird von einem eigenen Baustein der Memory Management Unit (MMU) ausgeführt. Ihre Aufgaben sind das Übersetzen von logischen Adressen zu physikalischen, ebenso wie der Speicherschutz (engl. Memory Protection). Kommt es zu einem transienten oder permanenten Fehler, werden die Adressen verfälscht und der Speicher liefert nicht die gewünschten Daten, sondern den Inhalt einer anderen Speicherstelle. Ebenso ist die MMU eine der Ursachen für ein verlorenes Update. Werden durch einen Fehler die falschen Daten zu einer Adresse geliefert, ist dies über die ANB-Codierung erkennbar. Werden Daten, welche nicht zum Sicherheitsprogramm gehören ausgelesen (z. B. ein Teil des Betriebssystems), ist dies durch die Konstante A erkennbar. Die Wahrscheinlichkeit, dass zufällige Daten ein korrektes A vorweisen können lautet $\frac{1}{A}$. Hinzu kommt das nicht-vorhandensein einer Signatur. Wenn jedoch durch einen Adressfehler, ein an sich korrekten Codewort, an der falschen Stelle geladen wird, ermöglicht die Signatur B die Erkennung. Da beide Softwarekanäle die Signaturen mit rechnen, fällt ein Fehler dieser Art auf. Ein permanenter Fehler kann natürlich beide Kanäle beeinflussen, jedoch ist auszuschließen, dass er das in beiden Kanälen gleich tut. Die unterschiedliche Programm- und Speicherstruktur in den Kanälen würde die Daten unterschiedlich verfälschen, wodurch die Abweichung beim internen Vergleich oder im Watchdog erkannt wird.
Der direkte Speicherzugriff (engl. Direct Memory Access, DMA) erlaubt Peripheriegeräten Daten in den Speicher zu schreiben oder zu lesen, ohne dass sie die CPU passieren müssen [16]. Die CPU wird nur mehr über den Anfang und das Ende einer Datenübertragung informiert, der Transport der Daten geschieht ohne das Zutun des Prozessors. In diesem Fall könnte durch einen Fehler auf einen falschen Speicherbereich geschrieben und sicherheitskritische Daten verändert werden. Da

[3] Kommunikationskanal welcher nicht nach den Anforderungen der IEC 61508 entwickelt wurde [EN12]

die Daten doppelt und diversitär vorhanden sind und im codierten Kanal zusätzlich durch A und B gesichert sind, kann eine Verfälschung dieser Art aufgedeckt werden.
Die Zugriffsstrategie auf den Bus, auch Arbitration genannt, regelt, in welcher Reihenfolge die unterschiedlichen Teilnehmer auf den Bus zugreifen dürfen. Kommt es hier zu einem Fehler, werden möglicherweise sicherheitskritische Daten nicht rechtzeitig übertragen. Die Auswirkung wäre wiederum ein nicht-einhalten der Deadlines und der Watchdog oder der Ausgangsknoten müssen dies detektieren.

Die Fehlfunktionen der CPU wurden zwar schon in den Fehlermodellen von Forins und Goloubevas in Abschnitt 4.1 und 4.1 behandelt, doch die Norm sieht teilweise eine andere Betrachtungsweise vor.
Bei den internen Registern ist eine gegenseitige Beeinflussung von Speicherstellen, Veränderung durch Soft-Errors und fehlerhafte Adressierung zu beachten. Die Beeinflussung von Speicherstellen und die Veränderung durch transiente Fehler kann gemeinsam betrachtet werden. Die uncodierten Daten sind anfällig für diese Art von Fehlern, nicht jedoch die codierten. Die Informationsredundanz durch die Codierung erkennt mit Wahrscheinlichkeit von $\frac{1}{A}$ Verfälschungen der Daten, wobei die zusätzliche Signatur die Erkennungswahrscheinlichkeit noch erhöht. Veränderung der Adressierung, z. B. dass die Daten in ein falsches Register geladen werden, können den Kontrollfluss verändern. Dies wird jedoch durch die Kombination aus codiertem und uncodiertem Kanal erkannt. Wenn in einem Kanal der Programmablauf korrumpiert wird, erkennt der zweite Kanal dies an der Abweichung der entstehenden Signaturen.
Kommt es zu Veränderungen der Codierung[4] oder Ausführung kann die ANB-Codierung dies aufdecken. Der Punkt ist der Untersuchung in Forins Fehlermodell ähnlich, wobei veränderte Ausführung als Operationsfehler und veränderte Codierung als Operatorenfehler angesehen werden können. Eine Verfälschung des Flagregisters wirkt sich in einer Veränderung des Kontrollflusses aus. Diese Register werden bei einer Vielzahl von mathematischen Operationen und Vergleichen entsprechend dem Ergebnis gesetzt. Eine Abfrage dieser, bei der Verwendung bedingter Sprungbefehle, steuert den Kontrollfluss [Bac03, S. 86]. Flagregister spielen eine wichtige Rolle bei If-Abfragen und endlichen Schleifen. Endliche Schleifen bestehen im Prinzip aus einer Endlosschleife mit einer If-Abfrage zur Beendigung. Das Ergebnis jedes Vergleichs endet als Bit in einem Flagregister. Eine codierte If-Abfrage wie sie in [Sch11, S. 48] gezeigt wird ändert nichts an der Tatsache, dass selbst die verzweigtesten Vergleiche in einem einzigen Bit im Flagregister enden und diese Bit die weitere Ausführung beeinflusst. Bei statischen Programmabläufen kann anhand der vorkalkulierten Signaturen ein Fehler dieser Art aufgedeckt werden, bei dynamischen Anläufen ist dies nicht der Fall und birgt das Risiko für einen gefährlichen Ausfall. Schutz davor bietet nur die Kombination mit einem zweiten Berechnungskanal. Läuft dieser auf der gleichen Hardware ist eine diversitäre Softwareimplementierung entscheidend, sodass ein permanenter Fehler im Flagregister nicht beide Kanäle in der gleichen Weise verändert.
Weitere zu beachtende Elemente in der CPU sind der Stack-Pointer und der Program-Counter. Beides sind Register, welche eine Adresse beinhalten. Der Stack-Pointer zeigt auf das oberste Element im Stapelspeicher [Bac03, S. 81] und der Program-Counter auf den als nächsten auszuführenden Befehl im Speicher [Bac03, S. 80]. Der Stack wird als schnell zugänglicher Zwischenspeicher für Daten und Adressen verwendet, folglich können Daten und Adressen verfälscht werden. Eine Veränderung des Stack-Pointers kann weitläufige Auswirkungen haben: Rücksprungadressen gehen verloren, Variablen werden vertauscht, oder Variablen werden mit Adressen vertauscht (und umgekehrt). Da im Stack Daten aller Softwarekomponenten vorhan-

[4] In diesem Zusammenhang ist nicht die arithmetische Codierung angesprochen, sondern der auszuführende Programm-/Assembler-Code

den sind, ist ein Fehler nicht auf einen Softwarekanal beschränkt. Ein Beispiel eines Stacks ist in Abb. 4.2 dargestellt. Die Zugehörigkeit der Daten zu der jeweilige Softwarepartition ist durch „Kanal 1“, „Kanal 2“, „GPOS“ und „Hypervisor“ gekennzeichnet, die Binärzahl zeigt Adresse der Speicherstelle. Durch den dargestellten Bitfehler im Stack-Pointer werden Daten aller Partitionen korrumpiert. Das resultierende Verhalten ist schwer zu beschreiben, jedoch kann davon ausgegangen werden, dass Daten und Programmfluss stark verändert werden. Der Watchdog erkennt dies, indem Signaturen zu früh, zu spät oder gar nicht an ihn gesendet werden und leitet einen sicheren Halt ein. Ein ähnliches Verhalten ist bei der fälschlichen Veränderung des Program-Counters zu erwartet, jedoch mit zwei Adaptionen. Erstens ist der Program-Counter für den gesamten Speicherbereich verantwortlich und nicht nur für den vergleichsweise kleinen Stack. Zweitens werden durch die fehlerhafte Veränderung des Program-Counters keine Daten gelöscht. Bei dem Stack gelten alle Daten oberhalb des Stack-Pointers als veraltet und können ohne Bedenken überschieben werden. Das durchaus vielfältige Fehlverhalten wird in dem Teil 4.3 mit einem dafür spezialisierten Fehlermodell untersucht.

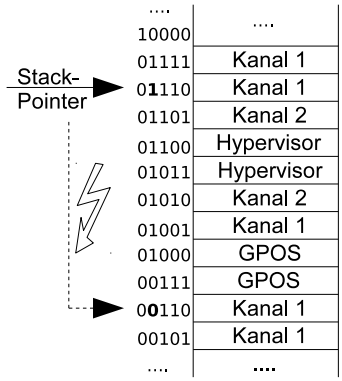

Abbildung 4.2: Bit-Flip im Stack-Pointer

Ein ähnlich zufälliges Verhalten ist bei Fehlern der Interrupt-Behandlung zu erwarten. Die hardwaremäßige Unterbrechung des Programmablaufs ist nicht immer vorhersehbar und somit nicht im Vorhinein festzulegen. Keine oder ständige Interrupts ebenso wie die gegenseitige Beeinflussung von Interrupts müssen untersucht werden. Interrupts einer CPU können interne oder externe Ursachen haben. Intern wäre z. B. eine Division durch Null, eine Schutzverletzung oder ein Page Fault[5] [Int13, S. 6-10], externe Ursachen können eine Benutzerinteraktion oder eine abgeschlossene Datenübertragung per Direct Memory Access (DMA) sein. Betrachtet man die Möglichkeit der ständigen Interrupts kann davon ausgegangen werden, dass das Anwenderprogramm keinen Fortschritt mehr erzielen kann, da die Abarbeitung ständig unterbrochen wird. Dieser Umstand kann vom Watchdog erkannt werden, da dieser unabhängig von der CPU arbeitet und die generierte Signaturen nicht nur auf inhaltliche, sondern auch auf zeitliche Korrektheit überprüft. Ist die Interruptverarbeitung dahingehend gestört, dass keine mehr ausgelöst werden, so ist die Abarbeitung zwar nicht dauerhaft gestört, doch ein korrektes Ergebnis wird mit hoher Wahrscheinlichkeit nicht mehr generiert. Werden z. B. die Interrupts der Netzwerkschnittstelle über eingehende Pakete verloren, können keine Ausgangsdaten mehr berechnet werden und der Watchdog, bzw. der sichere Ausgangsknoten wird dies bemerken. Über die Interrupt-Problematik lässt sich allgemein sagen, dass ein komplexes Zusammenspiel aus Interrupts und ungestörter Abarbeitung nötig

[5]Ein Page Fault tritt ein, wenn die benötigten Daten nicht im Hauptspeicher sondern auf der langsameren Festplatte gesucht werden müssen [HJL02, S. 939].

ist, um gültige Ergebnisse rechtzeitig zu liefern. Durch die mehrschichtige Überprüfung ist es nur schwer möglich, durch Interruptfehler unentdeckt, falsche Ergebnisse zu erzeugen. Die Programmdiversität der zwei parallel arbeitenden Sicherheitsapplikationen erhöhen die Aufdeckungswahrscheinlichkeit zusätzlich, da es ausgeschlossen werden kann, dass beide Kanäle zur gleichen Zeit die gleichen Daten verarbeiten und so die Berechnung am gleichen Punkt stören.
Eine Sonderstellung besitzt der Reset-Schaltkreis, dessen Aufgabe es ist, alle internen Bereiche der CPU auf einen definierten Anfangszustand zu setzen. Er wird in der IEC 61508 ebenfalls als Interrupt eingeteilt. Ein fälschliches Eintreten würden den Kontrollfluss gänzlich zerstören, sodass keine Berechnungen mehr durchgeführt werden können. Dies ist im Watchdog und im Ausgangsknoten leicht erkennbar, wodurch ein sicherer Halt eingenommen werden kann. Wird durch einen fehlerhaften Reset-Schaltkreis nur ein Teil der CPU in den richtigen Anfangszustand zurückgesetzt, sollte dies ebenfalls keine Auswirkungen haben, denn eine gewissenhafte Programmierung der Sicherheitsanwendung setzt voraus, dass alle verwendeten Komponenten vom Programmierer zusätzlich in Software auf einen definierte Abgangszustand gesetzt werden, bevor mit der Berechnung begonnen wird. Ist dies nicht möglich, da aufgrund eines nicht ausreichend initialisierten CPUs das Programm gar nicht starten kann, werden trotzdem keine falschen Daten über den Feldbus versandt, denn der Watchdog muss jedes Datenpaket gezielt freigeben. Die Freigabe erfolgt erst, wenn die CPU regelmäßig korrekte Signaturen an den Watchdog übermittelt.

Die Daten im nicht veränderlichen Speicher, wie der Festplatte oder einem Flash-Speicher, werden über Prüfsummen abgesichert. Somit kann eine Veränderung während der Speicherung und bei der Übertragung erkannt werden. Wichtig ist es, dass Konstanten bereits zur Compilezeit entsprechend ANB-codiert sind. Somit wird verhindert, dass durch einen Adressfehler unerkannterweise ein falscher oder vertauschter Wert geladen wird. Zusätzlich wird das Prinzip der Informationsredundanz neben der Codierung ein weiteres Mal angewandt, da die sicherheitskritischen Programme einmal mit codierten und einmal mit uncodierten Daten vorliegen. Die Instruktionen zum Verarbeiten der Daten liegen ebenso doppelt und diversitär vor, da sie mit zwei unterschiedlichen Compilern (oder Compilereinstellungen) erzeugt wurden.

Im veränderlichen Speicher wie dem Arbeitsspeicher und den Caches sorgt die ANB-Codierung für die Fehleraufdeckung. Stuck-at Fehler, das Übersprechen von Speicherstellen oder Daten-/Adressbus, Veränderung durch Soft-Errors und keine, falsche oder mehrfache Adressierung wird in den Fehlermodellen von Forin in 4.1 und Goloubevas in 4.1 genau analysiert.

Ausfälle beim Takt sind kritisch zu betrachten, da sie den Hypervisor und die beiden sicherheitskritischen Softwarekanäle in gleicher Weise beeinflussen. Eine Aufdeckung rein in Software ist somit nicht oder nur sehr verzögert, möglich. Eine der verzögerten Aufdeckungsmöglichkeiten wäre, wenn der sichere Ausgangsknoten bemerkt, dass die Datenpakete zu spät ankommen. Früher lässt sich eine Veränderung des Takts durch den Watchdog erkennen. Dieser besitzt aufgrund seines eigenen Quarzes eine eigene Zeitbasis. Kommen nun die Signaturen der Softwarekanäle nur etwas außerhalb des vorgeschriebenen Zeitfensters kann sofort der Netzwerkport geschlossen und ein sicherer Halt ausgeführt werden.

Die weiteren Punkte in der Tabelle A.1 der IEC 61508 [IEC10a, Teil 2] zu Kommunikation, Sensoren, Stellglieder sind hier nicht zu betrachten, da die bei der Analyse der Sicherheitssteuerung unerheblich sind.

4.3 Kontrollfluss

Betrachtet man das Fehlermodell von [Gol06], dargestellt in Abb. 2.8, lässt sich dieses bei der hier durchgeführte Analyse gut anwenden, da die Einteilung in sprungfreie Basic Blocks dem Aufbau eines Programms in Assembler sehr nahe kommt. Codeblöcke in Assembler bestehen aus einem Label, welches das Ziel eines Sprunges ist und immer den Anfang eines Codeblocks markiert. Am Ende wird in einen weiteren Block gesprungen oder der im Assembler-Code folgende Block ausgeführt. Hier handelt es sich gewisser Weise auch um einen Sprung, wenn auch nur um eine Adressstelle weiter.

Tritt während der Ausführung des Programms ein Fehler vom Typ 1 (verbotener Sprung) auf, so wird in einen Speicherbereich angesprungen, welcher eigentlich nicht betreten werden darf. Dies könnte ein versuchter Sprung von einem Softwarekanal in den anderen sein. Eine Speicherverletzung wie diese muss der Hypervisor aufdecken und verhindern. Speicherisolation wie sie hier benötigt wird, ist eine der wichtigsten Aufgaben eines Hypervisors. Eine typische Fehlerreaktion ist das Anhalten aller Softwarepartitionen folglich bekommt der Watchdog keine Signaturen mehr und schaltet alle sicherheitskritischen Ausgänge ab.

Die Auswirkungen eines Fehlers vom Typ 2 bis 5 hängen zu einem großen Teil vom angesprungenen BB ab. Wird nach dem falschen Sprung wieder in den Ursprungsblock zurückgekehrt ist die Aufdeckung deutlich erschwert. Die Ursache dessen ist, falls der angesprungene Block nur wenig an den Daten ändert und somit die Veränderung im Ergebnis oder im Kontrollfluss nur gering ist. Wird nach dem fälschlich angesprungenen Block nicht zurückgesprungen und ein weiterer nicht im Kontrollfluss vorgesehener Block ausgeführt, sind die Abweichungen von Daten und Kontrollfluss von den Soll-Werten deutlich größer und können leichter detektiert werden.

Die Fehlertypen 2 bis 5 lassen sich in dieser Arbeit gemeinsam behandeln. Das Fehlermodell wurde für sogenannte Blocksignaturen entwickelt [Gol06, S. 68]. Dabei wird in jedem Basic Block am Anfang und vor dem Weitersprung eine Signatur berechnet, anhand welcher erkannt werden kann, ob der Block korrekt von der ersten bis zur letzten Instruktion ausgeführt wurde. Drei Nachteile ergeben sich jedoch dabei. Erstens können die zusätzlichen Befehle der Signaturberechnung Ursache weiterer Fehler sein. Zweitens ist für eine zusätzliche Signaturberechnung in jedem BB ein enormer zusätzlicher Rechenaufwand nötig. Drittens sind Fehler vom Typ 5 nicht aufdeckbar, da hierbei die Blocksignatur nicht verändert werden würde.

Bei dem Konzept in Kapitel 3 wird durch die ANB-Codierung ein anderer Weg der Kontrollflusssicherung genommen. Es werden nicht in jedem Basic Block zusätzlich Informationen am Anfang und Ende berechnet, sondern nur dort, wo sich Daten verändern und sich ein Kontrollflussfehler auswirken würde. Ein Sprung vom Typ 2 bis 5 lässt sich vereinfacht ausdrücken als das Überspringen von Befehlen der folgenden drei Kategorien: Datenverschiebung, arithmetische Operationen, Kontrollflussoperationen.
Am einfachsten aufzudecken ist das Überspringen arithmetischer Operationen. Diese sind durch die Signaturen ANB-Codierung und dem zweiten uncodierten Kanal gesichert. Wird nur eine codierte mathematische Operation übergangen, wird der Watchdog dies bemerken und in den Fail-Silent Zustand übergehen.
Eine übersprungene und nicht ausgeführte Datenverschiebung zu erkennen ist schwerer. Ein Kopiervorgang, wie es bei einer Variableninitialisierung vorkommt, ist nicht durch einen Code gesichert und kann in erster Instanz nicht aufgedeckt werden. Erst wenn die fehlerhafte Zuweisung zu Kontrollflussabweichung führt, kann dies detektiert werden. Wenn zum Beispiel bei der Initialisierung einer Schleifenvariable ein Fehler passiert, kann dies erst durch das zu häufige oder

seltene Ausführen der Schleife, verglichen mit dem zweiten Kanal, erkannt werden.
Führt ein Fehler dazu, dass mindest eine Kontrollflussoperation übersprungen wird, lässt sich dies zurückführen, dass in dem gerade ausgeführten Basic Block der letzte Befehl des Bodys fälschlicherweise als ein Sprung interpretiert wurde. Der Sprung an sich kann nicht detektiert werden, jedoch schon seine Auswirkungen auf die folgenden arithmetischen Operationen.

In diesem Kapitel wurde das Konzept theoretisch analysiert und überprüft, ob es den Anforderungen der Norm IEC 61508 und den Fehlermodellen aus der Wissenschaft standhalten kann. Die Fehlermodelle von Forin und Goloubevas überschneiden sich teilweise, jedoch in anderen Punkten hat die gering unterschiedliche Ansichtsweise große Auswirkungen auf die Analyse. Die von Forin propagierten verlorenen Updates stellen eine Herausforderung für die ANB-Codierung dar. Durch die Definition der Konstante A als Primzahl und der Signatur B größer 1 und kleiner A ist es möglich auch ohne Zeitstempel D verlorene Updates erkennbar zu machen. Das Fehlermodell der IEC 61508 berücksichtigt eine Vielzahl von physikalischen Effekten und verlangt die Auseinandersetzung dieser mit den möglichen Auswirkungen auf die Sicherheitsfunktion.
Nach der Analyse des Fehlermodells für den Kontrollfluss zeigt sich eine Eigenschaft des Konzepts. Für die Aufdeckung der arithmetischen Fehler ist die ANB-Codierung gut geeignet. Eine Absicherung des Kontrollflusses ist bei dynamischen Programmabläufen nicht möglich. Dazu bedarf es eines zweiten Rechenkanals, welcher die Nutzdaten, wie auch die Signaturen des anderen Kanals mitrechnet. Erst jetzt kann in Kombination mit dem Watchdog die umfangreiche Sicherheit der einkanaligen, hardwareunabhängigen Lösung argumentiert werden.

5 Umsetzung

In diesem Kapitel wird der Testaufbau und die implementierten Programme diskutiert. Neben der Mathematik-Bibliothek zum codierten Rechnen wurden noch eine Fehlerinjektionssoftware, eine Parametersuche und zwei Beispielapplikationen erstellt.

5.1 Testaufbau

Um eine praktische Untersuchung des Konzepts aus Kapitel 3 durchzuführen, wurde eine Testimplementierung erstellt. In dieser Arbeit nicht relevante Teile wurden entfernt um die Analyse fokussiert durchführen zu können. Der Testaufbau ist in Abb. 5.1 dargestellt.

Der Schwerpunkt dieser Untersuchung ist die Fehleraufdeckung der kombinierten Softwareapplikationen, unabhängig von der Hardware. Betrieben wird diese Testumgebung auf einer Hardware, an welche keine safety-spezifischen Anforderungen gestellt werden müssen. Bei der Durchführung der Tests handelt es sich um eine Intel Q9400S [9] CPU mit 8 GB Arbeitsspeicher. Darauf wird ein GPOS betrieben, in diesem Fall Debian 7.4 Wheezy 64 Bit [2]. Debian zeichnet sich durch hohe Performance und Stabilität aus. Die darauf aufbauenden Softwarekomponenten Testsuite, Fehlerinjektion und Sicherheitsapplikationen inklusive Mathematikbibliothek (siehe Abschnitt 5.2) werden in den folgenden Unterpunkten erklärt.

Im Vergleich mit dem Aufbau in 3.1 lassen sich diverse Unterschiede erkennen. Der Hypervisor wird nicht betrachtet und bleibt daher im Testaufbau unberücksichtigt, ebenso das Gast-GPOS. Hardwaremäßig werden der Watchdog und die COM-Schnittstelle nicht weiter betrachtet und daraus folgend die internen Datenkommunikationsschnittstellen wie PCIe und die analogen Anschlüsse des Watchdogs.

Der Hypervisor entzieht sich der Betrachtung, da es sich dabei um eine bereits am Markt befindliche, nach IEC 61508 Teil 3 entwickelte, Software handelt. Produkte mit einer Freigabe von bis zu SIL 3 sind verfügbar [19]. Somit kann angenommen werden, dass die Isolation zwischen den Partitionen in einer notwendigen, und daher nicht weiter zu untersuchenden, Form vorhanden ist. Das GPOS als Gast des Hypervisors wird in der der Evaluierung nicht weiter betrachtet, da es sich um eine optionale Erweiterung des Konzepts handelt. Einflüsse von dem unzertifizierten Gastsystem müssen durch den Hypervisor isoliert werden und dürfen keine Auswirkungen auf die sicherheitskritischen Applikationen haben. Der Watchdog wird als physikalische Zusatzkomponente nicht in den Testaufbau aufgenommen. Seine Aufgabe, die Rechenergebnisse der Sicherheitsapplikationen

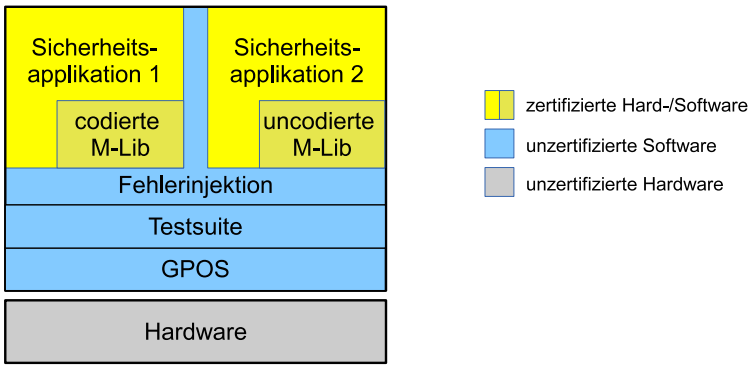

Abbildung 5.1: Architektur des Testaufbaus

und deren Signaturen zu vergleichen, wird von der Testsuite übernommen und fließt dort direkt in die Ergebnisse der Fehlerinjektion ein. Die Kommunikationsschnittstelle wird nicht in den Testaufbau eingeführt. Es werden bei der hier durchgeführten Untersuchung die Ergebnisse den Sicherheitsapplikationen direkt entnommen und ausgewertet, noch bevor sie in ein sicherheitskritisches Datenpaket verpackt werden würden. Zusätzlich wird die Kommunikationsschnittstelle als nicht zertifiziert gehandhabt, weshalb sie keinen gefahrbringenden Einfluss auf die Sicherheitsfunktion besitzen darf. Ebenso wie die Kommunikationsschnittstelle zum Feldbus sind die internen Kommunikationswege wie PCIe und die analogen Leitungen zum Watchdog nicht zu zertifizieren. Tritt hier ein Fehler auf, darf dies nur die Verfügbarkeit und nicht die Sicherheit beeinflussen. Aus diesem Grund werden die internen Datenkommunikationswege nicht in dem Testaufbau untersucht.

5.2 Mathematik Bibliothek

Im Zuge der Arbeit wurde eine Mathematik-Bibliothek in der Programmiersprache C [Rit90] erstellt, welche den Umgang mit Coded Processing so weit wie möglich vereinfachen und trotzdem zu performanten Lösungen führen soll.

Codierte Verarbeitung

Die Bibliothek ist für ANB-Codierung, wie in Kapitel 3 erläutert, ausgeführt. Die Speicherung der Daten ist mittels Struktur implementiert und beinhaltet immer die codierten, sowie die uncodierten Daten und die Signatur B. Im codierten Zweig dürfen die uncodierten Daten nicht zur Berechnung herangezogen werden, jedoch ist die Fehlersuche anhand der uncodierten Daten deutlich einfacher. Die uncodierten Daten werden im codierten Kanal nie explizit gerechnet, sondern immer aus den codierten extrahiert.

Für umfangreiche arithmetische Berechnungen wurden die Addition, Multiplikation, Subtraktion und Division implementiert. Darauf aufbauend wurden Bitschiebeoperationen nach links und rechts, sowie die Potenz-Funktion erstellt. Für jede der arithmetischen Operationen ist eine eigene Signaturkorrektur eingebaut. Diese wird benötigt, wenn die Signatur B im Zuge einer korrekten Berechnung den Rahmen $1 < B < A$ (siehe Bedingung 2.9) verlässt. Die Rechenprobe anhand der Modulo-Operation wird nach jedem Rechenschritt durchgeführt.

Für die Steuerung des Kontrollflusses sind drei codierte Vergleichsoperationen implementiert. Es ist möglich *größer*, *kleiner*, *gleich* Vergleiche mit codierten Daten durchzuführen. Da ein direkter Vergleich nur bei AN-codierten Daten durchführbar ist, muss für ANB- (und ANBD-) codierte Daten ein anderer Weg gefunden werden. In der hier implementierten Lösung wird der Vergleich anhand einer Subtraktion durchgeführt. Die zu vergleichenden Variablen werden subtrahiert und das Ergebnis ausgewertet, wobei sich die Bedingungen Eq. 5.1, 5.2, 5.3 für die codierten Werte x_c, y_c und deren uncodierte Werte x_f, y_f ableiten lassen.

$$x_c - y_c < 0 \text{ folglich ist } x_f < y_f \tag{5.1}$$

$$x_c - y_c > A \text{ folglich ist } x_f > y_f \tag{5.2}$$

$$1 < x_c - y_c < A \text{ folglich ist } x_f = y_f \tag{5.3}$$

Ist das Ergebnis der Subtraktion kleiner Null (Eq. 5.1), muss $x_f < y_f$ gelten. Der Einfluss der Signatur B wird durch die Subtraktion aufgehoben. Der Vergleich $x_c > y_c$ in Eq. 5.2 verläuft ähnlich. Ist die Differenz größer als A, ist die uncodierte erste Variable größer als die Zweite. Befindet sich die Differenz zweier codierter Variablen im Bereich zwischen 1 und A, sind die uncodierten Variablen gleich groß (siehe Eq. 5.3). Der Unterschied zweier unterschiedlicher, uncodierter Variablen muss sich codiert in einem Unterschied größer A widerspiegeln. Ist die Differenz geringer als A, so sind die uncodierten Variablen folglich identisch. Die hier aufgeführten Vergleiche sind nur gültig solange für B die Bedingung $1 < B < A$ (Eq. 2.9) erfüllt ist.

Hierbei zeigt sich ein Problem der codierten Vergleiche. Die Formeln Eq. 5.1, 5.2, 5.3 arbeiten zwar mit codierten Werten, jedoch wird der Vergleich kleiner Null, größer A zwischen 1 und A im Prozessor schlussendlich in einem bedingten Sprung enden. Dieser Sprung ist nur von einem einzigen Bit im Flag-Register abhängig und somit anfällig für gefährliche Fehler. Wird genau zum Zeitpunkt des Sprunges der Wert im Flag-Register verfälscht, wird der falsche Teil der Verzweigung durchlaufen. Dieser Fehlerfall kann durch codierte Verarbeitung nicht aufgedeckt werden. Hierzu ist der Vergleich mit dem zweiten Kanal nötig, sofern dieser nicht den gleichen Fehler begeht. Die Wahrscheinlichkeit dazu ist jedoch durch die diversitäre Programmierung gering.

Für die uncodierte Berechnung kann anhand der Übergabeparameter beim Programmstart festgelegt werden, ob mit uncodierten Variablen hantiert wird und zusätzlich, unabhängig davon, die Signaturen für den Vergleich mit der codierten Partition erstellt werden oder codiert gerechnet werden soll. Somit kann die gleiche Bibliothek für codierte und uncodierte Berechnungen verwendet werden. Der Quellcode ist aufgrund der unterschiedlichen Aufgaben bereits diversitär ausgeführt und wird bei Verwendung in zwei unterschiedlichen Kanälen mit unterschiedlichen Compilern erstellt. Somit wird eine größtmögliche Diversität erreicht, damit im Betrieb transiente und permanente Fehler zu unterschiedlichen Ergebnissen führen, welche wiederum durch einen Vergleichsvorgang aufgedeckt werden können.

Zum Codieren wie zum Decodieren stehen dem Benutzer eigene Funktionen zur Verfügung um den Vorgang einfach durchführen zu können. Die Probe der Berechnung nach den Regeln der ANB-Codierung wird nach jeder arithmetischen Operation durchgeführt. Wann die Signaturen und Ergebnisse zwischen dem codierten und uncodierten Kanal verglichen werden, kann der Benutzer entscheiden. Ein Vergleich nach jeder arithmetischen Operation ist jedoch nicht zu empfehlen, da durch den zusätzlichen Perfomance-Aufwand des Datenextrahierens und Vergleichens unnötige Verzögerungen entstehen. Sinnvoller ist es die Häufigkeit der Vergleichsvorgänge der Berechnung

anzupassen. Bei Vorgängen mit einer starken Fehlerfortpflanzung wie z. B. Matrixmultiplikationen reicht ein Vergleich am Ende, wodurch viel Rechenzeit gespart werden kann.

Es ist anzumerken, dass bei der Multiplikation das Ergebnis speziell angepasst werden muss. Wie die Formel 2.13 in Abschnitt 2.3 zeigt, wird durch eine Multiplikation die A und B Terme vermischt werden. Eine weitere Berechnung oder Probe ist damit nicht möglich. Eine Korrektur wie sie in [Sch11, S. 62] Eq. 2.13 angewandt wird, ist bei genauerer Betrachtung nicht zu empfehlen, da sie uncodierte Werte benötigt. Die Verwendung von uncodierten Variablen sollte im codierten Zweig vermieden werden. Ein hier entwickelter und in Eq. 5.4 dargestellter Rechenweg umgeht das Problem indem die Korrektur mit AN-codierten Variablen durchgeführt wird. Der zu korrigierende Wert ist z'_c und der korrigierte z_c. Sollte bei der Anpassung ein Fehler passieren und das Ergebnis verfälscht werden, würde er dennoch aufgedeckt, da das Ergebnis nicht die Form von Eq. 2.14 besitzt.

$$z_c = \frac{((z'_c - ((x_c - B_x) \cdot B_y + (y_c - B_y) \cdot B_x)) + (A-1) \cdot B_x \cdot B_y)}{A} \tag{5.4}$$

Uncodierte Verarbeitung

Der uncodierte Zweig rechnet zusätzlich zu den Nutzdaten die Signaturen des codierten Programmteils mit. Dabei ist es wichtig, dass die Kalkulation ein deterministisches Verhalten aufweist. Bei Berechnungen ist das Verhalten der Signatur durch die Rechenoperation festgelegt, bei der Neucodierung von Variablen muss aber im codierten und uncodierten Zweig die gleiche Signatur gewählt werden um die Vergleichbarkeit zu gewährleisten. Die Implementierung der Mathematik-Bibliothek lässt das Problem der deterministischen Signaturfindung anhand eines Pseudozufallsgenerators, welcher in beiden Kanälen gleich initialisiert wird und somit die gleichen Zahlenfolgen generiert. Ein Mehrfachvergeben einer Signatur ist somit nicht ausgeschlossen, kann aber auch durch andere Verfahren nicht verhindert werden. Angenommen, die Signaturen werden einer zur Compile-Zeit festgelegt und im Betrieb einer Liste entnommen und aus dieser gelöscht. Somit könnte jede Signatur nur ein Mal geben werden. Im Laufe der Berechnung ändern sich die zugewiesenen Signaturen jedoch stetig, wodurch Mehrfachvergaben wieder möglich sind. Eine ausgeklügelte Signaturfindung ist folglich nicht nötig, da sie das Problem von mehrfach vergebenen Signaturen nicht lösen kann.

Signaturberechnungen

Im Laufe der Berechnungen wird die Bedingung $1 < B < A$ verletzt. Ein Überschreiten der oberen Grenze kommt vor allem bei Additionen und Multiplikationen vor. In diesem Fall wird durch eine Modulo-Operation durch A die Signatur wieder in den erlaubten Zahlenbereich verschoben.
Bei der Berechnung von vorzeichenbehafteten Werten kommt es zu Verletzungen der unteren Grenze. Würde hier eine Modulo-Operation angewendet werden, wäre das Ergebnis fehlerhaft. Ist die Signatur (unabhängig vom Ergebnis der codierten Daten) negativ, muss A addiert werden um wieder in den positiven Bereich zu gelangen. Eine darauffolgende Modulo-Operation ist nicht mehr nötig, da ein Überschreiten der oberen Grenze nicht möglich ist, dazu siehe ebenso Abschnitt 4.1.

5.3 Fehlerinjektionssoftware und Testsuite

Zur Untersuchung der Fehleraufdeckungswahrscheinlichkeit der zweikanaligen Berechnung mit codierten und uncodierten Daten wurde eine Software erstellt, welche die Ausführung von Programmen verändern kann. Fehler können in jedes Programm injiziert werden, sofern der Quellcode in Assembler-Sprache vorliegt. Die Fehlerinjektion geschieht auf Assembler-Ebene, dazu wird der Assembler-Code eingelesen und je nach Fehlerinjektionsoptionen modifiziert. Der Schwerpunkt der Injektionen liegt auf transienten Fehlern. Dies ist durch Untersuchungen begründet, welche zeigen, dass ANB-codierte Berechnungen durch transiente Fehler am verwundbarsten sind [Sch11, S. 160]. Laut IEC 61508 sind systematische Fehler in der Hardware weitgehend auszuschließen, da kommerzielle CPUs strengen Entwicklungsverfahren und Tests zu Grunde gelegt sind [IEC10a, Teil 2, Klausel 7.4.6.1]. Implementiert wurde die Software in Python [15], da eine Vielzahl von Text-Strings bearbeitet werden und dies in Python effizient zu bewerkstelligen ist.

Die zu injizierenden Fehler basieren auf dem Fehlermodell von Forin [For89] und sind in Abschnitt 2.2 erklärt. Es besteht die Möglichkeit den Operator einer Instruktion auszutauschen oder beliebige Bitfehler im Operanden zu erzeugen. Die dazugehörige Auswahl geschieht über eine Konfigurationsdatei. Wird ausgewählt, dass der Operator vertauscht werden soll, wird aus einer Liste von parameterkompatiblen Instruktionen zufällig eine ausgewählt und die Originalinstruktion ersetzt. Die Parameterkompatiblität ist notwendig, damit das Programm überhaupt ausführbar bleibt. Würde zum Beispiel ein `mov %rax, %rbx` gegen ein `call %rax, %rbx` ausgetauscht, würde der Assembler sofort einen Fehler erkennen und das Programm nicht übersetzen.

Wird eine Fehlerinjektion gestartet, wird zuerst der Assembler-Quellcode des Programms eingelesen. Durch die Eingabe der Testoptionen in der Konfigurationsdatei wird nun der Quellcode nach den zu verfälschenden Parametern durchsucht. Dies kann einerseits eine Instruktion sein, oder ein Operand. Nun wird eine Liste erstellt, welche die Zeilennummern enthält, in welchen der Befehl oder Operand vorkommt. Aus dieser Liste wird zufällig eine Zeilennummer gewählt, welche in weiterer Folge verfälscht werden soll.
Da der Assembler-Code typischerweise auch Codeteile enthält, welche nie ausgeführt werden, würde eine Fehlerinjektion darin keine Auswirkung auf das Ergebnis besitzen. Folglich muss ermittelt werden, ob und wie oft die zufällig gewählte Zeile ausgeführt wird. Dazu wird der Assembler-Code direkt vor der zu verfälschenden Zeile um eine Ausgabe erweitert. Immer bevor die Zeile ausgeführt wird, wird ein Zählzeichen am Bildschirm ausgegeben. Es ist wichtig, dass dieses Zählzeichen nicht in der regulären Programmausgabe auftritt, da sonst die automatische Auswertung fehlschlagen kann. Das Zählzeichen kann ebenso in der Konfigurationsdatei festgelegt werden. Wird die Zeile nie ausgeführt, wird die Fehlerinjektion an dieser Stelle beendet. Ist die Ausführungsanzahl größer Null, wird zufällig eine der Ausführungen ausgewählt und verfälscht. Nun trennen sich sie Ausführungswege für Operanden- und Operatorenverfälschung.

Operandenverfälschung Hierbei wird aus einer Liste von parameterkompatiblen Instruktionen zufällig eine ausgewählt und mit den gleichen Operanden wie die Originalinstruktion verknüpft. Diese adaptierte Befehlszeile wird unter den Originalbefehl eingefügt, wobei zusätzlich noch ein bedingter Sprung und zwei Labels hinzugefügt werden. Das erstellte Konstrukt ermöglicht es, dass immer der Originalbefehl ausgeführt wird, mit der Ausnahme, wenn die Ausführungsanzahl die vorher (zufällig) bestimmte Anzahl erreicht. In diesem Fall wird zum fehlerhaften Befehl gesprungen, dieser ausgeführt und das Programm dann weiter normal exekutiert.

Operatorenverfälschung Soll der Operator durch einen oder mehrere Bitfehler verfälscht werden, kann der Benutzer in der Konfigurationsdatei wählen, in welchem Register und ob ein Stuck-at 1, Stuck-at 0 oder ein Bitflip injiziert werden soll. Zusätzlich besteht die Möglichkeit selbst die zu verfälschenden Bitstellen auszuwählen, oder dies durch eine zufällig generierte Zahl zu erledigen. Wird die zufällige Zahl gewählt, muss der Benutzer angeben, wie viele Bitstellen beeinflusst werden sollen. Bei der Operatorenverfälschung wird ebenso wie bei der Operandenverfälschung der originale Assembler-Code erneut adaptiert, jedoch dass bei einer bestimmten Ausführungszahl ein Registerwert verfälscht wird.

Im darauf folgenden Schritt wird das fehlerinjizierte Programm ausgeführt und die Ergebnisse ausgegeben. Diese werden in weiterer Folge von der Testsuite ausgewertet.

Die implementierte Fehlerinjektionssoftware bietet viele Vorteile für das gezielte Untersuchen des Programmverhaltens. Jeder zufällig gewählte Parameter wie Zeilennummer, Ausführungszahl und Bitstellen der Bitflips kann optional dezidiert vom Benutzer angegeben werden. Dies ist für eine genaue Untersuchung des Fehlerverhaltens sinnvoll, hingegen für eine große Anzahl an Tests und eine statistische Auswertung ist die zufällige Fehlerplatzierung vorteilhaft. Für die genaue Analyse bei einer großen Anzahl von Fehlerinjektionen werden alle injizierten Fehler in einer Log-Datei mitgeschrieben und können zurückverfolgt werden. Dies ermöglicht es, mehrere Fehlerinjektionsserien mit den gleichen Fehlern durchzuführen.

Beinhaltet die zu testende Software Abschnitte, welche nicht verfälscht werden dürfen, können diese in der Konfigurationsdatei als *Vital* angegeben werden. Im Zuge dieser Untersuchung wurde beispielsweise die Ausgabe der Ergebnisse als vital gekennzeichnet. Dies ist damit begründet, dass die Ergebnisse nativ ausgegeben werden und nicht als sicherheitskritisches Datenpaket, wie im Konzept vorgesehen.

Die Fehlerinjektionssoftware ist eigenständig ausführbar, jedoch nur für Einzelinjektionen geeignet. Um eine große Anzahl von Fehlerinjektionen durchzuführen, wurde eine Testsuite entwickelt, welche die Tests automatisiert. Ihre Aufgaben sind es, große Anzahlen von Fehlerinjektionen zu starten und die Ergebnisse auszuwerten. Auch diese Software wurde in Python [15] erstellt. Python eignet sich für die Testsuite besonders gut, da die Auswertung der Ergebnisse von der zu testenden Applikation abhängt. Die Fehlerinjektionssoftware übergibt die Ausgabe des zu testenden Programms als String an die Testsuite. Die automatische Speicherallozierung, das Überladen von Parametern vereinfacht die Auswertung der Fehlerinjektion. Verglichen wird das Ergebnis mit dem sogenannten „Golden Run“, einer Referenzausführung des Programms, in welcher keine Fehler injiziert wurden. Voraussetzung hierfür ist natürlich, dass während der Durchführung des „Golden Runs“ keine unentdeckten Fehler aufgetreten sind und das Ergebnis selbst korrekt ist.
Die Interpretation des Vergleichs des Golden Runs mit dem Ergebnis der Fehlerinjektion obliegt dem Benutzer. In der hier durchgeführten Untersuchung wird zwischen

- unerwartetem Programmende,
- korrektem Ergebnis,
- durch Codierung aufgedeckte Fehler und
- SDCs unterschieden.

Die ersten drei Fehlerkategorien werden als nicht gefährlich eingestuft, da offenkundig das Ergebnis nicht korrekt sein kann. Die Kategorie SDC beinhaltet die gefährlichen und nicht durch Diagnosemaßnahmen aufgedeckten Fehler, welche es zu verhindern gilt.

5.4 Beispielanwendung

Zum Testen des Konzepts, der Codierung und der Implementierung der Mathematik-Bibliothek wurden zwei Beispielanwendungen erstellt. Eine Matrizenmultiplikation und eine Überwachung anhand eines Funktionsblockdiagramms stellen zwei typische Anwendungen dar, wie sie in sicherheitskritischen Systemen vorkommen können.

Matrizenmultiplikation

Matrizenmultiplikationen sind in der Robotik sehr häufig. Das Umrechnen von Inertialsystemen ineinander kann durchaus ausführliche Matrizenmultiplikationen benötigen. Zur Untersuchung dieser wurden zwei 10x10 Matrizen erstellt, welche miteinander multipliziert werden. Der Kern der Berechnung besteht pro codiertem und uncodiertem Zweig aus je 10^3 Multiplikationen. Das Herzstück der Multiplikation für die native Berechnung ist im Folgenden im C-Code dargestellt.

```
for(i=0; i<DIM; i++)
   for(k=0; k<DIM; k++)
      for(j=0; j<DIM; j++)
         m3[i][k] += (m1[i][j] * m2[j][k]);
```

DIM steht für die Dimension der Matrizen, *m1* und *m2* sind die Matrizen, welche multipliziert und auf *m3* geschrieben werden. Bei der codierten Implementierung werden die arithmetischen Befehle durch die entsprechenden codierten Gegenstücke ersetzt.

Während der Berechnung wird im codierten Zweig überwacht, ob die Codierung verletzt wurde. Tritt hier eine unerlaubte Abweichung ein, wird das Ergebnis als falsch angesehen, unabhängig davon, ob das Ergebnis schlussendlich korrekt ist oder nicht.
Der Schwerpunkt dieser Beispielanwendung liegt in der Überwachung von arithmetischen Fehlern. Matrizenmultiplikationen beinhalten eine Vielzahl arithmetischer Operationen und einen relativ einfachen und vorhersehbaren Kontrollfluss. Die Fehlerfortpflanzung ist bei arithmetischen Berechnungen wie dieser sehr groß. Eine Abweichung während der Berechnung wirkt sich mit hoher Wahrscheinlichkeit ebenso auf das Ergebnis aus.

Bei der Implementierung wurden die Eingangsdaten (die Matrizen) fix in den Quellcode hinein programmiert. Das ist begründet, da die Daten auf dem Weg in die Sicherheitsapplikation nicht durch die Fehlerinjektion verfälscht werden dürfen. In dem Konzept in Kapitel 3 werden die Daten als sicherheitskritisches Datenpaket in die Applikation geladen, wodurch Verfälschungen mit sehr hoher Wahrscheinlichkeit erkannt werden können. Im Testaufbau in Abb. 5.1 ist dieser Teil nicht vorhanden, somit muss auf andere Art und Weise garantiert werden, dass die Daten korrekt in die Applikation geladen werden. Nach der Berechnung werden die Daten ausgegeben und sind ebenso nicht durch ein sicheres Feldbusprotokoll geschützt. Um Verfälschungen in diesem Teil der Untersuchung zu verhindern, wurde die Ausgabe durch die *Vital*-Option von der Fehlerinjektion ausgeschlossen.

Sicherheitssteuerung

Eine Sicherheitssteuerung im Funktionsplan stellt das Gegenteil einer Matrizenmultiplikation dar. Der arithmetische Anteil der Berechnungen ist sehr gering, das Programm besteht zu einer großen Anteil aus Sprüngen, welche einen Kontrollfluss mit vielen Zweigen und Verzweigungen darstellt.

Die Fehlerfortpflanzung ist eher gering. Wird der Kontrollfluss beschädigt, wirkt sich das stark auf das Ergebnis aus, jedoch existieren viele Programmteile, in welchen ein Fehler keine Auswirkungen hat. Beispielsweise pflanzt sich ein Fehler am Eingang eines booleschen AND-Gatters nicht fort, wenn mindestens einer der übrigen Eingänge logisch Null ist.

Das implementierte Beispiel einer Sicherheitssteuerung ist für die Überwachung eines Roboters (siehe Abb. 5.2) ausgelegt. Sie besteht aus einem Roboter, welcher durch die Sicherheitssteuerung abgeschaltet werden kann. Zur Überwachung des Arbeitsraumes ist ein Lichtgitter und ein Schutzkäfig mit einer Sicherheitstür eingesetzt. Der Bediener verfügt über eine Zweihandsteuerung und einen Freigabeschalter. Der Roboter wird in Betrieb genommen, wenn beide Taster der Zweihandsteuerung und der Freigabeschalter gedrückt sind und Lichtgitter und Sicherheitstür keine Verletzung des Arbeitsraums melden. Nun kann der Freigabeschalter losgelassen werden und der Roboter arbeitet weiter. Wird der Zweihandschalter losgelassen oder melden Lichtgitter oder Sicherheitstür eine Arbeitsraumverletzung, wird der Roboter sofort angehalten. Das zugehörige Funktionsblockdiagramm ist in Abb. 5.3 angeführt. Die Eingänge stehen für die Sicherheitstür (Tür), das Lichtgitter (LG), die Taster der Zweihandsteuerung (T1 und T2) und den Freigabeschalter (FG). Ausgang steuert den Roboter, wobei logisch Null als Stillstand interpretiert wird und logisch Eins als Betrieb. Das Flip-Flop am Ausgang speichert den Zustand, wenn nach einer erfolgreichen Inbetriebnahme der Freigabeschalter wieder losgelassen wird. Es ist wichtig, dass bei dem Flip-Flop das Rücksetzsignal (R) höhere Priorität besitzt als das Setzsignal (S).

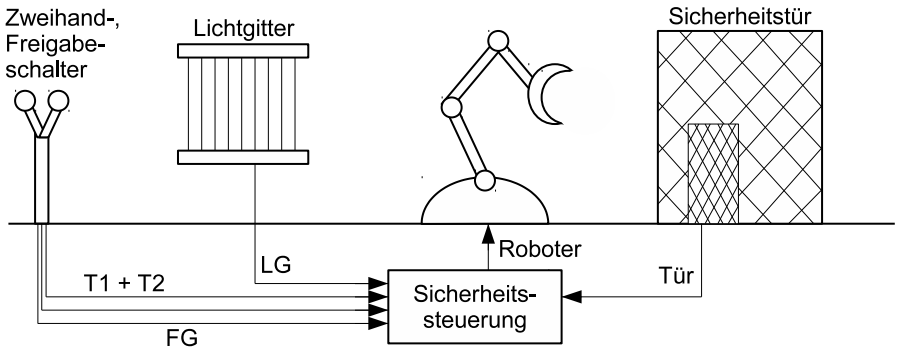

Abbildung 5.2: Implementiertes Beispiel für die Applikation „Sicherheitssteuerung“

In der Beispielimplementierung wurden für die Umsetzung des Funktionsblockdiagramms die logischen Gatter AND, NOT und RS-Flip-Flop implementiert. Für die logischen Signale Eins und Null wurden zwei codierte Zahlen definiert, welche als „high“ oder „low“ interpretiert werden. Das invertierende Gatter (NOT) tauscht den Eingangswert gegen den alternierenden Wert aus. Bei dem AND-Gatter wurden vier Eingänge implementiert. Da nicht immer alle genutzt werden, ist es wichtig die nicht verwendeten auf logisch Eins zu setzen. Die Auswertung in dem Gatter beruht auf der codierten Addition. Entspricht das Ergebnis einer vordefinierten, codierten Vier wird der Ausgang gesetzt. Für die logischen Verknüpfungen werden die Abfragen Eq. 5.1, 5.2, 5.3 genutzt.

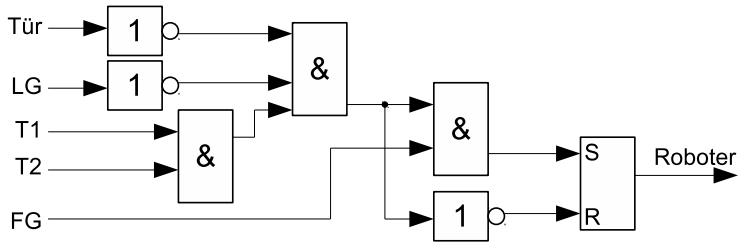

Abbildung 5.3: Funktionsblockdiagramm für die Sicherheitssteuerung in Abb. 5.2

5.5 Parametersuche

Um Werte mit einer möglichst hohen Fehleraufdeckung für die Konstante A und die Signatur B zu finden wurde ein Programm entwickelt, welches die Hammingdistanz zwischen Binärzahlen bestimmt.

In einem ersten Schritt wird die Hammingdistanz von AN-codierten Werten bestimmt. Eingabeparameter für das Programm sind einerseits die potentiellen As und der damit zu codierende Zahlenbereich. Für die As wurde eine Liste von Primzahlen erstellt, welche auf unterschiedlich große Zahlenbereiche angewendet wird. Jede codierte Zahl wird mit allen anderen, gleich codierten, vergleichen und die minimale und durchschnittliche Hammingdistanz berechnet. Es wird nicht der volle mögliche Zahlenbereich untersucht, da der benötigte Rechenaufwand exponentiell steigt. Die Zahlenbereiche wurden bei jedem Berechnungsdurchgang erhöht um mögliche Trends erkennen zu können.

Eine frühe Implementierung der Hammingdistanzberechnung wurde in Python geschrieben, es zeigte sich jedoch, dass eine Implementierung in C deutlich performanter ist. In C lassen sich einfach die unterschiedlichen Bitstellen mittels einer Exor-Verknüpfung ermitteln. Jeder Unterschied in einer Bitstelle wird durch eine logische Eins repräsentiert. Zum Abschluss muss die Anzahl der resultierenden Einsen bestimmt werden. Diese Implementierung basiert großteils auf booleschen Operationen, weshalb sie in der CPU schnell ausgeführt werden können.

Die Anzahl der möglichen Primzahlen lässt sich anhand des Primzahlsatzes von Gauß in Eq. 5.5 [Gol04] abschätzen. Wobei a und b für die Grenzwerte stehen, in welchem die Anzahl der Primzahlen approximiert werden soll. Li steht für den Integrallogarithmus. Anhand von Eq. 5.5 kann ermittelt werden, dass durch 24 Bit (siehe Abschnitt 2.3) ca. 1.078.000 Primzahlen dargestellt werden können.

$$\pi(x) \approx Li(x) = \int_a^b \frac{1}{ln(t)} \tag{5.5}$$

Eine Hammingdistanzberechnung mit dieser Menge an Primzahlen würde sehr lange dauern, weshalb für A eine Liste mit 78.330 Zahlen erstellt wurde, welche die Primzahlen von 1.009 bis 999.983 beinhaltet. Es fällt auf, dass in Abschnitt 2.3 von mindest 24 Bit breiten As gesprochen wird und für die größte hier untersuchte Primzahl nur $ld(999.983) \approx 19,93$, ergo 20 Bit verwendet werden. Die Ursache liegt darin, dass für die Untersuchung eine lückenlose Primzahlenfolge benötigt wird. Diese zu erstellen ist selbst sehr rechen- und zeitaufwändig und wie die Untersuchung in Abschnitt 6.1 zeigt auch nicht weiter relevant.

In diesem Kapitel wurden die Details für die Fehlerinjektionsuntersuchung diskutiert. Die Architektur aus Kapitel 3 wurde verändert, um die Tests so aussagekräftig wie möglich zu gestalten.

Die Implementierung der Mathematik-Bibliothek inklusive der codierten If-Abfragen wurde vorgestellt. Darauffolgend wurde die Fehlerinjektionssoftware und ihr Aufbau erklärt. Zur automatischen Testdurchführung wurde eine Testsuite erstellt, welche die Fehlerinjektionssoftware bedient und die Ergebnisse auswertet. Zwei Beispielapplikationen wurden mit der Mathematik-Bibliothek realisiert, einerseits eine Matrizenmultiplikation und andererseits eine Sicherheitssteuerung. Die erste Applikation benötigt viele arithmetische Verknüpfungen, wobei die zweite großteils boolesche Operationen beinhaltet. So kann das Verhalten der Mathematik-Bibliothek in zwei grundverschiedenen Anwendungen untersucht werden.

Die ausgeführten Tests mit der Mathematik-Bibliothek und der Fehlerinjektionssoftware werden im folgenden Kapitel präsentiert.

6 Evaluierung

In diesem Kapitel werden die Ergebnisse der Untersuchungen präsentiert und diskutiert. Zuerst wird die Suche nach optimalen Codierungsparametern durchgeführt, darauffolgend werden die Ergebnisse der Benchmarks durchgeführt, welche die zusätzliche Rechenlast durch codierte Verarbeitung aufzeigen sollen. Weiter wird das Verhalten der Beispielprogramme bei Fehlerinjektion analysiert. Am Ende befindet sich eine Abschätzung der Umsetzbarkeit für das vorgeschlagene Konzepts.

6.1 Parameterwahl

Geht man von einem 64 Bit Prozessor aus, welcher 40 Bit für Daten und 24 Bit für die Codierung (A und B) verwendet, so ergeben sich bei empirischer Berechnung für die Hammingdistanz $2^n \cdot (2^n - 1)$ Vergleiche, wobei für $n = 40$ gilt. Folglich wären alleine $1,2 \cdot 10^{24}$ Vergleichsoperationen pro Primzahl nötig. Eine beispielhafte Berechnung der Hammingdistanz für AN-codierte Zahlen 0 bis 10.000 codiert mit den Primzahlen zwischen 1.009 und 999.983 benötigt auf einem AMD X2 3800+ Prozessor ca. 30h. Hochgerechnet auf einen Zahlenraum von bis 2^{40} ergibt sich eine Berechnungsdauer von ca. $10^{22}\,s$ oder umgerechnet über 10^{18} Tage. Dieser Wert ist natürlich theoretischer Natur, veranschaulicht aber die Komplexität der Berechnung.

Da die Berechnung des vollen 40 Bit Zahlenraums nicht mit vertretbarem Aufwand errechnet werden kann, wurden zuerst die Zahlenbereiche von 0 bis 10, 100, 1000 und 10000 untersucht um zu überprüfen, ob sich ein Trend abzeichnet und die zu untersuchenden Zahlen (und somit die benötigte Rechenzeit) eingeschränkt werden könnten. Eine erste, offensichtliche Einschränkung lässt sich nach der Hammingdistanzberechnung von 0 bis 10 anschreiben. Primzahlen, welche in diesem geringen Zahlenraum eine schlechte Hammingdistanz aufweisen, besitzen in größeren Zahlenbereichen eine gleiche oder geringere Hammingdistanz. Durch diese Festlegung kann ein Teil der Primzahlen aus der weiteren Betrachtung ausgenommen werden. Zu stark darf das Suchfenster jedoch nicht eingegrenzt werden, denn Primzahlen mit einer hohen Hammingdistanz im Zahlenbereich 0 bis 10 können bei größeren Zahlenräumen stark an Hammingdistanz verlieren. Beispielsweise ermöglicht ein A von 379.811 im Zahlenbereich von 0 bis 10 eine Hammingdistanz von 9, im Zahlenbereich von 0 bis 1.000 jedoch nur mehr 3. Andererseits besitzt ein A von 234.917 im Bereich 0 bis 10 eine Hammingdistanz von 7 und verliert bei einer Vergrößerung des Zahlenraums auf 0 bis 1.000 nicht an Binärabstand. Für die Untersuchung kann ein immer kleiner werdender Bereich für die Primzahlen ausgewählt und so die Berechnung verkürzt werden.

Betrachtet man neben der minimalen die durchschnittliche Hammingdistanz der Zahlenbereiche 0 bis 10, 100, 1.000, 10.000 codiert mit den Primzahlen von 1.009 bis 999.983 zeigt sich, dass die durchschnittliche Distanz mit größer werdendem Zahlenbereich ansteigt. In Abb. 6.1 ist dies dargestellt. Auf der Abszisse sind die Primzahlen aufgetragen und auf der Ordinate der durchschnittliche Binärabstand. In den Klammern der Legende steht der obere Grenzwert der einzelnen Simulationsdurchgänge.

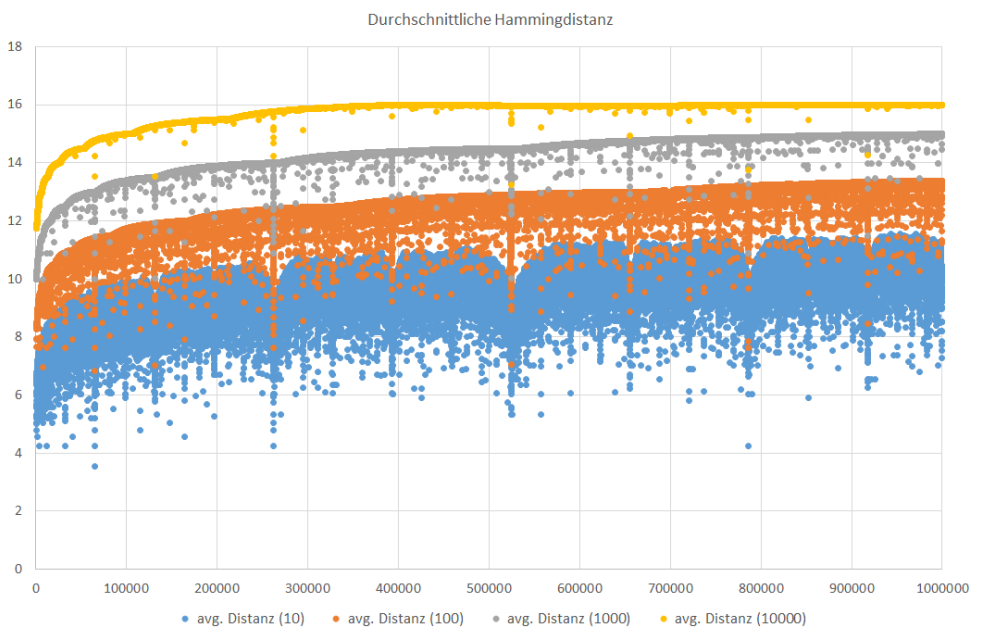

Abbildung 6.1: Durchschnittliche Hammingdistanz aufgetragen über den Primzahlen von 1009 bis 999983 angewandt auf verschiedene Zahlenbereiche.

Dass die Hammingdistanz mit größer werdendem Zahlenbereich ansteigt ist naheliegend, da große Zahlen mehr Raum für größere Hammingdistanzen bieten. Der Schein trügt jedoch, da im Gegensatz dazu die minimale Hammingdistanz bei steigenden Zahlenbereichen sinkt, da mehr potentielle binäre Nachbarn existieren. Auffällig sind in Abb. 6.1 manche Primzahlen, welche eine deutlich schlechtere mittlere Distanz aufweisen als der Rest. Die Ursache liegt in der Binärdarstellung der Zahlen. Zahlen mit einer annähernd durchmischten, ausgeglichenen Anzahl von binären „0“ und „1“ schneiden gut ab, Zahlen mit „1“er oder „0“er lastigem Aufbau schlecht. Vier Beispiele aus dem Intervall 0 bis 1000 sind in Tabelle 6.1 aufgeführt. Die aufgeführten Zahlen sind Primzahlen und aus einem ähnlichen Intervall. Die unterschiedliche Binärdarstellung wirkt sich stark auf die minimale und durchschnittliche Hammingdistanz aus.

Tabelle 6.1: Vergleich von Zahlen mit guten/schlechten Codierungseigenschaften

A dezimal	A binär	avg. Hammingdistanz	min. Hammingdistanz
262147	0b01000000000000000011	14,248535	3
297893	0b01001000101110100101	15,856491	6
524287	0b01111111111111111111	13,230166	1
583127	0b10001110010111010111	15,956434	6

Bei diesen Berechnungen muss berücksichtigt werden, dass bisher nur die Binärabstände von AN-

Codierungen untersucht wurden. In der verwendeten ANB-Codierung kann eine Signatur B eine gute Hammingdistanz bei AN-codierten Zahlen wieder reduzieren. Im schlechtesten Fall kann es sogar soweit führen, dass die Hammingdistanz der im Speicher liegenden Variablen 1 ist. Diese Reduktion der Distanz ist jedoch nicht so kritisch wie es anfänglich scheint. Eine ANB-codierte Zahl ist nur durch die angewendete, dynamische Signatur verifiziert, welche zusätzlich zum codierten Wert gespeichert sein muss. Würde ein Bitfehler eine codierte Variable verändern und in eine andere codierte Variable überführen, wäre der Fehler dennoch erkennbar, da der codierte Wert nur in Kombination mit einer passenden Signatur gültig ist. Folglich müsste für einen nicht erkennbaren Fehler der codierte Wert wie auch die Signatur in spezieller Art und Weise verfälscht werden um die Anforderungen einer gültigen Codierung zu erfüllen. Davon abgesehen sind die Daten laut Konzept in Abb. 3.1 in der uncodierten Partition ein weiteres Mal in diversitäterer Darstellung vorhanden. Es müssten für einen unentdeckten Fehler drei unabhängige Zahlen in sehr spezieller Weise verändert werden um wiederum ein gültiges, übereinstimmendes und dennoch falsches Ergebnis zu generieren.

6.2 Ausführungszeit

Im Folgenden wird die Laufzeit und die Perfomance-Einbuße durch die Codierung in den erstellten Applikationen untersucht. Dazu wird die gleiche Aufgabe einmal codiert, einmal uncodiert mit Signaturberechnung und einmal nativ, ohne jede Codierungserweiterung ausgeführt. Die native Implementierung wird nur in diesem Abschnitt verwendet, um den gesteigerten Ressourcenbedarf zu messen. Die Messungen wurden auf dem in Abschnitt 5.1 erwähnten Intel Q9400S ausgeführt. Übersetzt wurden die Programme mit dem GNU Compiler Collection (GCC) Version 4.7.2 und Optimierungsstufe -O0 (keine Optimierung) [3]. Die Messung der Laufzeit wurde mit der Linux Standardfunktion *time* [6] durchgeführt. Da diese Funktion vor allem bei kleinen Ausführungszeiten sehr ungenau wird, wurden die Programme mehrfach ausgeführt.

Eine einzelne Matrizenmultiplikation ist selbst codiert sehr schnell durchgeführt, weshalb eine Vielzahl von Durchläufen ausgeführt und das arithmetische Mittel errechnet wurde, um somit in einen Laufzeitbereich zu kommen, welcher mit der *time* Funktion zuverlässig gemessen werden kann. Die Ergebnisse der durchgeführten Messungen sind in Tabelle 6.2 ersichtlich. Es zeigt sich, dass im Vergleich zur nativen Ausführung Perfomanceeinbußen von ca. Faktor 13 auftreten. Dieser Wert ist um Faktor 10 besser als der in der anderen Publikationen [Sch11, S. 166] für ANB-Codierung ermittelt. Die uncodierte Berechnung mit zusätzlicher Signaturberechnung ist ungefähr um Faktor 7 langsamer als die native Ausführung. Dieses Ergebnis passt ins Bild, da der uncodierte Berechnungszweig deutlich simpler als der codierte ist. Bei jeder Berechnung ist neben dem eigentlichen Ergebnis noch die Signatur zu kalkulieren. Hierbei handelt es sich meist um nur eine weitere arithmetische Operation und fallweise eine Signaturkorrektur, falls die Bedingung Eq. 2.9 verletzt wird. Auffallend ist der geringe Performanceunterschied zwischen codierter und uncodierter Berechnung. Dieser ist nur Faktor zwei, obwohl die codierte Berechnung vor allem bei Multiplikationen komplex ist.

Die zweite Beispielapplikation, die Sicherheitssteuerung, wurde ebenso mehrfach ausgeführt um in einen korrekt messbaren Bereich der Ausführungszeit zu gelangen. Die arithmetischen Mittelwerte der Ausführungszeit sind in Tabelle 6.3 wiedergegeben. Auffällig ist der starke Performance-Verlust im Vergleich zur nativen Berechnung. Die verlängerte Programmlaufzeit von Faktor 50 ist darin begründet, dass die native Ausführung mit boolescher Logik und booleschen Variablen durchgeführt wurde. Diese ist deutlich schneller als eine Berechnung mittels Integerwerte, wie

Tabelle 6.2: Performance der Matrizenmultiplikation bei 10^6 Ausführungen

Ausführungsmodus	Laufzeit [s]	Performance-Faktor
codiert	93,3	12,8
uncodiert	49,3	6,7
nativ	7,3	1

es bei der codierten und uncodierten Ausführung der Fall ist. Betrachtet man wieder den Unterschied zwischen codierter und uncodierter Berechnung, so fällt ein Unterschied von Faktor 2 auf, wie er schon bei der Matrizenmultiplikation vorkam. Damit zeigt sich, dass der Performance-Unterschied von codiert bzw. uncodiert mit Signatur großteils von der deutlich performanteren Implementierung abhängt.

Tabelle 6.3: Performance der Sicherheitsapplikation

Ausführungsmodus	Anzahl	Laufzeit [s]	Performancefaktor
codiert	10^7	81,2	49,2
uncodiert	10^7	36,0	21,8
nativ	10^8	16,5	1

Untersuchungen in der Literatur haben für ANB-Codierung einen Performanceverlust von bis zu Faktor 130 [Sch11, S. 166] berechnet. Die Ursache dafür liegt, soweit abschätzbar, in der unterschiedlichen Implementierung der Mathematik-Bibliothek und der Codierung. Die hier vorgestellte Implementierung einer ANB-Codierung wurde in C geschrieben. In der erwähnten Literatur ist dies in der Low Level Virtual Machine (LLVM) [10] geschehen. Dieser Compilerunterbau bietet einen Nachteil. Wenn die Erweiterungen von Hand geschrieben werden, ist es möglich, dass nicht das volle Optimierungspotential ausgeschöpft wird, wie es der Fall ist, wenn die Aufgaben von einem bewährten und optimierten Compiler wie dem GCC verarbeitet werden. Zweitens ist die hier verwendete Codierung aufgrund der dynamischen Signatur B performanter aufgebaut, da nicht nach jeder Berechnung inklusive Probe das Ergebnis auf das statische B zurückgeführt werden muss.

6.3 Fehleraufdeckung

Dieser Abschnitt behandelt die Ergebnisse der Fehlerinjektionen in die Matrizenmultiplikation und die Sicherheitsapplikation. Zuerst wird die Fehleraufdeckungswahrscheinlichkeit der codierten Ausführung untersucht, in einem weiteren Schritt wird dann die codierte Berechnung mit der uncodierten (inklusive Signaturberechnung) verglichen. Das Verhalten eines Programmes wurde in folgende Kategorien eingeteilt.

Programmende: In diese Kategorie fallen Testdurchgänge, in welchen das Programm unerwartet beendet wurde (z. B. durch eine Speicherzugriffsverletzung). Erkannt wird dies daran, dass kein oder nur ein unvollständiges Ergebnis ausgegeben wird.

Korrekt: Liefert das Programm trotz der Fehlerinjektion ein Ergebnis, welches dem erwarteten entspricht, fällt es in diese Kategorie.

Check: Diese Kategorie bezeichnet Fehler, welche anhand der internen Codierungsüberprüfung mittels Modulo-Probe aufgedeckt werden. Diese findet nach jedem Rechenschritt statt.

SDC: Silent Data Corruptions stellen die gefährlichen, unentdeckten Fehler dar. Das Ergebnis der Berechnung ist falsch, jedoch hat keine der Diagnosemaßnahmen die Abweichung erkannt.

6.3.1 Matrizenmultiplikation

In die Matrizenmultiplikation wurden Fehler mit der Fehlerinjektionssoftware Operanden und Operatorenfehler injiziert. Die Operanden wurden transient durch andere ersetzt und die Operatoren durch einen oder mehrere Bit-Flips verfälscht.

Operatorenverfälschung

Bei der Operatorenverfälschung wurde der Assembler-Code der Matrizenmultiplikation analysiert. Vorkommende Assembler-Instruktionen wurden bei einer der Ausführungen durch andere ersetzt. Das Ergebnis der codierten Fehlerinjektion ist in Abb. 6.2 dargestellt.

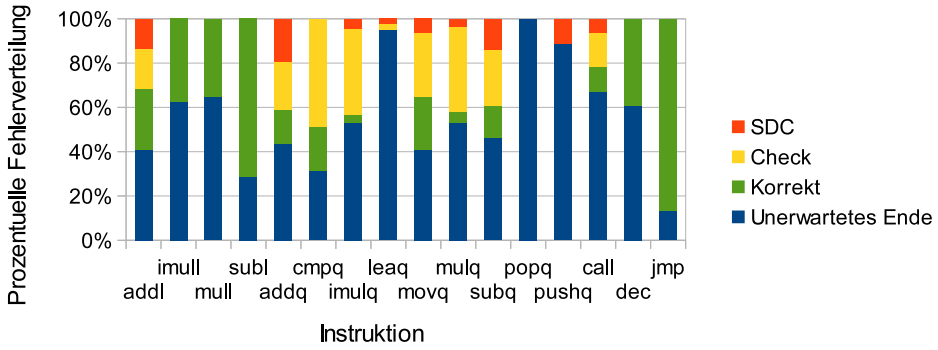

Abbildung 6.2: Verteilung der Fehler bei Operatorenverfälschung der codierten Matrizenmultiplikation

Auffällig ist trotz Codierung die hohe Rate an SDCs bei den Operationen `addl` und `addq`. Dies ist darin begründet, dass die Matrixmultiplikation drei geschachtelte Schleifen beinhaltet, in welchen die Indexvariablen nicht durch die Codierung geschützt sind. Ein Fehler in diesen Variablen verändert den Programmfluss und das Ergebnis. Da der Programmfluss nur durch Vergleich mit dem zweiten Kanal überwacht werden kann, treten in dieser Untersuchung viele SDCs auf. Insgesamt ist die Wahrscheinlichkeit für einen SDC bei 5 %. Die Wahrscheinlichkeit, dass das Programm zu einem vorzeitigen Ende kommt ist am höchsten, vor allem wenn die Stack-Operationen wie `pushq`, `popq` oder `leaq`[1] beeinflusst werden.

Das Ergebnis der Fehlerinjektion in die uncodierte Berechnung ist in Abb. 6.3 dargestellt. Das Programmverhalten bei der codierten und der uncodierten Ausführung zeigt Parallelen. Die Wahrscheinlichkeit für ein unerwartetes Programmende ist bei den Befehlen `addl`, `imull`, `mull`, `subl` und `addq` in den beiden Ausführungen annähernd gleich. Ebenso ist die Wahrscheinlichkeit, dass beide Ausführungen der Stack-Operation unerwartet beendet werden, sehr hoch.

[1] `leaq`: Load Effective Address

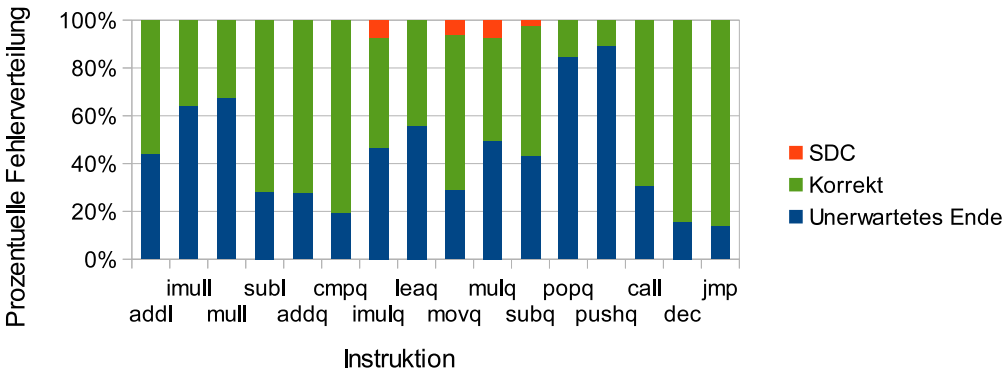

Abbildung 6.3: Verteilung der Fehler bei Operatorenverfälschung der uncodierten Matrizenmultiplikation mit Signaturberechnung

Der Vergleich der codierten Berechnung mit der uncodierten zeigt ein überraschendes Bild. Die uncodierte Ausführung scheint unempfindlicher gegenüber Fehlerinjektionen zu sein als ihr codiertes Pendant. Die Wahrscheinlichkeit für einen SDC liegt im Mittel über alle Fehler bei 1,4 %. Nur die Multiplikationen und der Kopierbefehl `movq` scheinen verwundbar. Eine erste Vermutung, dass die Berechnung einer Matrizenmultiplikation für diese Analyse ungeeignet ist, da möglicherweise eine Fehlerfortpflanzung nicht gegeben ist, kann durch das Ergebnis der Fehlinjektion in die native Berechnung entkräftet werden. Das Ergebnis der Fehlerinjektion in die native Implementierung ist in Abb. 6.4 dargestellt. Bei der nativen Berechnung sind weniger Instruktionen eingetragen, da durch einen simpleren Programmaufbau nicht alle Instruktionen der codierten und uncodierten Implementierung vorhanden sind. Die native Ausführung zeigt eine hohe Fehleranfälligkeit auf Operatorenfehler, am Stärksten die Multiplikationen, gefolgt von den Additionen. Vereinzelt ist eine SDC-Wahrscheinlichkeit von bis zu 96 % bei `mulq` möglich, die mittlere SDC-Rate beläuft sich auf 54,7 %. Die Befehle `leaq`, `movq` und `addl` führen mit einer hohen Wahrscheinlichkeit zu einem unerwarteten Programmende.

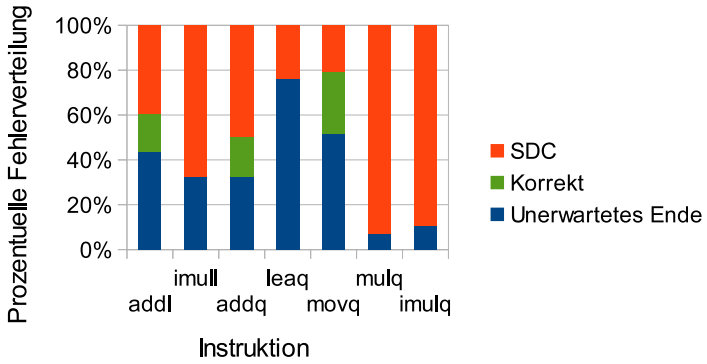

Abbildung 6.4: Verteilung der Fehler bei Operatorenverfälschung der uncodierten Matrizenmultiplikation mit Signaturberechnung

Dennoch bleibt die Frage zu klären, warum die uncodierte Implementierung mit Signaturberechnung eine derart niedrige Rate für SDCs aufweist, wenn der Beweis erbracht ist, dass der verwendete Algorithmus einer Matrixmultiplikation eine starke Fehlerfortpflanzung aufweist. Eine Diskussion dieser Ursachen ist in Abschnitt 6.3.4 durchgeführt.

Operandenverfälschung

Bei der Operandenverfälschung werden ein bis drei Bit-Flips in einem der CPU-Register eingebaut. Das Ergebnis für die codierte Berechnung ist in Abb. 6.5 dargestellt. Auf der Abszisse sind die Register und die Anzahl der injizierten Bit-Flips eingetragen.

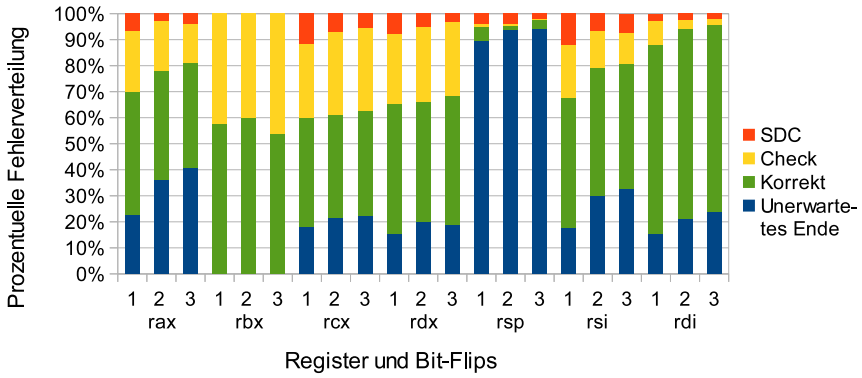

Abbildung 6.5: Verteilung der Fehler bei Operandenverfälschung der codierten Matrizenmultiplikation

Die Abb. 6.5 zeigt pro Register ein sehr ähnliches Verhalten. Die Wahrscheinlichkeit für ein unerwartetes Programmende steigt etwas mit der Anzahl der eingebauten Fehler. Die Anzahl der korrekt ausgeführten Berechnungen wird hingegen scheinbar nicht durch die Anzahl der Bitfehler beeinflusst. Auffällig ist, dass ein einzelner Bitfehler mit einer höheren Wahrscheinlichkeit zu einem SDC führen als ein Doppel- oder Dreifachfehler. Die SDC-Rate ist im Mittel 4,6 %. Eine Veränderung des Registers `rsp` führt bei einem Bitfehler mit einer auffallend hohen Wahrscheinlichkeit von mehr als 89 % zu einem unerwarteten Programmende. Bei diesem Register handelt es sich um den Stapelzeiger, welcher immer auf das oberste Element des Stacks zeigt. Die Empfindlichkeit auf Veränderungen den Stack betreffend, hat sich auch bei der Operatorenverfälschung in Abb. 6.2 für die Instruktionen `pushq` und `popq` gezeigt.

Die Ergebnisse der Fehlerinjektion in die Prozessorregister während der uncodierten Ausführung sind in Abb. 6.6 ersichtlich. Auch hier zeigt sich ein ähnliches Bild, mit einer etwas erhöhten mittleren SDC-Rate von 4,3 %, wie bei der Operandenverfälschung in der codierten Ausführung. Die Wahrscheinlichkeit für ein unerwartetes Programmende ist vom Register abhängig, jedoch nur gering von der Anzahl der injizierten Fehler. Mit jedem Bitfehler steigt die Wahrscheinlichkeit für ein unerwartetes Ende ein wenig. Der Stackpointer ist wieder das anfälligste Register für Programmabbrüche. Die Verteilung der SDCs ist der codierten Ausführung sehr ähnlich und es zeigt sich erneut das Phänomen, dass die uncodierte Berechnung nicht anfälliger für SDCs ist als die codierte Ausführung.

Die zur Kontrolle durchgeführte Operandenverfälschung in die native Implementierung ist in Abb. 6.7 dargestellt. Wie schon bei der codierten und uncodierten Ausführung steigt die Wahrscheinlichkeit für ein unerwartetes Programmende pro Register mit der Anzahl der Bitfehler. Am

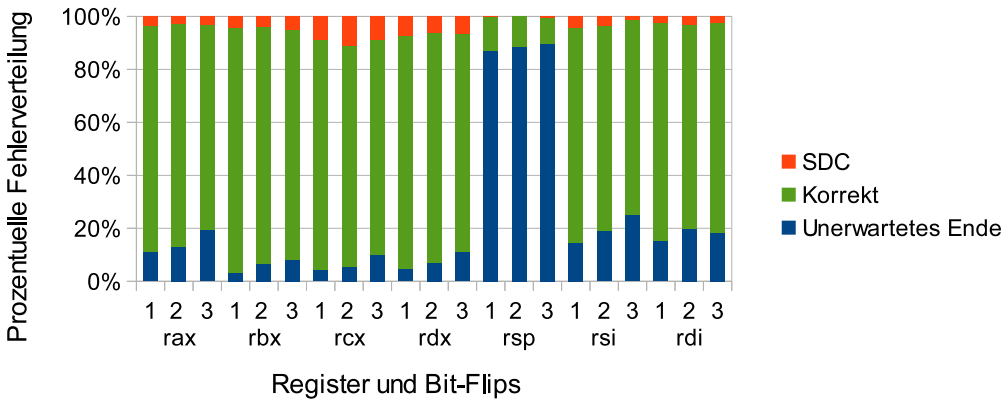

Abbildung 6.6: Verteilung der Fehler bei Operandenverfälschung der uncodierten Matrizenmultiplikation mit Signaturberechnung

anfälligsten ist wieder das Register für den Stackpointer `rsp`, doch auch der Akkumulator `rax` ist hier sehr empfindlich bei einer Wahrscheinlichkeit von ca. 58 % für einen Bitfehler und ansteigender Wahrscheinlichkeit bei mehreren Bitfehlern. Die Wahrscheinlichkeit für einen SDC ist wie bei der codierten Untersuchung in Abb 6.5 bei Einzelfehlern am höchsten und nimmt bei steigender Anzahl der Bitfehler ab. Insgesamt liegt die Wahrscheinlichkeit für einen SDC bei 11,9 %.

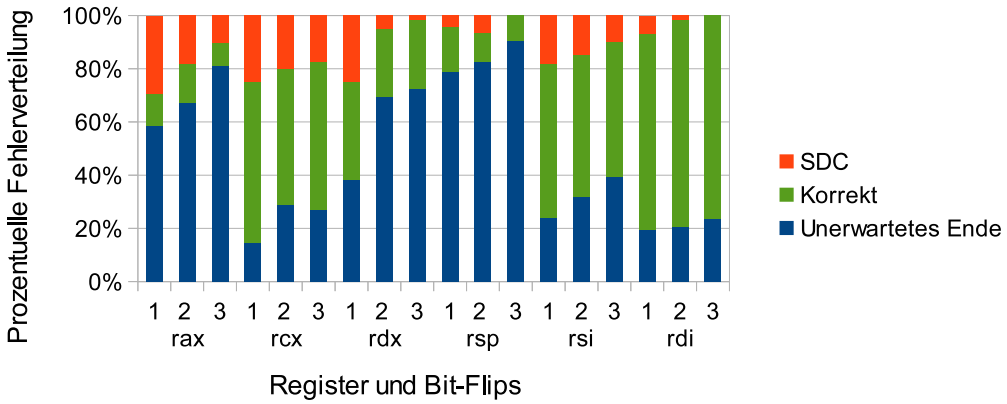

Abbildung 6.7: Verteilung der Fehler bei Operandenverfälschung der nativen Matrizenmultiplikation

6.3.2 Sicherheitsapplikation

Die Sicherheitsapplikation, welche eine Beispielsteuerung für einen Roboter darstellt, wurde der gleichen Fehlerinjektion wie die Matrizenmultiplikation unterzogen. Die implementierte Sicherheitsapplikation unterscheidet sich stark von der Matrizenmultiplikation. Die Matrizenmultiplikation benötigt exzessiv arithmetischer Operationen, während die Sicherheitsapplikation großteils aus If-Abfragen besteht und nur wenigen arithmetischen Berechnungen.

Operatorenverfälschung

Die Ergebnisse der Operatorenverfälschung sind in Abb. 6.8 dargestellt. Verfälschungen von `pushq` und `popq` führen wieder, wie bei der codierten Matrixmultiplikation, mit einer sehr hohen Wahrscheinlichkeit zu einem unerwarteten Programmende. Gefährliche Fehler treten hier am häufigsten bei Additionen, Kopiervorgängen und Funktionsaufrufen (`call`) statt, im Mittel haben 2,0 % zu einem gefährlichen Ausfall geführt.

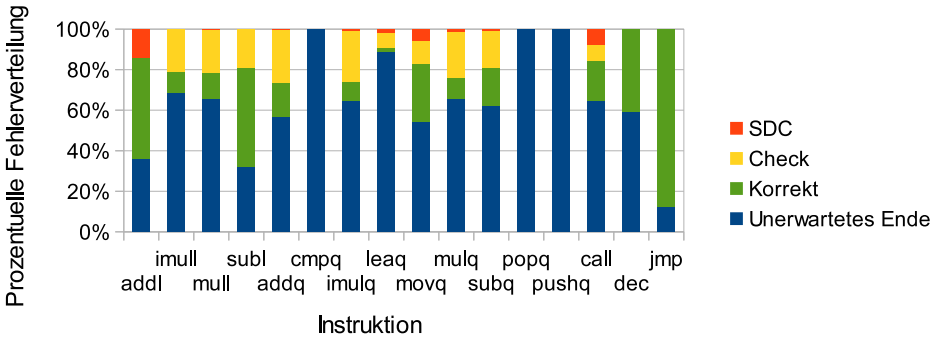

Abbildung 6.8: Verteilung der Fehler bei Operatorenverfälschung der codierten Sicherheitsappikation

Die uncodierte Ausführung mit Signaturberechnung in Abb. 6.9 zeigt ein ähnliches Verhalten, wie es schon bei der Matrixmultiplikation der Fall war. Die uncodierte Ausführung scheint nicht fehleranfälliger zu sein als die codierte Berechnung. Die Befehle, welche bei der uncodierten Sicherheitsapplikation zu SDCs führen, sind bei der uncodierten Ausführung im geringen Maße erhöht. Die durchschnittliche Wahrscheinlichkeit für einen SDC liegt hier bei 3 %.

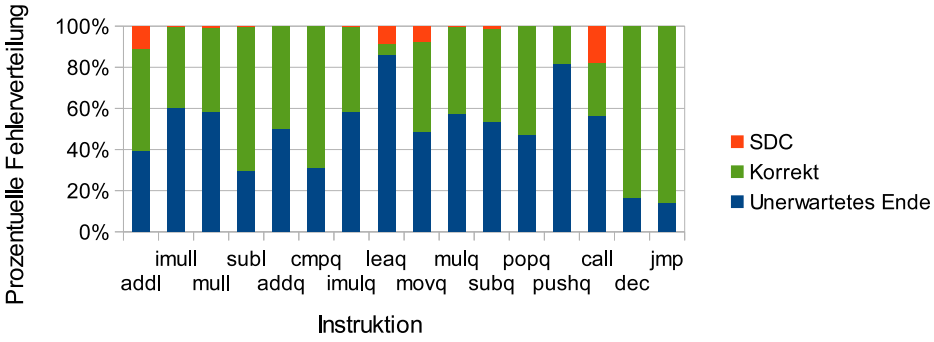

Abbildung 6.9: Verteilung der Fehler bei Operatorenverfälschung der uncodierten Sicherheitsappikation

Betrachtet man die native Berechnung in Abb. 6.10 zeigt sich wieder das Ergebnis, dass die Anwendung prinzipiell anfällig auf Operatorenfehler reagiert. Die verfälschten Befehle sind nicht die gleichen wie bei den beiden bisherigen Abbildungen. Die Ursache liegt darin, dass die native Ausführung keine 64 Bit Variablen und entsprechende Befehle benötigt, weshalb vom Compiler meist die 32 Bit Register und Instruktionen verwendet werden. Diese sind erkennbar an der Endung „l“ für 32 Bit Befehle, anstatt einem „q“ für 64 Bit. Durchschnittlich endeten 21,0 % der

fehlerhaften Ausführung mit einer SDC. Am anfälligsten ist der Befehl `jne`[2], gefolgt von `addl` und `cmpl`[3]. Diese stellen in der nativen Ausführung das Herzstück dar. In der Applikation werden, ebenso wie bei der codierten/uncodierten Version, die logischen Ergebnisse durch eine Addition für das UND-Gatter und mehrere If-Abfragen errechnet. Wird eine davon verfälscht, wirkt sich das mit hoher Wahrscheinlichkeit auf das Ergebnis aus. Ein unerwartetes Programmende ist wieder bei den Stack-Operationen `pushq` und `popq` am wahrscheinlichsten.

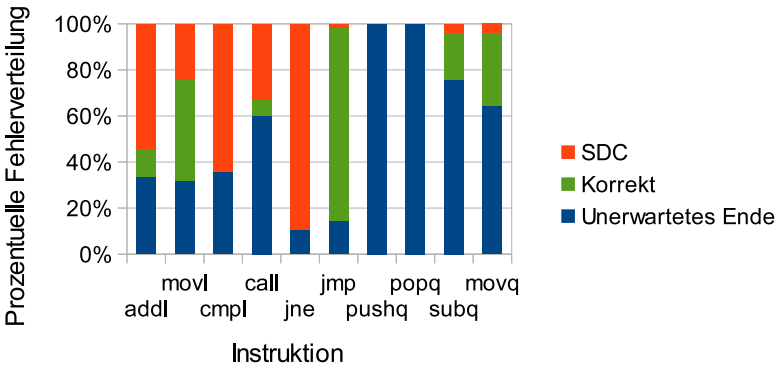

Abbildung 6.10: Verteilung der Fehler bei Operatorenverfälschung der nativen Sicherheitsappikation

Operandenverfälschung

Bei der Operandenverfälschung der Sicherheitsapplikation wurde gleich vorgegangen wie bei der Matrixmultiplikation. Es wurden ein bis drei Fehler in die verwendeten Prozessorregister injiziert, um Speicherfehler zu simulieren. Das Register und die Anzahl der eingebauten Bit-Flips kann wieder an der Abszisse abgelesen werden.

Die Fehlerinjektion der codierten Ausführung ist in Abb. 6.11 ersichtlich. Mit steigender Anzahl von Fehlerinjektionen steigt die Wahrscheinlichkeit für ein unerwartetes Programmende bei allen Registern an. Die Wahrscheinlichkeit für einen SDC ist, mit Ausnahme des Registers `rdx`, bei einzelnen Bitfehlern höher als bei mehreren und liegt im Mittel bei 2,1 %. Das Gesamtergebnis ähnelt sehr dem Ergebnis der codierten Matrizenmultiplikation in Abb. 6.5. Das Stack-Pointer-Register `rsp` fällt wieder durch die hohe Wahrscheinlichkeit an Programmabbrüchen auf.

Ein ebenso konsistentes Bild zeigt sich bei der Operandenverfälschung der uncodierten Ausführung mit Signaturberechnung, dargestellt in Abb. 6.12. Wieder sind die Ergebnisse der codierten Operandenverfälschung in Abb. 6.6 ähnlich. Die mit der Anzahl an injizierten Bitfehlern steigende Wahrscheinlichkeit für ein unerwartetes Programmende und das empfindliche Reagieren auf Veränderungen des Stackregisters sind auch hier zu beobachten. Die Anzahl der SDCs ist wieder gering, im Mittel 2,7 %.

Zum Vergleich wurde die native Implementierung ebenfalls durch Bit-Flips beeinflusst. Das Ergebnis in Abb. 6.13 zeigt keine Ähnlichkeit zu den bisherigen Ergebnissen. Bedingt ist dies durch

[2] jne: jump not equal
[3] cmpl: compare

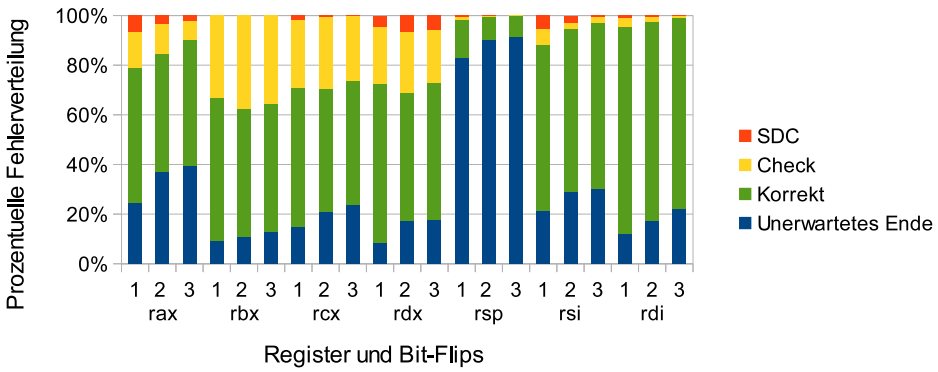

Abbildung 6.11: Verteilung der Fehler bei Operandenverfälschung der codierten Sicherheitsappikation

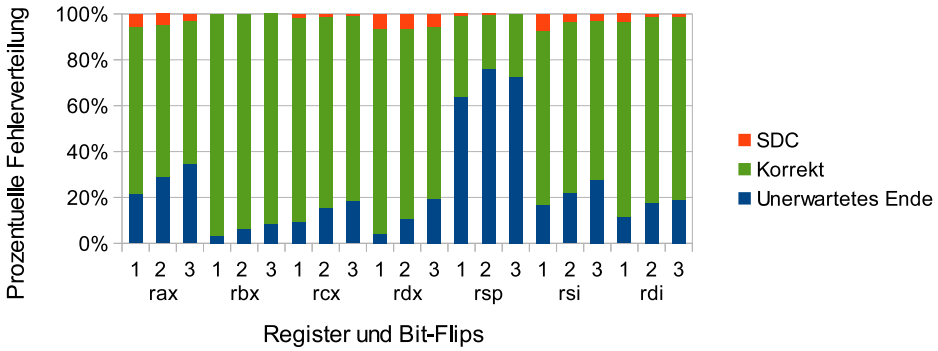

Abbildung 6.12: Verteilung der Fehler bei Operandenverfälschung der uncodierten Sicherheitsappikation mit Signaturberechnung

den strukturell unterschiedlichen Aufbau des Programms. Während bei der codierten und uncodierte Implementierung das Ergebnis mit Hilfe der Grundrechenarten erstellt wurde, ist bei der nativen dies nicht nötig. Beispielsweise werden Vergleiche direkt als If-Abfragen durchgeführt und basieren nicht, wie bei der codierten bzw. uncodierten Version, auf einer Subtraktion. Durch den einfacheren Aufbau und die reduzierte benötigte Bitbreite werden meist nur die 32 Bit Register verwendet. Das Register `ebx` wird während der ganzen Ausführung nicht benötigt. Die Nutzdaten des Programms werden nur in Register `eax` prozessiert, weshalb dieses am anfälligsten auf SDCs ist. Veränderungen in den anderen Registern führen fast immer zu einem Programmabbruch. Die Wahrscheinlichkeit für einen SDC liegt über alle Register bei 6,0 %. Betrachtet man nur das `eax`-Register liegt die Wahrscheinlichkeit bei 42,3 %.

6.3.3 Fehlerinjektion in codierte und uncodierte Ausführung

Da bei beiden Beispielapplikationen die codierte und uncodierte Verarbeitung untersucht wurde, wäre der nächste Schritt ein Parallelbetrieb einer codierten und einer uncodierten Instanz nebeneinander. Ein Einzelfehler wäre in dieser Konfiguration immer aufdeckbar, da er nur eines

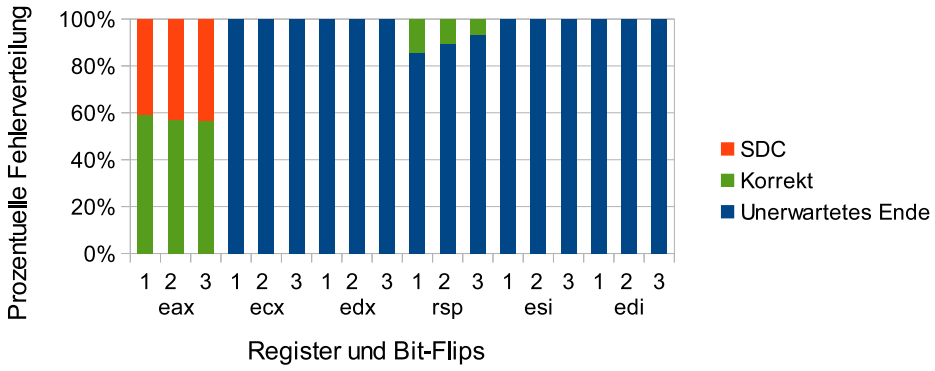

Abbildung 6.13: Verteilung der Fehler bei Operandenverfälschung der nativen Sicherheitsappikation

der beiden Programme betreffen kann. Somit würde eine fehlerinjizierte Ausführung mit einer fehlerfreien verglichen werden. Das Ergebnis dieser Untersuchung ist ohne Durchführung des eigentlichen Tests vorhersehbar. Jeder Fehler wird erkannt, da ein Ergebnis korrekt sein muss und das andere entweder mit diesem übereinstimmt, wenn sich der injizierte Fehler nicht auf das Ergebnis auswirkt, oder davon abweicht und durch den Vergleich mit der korrekten Ausführung erkannt werden kann. Diese Art der Untersuchung würde folglich keine verwertbaren Daten liefern, da das Verhalten von vornherein feststellbar ist.

Die Untersuchung kann dahingehend adaptiert werden, dass in jede der beiden Instanzen ein Fehler injiziert wird. Betrachtet man die bisherigen Ergebnisse der getrennten Ausführungen und Injektionen, so lässt sich aus der durchschnittlichen Wahrscheinlichkeit für einen SDC berechnen, wie viele Durchläufe nötig sind, damit beide Applikationen einen SDC generieren. Wenn beispielsweise in die codierte und uncodierte Ausführung der Matrixmultiplikation ein Operatorenfehler injiziert wird, ist die Wahrscheinlichkeit, dass in beiden Anwendungen bei der gleichen Ausführung ein SDC auftritt 0,07 %. Dies ergibt sich aus der Wahrscheinlichkeit für einen SDC in der codierten Ausführung (5 %), multipliziert mit der Wahrscheinlichkeit bei der uncodierten Ausführung (1,4 %). Anders interpretiert führt ungefähr jede 1.428te Ausführung zu einem SDC in beiden Kanälen. Diese Konstellation führt jedoch nicht zwingend zu einem gefährlichen Ausfall des Gesamtsystems, denn das ist nur der Fall, wenn beide Berechnungen das gleiche, falsche Ergebnis liefern. Dass die Wahrscheinlichkeit hierfür sehr gering ist, zeigt sich aus der Analyse der Daten, welche bei der getrennten Fehlerinjektion in codierte und uncodierte Ausführung erstellt wurden.

Bei insgesamt 35.721 in die codierte und uncodierte Implementierung der Matrixmultiplikation durchgeführten Fehlerinjektionen entstanden 1.407 SDCs. Eine Aufzeichnung der Ergebnisse ergab, dass keine zwei SDCs das gleiche, falsche Ergebnis erzeugten. Somit wäre ein Parallelbetrieb und das Warten auf eine übereinstimmende gefährliche Abweichung nicht zielführend. Ein ähnliches Bild zeigt sich, wenn die Ergebnisse der Sicherheitsapplikation untersucht werden. Bei 38.991 Fehlerinjektionen in die codierte und uncodierte Berechnung, wurden 958 SDCs gefunden. Auch hier zeigt der Vergleich der Ergebnisse, dass keine zwei Mal das Ergebnis auf die gleiche Weise verfälscht wurde.

6.3.4 Diskussion des Messverfahrens

Im Laufe der Testdurchführung haben sich mehrere Schwachstellen der Analyse auf Assembler-Ebene gezeigt, welche im Folgenden diskutiert werden.

Das auffällige Verhalten der Messergebnisse, nämlich dass die uncodierte Ausführung einerseits sicherer als die native scheint und andererseits sogar sicherer als die codierte, kann auf mehrere, potentielle Ursachen zurückgeführt werden.

Der geringe Unterschied an SDCs zwischen der codierten und der uncodierten Ausführung kann eine Auswirkung der Programmstruktur sein. Wenn man davon ausgeht, dass es Teile des Kontrollflusses gibt, welche das Endergebnis nicht beeinflussen, würde eine Verfälschung in diesen das Ergebnis nicht verändern. Ist dieser Teil der Probe, wird der Fehler mit der hohen Wahrscheinlichkeit von $1/A$ zur Laufzeit aufgedeckt und das Programm aufgrund eines Fehlers beendet. Dies trifft natürlich nur während der codierten Berechnung zu, da die uncodierte Berechnung keine Probe besitzt. Die hohe Fehleraufdeckung führt zu einer reduzierten Verfügbarkeit im codierten Zweig.

Vergleicht man die native Ausführung mit der uncodierten, so kann eine Begründung für den geringen Anteil an SDCs ebenso in dem Kontrollfluss liegen. Die codierte und uncodierte Implementierung weisen einen komplizierteren Kontrollfluss auf als die native Berechnung, welche am Beispiel der Matrixmultiplikation nur aus drei Schleifen, einer Addition und einer Multiplikation bestehen. Dieser straffe Kontrollfluss führt dazu, dass sich ein Fehler mit hoher Wahrscheinlichkeit auf das Endergebnis auswirkt. Die codierte und uncodierte Matrixmultiplikation mit Signaturberechnung ist komplizierter aufgebaut und besitzt Rechenzweige, welche nicht direkt das Endergebnis beeinflussen. Folglich ist es verständlich, dass bei einer ausführlicheren Implementierung die Auswirkung auf das Ergebnis geringer ist. Für die Sicherheitsapplikation gelten, im übertragenen Sinne, die gleichen Argumente.

Aus dem Ergebnis, dass die uncodierte Implementierung weniger SDCs besitzt, darf nicht der Schluss gezogen werden, dass ein umfangreicherer Kontrollfluss zu einem sichereren Programm führt. Die Ursache ist viel eher eine geringere Beeinflussung durch die Fehlerinjektion. Die native Implementierung benötigt deutlich weniger Instruktionen, wodurch die Anzahl der Fehler pro Codezeile höher ist als bei der codierten bzw. uncodierten Implementierung.

Es zeigte sich, dass der Assembler eine Einschränkung bei der Durchführung der Fehlerinjektionen darstellt. Die Aufgabe des Assemblers ist es, den Assembler-Code in eine ausführbare Datei zu übersetzen. Er verfügt typischerweise über keine quellcodeverändernden Verfahren, wie Optimierungen oder ähnliches, jedoch beinhaltet der GNU Assembler eine einfache Syntaxprüfung, welche gewisse Fehler erkennt und die Übersetzung verhindert. Beispielsweise erkennt er, wenn dem Kopierbefehl `mov` nur ein Parameter anstatt zweien übergeben wird. Somit sind gewisse Fehlerkombinationen nicht ausführbar, weil sie den Assembler nicht passieren können. Deren Auswirkungen auf die Messergebnisse können nur schwer abgeschätzt werden, jedoch ist es sehr wahrscheinlich, dass von Grund auf falsche Instruktionen im Prozessor eine Exception auslösen, welche die Berechnung unterbrechen und somit eine unentdeckte Verfälschung des Ergebnisses verhindern.

Die Analyse auf Assembler-Ebene zeigt bei näherer Betrachtung der Messergebnisse einen weiteren Schwachpunkt. Die Ergebnisse der Fehlerinjektion sind stark abhängig von dem Vorkommen der Assembler-Instruktionen. Kommt ein Befehl in einem Programm nur einmal, an einer unkritischen Stelle vor, so führt eine Verfälschung dieses Befehls möglicherweise zu keiner Beeinflussung

des Ergebnisses. Eine naheliegende Interpretation, dass der Befehl keine gefährliche Beeinflussung des Ergebnisses verursacht, ist nur für diese eine Anwendung gültig und darf nicht verallgemeinert werden. Eine andere Applikation, welche den Befehl öfters verwendet, kann ein deutlich anderes Verhalten zeigen. Selbst wenn nur ein anderer Compiler für das gleiche Programm verwendet wird, kann das Ergebnis der Fehlerinjektion abweichen, da ein anderer Compiler die Assembler-Instruktionen anders kombiniert.

6.4 Abschätzung der Umsetzbarkeit

Mit den gewonnenen Erkenntnissen aus der theoretischen und praktischen Analyse soll nun abgeschätzt werden, ob das vorgeschlagenen Konzept in Kapitel 3 realisierbar ist und die gestellten Anforderungen erfüllen kann.

Von dem theoretischen Standpunkt aus, liefert die codierte Verarbeitung von Daten ein in sich geschlossenes System mit einer hohen Fehlererkennungswahrscheinlichkeit im Falle einer Codeverletzung. Es muss jedoch bedacht werden, dass Softwareprogramme weitere Anforderungen besitzen. Das Problem eines Kontrollflusses, in welchem sich Rechenwege auftrennen und möglicherweise wieder treffen, führen zu Schwachpunkten in der Fehleraufdeckung. Beispielweise sind If-Abfragen in codierter Form nicht möglich. Ein Vergleich zwischen Variablen führt unabhängig von der Codierung früher oder später zu einem Ergebnis „wahr“ oder „falsch“, anhand dessen über die weitere Berechnung entschieden wird. Diese booleschen Werte können nicht codiert werden, da der Prozessor intern Entscheidungen anhand einzelner, uncodierter Bits trifft.
Ein weiteres Problem, für welches Coded Processing keine Lösung liefert, ist die Handhabung von Indexvariablen. Werden in der Softwareapplikation Felder verwendet, werden deren Elemente typischerweise durch die Angabe des Indexes angesprochen. Würde der Index codiert, müsste ein großer Speicherbereich für das Feld angelegt werden, von welchem nur ein kleiner Teil verwendet wird. Ebenso müsste dann von der dynamischen Signatur B Abstand genommen werden, da es sehr aufwändig wäre, den Inhalt einer Speicherstelle, welche kontinuierlich ihren Speicherplatz wechselt, aufzufinden. Wird für den Index der Felder die Signatur entfernt und nur AN-Codierung verwendet, besteht weiterhin das Problem des großen Speicherverbrauchs, da die Daten weit verstreut im Speicher liegen würden. Ein Vorteil dieser Verteilung der Daten im Speicher wäre allerdings, dass logisch benachbarte Speicherstellen im (physikalischen) Speicher weiter voneinander getrennt liegen und so eine gegenseitige Beeinflussung von Speicherstellen reduziert werden könnte.

Die erwähnten Schwachpunkte lassen sich durch die doppelte und diversitäre Berechnung jedes Ergebnisses entkräften, jedoch nicht mit der vollen Wirksamkeit, wie es ein mathematisch geschlossenes System bieten könnte. Eine genaue Untersuchung etwaiger Fehler gemeinsamer Ursache sind somit nötig, um nachweisbare Sicherheit zu erreichen.

Passende Parameter für die Codierung zu finden stellt eine herausfordernde Aufgabe dar, da der benötigte Rechenaufwand um dies zu bewerkstelligen enorm ist. Dennoch kann es prinzipiell als machbar angesehen werden. Eine minimale Hammingdistanz lässt sich durch die Verwendung der dynamischen Signatur B nicht bestimmen. Dem wird entgegengewirkt, da die Nutzdaten für die Berechnung doppelt und in diversitärer Form vorliegen.

Die Analyse der CCFs stellt eine der größten Herausforderungen des in Kapitel 3 vorgestellten Konzepts dar. COTS-Hardware verfügt über mehr Funktionen als die bisher verwendeten CPUs

in Sicherheitssteuerungen (siehe Abschnitt 2.1.8). Es muss bedacht werden, dass für eine Zertifizierung alle Elemente der Hardware untersucht werden und ihre potentielle Gefährdung auf das Gesamtsystem analysiert werden müssen. Dies gilt ebenso für nicht benötigte, jedoch in Hardware vorhandene Komponenten, wie z. B. den Audiocontroller. Eine Untersuchung dieser Art ist einerseits sehr zeitintensiv und andererseits kann es in weiterer Folge zu Hardwareabhängigkeiten führen, welche laut Anforderungen eigentlich verhindert werden sollen.

Um die Berechnung doppelt durchführen zu können, ist ein Hypervisor notwendig. Dieses Stück Software muss ebenfalls zertifiziert sein. Am Markt befindliche Hypervisorlösungen können bereits eine Zertifizierung nach SIL 3 vorweisen. Dennoch muss beachtet werden, dass die Verwendung zertifizierter Komponenten kein Garant dafür ist, dass das zusammengestellte Gesamtsystem die Anforderungen an das Sicherheitslevel erfüllen. Das Zertifikat der Software bestätigt nur, dass sie nach einer entsprechenden Norm entwickelt und überprüft wurde, jedoch nicht, dass sie für ein spezielles Einsatzgebiet geeignet ist, wie z. B. wie hier zwei sicherheitskritische Rechenkanäle voneinander zu isolieren um virtuelle Zweikanaligkeit zu erreichen.

Die Abschätzung der zusätzlich benötigten Rechenleistung durch Coded Processing hat gezeigt, dass mit der hier gewählten Form einer Mathematik-Bibliothek in C große Ressourcen-Einsparungen, verglichen mit anderen Untersuchungen [Sch11, S. 166], möglich sind. Dem dennoch gesteigerten Ressourcenverbrauch kann entgegengewirkt werden, da in dem hier verwendeten Ansatz tagesaktuelle Hardware verwendet werden kann und nicht, wie im Anschnitt 2.1.8 diskutiert, sogenannte betriebsbewährte Hardware. Mit der erhöhten Rechenleistung der Prozessoren kann den Performanceverlusten entgegengewirkt werden. Somit ist eine Verwendung von Coded Processing vom Standpunkt des Ressourcenverbrauchs möglich.

Die Analyse auf Assembler-Ebene inklusive Fehlerinjektion hat mehrere Erkenntnisse zur Durchführbarkeit von ANB-codierten Anwendungen gezeigt. Eine eingehende Analyse der Messergebnisse ist in Abschnitt 6.3.4 durchgeführt. Für die Umsetzbarkeit muss hinzugefügt werden, dass für eine Zertifizierung keine empirische Untersuchung wie die durchgeführte entscheidend ist. Anhand der Untersuchung kann eine Richtung für die weitere Analyse gezeigt werden, ein Argument für die Sicherheit des Systems stellt sie jedoch nicht dar. Weiters zeigten die Ergebnisse der Fehlerinjektion eine starke Applikationsabhängigkeit, welche für die Erstellung generischer Sicherheitskomponenten hinderlich ist. Eine theoretische Analyse des Instruktion-Sets scheint notwendig, welche zeitaufwändig sein kann und möglicherweise zu Hardwareabhängigkeiten führen kann.

In der theoretischen Analyse anhand der Fehlermodelle aus der Literatur und der Norm schneidet das Konzept gut ab. Operanden- und Operatorenverfälschungen können durch die Codierung und die virtuelle Zweikanaligkeit erkannt werden. Für die Hardwarephänomene, welche außerhalb der Fehleraufdeckungsmöglichkeiten durch die Software stehen, kann der Watchdog Abhilfe schaffen. Die Funktionen und der Aufbau des Watchdogs sind simpel gehalten, wodurch dieser in der Entwicklung und Zertifizierung keine Herausforderung darstellen sollte.

7 Conclusio

Diese Arbeit zeigte ein Konzept für eine Sicherheitssteuerung für industrielle Anwendungen. Ausgehend von den am Markt befindlichen Lösungen wurde ein Konzept entwickelt, welches größtmögliche Hardwareunabhängigkeit ermöglichen sollte. Die Sicherheitsfunktionen wurden weitestgehend in Software abgebildet um eine Hardwareunabhängigkeit zu erreichen. Ziel dieser war es, die Hardwarekomponenten weitestgehend aus dem Zertifizierungsprozess der Sicherheitssteuerung zu entfernen. Somit wird es möglich, rein durch die Verwendung der speziellen Software, auf aktueller, rechenstarker Hardware sicherheitsrelevante Berechnungen durchzuführen. Den Performance-Einbußen durch die Verwendung der Codierung wird entgegengesetzt, dass statt betriebsbewährter, eingebetteter Hardware tagesaktuelle Hardwarekomponenten verwendet werden können, welche über ein vielfaches an Rechenleistung verfügen. In weiterer Folge kann die Verschmelzung von Steuerung und Sicherung verwendet werden, um die Effizienz und Sicherheit von industriellen Robotern zu erhöhen. Durch die einfache Erweiterung in Software ist es möglich auch Steuerungen sicherer zu machen, bei welchen bisher keine Sicherheitsmaßnahmen benötigt wurden.

In der Technologieanalyse in Kapitel 2 wurde ein Überblick über die relevanten Normen der funktionalen Sicherheit für das industrielle Umfeld gegeben. Im weiteren wurden Prinzipien vorgestellt, welche es ermöglichen, die funktionale Sicherheit von Bauelementen zu erhöhen. Ein gebrachtes Beispiel ist die Redundanz in ihren verschiedenen Ausformungen, wie 1oo2 oder 2oo2. Einen wichtigen Teil der Technologieanalyse stellt die Diskussion des Coded Processing dar. Dieses Verfahren verwendet eine arithmetische Codierung um etwaige Rechenfehler in arithmetischen Operationen aufdeckbar zu machen.

Im Kapitel 3 wurde ein Konzept für eine Sicherheitssteuerung vorgestellt, welches den Anforderungen nach SIL 3 nach IEC 61508 [IEC10a] bzw. PL e nach EN 13849 [EN08] gerecht werden soll. Es basiert auf einer einkanaligen, unzertifizierten Hardware. Den hohen Anforderungen der Norm an die SFF bzw. DC wird großteils in Software Rechnung getragen. Dazu wird jede Berechnung doppelt und in diversitärer Form durchgeführt. Somit sollen Hardwarefehler weitgehend für die Software erkennbar gemacht werden. Die diversitäre Darstellung und Berechnung der Daten wird durch Coded Processing in einem der Berechnungskanäle erreicht. Für Fehler, welche nicht eindeutig in der Software detektiert werden können, wurde eine zusätzliche Hardwareeinheit hinzugefügt. Diese Komponente, genannt Watchdog, überwacht die Umgebungsparameter der Sicherheitssteuerung. Zusätzlich fungiert er als Vergleicher, welcher die Gleichheit der Ergebnisse der Berechnungskanäle überwacht. Detektiert der Watchdog einen Fehler, kann er aufgrund seiner Position als Buswächter die Netzkommunikation trennen und somit verhindern, dass fehlerhafte Daten über den Feldbus versendet werden. Bei der Erarbeitung der Aufgaben des Watchdogs

wurde darauf geachtet, dass er zur Durchführung dieser nur geringe Rechenleistung benötigt. Somit wird es möglich, den Watchdog einerseits so simpel wie möglich zu entwerfen und andererseits zukünftige Performance-Optimierungen des Watchdog in abschätzbarer Zeit zu verhindern. Die immer leistungsfähiger werdenden PCs erzeugen mehr Daten, welche verglichen werden müssen und stellen steigende Ansprüche an den Watchdog. Durch die einfachen und dennoch wichtigen Überwachungsaufgaben kann davon ausgegangen werden, dass der Watchdog für weitere PC-Generationen nicht angepasst werden muss.
Die entwickelte Codierung orientiert sich an arithmetischen Codierungen, wie sie in anderen sicherheitsrelevanten Bereichen eingesetzt werden. Für das Umfeld der industriellen Steuer- und Sicherungseinrichtungen wurde eine Codierung erarbeitet, welche den gegebenen Anforderungen besser entspricht. Dazu mussten allfällige Korrekturfunktionen für alle Berechnungsarten und Bedingungen für optimale Codierungsparameter gefunden werden.

Dieses Konzept wurde in Kapitel 4 anhand von Fehlermodellen aus der Wissenschaft und der Norm analysiert. Die verwendeten Fehlermodelle bauen auf Systemebene auf. Untersuchungen anhand tiefer liegender Modelle, wie des Stuck-at-Modells, sind bei komplexen Elementen nicht durchführbar. Zusätzlich wurde das Fehlermodell der IEC 61508 Teil 2 detailliert betrachtet. Dieses behandelt in erster Linie physikalische Phänomene und deren mögliche Auswirkungen auf die Hardware, weitab von der Betrachtung auf Systemebene. Bei allen Analysen wurde darauf geachtet, ob spezifische, hardwareabhängige Fehler auftreten. Dies würde der Aufgabenstellung widersprechen und keine zu bevorzugende Lösung darstellen. Die durchgeführte Analyse zeigte, dass das vorgestellte Konzept für die Fehler der Modelle passende Gegenmaßnahmen liefert und keine Hardwareabhängigkeiten aufdeckte.

Zur Durchführung einer praktischen Analyse des Konzeptes, wurden in Kapitel 5 mehrere Programme erstellt. Allen voran eine Mathematik-Bibliothek, welche die Codierung, das codierte Rechnen und die Decodierung übernimmt. Anhand dieser Bibliothek soll die Verwendung von Coded Processing für den Anwender vereinfacht werden, damit dieser, im besten Fall, keinerlei Wissen über Coded Processing benötigt. Die Entwicklung der Mathematik-Bibliothek zeigte sich als sehr zeitintensiv, da für jede Rechenoperation Sonderfälle in der Codierung bestehen, welche erkannt und korrigiert werden müssen. Als herausfordernd zeigte sich, dass einerseits die Performance der Bibliothek hoch bleiben sollte und andererseits das Hantieren mit uncodierten Daten unbedingt verhindert werden musste. Dazu wurden neue Korrekturverfahren entwickelt, welche eine Verbesserung, zu den in der Literatur präsentierten, darstellen. Für den codierten Kontrollfluss konnten If-Abfragen erstellt werden, welche intern nicht auf uncodierten Werten basieren, wie es ebenso in der Literatur vorgeschlagen wurde. Somit ist es möglich, die Fehleraufdeckungswahrscheinlichkeit im Vergleich zu anderen, in wissenschaftlichen Publikationen präsentierten, Lösungen zu steigern.
Aufbauend auf diese Bibliothek wurden zwei Beispielanwendungen implementiert. Einerseits eine Matrizenmultiplikation, wie sie in der Kinematik häufig vorkommt und andererseits eine Sicherheitsapplikation mit logischen Gattern. Das Verhalten dieser Anwendungen in einem Fehlerfall wurde anhand einer Fehlerinjektion untersucht. Da sich am Markt keine passende Software befand, musste ebenso eine entwickelt und implementiert werden. Diese kann, zur Laufzeit des zu testenden Programms, nach den Fehlermodellen auf Systemebene typische Fehler erzeugen, wie sie ausgelöst durch Hardwarefehler vorkommen können. Die Fehlerinjektionssoftware muss so entwickelt werden, dass sie für das zu testende Programm vollkommen transparent wirkt, um nicht ungeplante Fehler in der Ausführung zu erzeugen und die Evaluierung zu verfälschen.

Die Ergebnisse der Fehlerinjektion und weitere Parameter der Beispielanwendungen wurden in Kapitel 6 gezeigt. Neben einer Untersuchung idealer Codierungsparameter, wurden Benchmarks

erstellt, welche die Leistungsfähigkeit der Mathematik-Bibliothek prüften und bewiesen. Die Performance liegt um ca. Faktor 10 über den in der Wissenschaft publizierten Werten. Ursachen dafür sind die Implementierung in C und die veränderte Codierung. Gegen das dynamische Verhalten der Signatur B wurde nicht angekämpft, sondern die dynamische Eigenschaft ausgenützt, um bei reduziertem Rechenaufwand dennoch eine hohe Fehleraufdeckung zu erreichen. Weiters kann durch das nunmehr dynamische Verhalten der entwickelten Codierung eine feinmaschige Kontrollflussüberwachung ermöglicht werden. Die Genauigkeit der Überwachung kann vom Programmierer festgelegt und somit der Applikation angepasst werden.
Bei der Fehleraufdeckung der Fehlerinjektionen zeigte sich ein auf den ersten Blick überraschendes Bild. Die codierte Berechnung schien gleich, oder sogar schlechter als die uncodierte. Dieses Verhalten konnte in der darauffolgenden Diskussion geklärt werden. Dennoch zeigte sich, dass eine Analyse des Konzepts auf Assembler-Ebene nicht zu idealen Ergebnissen führt. Eine starke Applikationsabhängigkeit erschwert die Verallgemeinerung der Ergebnisse. An diesem Punkt kann für weitere Arbeiten eingehakt werden, um ein besseres Analyseverfahren zu entwickeln und die hier präsentierten Ergebnisse zu verfeinern.

Appendix Glossar

Common Cause Failure (dt. Ausfall infolge gemeinsamer Ursache) Dieser Ausfall, der das Ergebnis einer oder mehrerer Ereignusse ist, die gleichzeitige Ausfälle von zwei oder mehreren getrennten Kanälen in einem mehrkanaligen System verursachen und zu einem Systemausfall führen [IEC10a, Teil 4, Clause 3.6.10].

Deadline Die Deadline bestimmt den Zeitpunkt, zu welchem die Berechnung spätestens beendet sein muss [Wör05, S. 355].

Error (dt. Abweichung) Die Nichtübereinstimmung zwischen Rechenergebnissen, beobachteten oder gemessenen Werten oder Beschaffenheiten und den betreffenden wahren, spezifizierten oder theoretisch richtigen Werten oder Beschaffenheiten [IEC10a, Teil 4, Clause 3.6.11].

Failure (dt. Ausfall) Die Beendigung der Fähigkeit einer Funktionseinheit, eine geforderte Funktion bereitzustellen oder Betrieb einer Funktionseinheit in irgendeiner Art anders als gefordert [IEC10a, Teil 4, Clause 3.6.4].

Fault (dt. Fehler) Die nicht normale Bedingung, die eine Verminderung oder den Verlust der Fähigkeit einer Funktionseinheit verursachen kann, eine geforderte Funktion auszuführen [IEC10a, Teil 4, Clause 3.6.1].

Hypervisor Unter einem Hypervisor versteht man ein Programm, welches eine virtuelle Betriebsumgebung für andere Programme bereitstellt [Fox12].

Risiko Kombination aus der Wahrscheinlichkeit, mit der ein Schaden auftritt, und dem Ausmaß des Schadens [IEC10a, Teil 4, Clause 3.1.6].

Sicherheit Die Freiheit von unvertretbarem Risiko [IEC10a, Teil 4, Clause 3.3.11].

Verfügbarkeit Die Verfügbarkeit eines Systems ist die Wahrscheinlichkeit, dass das System zu einem bestimmten Zeitpunkt korrekt funktioniert [Bör04, S. 5].

Wissenschaftliche Literatur

[Bac03] Backer, Reiner: *Assembler: Maschinennahes Programmieren von Anfang an. Mit Windows-Programmierung.* Rowohlt Taschenbuch Verlag, 2003 (rororo Taschenbücher). – ISBN 9783499612244.

[Ben98] Benso A., Prinetto P., Rebaudengo, M., Reorda M. Sonza: EXFI: A Low-cost Fault Injection System for Embedded Microprocessor-based Boards. In: *ACM Trans. Des. Autom. Electron. Syst.* 3 (1998), Oktober, Nr. 4, S. 626–634. – ISSN 1084–4309.

[Ben03] Benso A., Prinetto P. (Hrsg.): *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation.* Boston, MA, USA : Kluwer Academic Publishers, 2003. – ISBN 1–4020–7589–8.

[Bia09] Bianca Schroeder, Eduardo Pinheiro, Wolf-Dietrich Weber: DRAM Errors in the Wild: A Large-Scale Field Study. (2009.)

[Bör04] Börcsök, Josef: *Electronic Safety Systems.* 1. Hüttig GmbH, 2004. – ISBN 3–7785–2985–2944–7.

[Bör06] Börcsök, Josef: *Funktionale Sicherheit, Grundzüge sicherheitstechnischer Systeme.* Hüthig, 2006.

[Bra12] Braun Juergen, Mottok Juergen, Miedl Christian, Geyer Dirk, Minas Mark: Increasing the Reliability of single and multi core systems with software rejuventation and coded processing. In: *Automotive - Safety & Security 2012*, 2012. – ISBN 978–3–88579–604–6.

[Bro60] Brown, David T.: Error Detecting and Correcting Binary Codes for Arithmetic Operations. In: *Electronic Computers, IRE Transactions on* EC-9 (1960), Sept, Nr. 3, S. 333–337. – ISSN 0367–9950.

[CG92] Choi G.S., Iyer R.: FOCUS: an experimental environment for fault sensitivity analysis. In: *Computers, IEEE Transactions on* 41 (1992), Dec, Nr. 12, S. 1515–1526. – ISSN 0018–9340.

[Cla09] Claude Hennebert, Gerard Gumo: Xception: Software Fault Injection and Monitoring in Processor Functional Units. (2009.)

[EN08] EN: EN 13849: Sicherheit von Maschinen - Sicherheitsbezogene Teile von Steuerungen. (2008.)

[EN11] EN: Industrielle Kommunikationsnetze - Profile - Teil 3-3: Funktional sichere Übertragung bei Feldbussen. (2011.)

[EN12] EN: Industrielle Kommunikationsnetze - Profile - Teil 3-18: Funktional sichere Übertragung bei Feldbussen - Zusätzliche Festlegungen für die Kommunikationsprofilfamilie 18. (2012.)

[For89] Forin, P.: Vital Coded Microprocessor: Principles and Application for various Transit Systems. In: *Proc. IFAC-GCCT* (1989.)

[Fox12] Fox, Dirk: Hypervisor. In: *Datenschutz und Datensicherheit - DuD* 36 (2012), Nr. 1, S. 54–54. – ISSN 1614–0702.

[Fuc96] Fuchs, Emmerich: An Evaluation of the Error Detection Mechanisms in MARS using Software-Implemented Fault Injection. In: *Dependable Computing — EDCC-2, Second European Dependable Computing Conference, October 1996, Taormina, Italy Springer-Verlag, Lecture Notes in Computer Science, Volume 1150, pp. 73-90* (1996.), Oct.

[Gar66] Garner, Harvey L.: Error Codes for Arithmetic Operations. (1966.)

[GG09] Gaiswinkler G., Gerstinger A.: Automated software diversity for hardware fault detection. (2009), Sept, S. 1–7. – ISSN 1946–0759.

[Gol04] Goldfeld, D.: The Elementary Proof of the Prime Number Theorem: An Historical Perspective. In: Chudnovsky, David (Hrsg.) ; Chudnovsky, Gregory (Hrsg.) ; Nathanson, Melvyn (Hrsg.): *Number Theory*. Springer New York, 2004. – ISBN 978–1–4612–6490–3, S. 179–192.

[Gol06] Goloubeva Olga, Maurizio Rebaudengo, Matteo Sonza Reorda, Massimo Violante: *Software Implemented Hardware Fault Tolerance*. 1. Springer, 2006. – ISBN 0–387–26060–9.

[Hen94] Henrique Madeira, Francisco Moreira, João Gabriel Silva: RIFLE: A general purpose pin-level fault injector. In: Echtle, Klaus (Hrsg.) ; Hammer, Dieter (Hrsg.) ; Powell, David (Hrsg.): *Dependable Computing - EDCC-1* Bd. 852. Springer Berlin Heidelberg, 1994. – ISBN 978–3–540–58426–1, S. 197–216.

[HJL02] Hennessy John L., Patterson David A.: *Computer Architecture: A Quantitative Approach*. 3rd. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2002. – ISBN 1–55860–596–7.

[IEC03] IEC: Funktionale Sicherheit - Sicherheitstechnische Systeme für die Prozessindustrie. (2003.)

[IEC10a] IEC: IEC 61508: Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbarer elektronischer Systeme. (2010.)

[IEC10b] IEC: IEC 62061: Sicherheit von Maschinen - Funktionale Sicherheit sicherheitsbezogener elektrischer, elektronischer und programmierbarer elektronischer Steuerungssysteme. (2010.)

[Int10] Intel: Intel®Core™i7-900 Desktop Processor Extreme Edition Series and Intel®Core™i7-900 Desktop Processor Series, Datasheet, Volume 1. (2010.)

[Int13] Intel: Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture. (2013.)

[ISO12] ISO: Road vehicles - Functional safety. (2012.)

[Joã09] João Carreira, Henrique Madeira, João Gabriel Silva: Xception: Software Fault Injection and Monitoring in Processor Functional Units. (2009.)

[Kor07] Koren Israel, Krishna C. Mani: *Fault-Tolerant Systems*. Morgan Kaufmann Publishers, 2007. – ISBN 978–0–12–088525–1.

[Liu72] kai Liu, Chao: *Error-Correcting-Codes in Computer-Arithmetic*. United States Government, 1972.

[Lo92] Lo, Jien-Chung: Reliable floating-point arithmetic algorithms for Berger encoded operands. (1992.), Oct, S. 110–113

[Lo94] Lo, Jien-Chung: Reliable floating-point arithmetic algorithms for error-coded operands. In: *Computers, IEEE Transactions on* 43 (1994), Apr, Nr. 4, S. 400–412.. – ISSN 0018–9340

[Man13] Mandl, Peter: *Grundkurs Betriebssysteme*. 3rd. Wiesbaden, Deutschland : Springer Vieweg, 2013. – ISBN 978–3–8348–1897–3.

[Mir92] Miremadi G., Harlsson J., Gunneflo U., Torin, J.: Two software techniques for on-line error detection. In: *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, 1992, S. 328–335.

[Nah02] Nahmsuk Oh, Subahshish Mitra, Edward J. McCluskey: Technical Report: ED4I: Error Detection by Diverse Data and Duplicated Instructions. In: *Center for Reliable Computing* (2002.)

[Oze92] Ozello, Patrick: The Coded Microprocessor Certification. In: *safety of Computer Control Systems (Safecomp'92)*, 1992. – ISBN 0–08–041893–7.

[PF82] Patel, J.H. ; Fung, L.Y.: Concurrent Error Detection in ALU's by Recomputing with Shifted Operands. In: *Computers, IEEE Transactions on* C-31 (1982), Nr. 7, S. 589–595. – ISSN 0018–9340.

[Rao72] Rao Thammavarapu R.N., Monteiro P.: Multiresidue codes for double error correction. (1972), May, S. 1–13.

[Rei12] Reif, Konrad: *Automobilelektronik*. 4. Vieweg+Teubner Verlag, 2012.

[Ric02] Richard Krüger, Andreas Schenk, Frank Schiller: DE10219501B4 System und Verfahren zur Verbesserung von Fehlerbeherrschungsmassnahmen insbesondere in Automatisierungssystemen. (2002.)

[Rit90] Ritchie Dennis W., Kernighan Brian W.: *Programmieren in C*. 2nd. Carl Hander Verlag München Wien, 1990. – ISBN 3–446–15497–3.

[RV07] Raoul Velazco, Fabien F.: Error Rate Prediction of Digital Architectures: Test Methodology and Tools. In: *Radiation Effects on Embedded Systems*. Springer Netherlands, 2007. – ISBN 978–1–4020–5645–1.

[Sch07] Schrimpf, R. D.: Radiation Effects in Microelectronics. In: Valazco Raoul, Fouillat Pascal, Reis Ricardo (Hrsg.): *Radiation Effects on Embedded Systems*. P.O. Box 17, 3300 AA Dordrecht, Nederlands : Springer, 2007, S. 233–258.

[Sch11] Schiffel, Ute: *Hardware Error Detection Using AN-Codes*, Technische Universität Dresden, Diss., Juni 2011.

[Sem05] Semiconductor, Tezzaron: Soft Errors in Electronic Memory - A White Paper. (2005.)

[Seu95] Seungjae Han, Harold A. Rosenberg, Kang G. Shin. *DOCTOR: An IntegrateD SOftware Fault InjeCTiOn EnviRonment*. 1995.

[Wör05] Wörn, Heinz: Echtzeitbetriebssysteme. In: *Echtzeitsysteme*. Springer Berlin Heidelberg, 2005 (eXamen.press). – ISBN 978–3–540–20588–3, S. 343–442.

[Wra10] Wratil Peter, Michael Kieviet: *Sicherheitstechnik für Komponenten und Systeme*. 2. VDE Verlag GMBH, 2010. – ISBN 978–3–8007–3276–0.

Internet Referenzen

[1] Standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[2] March 2014. `https://www.debian.org/`, Aufgerufen am 08.03.2014.

[3] March 2014. `http://gcc.gnu.org/`, Aufgerufen am 29.03.2014.

[4] AMD. March 2014. `http://www.amd.com/us/solutions/servers/virtualization/Pages/virtualization.aspx`, Aufgerufen am 15.03.2014.

[5] ARM. Dezember 2013. `http://www.arm.com/products/processors/index.php`, Aufgerufen am 14.12.2013.

[6] die.net. March 2014. `http://linux.die.net/man/1/time`, Aufgerufen am 29.03.2014.

[7] Ethernet POWERLINK Standardization Group. Dezember 2013. `http://www.open-safety.org/`, Aufgerufen am 15.12.2013.

[8] Intel. March 2014. `http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/vt-directed-io-spec.html`, Aufgerufen am 15.03.2014.

[9] Intel. March 2014. `http://ark.intel.com/de/products/40815/Intel-Core2-Quad-Processor-Q9550S-12M-Cache-2_83-GHz-1333-MHz-FSB`, Aufgerufen am 08.03.2014.

[10] LLVM Team. March 2014. `http://llvm.org/`, Aufgerufen am 29.03.2014.

[11] LynuxWorks. Januar 2014. `http://www.lynuxworks.com/partners/show_product.php?ID=15`, Aufgerufen am 30.01.2014.

[12] Open Source Initiative. March 2014. `http://opensource.org/licenses/bsd-license.php`, Aufgerufen am 15.03.2014.

[13] OSADL - Open Source Automation Development Lab. Dezember 2013. `https://www.osadl.org/`, Aufgerufen am 15.12.2013.

[14] Peripheral Component Interconnect Special Interest Group. Dezember 2013. `http://www.pcisig.com/specifications/pciexpress/`, Aufgerufen am 18.12.2013.

[15] Python Software Foundation. March 2014. `https://www.python.org/`, Aufgerufen am 23.03.2014.

[16] Red Hat, Inc. November 2007. `http://people.freebsd.org/~lstewart/articles/cpumemory.pdf`, Aufgerufen am 09.04.2014.

[17] Siemens AG. Januar 2014. `http://www.freepatentsonline.com/DE10219501B4.html`, Aufgerufen am 10.01.2014.

[18] Sysgo AG. March 2014. `http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/embedded-virtualization/`, Aufgerufen am 19.03.2014.

[19] SYSGO AG. March 2014. `http://www.sysgo.com/news-events/press/press/details/article/sysgos-pikeos-receives-official-tuev-certificate/`, Aufgerufen am 08.03.2014.

[20] Wind River. March 2014. `http://windriver.com/solutions/virtualization/`, Aufgerufen am 19.03.2014.

Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Wien, am 12.05.2014 _____

Bernd Thiemann