technische universität
dortmund

Masterarbeit

Arithmetische Fehlercodierung in der Praxis: Korrekte Anwendung und neue Ansätze

Gerit Maldaner
August 2023

Gutachter:
Prof. Dr. Peter Ulbrich
Dr. Alexander Lochmann

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl 12
Arbeitsgruppe Systemsoftware
https://sys.cs.tu-dortmund.de/de/

Abstract

Transiente Hardwarefehler stellen eine zunehmende Herausforderung in modernen elektronischen Systemen dar, da sie unvorhersehbar auftreten und potenziell schwerwiegende Auswirkungen auf die Zuverlässigkeit und Funktionalität dieser Systeme haben können. Dies ist unter anderem darauf zurückzuführen, dass die Zuverlässigkeit der Hardware aufgrund ihrer steigenden Integrationsdichte immer weiter abnimmt.

Der CoRed-Ansatz ist ein softwarebasierter Ansatz zur Implementierung von Fehlertoleranz in sicherheitskritischen Systemen. Er kombiniert die Technik der Dreifachredundanz auf Software-Ebene mit der arithmetischen Fehlercodierung. Je nach Parametrisierung des CoRed-Ansatzes verbleiben jedoch vereinzelt unentdeckte Fehler, die mithilfe einer Fehlersimulation aufgedeckt werden konnten.

Für eine korrekte Funktionalität von CoRed müssen daher die Parameter der arithmetischen Codierung geeignet gewählt werden, da diese die Fehlererkennungsleistung des Ansatzes maßgeblich beeinflussen. Für die Auswahl der Schlüssel der arithmetischen Codierung existiert bereits ein etabliertes Verfahren [55, 47].

Auch für die Auswahl der drei Signaturen gibt es einige Ansätze. Ungeklärt ist jedoch beispielsweise, ob die Auswahl der Signaturen in Abhängigkeit von der minimalen Hamming-Distanz zwischen den Signaturen einen Einfluss auf die Fehlererkennungsleistung des Codes hat. In dieser Arbeit soll die Parametrisierung der arithmetischen Codierung am Beispiel des CoRed-Ansatzes untersucht werden. Der Fokus liegt dabei auf der Wahl der Signaturen.

Dazu wird zunächst die Verteilung der minimalen Hamming-Distanzen zwischen zwei bzw. drei Signaturen untersucht. Darauf folgend wird die Wirksamkeit der minimalen Hamming-Distanzen zwischen den Signaturen bewertet. Darüber hinaus soll die Arbeit von Hoffmann u. a. [24] näher betrachtet werden. Insbesondere soll untersucht werden, ob bestehende Abfragen in der CoRed-Implementierung modifiziert werden können, um die Fehlertoleranz weiter zu erhöhen. Zur Auswertung der Ergebnisse kommt das Werkzeug Fail* (Fault Injection Leveraged) zum Einsatz.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Auswirkungen und Ursachen für transiente Hardwarefehler sind seit den siebziger Jahren bekannt. Da ihr Auftreten in Standard-Hardware selbst schwer vollständig zu verhindern ist, wird seitdem an Lösungen für die Folgen transienter Hardwarefehler geforscht [60].

Eine der ersten Beobachtungen transienter Hardwarefehler machten Forschende im Jahr 1975 im Zusammenhang mit dem Betrieb von Kommunikationssatelliten. Sie stellten fest, dass kosmische Strahlung dafür verantwortlich war, dass die Einschaltspannung (engl. *turn-on voltage*) der verbauten Transistoren überschritten wurde. Dadurch kam es zu einer Ladungsänderung im Kondensator und in der Folge zu einer fehlerhaften Reaktion der Schaltkreise. [8]

Eine ähnliche Beobachtung machte Intel zu ungefähr derselben Zeit auf der Erde. Intel kämpfte Ende der siebziger Jahre mit unzuverlässigem Speicher in seinen neu entwickelten Chips, die zu Lieferschwierigkeiten der Mikroprozessoren an seiner Kunden führten. [36] Bereits zu dieser Zeit stieg die Integrationsdichte in RAM (Random Access Memory) [35]. Den geringeren Platzbedarf erreichte man damals wie heute durch kleinere Speicherzellen und gesteigerte Sensitivität des Leseverstärkers. Intel musste allerdings feststellen, dass sein Speicher mit steigender Integrationsdichte unzuverlässiger wurde. [36]

Ende der siebziger Jahre konnten die Intel-Mitarbeiter May und Woods eine Begründung für die Unzuverlässigkeit finden und veröffentlichen [35]. Die Anfälligkeit der Chips gegenüber transienten Fehlern ergab sich entsprechend ihrer Untersuchung aus einer Verkettung zweier Probleme: Zunächst machten sie die Bauweise des Speichers als Grund für den Fehler aus. Die Verkleinerung der Speicherzellen und die damit einhergehende gesteigerte Sensitivität des Leseverstärkers machte den Speicher erst richtig anfällig für das eigentliche Problem - eine radioaktive Kontamination mit Alpha-Partikeln der Gehäuse der Mikroprozessoren [35]. Intels Produktionsstätte für die Gehäuse befand sich an einem Fluss, an welchem ein kurzes Stück weiter flussaufwärts ein altes Uranbergwerk stand [36]. Die Kontamination der Gehäuse hatte, wie sich herausstellte, Auswirkungen auf die Zuverlässigkeit des Speichers.

Wenig später wurde festgestellt, dass Fehler aufgrund von kosmischer Strahlung nicht ausschließlich ein Problem des Weltalls sind. T. J. O'Gorman [40] führte Experimente in verschiedenen Höhenlagen durch und konnte dabei zeigen, dass auch auf Höhe des Meeresspiegels ein signifikanter Anteil transienter Fehler auf kosmische Strahlung zurückzuführen war. Er stellte auch fest, dass die kosmische Strahlung, die auf den untersuchten DRAM (Dynamic Random Access Memory) wirkte, mit steigender Höhe überproportional zunahm. [40]

Verstärkt wird die steigende Fehlerquote in Hardware durch immer höhere Integrationsdichten, die immer geringere Einschaltspannungen der Transistoren mit sich bringen [8]. Ein entgegenwirkender Effekt in diesem Szenario ist die reduzierte Größe der Speicherzellen, da mit einer Verkleinerung der Strukturbreite auch die Trefferwahrscheinlichkeit der Strahlungsteilchen sinkt. Die kritische Ladungsgrenze, die für einen Ladungswechsel sorgt, sinkt allerdings schneller als sich die Größe der Transistoren reduziert. Daher treten transiente Fehler durch die steigende Integrationsdichte häufiger auf als noch vor ein paar Jahren. [14] So kann beispielsweise auch in großen Serverfarmen ein nicht unerheblicher Anteil transienter Fehler beobachtet werden [51].

Somit sind transiente Hardwarefehler nicht nur ein Phänomen des Weltalls, sondern müssen auch verstärkt in Systemen berücksichtigt werden, die auf Höhe des Meeresspiegels eingesetzt werden. Um der erhöhten Fehleranfälligkeit aufgrund von vermehrten Strahlungseffekten und höheren Integrationsdichten entgegenzuwirken, gibt es zwei Kategorien von Ansätzen, die nicht-funktionalen Eigenschaften Sicherheit und Zuverlässigkeit in ein System einzubringen. Dabei handelt es sich um hardwarebasierte und softwarebasierte Ansätze [15]. Da hardwarebasierte Ansätze sehr teuer in der Anschaffung sowie im Betrieb sind, wird zur Kostensenkung häufig auf softwarebasierte Ansätze zurückgegriffen. Mitunter lassen sich außerdem hybride Lösungen beobachten. Ein bekanntes Beispiel hierfür ist die Flugsteuerung der Boeing 777 [61].

Sowohl für hardwarebasierte als auch für softwarebasierte Fehlertoleranz sind beispielsweise räumliche Redundanzmaßnahmen durch Replikation sowie Informationsredundanz z. B. durch Codierung denkbar. Neben der räumlichen Redundanz und der Informationsredundanz existieren weitere Redundanzmaßnahmen, die in dieser Arbeit nicht tiefer betrachtet werden sollen. Bei räumlichen Redundanzmaßnahmen werden Komponenten, wie beispielsweise Prozessoren, Speichereinheiten auf Hardware-Ebene oder auch Prozesse auf Software-Ebene, repliziert. Die Replikation ermöglicht dabei eine Fehlererkennung im Redundanzbereich, indem die Ausgaben der Replikate miteinander verglichen werden. Insbesondere an den Grenzen des Redundanzbereichs kann es zu kritischen Bruchstellen (engl. *SPoF (Single Point of Failure)*) kommen. Im Allgemeinen muss gewissenhaft abgewogen werden, ob eine erhöhte Komplexität den Erfolg von Redundanzmaßnahmen aufgrund einer erhöhten Angriffsfläche nicht gleich wieder zunichtemacht. [15]

Mit einer Codierung von Daten oder Kontrollflüssen auf Hardware- oder Software-Ebene kann das Auftreten von Fehlern unter eine gewisse Restfehlerwahrscheinlichkeit (vgl. Abschnitt 2.3.2) gedrückt werden. Dies ist möglich, da zusätzliche Informationen zu den Nutzdaten gehalten werden, die bei einem Datenabgleich das Erkennen und teilweise sogar das Korrigieren von Fehlern ermöglichen. Wird die Codierung mithilfe spezieller Hardware umgesetzt, erzeugt dies jedoch zusätzliche Kosten. Bei einer Codierung von Daten oder Kontrollflüssen auf Software-Ebene sorgt Codierung für einen großen Overhead, da der Aufwand für die zusätzliche Codierung der Daten berücksichtigt werden muss. Daher wird sie häufig als zu teuer angesehen, als dass sie zur Fehlererkennung oder -korrektur in großen Programmen eingesetzt werden könnte.

Ein softwarebasierter Ansatz, um die Schwachstellen beider Ansätze - Codierung und Redundanz - auszugleichen, ist *CoRed (Combined Redundancy)* (vgl. Abschnitt 2.5).

1.1 Der CoRed-Ansatz

Der CoRed-Ansatz löst die konzeptbedingten Probleme der Codierung und der Replikation, indem er die beiden Ansätze miteinander kombiniert. Dazu werden die sicherheitskritischen Software-Prozesse dreifach redundant ausgeführt. Lücken zwischen den redundanten Ausführungen werden durch arithmetische Codierung (genauer AN-Codes) gehärtet. Dabei wird die Codierung lediglich zur Absicherung der nicht replizierbaren Komponenten, genauer der Eingabereplikation und des Mehrheitsentscheiders, eingesetzt. Damit werden auch bisherige Lücken in der Redundanz auf Betriebssystem-Ebene abgesichert und eine vollständige Fehlererkennung im Fehlermodell ermöglicht. [57, 55]

Wörter werden AN-codiert, indem sie mit einem Schlüssel A multipliziert werden. Es existieren Erweiterungen der AN-Codes zu ANB- oder ANBD-Codes, bei welchen zusätzlich eine Signatur B und ggf. ein Zeitstempel D auf das Produkt addiert werden. Arithmetische Codierung bietet den Vorteil, dass sie gegenüber arithmetischen Operationen abgeschlossen ist. Eine sinnvolle Wahl der Parameter ist entscheidend für eine gute Fehlererkennungsleistung des Codes. Daher diskutiert P. Ulbrich in seiner Arbeit eine sinnvolle Wahl der Parameter. Dabei konzentriert er sich auf die geeignete Wahl der Schlüssel. [55]

In der Literatur wird für die Wahl der Codierungsschlüssel häufig eine Empfehlung für große Primzahlen gegeben [19, 44]. Diese Aussage kann P. Ulbrich in seiner Arbeit jedoch nicht bestätigen. Stattdessen schlägt er vor, die Codierungsschlüssel primär nach der Fehlererkennungsleistung des von ihnen erzeugten Codes auszuwählen. Sekundär sollen die Schlüssel nach der minimalen Hamming-Distanz der Codewörter ausgesucht werden. [55] Mithilfe von Experimenten berechnet und benennt er günstige Schlüssel zur Parametrisierung der Codierung. Dabei identifiziert er insbesondere fünf besonders geeignete 32-Bit-Schlüssel.

P. Ulbrich [55] wählt die Signaturen in seinen Arbeiten, sodass sie kleiner als der Schlüssel sind. Darüber hinaus beachtet er, dass die Distanzen zwischen allen ausgewählten Signaturen unterschiedlich und die Signaturen aufsteigend sortiert sind. Eine Empfehlung für bestimmte Werte, wie sie für die Schlüssel existiert, oder genauere Auswahlverfahren gibt er allerdings nicht. Darüber hinaus ist z.B. nicht bekannt, ob eine Berücksichtigung der minimalen Hamming-Distanzen zwischen den ausgewählten Signaturen einen Einfluss auf die Fehlererkennungsleistung des Codes hat.

Trotz der theoretisch vollständigen Fehlererkennung des CoRed-Ansatzes können in Fehlersimulationen abhängig von der Parametrisierung verbleibende unentdeckte Datenfehler (engl. *SDC (Silent Data Corruption)*) beobachtet werden. Die Gründe können vielfältig sein. So könnten diese Fehler z.B. auf ggf. verbleibende Lücken im Ansatz oder der Implementierung, spezifisches Verhalten von Compiler und Betriebssystem, eine nicht hinreichende Erforschung der Parameterwahl oder nicht ausreichend strenge Abfragen zur Fehlererkennung im Programmcode zurückgeführt werden.

Im Folgenden soll der Fokus insbesondere auf der Auswahl der Signaturen bei der Parametrisierung liegen. Hier existiert bisher die Annahme, dass diese eine eher untergeordnete Rolle bei der Parametrisierung spielen [55]. Aus diesem Grund werden Signaturen insbesondere in der Industrie häufig zufällig gewählt. Es ist allerdings unklar, ob diese Vermutung stimmt.

1.2 Ziele

Im Abschnitt 1.1 wurden bereits unerforschte Aspekte hinsichtlich der Parametrisierung der AN-Codes bzw. der verwandten ANB- und ANBD-Codes herausgestellt. Hierzu gehört insbesondere, dass es zwar bekannte Regeln für die Auswahl der Schlüssel gibt, die Signaturen meist allerdings eher zufällig gewählt werden. Dies gilt insbesondere im industriellen Kontext.

Schlüssel werden häufig, wie in 1.1 beschrieben, auf Basis ihrer Restfehlerwahrscheinlichkeit und der minimalen Hamming-Distanz zwischen den Codewörtern, die sie erzeugen, gewählt [55]. Anwendungsszenarien der arithmetischen Codierung, wie z.B. der CoRed-Ansatz, zeigen allerdings, dass nach wie vor Probleme bei der arithmetischen Codierung in Form von unentdeckten Datenfehlern existieren. Diese können potenziell mithilfe einer geschickten Auswahl von Signaturen reduziert bzw. gelöst werden. Das Ziel dieser Arbeit ist daher die Untersuchung der Signatur-Auswahl am Beispiel der CoRed-Implementierung. Die Signaturen und ihr Einfluss auf die erfolgreiche Fehlererkennung bei der ANB-Codierung sollen dabei hinsichtlich mehrerer Aspekte untersucht werden.

Es ist zu beobachten, dass ein Schlüssel, der eine größere minimale Hamming-Distanz der Codewörter erzeugt, eine bessere Fehlererkennungsleistung aufweist [55]. In dieser Arbeit soll daher unter anderem untersucht werden, ob eine ähnliche Beobachtung auch für die

Signaturen gemacht werden kann. Dabei soll erforscht werden, welchen Einfluss Signaturen, die zu einer besonders großen minimalen Hamming-Distanz führen, auf die Erkennungsleistung des Codes haben.

Darüber hinaus soll die Auswahl der Signaturen mithilfe von Regeln weiter eingeschränkt werden, mit dem Ziel, unentdeckte Datenfehler zu reduzieren. Denn es ist z.B. möglich, dass durch einen ungünstigen Fehler ein gültiges Codewort fälschlicherweise in ein anderes gültiges Codewort übergeht. Dabei kann es passieren, dass es sich um ein rechnerisch gültiges Codewort handelt, welches aus semantischer Sicht so gar nicht möglich ist. Mithilfe von geschickten Einschränkungen können solche Fehler in der Implementierung entdeckt und abgefangen werden [24]. In dieser Arbeit soll geprüft, ob es möglich ist, bestehende Abfragen im Programmcode strenger zu formulieren und dadurch eine bessere Fehlererkennungsleistung zu erreichen.

Sollten die genannten Untersuchungen zeigen, dass eine große minimale Hamming-Distanz der Codewörter zu einer verbesserten Erkennungsleistung des Codes führt, so wäre es günstig, die minimalen Hamming-Distanzen nicht errechnen zu müssen, sondern die Signaturen anhand von Regeln auswählen zu können. Daher wird im Rahmen dieser Arbeit nach Mustern in den minimalen Hamming-Distanzen für verschiedene Signaturen gesucht.

Um die Ergebnisse hinsichtlich der Signatur-Auswahl zu überprüfen, sollen Fehlersimulationen durchgeführt werden. Dazu soll die CoRed-Implementierung unterschiedlich parametrisiert und Fehlerinjektionsexperimenten unterzogen werden. Die Ergebnisse dieser Experimente werden miteinander verglichen, um (statistische) Aussagen über die Wirksamkeit der unterschiedlichen Parametrisierungen treffen zu können.

Damit ergeben sich die folgenden Ziele:

- Untersuchung der Signatur-Auswahl auf Basis der minimalen Hamming-Distanz der Codewörter.
- Untersuchung bestehender Abfragen im Programmcode auf Möglichkeiten strengerer Einschränkungen.
- Suchen von Mustern in den minimalen Hamming-Distanzen der Codewörter für unterschiedliche Signaturen.
- Überprüfung der Ergebnisse mithilfe von Fehlersimulationen.

1.3 Aufbau der Arbeit

Das Kapitel 2 befasst sich mit den theoretischen Grundlagen dieser Arbeit. Insbesondere wird auf die Grundlagen und Begriffe im Bereich der Fehlertoleranz eingegangen, bevor die arithmetische Codierung sowie die Dreifachredundanz als Bestandteile von CoRed

thematisiert werden. Das Kapitel schließt mit einem Überblick über die Technik der Fehlerinjektion und das Werkzeug FAIL*.

Es folgt das Kapitel 3 mit der Problembeschreibung. Diese wird eingeleitet durch die Beschreibung des Fehlermodells, des Systemmodells und des Anwendungsmodells dieser Arbeit. Anschließend wird der Stand der Forschung auf dem Gebiet der Parametrisierung der arithmetischen Codierung zusammengefasst. Schließlich folgt die eigentliche Problemanalyse. Diese ist angelehnt an Hoffmann u. a. [24] in drei Fallstricke unterteilt. Anhand dieser Fallstricke werden die offenen Punkte erläutert, die in dieser Arbeit untersucht werden sollen.

Für die im Kapitel 3 identifizierten offenen Punkte werden dann im Kapitel 4 Lösungsansätze erarbeitet. Die Lösungsansätze sind ebenfalls in die Fallstricke aus Hoffmann u. a. [24] untergliedert. Sie werden zudem um eine weitere Lösungsidee - die Beachtung der Code-Distanz zwischen den Signaturen der arithmetischen Codierung - erweitert.

Das Kapitel 5 beschäftigt sich mit der Umsetzung sowie dem Versuchsaufbau. Dieses Kapitel ist zweigeteilt. In den Lösungsansätzen wurde zuvor motiviert, dass die Parametrisierung der arithmetischen Codierung anhand der Code-Distanzen zwischen den Signaturen ein interessanter Lösungsansatz für das Problem der verbleibenden unentdeckten Datenfehler sein könnte. Aus diesem Grund wird zunächst beschrieben, wie diese Code-Distanzen berechnet werden. Im zweiten Teil des Kapitels wird dann auf die Vorgehensweise bei der Untersuchung der weiteren Lösungsansätze eingegangen.

Im Kapitel 6 erfolgt die Auswertung der Versuche. Dabei wird auf die gesteckten Ziele Bezug genommen und bewertet, inwieweit diese mit den erarbeiteten Lösungsansätzen erreicht wurden. Des Weiteren wird auf Einschränkungen im Versuchsaufbau eingegangen.

Im letzten Kapitel wird zunächst ein Überblick über verwandte Arbeiten außerhalb der Parametrisierung der arithmetischen Codierung gegeben. In der anschließenden Diskussion wird ein Randfall des CoRed-Ansatzes diskutiert, der im Zuge der Überprüfung dieser Arbeit aufgefallen ist. Bei diesem ist eine Fehlererkennung nicht möglich. Auf die Beschreibung des Randfalls folgt die Zusammenfassung dieser Arbeit. Abschließend wird ein Überblick über offene Punkte und ein Ausblick auf mögliche weitere Forschungsarbeiten gegeben.

Kapitel 2

Grundlagen

Um die Ergebnisse dieser Arbeit detailliert beschreiben zu können, müssen zunächst einige Grundlagen definiert und erläutert werden. Hierzu gehört insbesondere die Taxonomie von Avižienis u. a. [5], welche Fehler detaillierter differenzieren sowie kategorisieren.

Weiterhin soll eine kurze Einführung in das Konzept der Fehlertoleranzmaßnahmen gegeben werden. Diese werden in hardwarebasierte und softwarebasierte Maßnahmen differenziert, näher beschrieben und anhand von Beispielen eingeordnet. Dabei werden zwei softwarebasierte Fehlertoleranzmaßnahmen tiefer erläutert. Dabei handelt es sich um die Fehlercodierung - genauer die arithmetische Fehlercodierung - sowie die Replikation. In einem nächsten Schritt wird der CoRed-Ansatz beschrieben. Dieser nutzt sowohl arithmetische Codierung als auch Replikation.

Zuletzt wird auf die Fehlerinjektion als ein Mittel zur Überprüfung der Zuverlässigkeit von Redundanzmaßnahmen eingegangen. Hier soll insbesondere das Werkzeug *FAIL** thematisiert werden.

2.1 Begriffsdefinitionen

Um einen tieferen Einblick in Fehlertoleranzmaßnahmen geben zu können, müssen einige Begriffe und Konzepte definiert werden. Dabei handelt es sich insbesondere um den Begriff der Fehlerkette sowie die Differenzierung von Defekt, internem Fehler und Fehlverhalten (engl. *fault, error* und *failure*). Darüber hinaus sollen diese Begriffe weiter klassifiziert und anhand ihrer Auswirkungen auf ein System weiter eingeordnet werden.

2.1.1 Fehlerarten

Wie viele Arbeiten im Bereich der Fehlertoleranz nutzt auch diese Arbeit die Taxonomie von Avižienis u. a., die im Jahr 2004 veröffentlicht wurde. Da die von Avižienis u. a. eingeführten Definitionen und Begrifflichkeiten in englischer Sprache sind, werden in dieser Arbeit passende deutsche Übersetzungen definiert und auf die entsprechenden englischen

Begriffe von Avižienis u. a. verwiesen. Betroffen sind hier insbesondere die Definitionen von und Differenzierung zwischen *fault*, *error* und *failure*. Im Folgenden werden diese Begriffe als Defekt (engl. *fault*), interner Fehler (engl. *error*) und Fehlverhalten (engl. *failure*) bezeichnet. Sie differenzieren das, was im Deutschen allgemein als Fehler bezeichnet wird, anhand des Fortschritts, den der Fehler im System auf seinem Weg vom Fehlerursprung an die Systemoberfläche gemacht hat.

Ein **Defekt**, z. B. in Form eines Bitkippers, kann einen internen Fehler bewirken. Wird dieser Defekt aktiviert, so kann daraus ein interner Fehler entstehen (vgl. Abbildung 2.1b). Es kann allerdings auch passieren, dass es nie zu einer Aktivierung des Defekts kommt und er so von der nächst höheren Ebene unbemerkt bleibt (vgl. Abbildung 2.1a). Daher unterscheidet man zwischen gutartigen und bösartigen Defekten (engl. *benign* und *malicious* faults). **Gutartige Defekte** sind jene, die sich nicht weiter ausbreiten, wohingegen **bösartige Defekte** aktiviert werden und sich auf die nächst höhere Systemebene auswirken.

Unter einem **internen Fehler** versteht man den Teil eines Systemzustandes, der potenziell zu einem Fehlverhalten führen kann. Er entsteht aus einem bösartigen Defekt.

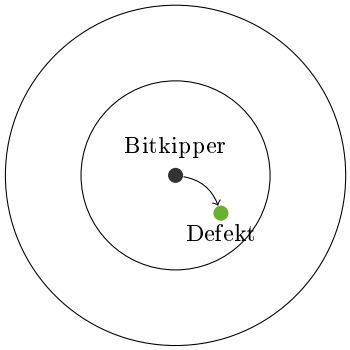
Als **Fehlverhalten** bezeichnet man sichtbare Fehler an der Oberfläche eines Systems. Diese können beispielsweise von Nutzer:innen festgestellt werden, weil sich ein Dienst nicht wie erwartet verhält (vgl. Abbildung 2.1b). Interne Fehler bedingen Fehlverhalten in der nächst höheren Ebene, falls sie aktiviert werden. [5]

Bösartige Defekte führen zu internen Fehlern, die Fehlverhalten produzieren können. Allerdings verursacht nicht jeder Defekt einen internen Fehler und auch nicht jeder interne Fehler führt zu Fehlverhalten des Systems. [15] Ein Fehlverhalten tritt insbesondere nur auf, wenn der fehlerhafte Systemzustand Teil des externen Systemzustands wird [5]. Die Unterschiede und Zusammenhänge werden in Abbildung 2.1 verdeutlicht.

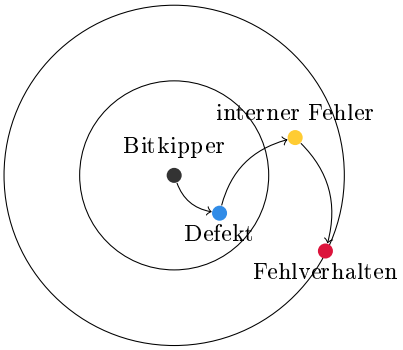
Das Propagieren des Fehlers über die verschiedenen Systemebenen bis potenziell an die Oberfläche wird als **Fehlerkette** bezeichnet. In Abbildung 2.1b ist diese Ausbreitung eines Fehlers bis hin zur Systemoberfläche abgebildet. Ob ein Defekt der einen Ebene in der Ebene darüber sichtbar wird, hängt zum einen davon ab, ob er aktiviert wird. Zum anderen kann er durch fehlerkorrigierende Maßnahmen maskiert werden. [36] In Abbildung 2.1a ist beispielsweise ein Defekt dargestellt, welcher noch nicht aktiviert wurde und sich daher auch nicht weiter ausbreiten kann.

Ist ein Fehlverhalten an der Oberfläche aufgetreten, so ist es möglich, dass dieses Fehlverhalten zu einem Defekt in einem anderen System führt, wenn dieses weitere System die fehlerhaften Ausgaben des ersten Systems entgegennimmt. So können sich Fehler auch über Systeme hinweg ausbreiten (vgl. Abbildung 2.1c). [5]

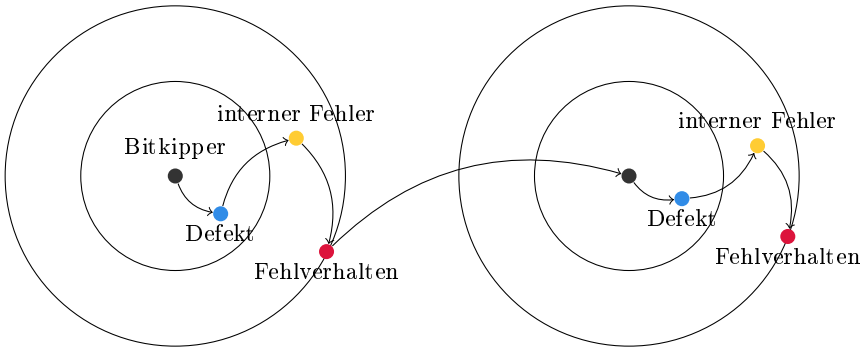
Zu dieser Begriffsdefinition lassen sich viele Beispiele finden: In einem ersten Beispiel lässt sich ein Softwarefehler betrachten und der Status des Softwarefehlers in die drei Fehlerklassen einordnen. Der Defekt ist in diesem Fall der in die Software eingebaute Fehler. Die Konsequenz, die der Fehler in der Software hat, lässt sich dann als interner Fehler



(a) Ein ruhender, noch nicht aktivierter Defekt.



(b) Ein Defekt breitet sich zum Fehlverhalten aus.



(c) Ein Defekt breitet sich zum Fehlverhalten aus. Das Fehlverhalten führt zu einem Defekt in einem weiteren System.

Abbildung 2.1: Fehlerausbreitung in den Systemebenen. Die Kreise symbolisieren die verschiedenen Systemebenen, über die sich ein Fehler ausbreiten kann. Abbildung 2.1a zeigt einen Defekt aufgrund eines Bitkippers im Systeminneren, welcher noch nicht aktiviert wurde und sich daher nicht auf andere Systemebenen ausgebreitet hat. Solche Defekte werden als ruhende Defekte bezeichnet. Abbildung 2.1b zeigt einen Defekt, der sich über höhere Systemebenen erst als interner Fehler und dann als Fehlverhalten bis an die Oberfläche fortgepflanzt hat. Deswegen wird der vorliegende Defekt auch als bösartig bezeichnet. Abbildung 2.1c zeigt ein mögliches Szenario der Fehlerausbreitung über Systemgrenzen hinweg. Das Fehlverhalten des linken Systems verursacht einen Defekt in einem weiteren System. Dort kann sich der Defekt ebenfalls weiter fortpflanzen, falls er aktiviert wird. Pflanzt sich der Fehler bis an die Systemoberfläche fort, kann sich auch dieses resultierende Fehlverhalten auf ein weiteres System auswirken. So kann sich ein Bitkipper ggf. sogar auf eine gesamte Systemlandschaft ausbreiten. (Abbildung nach Darstellungen aus Arbeiten von S. Mukherjee [36] und P. Ulbrich [55])

einordnen. Wird das betroffene Stück Quellcode durchlaufen, wird der interne Fehler aktiviert. Die Folge können beispielsweise Datenfehler sein. Wenn die durch den internen Fehler hervorgerufenen fehlerhaften Daten zum Versagen eines Dienstes führen, tritt ein Fehlverhalten auf. [23]

Das zweite Beispiel geht zurück auf die Motivation in Kapitel 1. Trifft ionisierende Strahlung ein Speichermedium, kann von einem Defekt gesprochen werden. Wenn diese Strahlung auf den Transistor wirkt und den Ladungszustands einer Speicherzelle ändert, ändert sich der Speicherinhalt (vgl. Abschnitt 2.1.2). Es liegt ein interner Fehler vor. Wird auf diesen Speicherinhalt lesend zugegriffen, wird der interne Fehler aktiviert. Wenn der Fehler im Speicherinhalt durch den Lesezugriff Einfluss auf den Systemzustand hat, kann es zu Fehlverhalten eines Dienstes kommen. [23]

Die Unterbrechung der Fehlerkette und damit der Fehlerausbreitung geschieht durch Fehlertoleranzmaßnahmen. Unter **Fehlertoleranz** versteht man das Verhindern des Auftretens von Fehlverhalten, falls ein Defekt bereits eingetreten ist, sodass dieser sich nicht an der Oberfläche durch Ausfall eines Dienstes oder durch Fehlverhalten bemerkbar machen kann. Neben Fehlertoleranzmaßnahmen gibt es noch weitere Herangehensweisen, um für Zuverlässigkeit und Sicherheit in einem sicherheitskritischen System zu sorgen. Dazu gehören die Fehlerprävention, die Fehlerbeseitigung sowie die Fehlervorhersage. [5]

2.1.2 Fehlerklassifizierung

Die Fehlerarten können hinsichtlich ihrer Eigenschaften differenziert werden. Hier soll zunächst eine Unterscheidung in ihrer Reproduzierbarkeit beziehungsweise anhand der Dauer ihres Auftretens erfolgen. **Permanente Fehler** (engl. *permanent fault, error* oder *failure*) sind Fehler, die zuverlässig und systematisch aktiviert werden können. [5] Beispiele für einen permanenten Fehler sind Herstellungsfehler oder abgebrochene Pins in einem Mikrocontroller.

Sporadische Fehler (engl. *intermittent fault, error* oder *failure*) sind solche, die nur hin und wieder und unter bestimmten Umständen reproduzierbar sind. Ein bekanntes Beispiel für einen sporadischen Fehler ist ein Wackelkontakt an einem Kabel oder ein teilweiser Kontakt aufgrund eines losen Pins an einem Mikrocontroller. Darüber hinaus gibt es transiente Fehler (engl. *transient fault, error* oder *failure*). [5]

Der Begriff **transienter Fehler** wurde zuerst von May und Woods im Jahr 1979 geprägt, wie in Kapitel 1 beschrieben. Sie definieren den Begriff als Ein-Bit-Fehler in Speichermedien, die einmalig und zufällig auftreten. Da solche Fehler weder aufgrund von Hardwarefehlern entstehen noch Schäden an der Hardware hinterlassen, bezeichnen die Autoren diese Fehler als „weich“ (engl. *soft*). Neben der Tatsache, dass aus transienten Fehlern keine Hardwareschäden entstehen, ist es sogar so, dass sie durch den nächsten Schreibzugriff im Speichermedium überschrieben werden. [35]

Transiente Fehler können anhand ihrer Schwere in drei Arten unterteilt werden: **SDC (Silent Data Corruption)**, **DUE (Detected Unrecoverable Error)** und **erkannte, behobene transiente Fehler**. Unter **SDCs** versteht man unerkannte Datenfehler. SDCs bleiben auch unentdeckt von potenziellen Fehlertoleranzmechanismen. Diese unerkannten Datenfehler sorgen häufig dafür, dass das System später Ergebnisse liefert, die nicht korrekt sind. [36]

DUEs sind Fehler, die zwar erkannt wurden, aber nicht behoben werden können. In den meisten Fällen werden DUEs als weniger schwerwiegend eingestuft. Das liegt daran, dass sie nicht unbemerkt zu weiteren korrumpierten Daten führen. [36]

Zuletzt gibt es eine dritte Art von transienten Fehlern. Dabei handelt es sich um Fehler, die korrigiert und an das System gemeldet wurden. Diese Fehler sind die am wenigsten folgenreiche Art, da sie keine negativen Konsequenzen für das System befürchten lassen und das System weiter arbeiten kann. [36]

2.2 Fehlertoleranzmaßnahmen

Fehlertoleranz ist das Verhindern einer Fehlerausbreitung im System durch Unterbrechen der Fehlerkette. Fehlertoleranzmaßnahmen greifen, nachdem ein Defekt bereits aufgetreten ist [5]. Das bedeutet, dass ein System einen Fehler mindestens erkennen und besser noch beheben können muss. Dies wird typischerweise durch Redundanz erreicht.

Redundanz liegt vor, wenn das System über mehr funktionale, technische Ressourcen verfügt, als für die eigentliche spezifizierte Funktionalität erforderlich wäre. Redundanz kann auf unterschiedliche Weise in ein System eingebracht werden. [16] Etablierte Formen sind die Informationsredundanz, die zeitliche, die funktionelle und die strukturelle Redundanz [16, 21].

Bei der **Informationsredundanz** werden neben den Nutzdaten zusätzliche Informationen mitgeführt [16]. Ein bekanntes Beispiel ist die ISBN.

Implementiert ein System **Zeitredundanz**, so wird dem ausführenden System zusätzliche Zeit für die Ausführung einer Funktion zur Verfügung gestellt. Wie diese Zeit genutzt wird, ist von System zu System unterschiedlich. [16] Zeitredundanz ermöglicht beispielsweise die Mehrfachausführung einer Funktion, um Fehler erkennen zu können [21]. Die zeitliche Redundanz definiert in erster Linie lediglich eine Höchstdauer für die Ausführung bis ein fehlerfreies Ergebnis bereitstehen soll [16].

Bei der **funktionellen Redundanz** implementiert ein System zusätzliche, im Normalbetrieb eigentlich nicht benötigte Funktionen, z. B. Testfunktionen. Diese Testfunktionen können dann beispielsweise die Ausgaben prüfen. [16]

Unter **struktureller Redundanz** versteht man die Erweiterung eines Systems um zusätzliche Komponenten. Diese Komponenten können gleichartig oder unterschiedlich sein und werden nicht für den spezifizierten Betrieb, sondern zur Fehlertoleranz benötigt. [16]

Im Verlauf dieses Kapitels wird jeweils ein Beispiel für die strukturelle Redundanz (Dreifachredundanz, engl. *TMR (Triple Modular Redundancy)*) und die Informationsredundanz (Fehlercodierung) gegeben. Für einen Überblick über zeitliche und funktionelle Redundanz werden die Arbeiten von Dubrova [15], Echtle [16] und Goloubeva u. a. [21] empfohlen.

Fehlertoleranzmaßnahmen in Form von Redundanz können grundsätzlich auf zwei Ebenen realisiert werden - auf Hardware-Ebene und auf Software-Ebene. Beide Arten verfolgen ähnliche Methoden. Lediglich die Implementierungsebene der Fehlertoleranz ist unterschiedlich. [21] Hardwarebasierte Fehlertoleranz setzt auf einer tieferen Ebene des Systems an, während softwarebasierte Fehlertoleranz auf einer höheren Systemebene angesetzt und feingranularer ist. Beide Ansätze bringen Vor- und Nachteile mit sich. Die folgenden Abschnitte sollen einen Überblick über die möglichen Implementierungsebenen und ihre Vor- und Nachteile geben.

2.2.1 Hardwarebasierte Fehlertoleranz

Ein etablierter Ansatz, die Fehlerkette zu durchbrechen und Fehlertoleranz in ein System einzubringen, sind hardwarebasierte Fehlertoleranzmaßnahmen. Dieser Ansatz ist aus Sicht der transienten Hardwarefehler leicht zu begründen, da hardwarebasierte Maßnahmen unmittelbar dort ansetzen, wo die Fehler auftreten.

Grundsätzlich unterscheidet man zwischen statischer und dynamischer Redundanz [16, 21, 15]. **Statische Redundanz** ist während des gesamten Systembetriebs aktiv und maskiert auftretende Fehler [16]. Sie wird insbesondere in sicherheitskritischen Bereichen eingesetzt, in denen hohe Verlässlichkeit sehr wichtig ist. Dazu gehören beispielsweise medizinische Systeme oder Systeme in Flugzeugen, wo Unterbrechungen oder fehlerhafte Berechnungen nicht akzeptiert werden können. [15]

Dynamische Redundanz hingegen wird erst bei Auftreten eines Fehlers aktiviert und kann diesen dann entsprechend der Redundanzmaßnahmen korrigieren [16]. Dementsprechend kann es hier zu kurzen Unterbrechungen kommen oder es können zwischenzeitlich Fehler vorliegen. Es wird lediglich gefordert, dass der Systemzustand in einer definierten Zeit wieder in einen Normalzustand zurückkehrt. Der Fokus liegt bei solchen Systemen auf hoher Verfügbarkeit.

Denkbar sind außerdem hybride Varianten abhängig vom Anwendungsfall. Dabei werden Fehler maskiert, um von vornherein fehlerhafte Ausgaben zu verhindern. Darüber hinaus werden dynamische Redundanzmaßnahmen eingesetzt, um das System wieder in einen fehlerfreien Zustand zu bringen, sollten dennoch Fehler aufgetreten sein. [15, 21]

Hardwarebasierte Replikation wird durch Mehrfachauslegung der betroffenen Hardware-Komponenten erreicht. Ein **Entscheider** (engl. *Voter*) entscheidet über die Ausgaben der Replikate und stellt eine Einigung (engl. *Consensus*) her.

Dabei gibt es verschiedene Arten von Entscheidern. Eine der am häufigsten verwendeten Varianten ist der Mehrheitsentscheider, der sich stets für den Wert entscheidet, den mehr als 50% der Replikate vorschlagen. Gibt es keinen solchen Wert, kann keine Einigung hergestellt werden. [15]

Da auf diese Art und Weise ganze Systeme repliziert werden, sind die Redundanzmaßnahmen sehr weitreichend. Um z. B. eine Dreifachredundanz aufbauen zu können, müssen alle redundant auszulegenden Hardware-Komponenten wie z. B. Prozessoren oder Speicher in mehrfacher Ausführung beschafft werden. Das geht primär mit höheren Kosten einher.

Die beschafften Komponenten benötigen sekundär außerdem mehr Platz und Energie und bringen zusätzliches Gewicht in das System ein im Vergleich zu einem nicht replizierten System. Hinzu kommen zusätzliche Zeitaufwände für die Entwicklung von Architekturmodellen für solche Systeme, die Herstellung selbst sowie das Testen. [15] Dabei haben die zusätzlichen Komponenten keinen weiteren Mehrwert im System außer der Fehlertoleranz. Ein möglicher Performance-Gewinn z. B. bleibt aus. Auf der anderen Seite kommt es durch die Fehlertoleranzmaßnahme auch nicht zu einem Performance-Verlust, da die Hardware-Komponenten ohnehin so bemessen wurden, dass sie für den spezifizierten Anwendungsfall passend sind. [55]

Hinzu kommt außerdem, dass die Selektivität der Replikation sehr gering ist [55]. Repliziert wird schlichtweg alles, was die replizierten Hardware-Komponenten verwendet. Man betrachte beispielsweise einen sicherheitskritischen Prozess, der eine relativ geringe Laufzeit ausmacht und weitere nicht sicherheitskritische Prozesse, die alle auf einem Prozessor ausgeführt werden. Aufgrund der Wichtigkeit des Prozesses ist es allerdings notwendig, dass dieser Prozess nicht fehlschlägt oder fehlerhafte Ergebnisse erzeugt. Daher wird der Prozessor repliziert. Durch den großen Redundanzbereich sind nun allerdings auch alle anderen nicht-sicherheitskritischen Prozesse durch diese Redundanzmaßnahmen geschützt. Das kann im weiteren Sinne erhöhte Betriebskosten zur Folge haben, da Prozesse redundant ausgeführt werden, für die dies nicht notwendig wäre.

Codierung wird auf Hardware- wie auch auf Software-Ebene durch das Einbringen zusätzlicher Information zu den Nutzdaten realisiert (z. B. Paritätsbit). Mithilfe dieser Informationsdaten wird eine Fehlererkennung oder Fehlerkorrektur ermöglicht. Ob eine Fehlerkorrektur oder bloß eine Fehlererkennung möglich ist, entscheidet dabei der verwendete Code. [44] Grundvoraussetzung für eine Fehlererkennung oder Fehlerkorrektur ist, dass nur ein gewisser Teil der Informationen verfälscht ist. Ist die Distanz zum ursprünglichen Codewort zu groß, ist eine Fehlerkorrektur nicht mehr möglich. [16]

In erster Linie werden bei der Codierung auf Hardware-Ebene lediglich die zu speichernden Daten über Speichermedien redundant abgesichert. Arbeitsspeicher mit fehlerkorrigierenden Eigenschaften wird beispielsweise ECC-RAM genannt (ECC). Eine Absicherung der Kontrollflüsse beispielsweise ist auf Hardware-Ebene nicht ohne Weiteres möglich. Hier kommen stattdessen häufig softwarebasierte Vorgehensweisen zum Einsatz, wie sie von

P. Forin [19] vorgestellt oder in einer Arbeit von Goloubeva u. a. [21] näher thematisiert wurden.

2.2.2 Softwarebasierte Fehlertoleranz

Softwarebasierte Fehlertoleranzmaßnahmen werden noch nicht so lange eingesetzt wie hardwarebasierte Fehlertoleranzmaßnahmen und sind noch nicht so weitreichend erforscht. Die Ansätze der softwarebasierten Fehlertoleranz sind jedoch vergleichbar zu hardwarebasierten Techniken. Dabei ist allerdings die Ebene, auf der die Redundanzmaßnahmen wirken, unterschiedlich. [15]

Soll softwarebasierte Redundanz eingesetzt werden, treten neue Herausforderungen auf. Denn einzelne Software-Komponenten sind in der Regel viel enger miteinander verknüpft und bauen viel stärker aufeinander auf, als dies bei Hardware-Komponenten der Fall wäre. Sie haben häufig viel mehr Schnittstellen und Abhängigkeiten untereinander als Hardware-Komponenten. Darüber hinaus treten existierende Fehler in Software meist von Beginn an auf und entstehen nicht erst durch altersbedingte Degeneration der Komponenten.

Die Frage nach der Korrektheit der Software-Module eröffnet einen ganz neuen und großen Bereich im Themengebiet des Testens, der Validierung und der Verifikation, insbesondere von Echtzeitsystemen. Hierauf kann an dieser Stelle allerdings nicht näher eingegangen werden, da es den Umfang der Arbeit überschreitet. W. Schütz [52] geht in seiner Arbeit auf Schwierigkeiten speziell im Hinblick auf das Testen von verteilten Echtzeitsystemen ein. Einen Einblick in formale Verifikation von Programmen geben Almeida u. a. [1]. Edwards u. a. [17] geben einen Überblick über den Entwurf eingebetteter Systeme. Dazu gehört unter anderem auch ein Überblick über die Verifikation und Validierung eingebetteter Systeme.

Zu der Problematik, wie und mit welchem Aufwand Fehler in Software gefunden oder ausgeschlossen werden können, kommt die Tatsache, dass sich Fehler in einem Modul aufgrund des hohen Integrationsgrades zwischen den Modulen häufig auch auf weitere Module auswirken.

Damit sind Software-Module insgesamt abhängig von der Umgebung, auf der sie ausgeführt werden und von den Eingabeparametern, die die Umgebung für sie erzeugt. Ein bekanntes Beispiel für Software, die auf einem alten System noch problemlos funktionierte, nach einer Portierung allerdings schwerwiegende Rechenfehler erzeugte, ist die Software der Ariane 4 bzw. Ariane 5. [15]

Darüber hinaus besteht häufig die Gefahr, dass in Software implementierte Fehlertoleranzmaßnahmen durch u. a. erhöhte Ausführungszeiten, erhöhten Speicherbedarf, erhöhte Rechenkapazitäten die Angriffsfläche für Fehler derart vergrößern, dass die Fehleranfälligkeit des Systems eher steigt als sinkt. [49].

Auch in der softwarebasierten Fehlertoleranz wird zwischen den zuvor beschriebenen Redundanzmaßnahmen Informationsredundanz, zeitliche, funktionelle und strukturelle Redundanz unterschieden. Darüber hinaus sind Codierung und Replikation (insb. Dreifachreplikation) auch bei der softwarebasierten Fehlertoleranz gängige Maßnahmen. [15]

Wie bereits in Abschnitt 2.2.1 erwähnt, gibt es in der softwarebasierten Redundanz einige Ähnlichkeiten zur hardwarebasierten Redundanz. Dabei können im Speziellen Parallelen zwischen der hardwarebasierten Replikation und der softwarebasierten Replikation gezogen werden [21, 15].

Die Technik der softwarebasierten Replikation wird ***N-version Programming*** genannt. Dabei werden Software-Teile N-fach repliziert und nebenläufig ausgeführt. Wie bei der hardwarebasierten Replikation können auch hier die Replikate gleichartig oder unterschiedlich sein, solange sie funktional äquivalent sind. Auch hier wird eine Einigung über die Ausgaben der Replikate mittels eines Entscheiders (engl. *Voter*) herbeigeführt. [15]

Meist werden generalisierte anstelle von anwendungsspezifischen Entscheidern eingesetzt. Neben einem Mehrheitsentscheider, der den häufigsten Wert als korrekt annimmt, gibt es weitere Alternativen. Hier ist zum einen der Mittelwert-Entscheider (engl. *Median Voter*) zu nennen, der den Mittelwert aller Ausgaben wählt. Weitere Entscheider wenden eine Technik an, bei der z. B. der gewichtete Mittelwert als korrekt angenommen wird oder Werte mit einer Abweichung von einer anwendungsspezifischen Schranke ϵ akzeptiert werden. [15]

Im Gegensatz zur Replikation mittels Hardware kann der Grad der Replikation hier feingranular gewählt werden. So kann eine Anwendung genauer abhängig von ihrem Schutzbedarf abgesichert werden. [55] Es ist denkbar, lediglich einige Anweisungen zu replizieren, bis hin zu ganzen Modulen oder Softwaresystemen. Bei letzterer Vorgehensweise wird beinahe der Redundanzbereich der hardwarebasierten Replikation erreicht. Die Replikation ganzer Systeme impliziert dabei allerdings einen höheren Ressourcenbedarf.

Wie bei der Replikation ganzer Systeme ist auch bei der Codierung auf Software-Ebene mit einem erhöhten Ressourcenbedarf zu rechnen. Prinzipiell können auf Software-Ebene Daten- und Kontrollflüsse durch Codierung abgesichert werden. Dabei ist mittels arithmetischer Codierung auch eine Fehlererkennung nach Anwendung arithmetischer Operationen möglich. [19, 15, 21]

Somit sind die Maßnahmen der Absicherung durch Codierung sehr weitreichend umsetzbar. Ein klarer Nachteil der Codierung ist allerdings der erhöhte Ressourcenbedarf, da Werte und Kontrollflüsse stets codiert werden müssen, bevor eine weitere Verarbeitung oder Programmausführung möglich ist. Es müssen zusätzliche Informationen mitgeführt werden und die Berechnung der Codes selbst sorgt für einen nicht unerheblichen Overhead. An dieser Stelle ist die Codierung auf Software-Ebene also vergleichbar mit der Codierung auf Hardware-Ebene.

2.3 Fehlercodierung

Codierung ist eines der am häufigsten verwendeten Verfahren zur Fehlererkennung beziehungsweise Fehlerkorrektur. Das liegt darin begründet, dass Codierung unabhängig von den zugrundeliegenden Daten angewendet werden kann. Eins der vermutlich bekanntesten und gleichzeitig einfachsten Beispiele für Codierungen ist die Prüfsumme in Kommunikationsprotokollen. Diese wird bei Datenübertragungen zusätzlich zu den eigentlichen Daten übermittelt. Abhängig von der Komplexität der Berechnungsvorschrift dient die Prüfsumme dem Empfänger zur Fehlererkennung und in einigen Fällen - abhängig von der Codierungsvorschrift - sogar zur Fehlerkorrektur.

Im Folgenden wird angenommen, dass die betrachteten Codes in Binärdarstellung vorliegen, da alle Berechnungen von Computern aufgrund ihrer Hardware in Binärdarstellung durchgeführt werden. Damit ist das Binärsystem auch für die Codierung mit Computern die natürliche Darstellung.

Grundsätzlich bedeutet **Codierung** nichts anderes als das Übersetzen eines n-Bit-Wortes mittels einer Codierungsvorschrift (engl. *coding scheme*) aus einer Definitionsmenge in ein Wort in einer Zielmenge bzw. in einem Bildraum (vgl. Abbildung 2.2). Eine **Codierungsvorschrift** ist eine injektive Abbildung. [31, 54] Da die Codierungsvorschrift dem Wort aus der Definitionsmenge k Prüfbits hinzufügt, ist die Kardinalität der Zielmenge wesentlich größer als die der Definitionsmenge. Die Prüfbits dienen der Fehlererkennung bzw. der Fehlerkorrektur. Damit haben codierte Wörter in der Zielmenge eine Länge von $n + k$ Bits. Enthalten die Definitionsmenge sowie die Zielmenge Binärwörter, ergibt sich somit für die Definitionsmenge eine Kardinalität von 2^n und für die Zielmenge eine Kardinalität von 2^{n+k}. [46]

Über eine Umkehrabbildung der Codierungsvorschrift können jene Codewörter, die im Bild liegen, zurück in Nutzwörter übersetzt werden.

Gewisse Codierungsvorschriften können abhängig von ihrer Beschaffenheit verschiedene Eigenschaften aufweisen. Bei arithmetischen Codes sind aufgrund ihrer Abgeschlossenheit gegenüber arithmetischen Operationen diese nicht nur in der Definitionsmenge, sondern auch direkt in der Zielmenge möglich. Das macht sie insbesondere interessant für das Anwendungsgebiet der Fehlertoleranz. [44, 46]

Das Hinzufügen von Prüfbits durch die Codierungsvorschrift kann auf zwei unterschiedliche Arten geschehen. Dies macht eine Unterscheidung von Codes in systematische und nicht-systematische bzw. separierte und nicht-separierte Codes notwendig (vgl. Abbildung 2.3). [46]

Systematische Codes sind binäre Codes der Länge $n+k$, die aus zwei Teilen bestehen (vgl. Abbildung 2.3a). Die ersten n Bits enthalten die zu codierenden Informationen. Die anderen k Bits dienen der Fehlererkennung oder der Fehlerkorrektur. [46] Hier besteht also eine klare Trennung der zu codierenden Information und der Prüfbits [22]. Dies ermöglicht

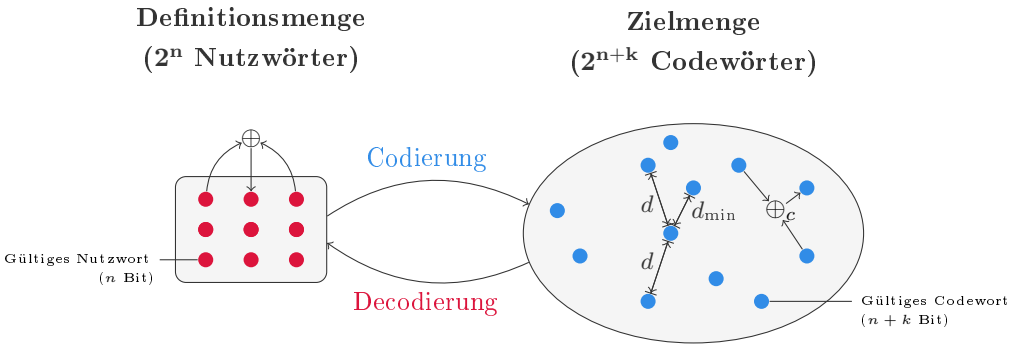
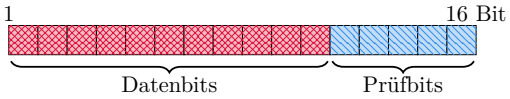


Abbildung 2.2: Funktionsweise von Codierungsverfahren. Die zu codierenden Wörter in der Definitionsmenge werden mithilfe einer Codierungsvorschrift in Wörter der Zielmenge übersetzt. Die Codierungsvorschrift fügt dabei dem n-Bit-Nutzwort k Bits hinzu. Dies vergrößert die Zielmenge auf 2^{n+k} Wörter im Vergleich zur Definitionsmenge mit 2^n Wörtern. So vergrößert sich auch der Abstand d zwischen zwei unterschiedlichen Codewörtern im Vergleich zum Abstand zweier Nutzwörter. Die Operation $\oplus$ verdeutlicht, dass für arithmetische Codes arithmetische Operationen sowohl in der Definitionsmenge als auch in der Zielmenge möglich sind, weil sie gegenüber arithmetischer Operationen abgeschlossen sind. (Abbildung entnommen aus Arbeiten von P. Ulbrich [55])

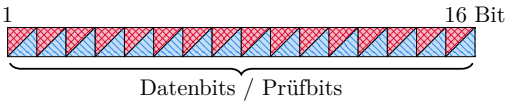
einen direkten Zugriff auf die codierten Informationen. Bei **nicht-systematischen Codes** liegen die zu codierenden Informationen und die Prüfbits vermischt vor (vgl. Abbildung 2.3b). Daher ist hier ein direkter Zugriff auf die codierten Informationen nicht möglich.

Bei **separierten Codes** werden die Prüfbits getrennt berechnet und gehalten (vgl. Abbildung 2.3c). Dies ist bei nicht-systematischen Codes nicht möglich, weshalb diese stets auch nicht-separiert sind. [46] Arithmetische Codes z. B. gehören zu den nicht-systematischen, nicht-separierten Codes. Abschnitt 2.3.1 thematisiert arithmetische Codes und ihre Codierungsvorschriften.

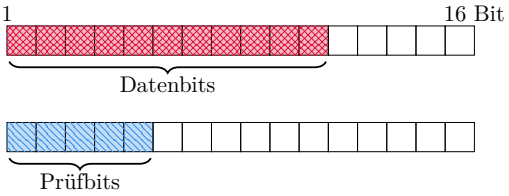
Die Tatsache, dass die Zielmenge einer Codierungsvorschrift meist bedeutend größer als ihre Definitionsmenge ist, hat zur Folge, dass ein größerer Abstand zwischen zwei Codewörtern in der Zielmenge vorliegt als zwischen ihren zwei Nutzwörtern in der Definitionsmenge (vgl. Abbildung 2.2). Der Abstand zwischen zwei Codewörtern bestimmt, ob eine Fehlererkennung und ggf. Fehlerkorrektur beispielsweise nach Auftreten eines transienten Fehlers möglich ist. Denn bei Auftreten eines Bit-Fehlers ist es möglich, dass bei geringem Abstand ein gültiges Codewort fälschlicherweise in ein anderes gültiges Codewort übergeht. In diesem Falle ist eine Fehlererkennung - geschweige denn eine Fehlerkorrektur - nicht mehr möglich. Daher kann insbesondere der minimale Abstand zwischen zwei gültigen Codewörtern von Interesse sein. Die Wahrscheinlichkeit, mit der ein solcher Fehler auftritt, wird als **Restfehlerwahrscheinlichkeit** bezeichnet. Die Restfehlerwahrscheinlichkeit für arithmetische Codes wird in Abschnitt 2.3.2 genauer beschrieben. [55]



(a) Systematischer Code mit $n = 11$ Datenbits und $k = 5$ Prüfbits.



(b) Nicht-systematischer Code. Daten- und Prüfbits liegen vermischt vor.



(c) Separierter, systematischer Code mit $n = 11$ Datenbits und $k = 5$ Prüfbits.

Abbildung 2.3: Vergleich (nicht-)systematischer und (nicht-)separierter Codes. Bei systematischen Codes werden die k Prüfbits getrennt von oder neben den n Datenbits gehalten. Werden die Prüfbits getrennt von den Datenbits gehalten, spricht man von separierten, systematischen Codes. Sind Daten- und Prüfbits vermischt, wird der Code als nicht-systematisch bezeichnet. (Abbildung entnommen aus Arbeiten von P. Ulbrich [55])

Für die Berechnung des Abstands zweier Codewörter können verschiedene Metriken herangezogen werden. Diese Metriken fallen abhängig von der Codierungsvorschrift unterschiedlich aus. Zwei mögliche Metriken zur Abstandsbestimmung bei arithmetischen Codes werden in Abschnitt 2.3.3 diskutiert. Dabei handelt es sich um die arithmetische Distanz und die Hamming-Distanz.

Neben arithmetischen Codes existieren noch weitere Codierungsverfahren bzw. Klassen wie z. B. *lineare Codes*. Die Arbeit von Peterson und Weldon Jr. [44] gibt einen breiteren Überblick über diese und weitere Codes, ihre Vor- und Nachteile sowie ihre Verwendung.

2.3.1 Arithmetische Codes

Arithmetische Codes sind solche Codes, die gegenüber arithmetischen Operationen abgeschlossen sind. Voraussetzung dafür ist, dass die Daten, auf denen die arithmetischen Operationen ausgeführt werden sollen, in codierter Form vorliegen und die Operationen auf den codierten Werten durchgeführt werden. Auf dem Ergebnis der arithmetischen Operation können Verfahren zur Fehlererkennung ausgeführt und das Ergebnis decodiert werden. Dies verdeutlicht die Abbildung 2.2. Das Symbol $\oplus$ steht für arithmetische Operationen und verdeutlicht, dass diese für arithmetische Codes sowohl in der Definitionsmenge als auch in der Zielmenge möglich sind. Für die Überprüfung nach Anwendung der Operation ist notwendig, dass die Operationen invariant sind. [15]

Arithmetische Codes sind insbesondere im Kontext fehlertoleranter Systeme von Interesse, da sie alle arithmetischen Berechnungen erlauben, die auch von einer ALU (Arithmetisch-logische Einheit) durchgeführt werden können. Daher können arithmetische Codes auf alle Wörter in allen Berechnungsschritten eines Programms angewendet werden und so während aller Berechnungen Fehlererkennung bzw.-korrektur ermöglichen. Insbesondere ermöglichen sie eine Erkennung von Fehlern, die durch transiente Fehler entstanden sind. [4]

Zu der Klasse der arithmetischen Codes gehören beispielsweise *Residue-Codes* und *AN-Codes* zusammen mit den Erweiterungen der *AN-Codes* zu *ANB-Codes* bzw. *ANBD-Codes*. Im Folgenden werden AN-Codes und ihren Erweiterungen beschrieben. Für eine Erläuterung der Residue-Codes seien [15] und [28] empfohlen. Alle Formen von AN-Codes sind nicht-systematisch und nicht-separiert [15].

Abhängig, ob AN-Codes, ANB-Codes oder ANBD-Codes betrachtet werden, können unterschiedliche Fehlerarten erkannt werden, wie im Folgenden beschrieben werden soll. Zu diesen Fehlern gehören Berechnungsfehler, Operatorfehler und Operandenfehler.

Unter **Operatorfehlern** versteht man solche Fehler, bei denen die Instruktion oder der Programmzähler von einem Fehler betroffen ist [19]. In der Konsequenz wird eine falsche Operation verwendet - z.B. eine Addition statt einer Subtraktion.

Bei **Operandenfehlern** ist einer der Operanden durch einen Fehler verfälscht [19]. Dies kann durch eine Verfälschung des Wertes, der eigentlich an der Adresse stehen sollte, oder durch ein verlorenes Update (engl. *lost update*) geschehen. Im letzteren Fall wird ein veralteter Wert für die Berechnung verwendet. Beispielsweise wird statt der Variable x die Variable y in der Berechnung verwendet oder das Programm greift auf einen alten Wert von x zu.

AN-Codes

AN-Codes wurden von P. Forin [19] vorgestellt. Gegeben einen Schlüssel $A > 1$ und einen zu codierenden Wert v, errechnet sich der codierte Wert v_c durch $v_c = A \cdot v$. Die Decodierung ist durch die Umkehrfunktion $v = \frac{v_c}{A}$ möglich. Kommt es bei der Division zu einem Rest, lässt sich feststellen, dass ein Fehler aufgetreten ist. Dies geschieht in der Praxis also mittels der Modulo-Operation $v_c \mod A$. Gilt $v_c \mod A \neq 0$, so ist ein Fehler aufgetreten. Ist $v_c \mod A = 0$, so war die Operation vermutlich fehlerfrei.

Dabei verbleibt eine geringe Restfehlerwahrscheinlichkeit von $p_{\mathtt{SDC}} \approx \frac{1}{A}$, bei der ein Fehler unerkannt bleibt [19]. In diesem Falle wird bei einem Fehler aus einem gültigen Codewort wieder ein gültiges Codewort. [20, 55]

AN-Codes unterstützen alle relevanten arithmetischen Operatoren [47]. Mithilfe von AN-Codes ist es außerdem möglich, Fehler im Wertebereich vollständig zu erkennen. Eine Erkennung von Operandenfehlern sowie Operatorfehlern fehlt allerdings. Denn es ist denkbar, dass die Operation selbst korrekt durchgeführt wurde. Falls aber in der Ope-

ration beispielsweise aufgrund eines Ladefehlers einer Adresse oder eines Operators ein falscher Operand oder ein falscher Operator bei der Berechnung verwendet werden, so ist das Ergebnis semantisch dennoch falsch. [19] Dies macht eine Erweiterung der AN-Codes notwendig, sodass auch diese Fehlerarten erkannt werden können.

ANB-Codes

ANB-Codes sind eine erweiterte Form der AN-Codes und ermöglichen zusätzlich die Erkennung von Operandenfehlern und Operatorfehlern. Dafür wird der Code um eine statische Signatur $B < A$ erweitert, sodass für eine Variable v, einen Schlüssel A und eine Signatur B_v gilt $v_c = A \cdot v + B_v$. Die Signatur wird durch eine statische Analyse bestimmt und ist somit vorab bekannt. [19] Signaturen sind variablenspezifisch [19] - d. h. für zwei uncodierte Variablen x und y, ihre Signaturen B_x und B_y und ihre Codewörter x_c und y_c gilt

$$x = y \iff B_x = B_y. \tag{2.1}$$

Die Fehlererkennung ist möglich mittels $v_c \mod A = B_v$. Decodiert wird mit $v = \frac{(v_c - B_v)}{A}$. Aufgrund der Erweiterung der Codes durch die variablenspezifische Signatur ist keine einfache Codierung der Division möglich. Stattdessen muss eine wiederholte encodierte Subtraktion mit entsprechendem Zeitaufwand erfolgen. Alternativ ist eine Absicherung der zu dividierenden Werte mittels AN-Codes mit den oben genannten Nachteilen möglich. ANB-Codes unterstützen daher alle arithmetischen Operatoren bis auf die direkte Division. Dabei können allerdings aufgrund der Art des Codes Korrekturmaßnahmen notwendig werden. [47]

Wegen der variablenspezifischen Signaturen ist eine erweiterte Fehlererkennung möglich. So kann mittels einer Überprüfung der Signaturen eine Überprüfung auf Operanden- sowie Operatorfehler erfolgen. Hierzu werden bei Operationen die Signaturen der berechneten Werte aus den statischen Signaturen abgeleitet. Würde ein falscher Operand verwendet werden, so würde sich auch die Signatur verändern und der Fehler würde entdeckt werden. Mit einem analogen Verfahren können auch Operatorfehler entdeckt werden. Dafür findet eine Vorberechnung der erwarteten Signaturen statt, um nach der eigentlichen Operation einen Abgleich der Ergebnissignatur mit der erwarteten Signatur durchführen zu können. [47]

Zur weiteren Fehlererkennung kann es notwendig werden, die Gleichheit zweier Codewörter festzustellen. Es lässt sich zeigen, dass die Äquivalenzprüfung mit bereits existierenden Mitteln leicht umzusetzen ist. Zwei uncodierte Wörter x und y sind genau dann

gleich, wenn die Differenz der codierten Wörter x_c, y_c und die Differenz der Signaturen B_x, B_y gleich sind, [55, 24]

$$\begin{aligned} x_c - y_c &= (A \cdot x + B_x) - (A \cdot y + B_y) \\ &= A \cdot x - A \cdot y + B_x - B_Y \\ &\overset{x=y}{=} B_x - B_y. \end{aligned} \tag{2.2}$$

Trotz der Erweiterung der AN-Codes um variablenspezifische Signaturen zu ANB-Codes, können diese weiterhin nicht alle Arten von Fehlern erkennen. So ist es möglich, dass aufgrund eines Bit-Fehlers eine Variable an einer falschen Adresse gespeichert wird. Im Verlauf des Programmes wird die Variable an der eigentlich korrekten Adresse erwartet und der Wert, der an dieser Adresse steht, wird ausgelesen. Zu diesem Zeitpunkt steht allerdings eine veraltete Version des erwarteten Wertes an dieser Adresse, da der eigentlich korrekte Wert durch den Bit-Fehler an einer falschen Adresse gelandet ist. Der aktualisierte Wert ist also verloren gegangen. Man spricht von einer verlorenen Änderung (engl. *Lost Update*). Solche Fehler können daher nicht durch die ANB-Codierung erkannt werden. Durch eine erneute Erweiterung der ANB-Codes zu ANBD-Codes wird jedoch eine Erkennung dieser Fehler möglich. [19]

ANBD-Codes

Bei der Erweiterung der ANB-Codes zu **ANBD-Codes** wird ein Zeitstempel D ergänzt. So wird eine vollständige Fehlererkennung im Sinne des Fehlermodells möglich. [19]

Ein Wert v lässt sich durch $v_c = A \cdot v + B_v + D$ codieren [19]. Während die Signaturen nach wie vor vorher ermittelt und statisch festgelegt werden, wird D dynamisch zur Laufzeit bestimmt. Die Bestimmung von D ist dabei abhängig von der Implementierung der genauen Codierung. Zur Fehlererkennung und Decodierung müssen allerdings sowohl der Zeitstempel D als auch die Signatur B_v, welche zur Codierung von v verwendet wurden, bekannt sein. [47]

ANBD-Codes ermöglichen dieselben arithmetischen Operationen wie ANB-Codes. Lediglich die Implementierung wie auch die Korrekturmaßnahmen, die bei einigen arithmetischen Operatoren benötigt werden, steigen in ihrer Komplexität. [47] Auch bei ANBD-Codes ist es möglich, die Gleichheit zweier Codewörter wie in Abschnitt 2.2 zu überprüfen [55, 24].

2.3.2 Restfehlerwahrscheinlichkeit

Die **Restfehlerwahrscheinlichkeit** ist die Wahrscheinlichkeit dafür, dass ein gültiges Codewort durch einen Fehler in ein anderes gültiges Codewort übergeht. Der Fehler bleibt dann von der Codierung unbemerkt.

Im Folgenden wird angenommen, dass die Wahrscheinlichkeit, mit der Verfälschungen eines Codewortes auftreten, gleichverteilt sind. Die Restfehlerwahrscheinlichkeit p_{sdc} lässt sich wie von R. A. Frohwerk [20] und P. Ulbrich [55] beschreiben durch

$$p_{\text{sdc}} = \frac{\text{Anzahl gültiger Codeworte} - 1}{\text{Anzahl möglicher Codeworte} - 1} = \frac{2^n - 1}{2^{n+k} - 1} \underset{n\to\infty}{\longrightarrow} \frac{1}{2^k}. \tag{2.3}$$

Für AN-Codes und ihre Erweiterungen ist die Restfehlerwahrscheinlichkeit etwa $\frac{1}{A}$, da den n Nutzdaten mit der Codierungsvorschrift $k = A$ Prüfbits hinzugefügt werden [55]. Die Restfehlerwahrscheinlichkeit für AN-Codes kann also für möglichst große Schlüssel deutlich gedrückt werden. Damit ist also die Robustheit des Codes direkt abhängig vom Schlüssel A. Aus mathematischer Sicht ist es aufgrund der Teilerfremdheit der Codewörter sinnvoll, Primzahlen als Schlüssel zu verwenden. [19]

In der Praxis fehlertoleranter Systeme hat sich dies als nicht sehr zielführend herausgestellt, da Computer in der Binärdarstellung rechnen. Hier sind vielmehr robuste Bitmuster entscheidend, bei denen ein Ein-Bit-Fehler nicht direkt dafür sorgt, dass ein gültiges Codewort in ein anderes gültiges Codewort übergeht. P. Ulbrich [55] ermittelt in seiner Arbeit fünf sogenannte „Super As“ für 32-Bit-AN-Codes, die sich aufgrund ihrer minimalen Hamming-Distanz (vgl. Abschnitt 2.3.3) besonders gut zur Parametrisierung der arithmetischen Codierung eignen.

Es wird insgesamt deutlich, dass die Restfehlerwahrscheinlichkeit eine theoretische Orientierung für die Robustheit des Codes gibt. [55] Es ist aber für die praktische Anwendung wesentlich wichtiger, wie viele bzw. welche Fehler ein Code tatsächlich tolerieren kann, bevor er degeneriert. Für die weitere Bewertung müssen daher weitere Metriken, wie z.B. die minimale Hamming-Distanz, herangezogen werden, die im Folgenden beschrieben werden sollen.

2.3.3 Metriken arithmetischer Codes

Zur Bewertung der Robustheit von Codewörtern wird eine geeignete Metrik benötigt. Die verwendete Metrik ermöglicht eine Aussage darüber, wie viele Bits eines Codewortes fehlerhaft sein können, während eine Fehlererkennung dennoch möglich ist. Die richtige Wahl einer Metrik ist essenziell für die Auswertung der Robustheit von Codewörtern. Dies liegt darin begründet, dass die Leistung der Codierungsvorschrift möglichst realistisch eingeschätzt werden soll. Eine Überschätzung der Fehlererkennungsleistung des Codes kann insbesondere in sicherheitskritischen Systemen fatale Auswirkungen haben. Aber auch eine zu pessimistische Einschätzung der Robustheit kann negative Konsequenzen haben. So könnte beispielsweise fälschlicherweise angenommen werden, dass weitere Härtungsmaßnahmen zur Absicherung eines Systems erforderlich sind. Dies könnte in höheren Kosten resultieren.

Wie in Abschnitt 2.3.2 beschrieben, ist die Restfehlerwahrscheinlichkeit keine geeignete Metrik für die genaue Einschätzung der Robustheit der gewählten Codierungsvorschrift, da

sie keine Aussage darüber treffen kann, welche Fehler die Codierung tatsächlich tolerieren kann. Alternativ stehen zwei weitere Metriken zur Auswahl. Zum einen kann die arithmetische Distanz zweier Codewörter betrachtet werden [44]. Diese Metrik wurde von Peterson und Weldon Jr. explizit für AN-Codes entwickelt. Alternativ kann die Hamming-Distanz als Metrik herangezogen werden. Bei der Hamming-Distanz handelt es sich um eine Metrik, die in der Codierungstheorie häufig im Kontext von binären Codes zum Einsatz kommt.

Arithmetisches Gewicht und arithmetische Distanz Das arithmetische Gewicht w_a sowie die arithmetische Distanz d_a werden von Peterson und Weldon Jr. als Metrik zur Bewertung von AN-Codes definiert [44].

Eine Zahl wird dargestellt als eine Linearkombination von Potenzen der Basis dieser Zahl. Für eine Zahl $N \in \mathbb{Z}$, $0 \leq |N| < r^n$ der Länge n mit $0 \leq N_i < r$ ergibt sich eine Darstellung $N = sgn(N) \cdot (N_{n-1}r^{n-1} + N_{n-2}r^{n-2} + \ldots + N_1 + N_0)$, wobei sgn die Vorzeichenfunktion ist. Die Zahl N wird geschrieben als $N_{n-1}N_{n-2}\ldots N_1N_0$. [44]

Das **arithmetische Gewicht** w_a wird definiert als die Anzahl der Summanden, die in der Darstellung von $N = a_nr^n + a_{n-1}r^{n-1} + \ldots + a_0$ ungleich Null sind. Dabei sind die Koeffizienten a_i betragsmäßig kleiner als r. Wesentlich dabei ist, dass auch negative Koeffizienten möglich sind. [44]

2.3.1 Beispiel. Man betrachte das Dezimalsystem, d. h. $r = 10$. Die Zahl 432 ergibt sich aus

$$432 = 4 \cdot 10^2 + 3 \cdot 10^1 + 2 \cdot 10^0.$$

Es sind drei Summanden vorhanden, deren Werte ungleich Null sind. Daraus resultiert ein arithmetisches Gewicht $w_a = 3$.

2.3.2 Beispiel. Man betrachte das Binärsystem, d. h. $r = 2$. Die Zahl $(0001\ 1001)_2$ setzt sich zusammen aus

$$\begin{aligned}(0001\ 1001)_2 &= 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^0 \\ &= 25.\end{aligned}$$

Es sind drei Summanden vorhanden, deren Werte ungleich Null sind. Daraus folgt ein arithmetisches Gewicht $w_a = 3$.

Die **arithmetische Distanz** d_a zweier Zahlen N_1 und N_2 ist definiert als das Gewicht der Differenz $N_1 - N_2$. Es gilt also $d_a = w_a(N_1 - N_2)$.

2.3.3 Beispiel. Im Dezimalsystem mit $r = 10$ und $N_1 = 1904$ und $N_2 = 1909$ ist die arithmetische Distanz $d_a = 1$, weil $1904 - 1909 = -5 = -5 \cdot 10^0$.

2.3.4 Beispiel. In der Binärdarstellung mit $r = 2$ und $N_1 = (0011\ 1000)_2 = 56$ und $N_2 = (0010\ 1000)_2 = 40$ ergibt sich eine arithmetische Distanz $d_a = 1$, weil $N_1 - N_2 = 16 = (0001\ 0000)_2 = 1 \cdot 2^4$.

Hamming-Gewicht und Hamming-Distanz Das **Hamming-Gewicht** eines Punktes $z = (z_1, \ldots, z_n)$ in einem n-dimensionalen Vektorraum ist definiert als die Anzahl der Werte z_i, die unterschiedlich zu Null sind [54].

Die **Hamming-Distanz** ist im Binärsystem die Anzahl der Bits, in denen sich zwei gleichlange Binärwörter x und y unterscheiden. Allgemein ist die Hamming-Distanz zweier Punkte $x = (x_1, \ldots, x_n)$ und $y = (y_1, \ldots, y_n)$ in einem n-dimensionalen Vektorraum die Anzahl der Koordinaten nach Wahl einer Basis r, in denen x und y sich unterscheiden. [54]

Es gilt also nach Stichtenoth [54]

$$w_h(z) := |\{i; z_i \neq 0\}| \tag{2.4}$$

und

$$d_h(x, y) := |\{i; x_i \neq y_i\}|. \tag{2.5}$$

2.3.5 Beispiel. Das Binärwort $z = (1100)$ hat ein Hamming-Gewicht $w_h = 2$, weil die Positionen zwei und drei ungleich Null sind.

2.3.6 Beispiel. Für die Binärwörter $x = (1100)$ und $y = (1010)$ ist die Hamming-Distanz $d_h = 2$, weil sich die beiden gleichlangen Binärwörter an den Positionen eins und zwei unterscheiden.

Code-Distanz Die **Code-Distanz** d_C eines Codes C ist die minimale Hamming-Distanz $d_{h_{\min}}$ zwischen zwei verschiedenen Codewörtern des Codes C. Die Code-Distanz erlaubt Aussagen darüber, ob eine Fehlererkennung oder sogar Fehlerkorrektur für diesen Code möglich ist. [15]

2.3.7 Beispiel. Für den Code $C = \{00, 01, 10, 11\}$ ist die Code-Distanz zwischen den Codewörter $d_C = 1$. Daher würde ein Ein-Bit-Fehler dazu führen, dass ein gültiges Codewort - z. B. 00 - in ein anderes gültiges Codewort - hier 01 oder 10 - übergeht. Damit sind fehlererkennende Eigenschaften nicht gegeben.

2.3.8 Beispiel. Für einen Code $C = \{000, 111\}$ mit einer Code-Distanz $d_C = 3$ existieren fehlerkorrigierende Eigenschaften für maximal Ein-Bit-Fehler [15]. Läge beim Codewort $c_1 = 000$ ein Ein-Bit-Fehler vor, so würde das Codewort zu $c_1' = 001$, $c_1'' = 010$ oder $c_1''' = 100$ verfälscht werden. Es ist aber weiterhin erkennbar, dass keine Verfälschung des Codewortes $c_2 = 111$ vorliegen kann, da eine Verfälschung eines Bits für c_2 entweder $c_1' = 110$, $c_1'' = 101$ oder $c_1''' = 011$ erzeugen würde.

Vergleich von arithmetischer Distanz und Hamming-Distanz Die arithmetische Distanz betrachtet Fehler, die vor einem Addierer anliegen können. Dabei wird von einer Verwendung von Ripple-Carry-Addierern (engl. *Ripple-Carry-Adder*) ausgegangen [44]. Daher können nach der Durchführung eines Addiervorgangs aufgrund von Überträgen (engl. *carry*) und der Funktionsweise der Ripple-Carry-Addierer im Allgemeinen gleich mehrere Bits im Ergebnis fehlerhaft sein [44]. Deswegen argumentieren Peterson und Weldon Jr., dass die arithmetische Distanz für arithmetische Codes akkurater sei als die Hamming-Distanz [44].

Die Hamming-Distanz betrachtet - im Gegensatz zur arithmetischen Distanz - Fehler, die nach einem Addiervorgang am Addierer auftreten können [55]. Damit ist die Hamming-Distanz nicht auf ein spezielles Design der ALU (Arithmetisch-logische Einheit) festgelegt. Stattdessen liefert die Hamming-Distanz auch aussagekräftige Ergebnisse, sofern andere Addiernetze wie beispielsweise Carry-Look-Ahead-Addierer oder andere schnellere Addiernetze verwendet werden [43].

Des Weiteren kann in einem komplexen Programm nicht davon ausgegangen werden, dass lediglich die Addierer der ALU verwendet werden. Eine heutige ALU stellt neben Addierern und logischen Einheiten typischerweise auch Multiplizierer und weitere Einheiten zur Verfügung[1]. Einen Überblick über Rechenschaltungen geben Mano und Ciletti [34].

Ein Nachteil bei Verwendung der Hamming-Distanz ist, dass keine Aussage mehr darüber getroffen werden kann, wie viele Fehler vor der Durchführung einer arithmetischen Operation vorlagen. Damit kann lediglich eine Aussage über die schlussendlich nach den Berechnungen vorliegende Zahl fehlerhafter Bits getroffen werden. Eine Aussage, wie viele Informationen vor oder während einer Reihe arithmetischer Operationen vorliegen, ist nicht möglich. [55]

Insgesamt sind die Einsatzmöglichkeiten der arithmetischen Distanz zu eingeschränkt, als dass sie sich als praxistauglich erweisen würde.

2.4 Strukturelle Redundanz

Strukturelle Redundanz kann in Form von Replikation implementiert werden und ist eine der häufigsten Fehlertoleranzmaßnahmen zur Verbesserung der Systemzuverlässigkeit. Replikation kann hardware- oder softwarebasiert erreicht werden. Die softwarebasierte Replikation hat sich aus der hardwarebasierten Replikation entwickelt. [15]

2.4.1 Replikation

Unter **Replikation** versteht man die Mehrfachauslegung von Modulen, Komponenten, Prozessen o.ä. mit dem Ziel der mehrfachen parallelen Ausführung zur Fehlererkennung [15].

[1] Es ist zwar möglich, die Einheiten einer ALU auf einfache Rechenschaltungen zu beschränken, dies schlägt sich allerdings ggf. auf die benötigte Rechenzeit für eine Operation nieder [43].

Die Replikation mit N Replikaten wird auch als N-fache Redundanz (engl. *N-Modular Redundancy*) bezeichnet [16]. Die einzelnen Instanzen der Replikation werden als **Replikate** bezeichnet. Je nach Anzahl der zu tolerierenden Fehler wird eine bestimmte Anzahl von Replikaten benötigt. Die Replikate werden parallel ausgeführt, die Ausgaben verglichen und zu einem Ergebnis zusammengeführt. [15]

Replikate können gleichartig oder verschiedenartig sein. Diversität zwischen den Replikaten ist insbesondere bei softwarebasierter Replikation möglich, aber nicht zwingend erforderlich. Sie wird durch die Überlegung motiviert, dass gleichartige Fehler in den Replikaten zu einer erhöhten Wahrscheinlichkeit des parallelen Ausfalls mehrerer Replikate führen können. Diversität kann daher helfen, robust gegenüber Gleichtaktfehlern zu sein, z. B. aufgrund von Entwurfsfehlern, Softwarefehlern und gängigen Fehlermustern bestimmter Implementierungen im Allgemeinen. Gleichzeitig wird dadurch jedoch der Entwicklungs- und Testaufwand erhöht. Insbesondere ist in den meisten Fällen anwendungsbezogenes Wissen erforderlich, um Diversität in den redundanten Komponenten umsetzen zu können. [21]

Darüber hinaus bestehen weitere Schwierigkeiten im Entscheidungsprozess, die durch die Diversität in den Replikaten entstehen. Denn eine Entscheidung über unterschiedliche Ausgaben der verschiedenen Replikate zu treffen, kann die Komplexität des Entscheiders erheblich erhöhen. [10, 16] Der CoRed-Ansatz setzt keine Diversität in den Replikaten ein, weshalb diese Möglichkeit hier nicht tiefer erläutert werden soll. Für einen tieferen Einblick wird die Arbeit von Bishop [10] empfohlen.

Die einzelnen Replikate müssen sowohl räumlich als auch zeitlich voneinander isoliert werden, um zu verhindern, dass Fehler in einzelnen Replikaten auf andere Replikate Einfluss nehmen. Räumlich könnte sich beispielsweise ein Fehler von einem Replikat auf ein weiteres ausbreiten. Aus zeitlicher Sicht ist es möglich, dass ein Replikat Ressourcen aufgrund eines Fehlers monopolisiert und diese den anderen Replikaten nicht weiter zur Verfügung stehen. So verhindert die zeitliche Isolation beispielsweise, dass ein Replikat die Ausführungs-Ressourcen aufgrund einer endlos laufenden Schleife blockiert. Insbesondere dürfen sich Replikate nicht auf Berechnungen anderer Replikate verlassen oder auf Nachrichten eines anderen Replikats warten. [55]

Der folgende Abschnitt beschäftigt sich mit der Dreifachredundanz (engl. *TMR (Triple Modular Redundancy)*). Diese soll als häufig verwendeter Fall der N-fachen Redundanz näher erläutert werden (vgl. Abschnitt 2.4.2).

2.4.2 Dreifachredundanz

Bei der **Dreifachredundanz** wird ein System, eine Komponente, ein Prozess o. ä. dreifach repliziert (vgl. Abbildung 2.4). Die drei Replikate werden parallel ausgeführt. Ein **Mehrheitsentscheider** (engl. *Majority Voter*) sucht nach einer Mehrheit in den Aus-

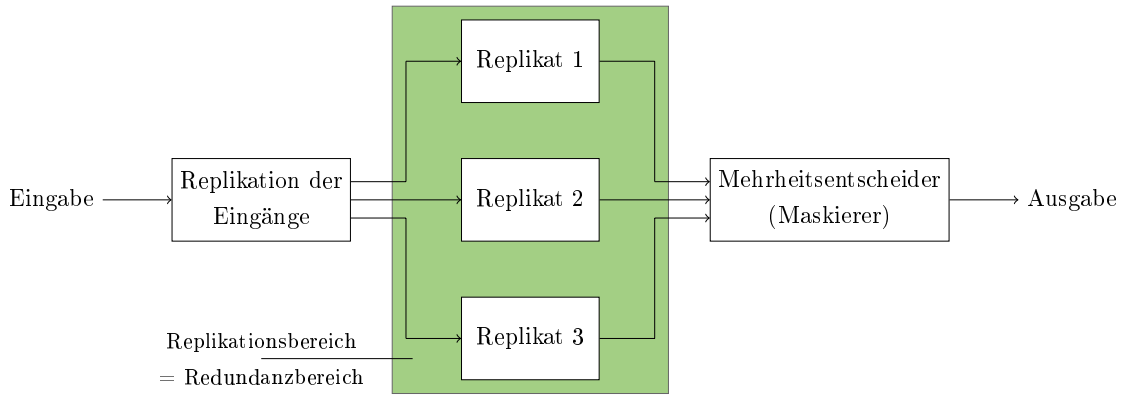


Abbildung 2.4: Schematischer Aufbau der Dreifachredundanz. Ein Programm und die Eingabe werden dreifach repliziert. Die replizierten Eingaben werden auf die Replikate verteilt. Die Ergebnisse aus den drei Replikaten werden in einen Mehrheitsentscheider gegeben. Dieser entscheidet sich für das Ergebnis der Mehrheit, falls eine Mehrheit existiert. Kann keine Mehrheit gefunden werden, stellt der Entscheider einen Fehler fest. Die Replikation der Eingänge und der Mehrheitsentscheider sind nicht Teil des Redundanzbereichs. (Abbildung entnommen aus einer Arbeit von P. Ulbrich [55])

gaben der Replikate. Er vergleicht dafür die Ausgaben der drei Replikate untereinander. Wird eine Mehrheit gefunden, ist diese das Ergebnis der replizierten Berechnung. Wird keine Mehrheit gefunden, signalisiert der Mehrheitsentscheider einen Fehler. [15] In diesem Fall kann das System z.B. durch eine erneute Berechnung versuchen, einen fehlerfreien Zustand wiederherzustellen [46]. Die Wahl einer ungeraden Anzahl von Replikaten, wie bei der Dreifachredundanz, ist sinnvoll, um Gleichstände bei der Mehrheitssuche zu verhindern [53]. Abgesehen davon existieren weitere Arten generalisierter Entscheider. [21, 15, 53] Auch der CoRed-Ansatz setzt Dreifachredundanz ein.

Jedes Replikat erhält die gleiche, replizierte Eingabe und führt dieselbe deterministische Berechnung durch, da auf Diversität verzichtet wird. Bei fehlerfreier Ausführung der Replikate sind ihre Ausgaben daher identisch, sofern sie im selben Zeitintervall erfolgen. Dies wird als **Replikdeterminismus** bezeichnet. Ohne Replikdeterminismus ist ein Mehrheitsentscheid nicht möglich. [45]

Der replizierte Bereich wird als **Replikationsbereich** (engl. *Sphere of Replication*) bezeichnet. Bei der Dreifachredundanz ist der Replikationsbereich gleichzeitig der Bereich, der von den Redundanzmaßnahmen betroffen ist, der sogenannte **Redundanzbereich**. Fehler innerhalb des Redundanzbereichs, die sich bis zu dessen Grenze fortpflanzen, können während des Vergleichs durch einen Mehrheitsentscheider erkannt werden. [36]

Abhängig von der Anzahl der Replikate und dem verwendeten Fehlermodell können im Allgemeinen mehr oder weniger fehlerhafte Replikate toleriert werden. Unter der Annahme, dass fehlerhafte Replikate f gar keine oder auch eine falsche Ausgabe produzieren

können, aber byzantinische Fehler ausgeschlossen sind, kann ein System maximal $2f + 1$ Fehler tolerieren. [27] Unter **byzantinischen Fehlern** versteht man Fehler, bei welchen eine Systemkomponente beliebig falsches Verhalten zeigt [29]. Wenn maximal ein Fehler gleichzeitig auftritt, ist die Dreifachredundanz folglich aufgrund der drei Replikate dazu in der Lage, diesen zu tolerieren.

Unabhängig davon, wie viele Replikate verwendet werden, bleibt jedoch der Entscheider eine kritische Fehlerstelle. Fällt der Mehrheitsentscheider aus, kann keine Entscheidung getroffen werden und das System versagt. Für dieses Problem gibt es verschiedene Lösungsansätze. Der vielleicht naheliegendste ist, den Mehrheitsentscheider zu replizieren. Der Mehrheitsentscheider wird also auch dreifach redundant ausgelegt und jeder replizierte Mehrheitsentscheider erhält als Eingabe die Ausgaben aller drei Replikate, die die Berechnung ausführen. [15]

Dieser Ansatz hat jedoch die Schwäche, dass am Ende des Prozesses drei neue Ergebnisse bereitstehen. Daher muss über die Ausgaben der replizierten Mehrheitsentscheider entschieden werden, falls der Rest des Systems nur mit einer Ausgabe rechnet. Die Lösung ist die Verwendung eines weiteren Mehrheitsentscheiders, der über die Ausgaben der replizierten Mehrheitsentscheider entscheidet, und das Problem der kritischen Fehlerstelle bleibt bestehen. [55]

Neben dem Mehrheitsentscheider gibt es weitere kritische Fehlerstellen in der Replikation. Dazu gehört u. a. die Systemkomponente, welche für die Replikation der Eingabedaten verantwortlich ist. Fällt diese aus, erhalten die Replikate keine Eingabedaten und das System befindet sich ebenfalls in einem Fehlerzustand. [55]

Ein Ansatz, der diese Probleme löst, ist der CoRed-Ansatz, welcher im folgenden Abschnitt beschrieben wird.

2.5 Kombinierte Redundanz (CoRed)

Der CoRed-Ansatz vereint die arithmetische Fehlercodierung und die Replikation zu einem ganzheitlichen, softwarebasierten Redundanzansatz (vgl. Abbildung 2.5). Wie alle anderen softwarebasierten Redundanzansätze ist selbstverständlich auch CoRed darauf angewiesen, dass die Hardware selbst nicht vollständig ausfällt. Ist diese Voraussetzung gegeben, eliminiert CoRed allerdings die verbleibenden Fehlerfälle in der arithmetischen Codierung und der Replikation. [55]

2.5.1 Aufbau des CoRed-Ansatzes

Berechnungen werden im CoRed-Ansatz mithilfe der Replikation (vgl. Abschnitt 2.4) redundant ausgelegt. Die verbleibenden kritischen Fehlerstellen der Replikation, der Replikator und der Mehrheitsentscheider, werden mithilfe der arithmetischen Fehlercodierung

(vgl. Abschnitt 2.3.1) abgesichert. Die replizierten Bereiche selbst werden nicht codiert, da sie bereits gehärtet sind und somit der Overhead, der sonst durch die Codierung entsteht, an dieser Stelle vermieden werden kann. CoRed übernimmt dafür die (De-)Codierung an den Übergängen zu den Replikaten; genauer übernimmt der Ansatz die Decodierung an den Replikateingängen sowie die Codierung an den Replikatausgängen. [55]

CoRed bietet darüber hinaus verschiedene Vorgehensweisen bei der Eingangsreplikation und bei der Ausgabe des Ergebnisses an. Es ist denkbar, dass die eingesetzte Hardware bereits eine codierte Eingabe für CoRed liefert. In diesem Fall muss ein Replikator lediglich die Eingabe replizieren und an die drei Replikate weiterreichen. Ist dies nicht gegeben, kommt ein Stellvertreter zum Einsatz, welcher zum frühestmöglichen Zeitpunkt den Eingabewert codiert und anschließend repliziert und an die Replikate weiterreicht. [55]

Darüber hinaus gibt es komplexere Lösungen, falls z.B. mehrere Sensorwerte als Eingabe verwendet werden sollen oder diese Eingaben leicht voneinander abweichen. Diese Lösung wird an dieser Stelle allerdings nicht weiter thematisiert, da sie für das Ziel dieser Arbeit nicht grundlegend ist und eine Erklärung den Umfang dieser Arbeit überschreiten würde. Für eine Erklärung wird die Arbeit von Ulbrich [55] empfohlen.

Nach erfolgreicher Berechnung der Werte in den Replikaten sowie dem Mehrheitsentscheid über die Ausgaben kann mithilfe eines codierten Ausgangs eine Ausgabe des codierten Ergebnisses erfolgen. Das kann sinnvoll sein, weil der Aktor[2], wenn er einen codierten Wert erhält, ebenfalls eine Fehlererkennung durchführen kann. Damit ist eine Ausweitung des Redundanzbereichs über die Grenzen von CoRed hinweg möglich. So kann die Software bis an ihre Grenzen vollständig redundant ausgelegt werden. [55]

Grundlegend für die Funktionsweise von CoRed ist der codierte Mehrheitsentscheider, welcher im Folgenden beschrieben werden soll.

2.5.2 Codierter Mehrheitsentscheider

Der codierte Mehrheitsentscheider nimmt die codierten Ausgaben x_c, y_c und z_c mit ihren zur Übersetzungszeit bekannten statischen Signaturen B_x, B_y und B_z aus den Replikaten entgegen und führt über diese Werte einen Mehrheitsentscheid durch. Der Mehrheitsentscheider verhält sich dabei „von außen betrachtet wie eine codierte Operation“ ([55]). Er nimmt die codierten Werte als Parameter an, führt Vergleiche auf den codierten Werten durch und gibt einen codierten Gewinner sowie eine Sprungsignatur als Ergebnis zurück (vgl. Algorithmus 2.1). [55]

Für die Vergleiche im Mehrheitsentscheider müssen die Werte aufgrund der Eigenschaft $x = y \iff x_c - y_c = B_x - B_y$ (vgl. Gleichung 2.2 in Abschnitt 2.3.1) nicht decodiert werden.

[2] Ein Aktor ist ein Bauelement, das Signale beispielsweise in Bewegung umwandelt und damit das Gegenstück zu einem Sensor.

Algorithmus 2.1: Der CoRed-Mehrheitsentscheider

```
1  Require: B_x, B_y, B_z                    % Konstante Signaturen der Operanden
2
3  function CORED_VOTE(x_c, y_c, z_c)
4    ZERO_LOCAL_STORAGE()
5    if ((x_c − y_c) = (B_x − B_y)) then
6      if ((x_c − z_c) = (B_x − B_z)) then
7        win_c ⟵ APPLY(x_c, (x_c − y_c) + (x_c − z_c))
8        return win_c, B_E ⟵^const (B_x − B_y) + (B_x − B_z) % Alle Ergebnisse identisch
9      else
10       win_c ⟵ APPLY(x_c, (x_c − y_c))
11       return win_c, B_E ⟵^const (B_x − B_y)          % z_c weicht ab
12     end if
13   else if ((x_c − z_c) = (B_x − B_z)) then
14     win_c ⟵ APPLY(x_c, (x_c − z_c))
15     return win_c, B_E ⟵^const (B_x - B_z)          % y_c weicht ab
16   else if ((y_c − z_c) = (B_y − B_z)) then
17     win_c ⟵ APPLY(y_c, (y_c − z_c))
18     return win_c, B_E ⟵^const (B_y − B_z)          % x_c weicht ab
19   else
20     win_c ⟵ ∅
21     SIGNAL_DUE()                                   % Keine Entscheidung möglich
22   end if
23 end function
24
25 function APPLY(v_c, B_dyn)
26   if (B_dyn > B_max) then SIGNAL_DUE()
27   return v_c + B_dyn
28 end function
```

Algorithmus 2.1: Implementierung des CoRed-Mehrheitsentscheiders. Der codierte Mehrheitsentscheider aus dem CoRed-Ansatz sucht in den codierten Werten x_c, y_c und z_c eine Mehrheit und gibt den Gewinner sowie die konstante Sprungsignatur als Ergebnis zurück. Die statischen Signaturen der Replikate B_x, B_y und B_z sowie die Sprungsignatur können bereits zur Übersetzungszeit festgelegt bzw. berechnet werden. Die dynamische Signatur errechnet sich aus den codierten Werten abhängig von der Mehrheit. Die `if`-Abfragen suchen nach der Mehrheit, indem die codierten Werte und ihre statischen Signaturen verglichen werden. Die *apply()*-Funktion addiert im Entscheidungsprozess die dynamisch berechnete Signatur auf das gewählte codierte Ergebnis. Die Decodierung findet später mit der statischen Signatur statt. Unter anderem so können Fehler entdeckt werden. (Algorithmus entnommen aus einer Arbeit von P. Ulbrich [55])

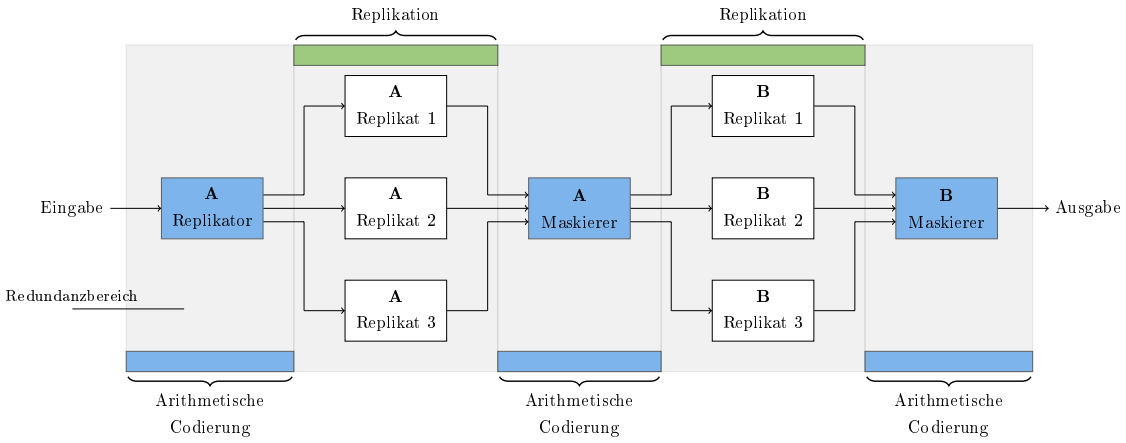


Abbildung 2.5: Schematischer Aufbau des CoRed-Ansatzes. CoRed nutzt sowohl Replikation als auch arithmetische Codierung, um Redundanz in ein System einzubringen. Dazu werden Berechnungen dreifach repliziert. Verbleibende kritische Fehlerstellen (die Eingangsreplikation sowie der Mehrheitsentscheider) werden mithilfe der arithmetischen Codierung codiert. Auf diese Weise können auch an den kritischen Fehlerstellen Fehler erkannt werden. (Abbildung entnommen aus einer Arbeit von P. Ulbrich [55])

CoRed führt eine sogenannte statische Sprungsignatur B_E für den Mehrheitsentscheider ein, um daraus später den Gewinner bestimmen zu können. Abhängig von den korrekten Replikaten unterscheiden sich die Sprünge im Entscheider und damit auch die Sprungsignatur. Die Sprungsignatur errechnet sich aus den zur Übersetzungszeit bekannten statischen Signaturen. Ihre möglichen Werte sind daher ebenfalls zur Übersetzungszeit bekannt. [55]

Für B_E gilt nach P. Ulbrich [55]

$$B_E = \begin{cases} (B_x - B_y) + (B_x - B_z), & \text{falls } x = y = z \\ (B_x - B_y), & \text{falls } x = y \\ (B_x - B_z), & \text{falls } x = z \\ (B_y - B_z), & \text{falls } y = z \\ \text{false}, & \text{sonst.} \end{cases} \tag{2.6}$$

Darüber hinaus führt CoRed eine dynamische Signatur B_{dyn} für den Mehrheitsentscheider ein, um Kontrollflussfehler feststellen zu können. Mögliche Kontrollflussfehler im Mehrheitsentscheider können in Sprungfehler und Sequenzfehler unterschieden werden. **Sprungfehler** bezeichnen das fehlerhafte Ausführen der falschen Verzweigung aufgrund eines Registerfehlers. Unter **Sequenzfehlern** versteht man das fälschliche Ausführen z.B. mehrerer Zweige aufgrund eines Fehlers im Befehlszähler (engl. *Instruction Pointer (IP)*). Die dynamische Signatur muss zur Laufzeit aus den codierten Werten basierend auf den Sprungentscheidungen berechnet werden. Mithilfe dieser Signatur ist dann eine Erkennung

von Kontrollflussfehlern möglich. Bei der Ausführung sind keine Kontrollflussfehler aufgetreten, falls $B_{\text{dyn}} = B_E$ gilt. [55]

Dabei gilt nach P. Ulbrich [55]

$$B_{\text{dyn}} = \begin{cases} (x_c - y_c) + (x_c - z_c), & \text{falls } x = y = z \\ (x_c - y_c), & \text{falls } x = y \\ (x_c - z_c), & \text{falls } x = z \\ (y_c - z_c), & \text{falls } y = z \\ \text{false}, & \text{sonst.} \end{cases} \tag{2.7}$$

Somit verhält sich CoRed, wie eingangs bereits beschrieben, „von außen wie eine codierte Operation“ ([55]). Abhängig davon, welche der Eingaben identisch sind, bildet der Mehrheitsentscheider einen Gewinner. Dieser besteht stets aus der ersten gültigen Variablen sowie der dynamischen Signatur B_{dyn}. [55]

Somit lässt sich der Mehrheitsentscheider, wie in der Arbeit von P. Ulbrich [55], durch folgende Funktion beschreiben:

$$\text{CoRed\_vote}(x_c, y_c, z_c) = (\text{win}_c, B_E), \tag{2.8}$$

wobei

$$\text{win}_c(x_c, y_c, z_c) = \begin{cases} x_c + (x_c - y_c) + (x_c - z_c), & \text{falls } x = y = z \\ x_c + (x_c - y_c), & \text{falls } x = y \\ x_c + (x_c - z_c), & \text{falls } x = z \\ y_c + (y_c - z_c), & \text{falls } y = z \\ \text{false}, & \text{sonst.} \end{cases} \tag{2.9}$$

Dabei ist der erste Teil des Tupels aus CoRed_vote der ermittelte Gewinner, der sich aus der Summe der codierten Variablen und der dynamischen Signatur zusammensetzt. Der zweite Teil des Tupels ist die statische Sprungsignatur. Soll das Ergebnis decodiert werden, wird die im Mehrheitsentscheider ermittelte statische Sprungsignatur vom Ergebnis abgezogen. Diese ist im fehlerfreien Fall äquivalent zur dynamischen Signatur. Dieser Schritt ermöglicht daher die Erkennung von Kontrollflussfehlern. Daraufhin kann das Ergebnis, wie in Abschnitt 2.3.1 beschrieben, decodiert werden. [55]

2.5.1 Beispiel. Angenommen, die Replikate 1 und 3 liefern ein korrektes Ergebnis, Replikat 2 ist fehlerhaft und es existieren keine Fehler im Mehrheitsentscheider Die Nummerierung der Replikate ist dabei angelehnt an 2.5. Replikat 1 liefert den Wert x_c, Replikat 2 den Wert y_c und Replikat 3 den Wert z_c. Damit gilt in diesem Beispiel $x = z$. Nach Gleichung 2.8 greift der dritte Fall und $\text{CoRed\_vote}(x_c, y_c, z_c) = (x_c + (x_c - z_c), (B_x - B_z))$. Im Algorithmus 2.1 wird das erste `if` in Zeile 5 nicht betreten. Stattdessen ist die Bedingung in Zeile 13 erfüllt. Daher gilt $win_c = x_c + (x_c - z_c)$ und $B_E = B_x - B_z$. Für eine

Decodierung muss die Sprungsignatur von win_c subtrahiert werden. Da nach Voraussetzung keine Fehler im Mehrheitsentscheider aufgetreten sind, gilt $B_E = B_{\text{dyn}}$ und somit $x = \frac{((win_c - B_E) - B_x - D)}{A}$ im Falle eines ANBD-codierten Codeworts.

Grundsätzlich ermöglichen sowohl der codierte Mehrheitsentscheider als auch die arithmetische Fehlercodierung lediglich eine Fehlererkennung und keine Fehlerkorrektur. Eine Fehlerbehebung ist im Zweifelsfall lediglich durch ein erneutes Durchführen des Mehrheitsentscheids möglich. Im Falle transienter Hardwarefehler ist es wahrscheinlich, dass eine zweite Ausführung fehlerfrei verläuft. [55]

Dies wird insbesondere im Falle eines Kontrollflussfehlers deutlich. Gilt beispielsweise $x = z$, d. h. y ist fehlerhaft und aufgrund eines Sprungfehlers wird fälschlicherweise $y = z$ angenommen, kann dies durch den Vergleich der Sprungsignatur mit der dynamischen Signatur erkannt werden. Aus dem Vergleich geht allerdings nicht hervor, ob stattdessen $x = z$ gilt oder alle Werte fehlerhaft sind.

Prinzipiell ist unter Einsatz von CoRed aber eine vollständige Fehlererkennung im Sinne des Fehlermodells möglich. Das Auftreten von Berechnungsfehlern ist in diesem Zusammenhang nicht möglich, da keine codierten Rechenoperationen verwendet werden. Operandenfehler kommen durchaus vor, werden allerdings mithilfe der arithmetischen Codierung erkannt. Operatorfehler wie auch Kontrollflussfehler sowie weitere Fehler, die zum Versagen des Mehrheitsentscheiders führen, werden aufgrund des Abgleichs der statischen Sprungsignatur mit der dynamischen Signatur bemerkt. [55]

2.6 Fehlerinjektion

Das Einbringen von Redundanzmaßnahmen sorgt prinzipiell für eine größere Angriffsfläche des Systems, da zusätzliche Funktionalitäten in das System eingebracht werden müssen, die für die Erfüllung der eigentlich spezifizierten Funktionalität nicht benötigt werden. Deswegen ist nicht garantiert, dass die Zuverlässigkeit des Systems durch Redundanz auch tatsächlich ansteigt. Daher ist es immer notwendig, Fehlertoleranzmaßnahmen hinsichtlich ihrer Funktionalität zu überprüfen. [15]

Dabei entsteht allerdings die Problematik, dass Fehlertoleranzmaßnahmen nicht Teil der eigentlichen Systemfunktionalität sind. Daher müssen zum Testen von Redundanzmaßnahmen explizit Fehler in das System eingebracht werden. Die betrachtete Fehlerklasse der transienten Hardwarefehler tritt jedoch nicht in einer solchen Menge auf, als dass sie für groß angelegte Testfälle ausreichen würde [51]. Daher sind spezielle Testmethoden notwendig, welche gezielt Fehler im System erzeugen, um die Zuverlässigkeit der Redundanz zu überprüfen. Dazu wird Fehlerinjektion eingesetzt. **Fehlerinjektion** injiziert auf reproduzierbare Art und Weise Fehler in ein System. So kann das Systemverhalten im Feh-

lerfall hinsichtlich seiner Fehlererkennung, seiner Fehlerausbreitung und seiner allgemeinen Zuverlässigkeit bewertet werden. [49]

Im Allgemeinen existieren bei der Fehlerinjektion hardwarebasierte und softwarebasierte Ansätze. Hinzu kommen außerdem simulationsbasierte Verfahren. Unter **hardwarebasierter Fehlerinjektion** versteht man das direkte Einbringen von Fehlern in bestimmte Testhardware. Wird **softwarebasierte Fehlerinjektion** angewendet, werden z. B. für Hardwarefehler typische Fehlermuster auf Software-Ebene in das System eingebracht. [62]

Bei **simulationsbasierten Ansätzen** wird ein System-Modell in einem Simulator simuliert. In dieser Simulation werden Fehler injiziert. [62] Ein Beispiel für simulationsbasierte Fehlerinjektionswerkzeuge ist FAIL* (Fault Injection Leveraged) von H. Schirmeier [49]. Auf Alternativen zur simulationsbasierten Fehlerinjektion wie softwarebasierte oder hardwarebasierte Fehlerinjektion wird an dieser Stelle nicht weiter eingegangen, da dies den Umfang der Arbeit überschreiten würde.

2.6.1 Modell und Begriffe der Fehlerinjektion

Das FARM-Modell beschreibt die Elemente, die in einer Fehlerinjektionskampagne verwendet werden. Dabei handelt es sich um ein Modell, welches die Voraussetzungen einer Fehlerinjektion beschreibt. Es geht nicht auf das Werkzeug oder die Umgebung ein, mit welchen die Fehlerinjektionskampagne durchgeführt werden soll. [21] Es wurde von Arlat, Crouzet und Laprie [2, 3] Anfang der neunziger Jahre präsentiert. Es ist ein gängiges Modell der Fehlerinjektion.

- **F**ault (Fehlerraum) $\mathbb{F}$ - beschreibt die Fehler, die in das System eingebracht werden.
- **A**ctivation (Aktivierungsmuster) $\mathbb{A}$ - beschreibt, wie die Fehler aus $\mathbb{F}$ verursacht werden können.
- **R**eadout (Messergebnisse) $\mathbb{R}$ - beschreibt, wie Messdaten bzgl. der Fehlerfolge erhoben werden können.
- **M**easure (Bewertung der Messergebnisse) $\mathbb{M}$ - beschreibt Metriken, mit denen eine Bewertung der Messdaten erfolgen kann.

Um den Fehlerraum $\mathbb{F}$ bestimmen zu können, müssen Annahmen hinsichtlich des Fehlermodells getroffen werden. Ziel dabei ist, die Realität möglichst genau abzubilden. [21]

Das Testen mittels simulationsbasierter Fehlerinjektion folgt einem festen Aufbau. Zu Beginn eines Testdurchlaufs wird stets ein Referenzlauf (engl. *Golden Run*) durchgeführt, bei welchem keine Fehler injiziert werden. Während des Referenzlaufs wird insbesondere der Fehlerraum reduziert, indem gewisse Injektionen ausgelassen werden (vgl. Abschnitt 2.6.2).

Durch die Reduktion der durchzuführenden Injektionen wird die Ausführungszeit der Kampagnen erheblich verbessert. Darüber hinaus können u.a. Ausgaben des Programms für spätere Vergleiche aufgezeichnet. [21]

Bei einem **Experiment** wird zu einem bestimmten Zeitpunkt an einer bestimmten Stelle ein Fehler $f \in \mathbb{F}$ injiziert [3]. Die Art der Injektion wird durch das Aktivierungsmuster $\mathbb{A}$ beschrieben [3].

Das Ergebnis der Injektion wird aufgezeichnet und durch das Fehlerinjektionswerkzeug interpretiert. Dieser Schritt wird durch die Menge $\mathbb{R}$ im FARM-Modell abgedeckt [3]. Damit kann ein Experiment durch das Tripel (f, a, r) beschrieben werden mit $f \in \mathbb{F}$, $a \in \mathbb{A}$, $r \in \mathbb{R}$ [21].

Mögliche Ergebnisse der Interpretation sind u.a. erkannte Fehler, unentdeckte Datenfehler, Timeouts sowie Injektionen ohne Effekt. [21] Werkzeuge wie FAIL* erlauben darüber hinaus das Erstellen eigener, feinerer Kategorien [49]. Diese Kategorien werden durch die Menge $\mathbb{M}$ beschrieben. Mehrere Experimente werden zu einer **Kampagne** zusammengefasst. Eine Kampagne beschreibt damit einen Testfall. [21]

Die Experimente einer Kampagne unterscheiden sich also in der Frage, wann bzw. wo Fehler injiziert werden [21]. Der Ort der Injektion sowie der Zeitpunkt in der Programmausführung spannen einen **Fehlerraum** (engl. *fault space*) auf, welcher die Anzahl der möglichen Experimente beschreibt. Dieser Fehlerraum kann abhängig von der Komplexität und der Dauer der Programmausführung immens groß werden, was direkte Auswirkungen auf die Größe und Laufzeit der Kampagne hat. Ziel ist eine möglichst große Abdeckung des Fehlerraums (engl. *Fault Space Coverage*), um eine möglichst große Anzahl von Testfällen und Testszenarien erreichen zu können. Dies ist aber aufgrund der schnell steigenden Größe des Fehlerraums oft kaum umsetzbar. [21]

Ein schematischer Ausschnitt aus einem Fehlerraum für Einbit-Fehler ist in Abbildung 2.6 dargestellt. Dabei ist jeder Punkt ein injizierbarer Fehler. Der Prozess, in den der Fehler injiziert werden soll, wird bis zum entsprechenden Zeitpunkt ausgeführt, das Bit wird gekippt und der Prozess läuft weiter.

2.6.2 Fault Injection Leveraged (FAIL*)

FAIL* ist ein Fehlerinjektionswerkzeug, welches von H. Schirmeier [49] entwickelt wurde. Es bietet die Möglichkeit der simulationsbasierten Fehlerinjektion auf Basis verscheidener Simulator-Backends (Bochs von K. P. Lawton [30], gem5 von Binkert u.a. [9] und QEMU von F. Bellard [7]) [49]. FAIL* bietet außerdem die Möglichkeit der hybriden Fehlerinjektion, die in einer Arbeit von H. Schirmeier [49] beschrieben wird, auf die an dieser Stelle allerdings nicht weiter eingegangen werden soll.

FAIL* ist also ein Framework zur Fehlerinjektion, das es ermöglicht, wiederverwendbare Fehlerinjektionsexperimente zu entwickeln. Insbesondere stellen die Autoren eine Reihe

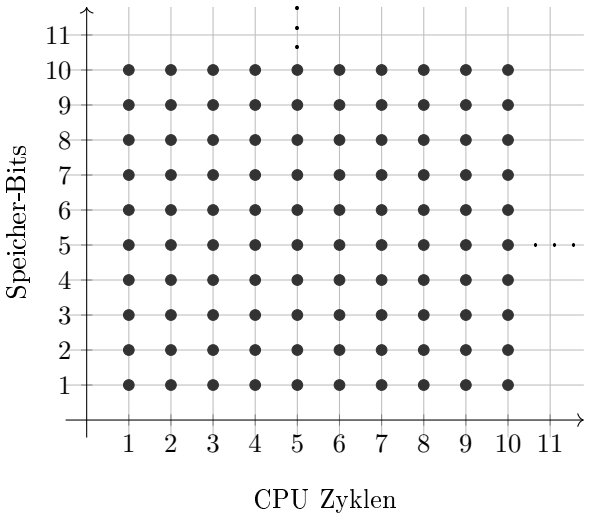


Abbildung 2.6: Fehlerraum für Einzelbit-Fehler. Jeder Punkt im Koordinatensystem repräsentiert einen injizierbaren Einzelbit-Fehler bestehend aus einem Zeitpunkt und einem Ort. Der Prozess läuft bis zum Zeitpunkt, zu dem injiziert werden soll. Ein Einzelbit-Fehler wird eingebracht und der Prozess wird weiter ausgeführt. Mithilfe einer dritten Dimension könnten Mehrbit-Fehler im Fehlerraum dargestellt werden. (Abbildung entnommen aus einer Arbeit von H. Schirmeier [49])

vordefinierter Komponenten zur Verfügung, die in das eigene Projekt eingebunden werden können und die Entwicklung von eigenen Kampagnen vereinfacht. [49] Es ist außerdem plattformunabhängig und frei verfügbar[3]. [50]

Aus technischer Sicht besteht FAIL* aus einem Server, welcher die Kampagne verwaltet, sowie einer Menge von Clients, auf denen die Einzelexperimente durchgeführt werden. Da die einzelnen Experimente einer Kampagne auf diese Weise nicht nur fachlich, sondern auch technisch unabhängig voneinander sind, bietet FAIL* Möglichkeiten zur Parallelisierung. Das reicht von einer Parallelisierung auf mehreren Kernen oder Prozessoren bis hin zur Parallelisierung auf mehreren Rechnern. Die Rechner müssen dabei nicht vor Ort stehen. Stattdessen können die Experimente über ein Netzwerk auf mehrere Rechner verteilt werden. Dies kann die Laufzeit einer Kampagne bereits erheblich verbessern. [49]

Um die Anzahl der Experimente, die durchgeführt werden müssen, und damit auch die Laufzeit zu reduzieren, bietet FAIL* außerdem die Möglichkeit der **Reduktion des Fehlerraums** (engl. *Fault Space Pruning*). Dabei werden insbesondere Injektionen ohne Effekt und idempotente Injektionen ermittelt. Diese werden bei der Kampagnen-Durchführung ausgelassen. Dadurch wird der Fehlerraum erheblich verkleinert. Diese werden in der FAIL*-Datenbank als Injektionsziel hinterlegt. In dieser Datenbank werden außerdem die

[3] FAIL* ist über GitHub verfügbar: https://github.com/danceos/fail

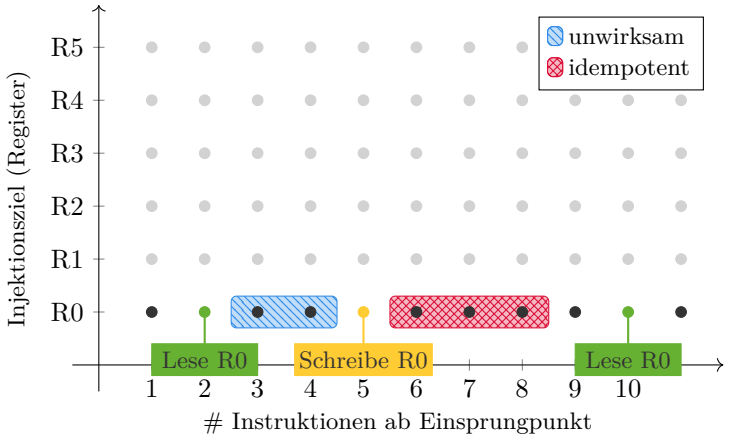


Abbildung 2.7: Beispiel für eine Fehlerraumreduktion. Unwirksame und idempotente Fehlerinjektionen können bei der Fehlerinjektion ausgelassen bzw. zusammengefasst werden, um den Fehlerraum zu verkleinern. Unwirksame Fehler sind z. B. gutartige Defekte in einem Register, die überschrieben werden, bevor sie von der Software ausgelesen und weiter verwendet werden. Die blau gestreift hinterlegten Injektionen sind also nicht notwendig. Idempotente Fehlerinjektionen sind solche, die dieselben Auswirkungen haben. So ist beispielsweise unwichtig, ob in dieser Abbildung der Fehler zum Zeitpunkt der sechsten, siebten, achten oder neunten Instruktion injiziert wird, da das Register erst mit der zehnten Instruktion ausgelesen wird. Die rot schraffierten Injektionen können somit ausgelassen werden. (Abbildung entnommen aus einer Arbeit von P. Ulbrich [56])

Ergebnisse der Experimente gespeichert. Die zu speichernden Informationen können dabei frei erweitert werden. [49]

Beispiele für idempotente und unwirksame Injektionen sind in Abbildung 2.7 dargestellt. Wird beispielsweise ein Fehler in ein Register injiziert, dessen Wert danach wieder überschrieben und vorher nicht ausgelesen wird, kann der injizierte Fehler keinen Effekt auf das Programm haben. Diese Fehler sind also gutartige Defekte und können daher von vornherein bei der Injektion ausgelassen werden. In der Abbildung 2.7 sind dies die blau gestreift hinterlegten Injektionen. Diese werden als **unwirksam** bezeichnet. Unter **idempotenten** Injektionen versteht man gleichartige Injektionen. Beispielsweise ist es unwichtig, zu welchem Zeitpunkt FAIL* einen Fehler in ein Register injiziert, solange auf dieses Register noch nicht zugegriffen wurde. Alle Instruktionen vor dem Registerzugriff sind idempotent, weil sie dieselben Auswirkungen hätten. In der Abbildung 2.7 handelt es sich um die rot schraffierten Injektionen. Diese können bei der Fehlerinjektion folglich ausgelassen werden. Dasselbe gilt für die unwirksamen Injektionen. [55]

Kapitel 3

Problemanalyse

Fehlersimulationen des CoRed-Mehrheitsentscheiders im Zuge dieser Arbeit zeigen, dass noch wenige, unerwartete SDCs auftreten. Eine mögliche Erklärung hierfür kann eine ungünstige Parametrisierung der Codierung sein. Wie in Abschnitt 1.2 beschrieben, soll daher die Auswahl der Signaturen untersucht werden. So kann potenziell durch eine geschicktere Wahl der Signaturen das Auftreten unentdeckter Datenfehler teilweise oder ganz verhindert werden. Aus diesem Grund wird im Folgenden lediglich die ANB-Codierung anstelle der ANBD-Codierung verwendet. Hinzu kommt, dass die Zeitstempel für eine zusätzliche Komplexität sorgen würden.

Dieses Kapitel soll mögliche Probleme und Lösungsansätze im Kontext der Parametrisierung aufzeigen. Zu diesem Zweck muss allerdings zunächst das dieser Arbeit zugrundeliegende Fehlermodell, Systemmodell und Anwendungsmodell definiert werden. Daraufhin wird der Stand der Forschung im Hinblick auf die Parametrisierung der ANB-Codierung zusammengefasst. Anhand eines Überblicks über die bestehenden Arbeiten können dann Probleme identifiziert werden. Diese Probleme werden anhand ihrer Ursache kategorisiert und näher erläutert. Daraufhin werden im folgenden Kapitel Lösungsansätze für diese Probleme erarbeitet, die später bewertet werden.

Es seien wie zuvor x, y und z uncodierte Werte, x_c, y_c und z_c ihre Codewörter, B_x, B_y und B_z ihre variablenspezifischen Signaturen und A der Schlüssel. Die Code-Distanz ist, wie in Abschnitt 2.3.3 definiert, die minimale Hamming-Distanz zwischen zwei verschiedenen Codewörtern des Codes C.

3.1 Fehlermodell, Systemmodell und Anwendungsmodell

Maßnahmen zur Fehlertoleranz können nicht unabhängig von äußeren Anforderungen und Gegebenheiten umgesetzt werden. Sie sind immer abhängig von der Art der Fehler, die toleriert werden sollen, sowie von den Systemanforderungen und Systemeigenschaften. Im Folgenden werden daher die Annahmen und Einschränkungen des Fehlermodells und des

Systemmodells dargestellt, die für diese Arbeit gelten sollen. Dabei entsprechen die in dieser Arbeit getroffenen Annahmen denen von P. Ulbrich [55], da die Ziele dieser Arbeit am Beispiel des dort vorgestellten CoRed-Ansatzes untersucht werden sollen.

3.1.1 Fehlermodell

Um Fehlertoleranz zu erreichen, müssen im Allgemeinen bestimmte Eigenschaften erfüllt sein, damit Fehler erkannt oder korrekte Werte wiederhergestellt werden können. Bei der Codierung dürfen z. B. nicht zu viele Bits fehlerhaft sein. Sind zu viele Bits fehlerhaft, wird die Codierung überbelastet und eine Fehlererkennung ist nicht mehr möglich.

Betrachtet man die Replikation, muss eine bestimmte Anzahl fehlerfreier Komponenten vorhanden sein, um einen Fehler erkennen bzw. einen korrekten Wert wiederherstellen zu können. [16] Die Anzahl der erforderlichen fehlerfreien Komponenten hängt von der Art und Anzahl der zu tolerierenden Fehler ab [16]. Dreifachredundanz z. B. kann typischerweise ein fehlerhaftes Replikat kompensieren [15]. In diesem Fall wird daher von einer Einzelfehler-Hypothese ausgegangen. Da der CoRed-Ansatz mit softwarebasierter Dreifachreplikation arbeitet, kann dieser einen Einzelfehler pro Replikationsebene bzw. pro Bereich tolerieren.

In dieser Arbeit werden - wie auch im CoRed-Ansatz - Einzelbit-Fehler betrachtet, da diese das Verhalten und Auftreten transienter Hardwarefehler widerspiegeln [55]. So argumentiert R. Baumann [6], dass der Einsatz von fehlerkorrigierenden Codes ausreicht, welche einen Einbit-Fehler korrigieren oder einen Zweibit-Fehler erkennen können, um eine erhebliche Reduktion der Fehlerrate erzielen zu können. Unterstützt wird diese Aussage durch praktische Untersuchungen von Maiz u. a. [33] und T. J. O'Gorman [40], durch die Feldstudie von Nightingale, Douceur und Orgovan [39] und die Feldstudie von Schroeder, Pinheiro und Weber [51]. Alle diese Studien konnten feststellen, das es sich bei einem sehr großen Teil der transienten Fehler um Einbit-Fehler handelt.

Somit ist die Einzelbit-Einzelfehler-Annahme für die einzelnen Replikate im CoRed-Ansatz, wie von P. Ulbrich [55] beschrieben, auch für diese Arbeit eine sinnvolle Grundannahme. Aus dieser Annahme entstehen außerdem Vorteile für die Fehlerinjektionsexperimente, welche in dieser Arbeit durchgeführt werden sollen. Die Reduktion des Fehlerraums auf Einzelbit-Fehler verkleinert den Fehlerraum und damit auch die Ausführungszeit der Fehlerinjektionsexperimente erheblich.

Weiterhin wird davon ausgegangen, dass keine byzantinischen Fehler auftreten. **Byzantinische Fehler** sind Fehler, bei denen eine Systemkomponente eine inkonsistente Sicht auf ihren Systemzustand an verschiedene andere Komponenten sendet. Das heißt, dass willkürlich richtige, falsche oder gar keine Nachrichten an andere Komponenten gesendet werden. Lamport, Shostak und Pease [29] erläutern dies anschaulich am Problem der byzantinischen Generäle. [29] Ein solches Verhalten kann z. B. in verteilten Systemen mit asynchroner Kommunikation im Fehlerfall auftreten [29, 59]. Die Konsensbildung ist in

diesem Fall aufwändig und erfordert spezielle Algorithmen [59]. Da der CoRed-Ansatz in einem Echtzeitsystem mit sicheren, synchronen Kommunikationsmedien eingesetzt wird, in dem eine konsistente Sicht auf die Zustände in den einzelnen Komponenten möglich bzw. realisierbar ist, kann für diese Arbeit die Betrachtung byzantinischer Fehler ausgeschlossen werden.

Neben der Einzelbit-Einzelfehler-Annahme sollen Fehler grundsätzlich in die drei von P. Forin [19] vorgestellten Fehlerklassen eingeordnet werden können. Dabei handelt es sich um Berechnungsfehler, Operatorfehler und Operandenfehler. Berechnungsfehler sind Fehler, bei denen eine Berechnung zu einem falschen Ergebnis führt [19].

3.1.2 Systemmodell und Anwendungsmodell

Aus der Fehlerhypothese und dem betrachteten CoRed-Ansatz mit seinen Fehlertoleranzmaßnahmen entstehen neben dem Fehlermodell außerdem Annahmen und Anforderungen an das Systemmodell. Das System, auf dem die Fehlertoleranz implementiert wird, muss zunächst eine strikte räumliche und zeitliche Fehlereingrenzung gewährleisten. Ein Fehler darf sich nicht von einem Replikat auf ein anderes ausbreiten. In diesem Fall wären mehrere Replikate fehlerhaft und die angenommene Fehlerhypothese wäre verletzt. Eine Fehlererkennung oder Fehlerkorrektur ist dann nicht mehr möglich. Dies kann durch räumliche Isolation vermieden werden. Zur Realisierung der räumlichen Fehlerisolation wird eine Speicherschutzeinheit (engl. *MPU (Memory Protection Unit)*) bzw. in einigen Experimenten eine Speicherschutzsimulation eingesetzt. So können z.B. die Adressräume der einzelnen Replikate voneinander isoliert werden. [55]

Um die zeitliche Isolation des auszuführenden Programms zu gewährleisten, wird ein Echtzeitbetriebssystem eingesetzt. Das Echtzeitbetriebssystem sorgt z.B. dafür, dass die Kapazitäten des Systems auf die Prozesse verteilt werden. So ist es nicht möglich, dass ein Prozess die Kapazitäten für sich alleine beansprucht. [55]

Das sicherheitskritische Programm muss über definierte Ein- und Ausgabeparameter verfügen und unterbrechungsfrei sein (engl. *run to completion*). Das Programm kann in die drei Phasen „Eingabe“, „Datenverarbeitung“ und „Ausgabe“ eingeteilt werden, wie es für sicherheitskritische Anwendungen häufig der Fall ist. [57] CoRed greift auf die Ein- und Ausgabe-Schnittstellen des sicherheitskritischen Programms zu und umgibt es mit Redundanzmaßnahmen. So kann der Redundanzansatz für viele sicherheitskritische Systeme eingesetzt werden.

Darüber hinaus muss sich das Programm deterministisch verhalten. Dadurch ist es möglich, Aussagen darüber zu treffen, ob sich das Programm auch im Fehlerfall korrekt verhält bzw. eine korrekte Fehlerbehandlung durchgeführt wird. Ein exakter Mehrheitsentscheider kann darüber hinaus nur dann das richtige Ergebnis ermitteln, wenn sich die Replikate replikdeterministisch verhalten. Das bedeutet, dass Replikate, die fehlerfrei arbeiten und

die gleiche Eingabe zu etwa demselben Zeitpunkt erhalten haben, auch die gleiche Ausgabe liefern [55].

3.2 Stand der Forschung

Die ANB-Codierung wie in der Folge auch die CoRed-Implementierung erfordern die Einhaltung einiger Regeln bei der Parametrisierung. Aus Arbeiten von U. Schiffel [47], P. Ulbrich [55] und Hoffmann u. a. [24] wird ersichtlich, dass Primzahlen, entgegen der mathematischen Erwartung, nicht zwangsläufig die beste Wahl für den Schlüssel A in der ANB-Codierung sind. Ganz im Gegenteil identifiziert P. Ulbrich [55] in seiner Arbeit fünf sogenannte „Super As“, welche sich besonders gut als Codierungsschlüssel im 32-Bit Wertebereich aufgrund besonders großer Hamming-Distanzen eignen. Keiner dieser Schlüssel ist eine Primzahl. Sie weisen jeweils eine Code-Distanz von sechs auf und können daher fünf Bit-Fehler erkennen. Im 8-Bit Bereich errechnet P. Ulbrich [55] eine beste Code-Distanz von vier für $A = 185$ und $A = 233$.

Wichtig anzumerken ist, dass AN-Codes nicht-systematische Codes sind. Das bedeutet insbesondere, dass die Code-Distanz in der Binärdarstellung nicht notwendigerweise abhängig von den durch die Codierungsvorschrift eingebrachten k Prüfbits ist, die den Schlüssel A repräsentieren. Vielmehr ist die Code-Distanz auch abhängig von der binären Repräsentation der Codierung $A \cdot v$. Das bedeutet, dass größere Schlüssel nicht notwendigerweise auch eine größere Code-Distanz erzielen. [24]

Bei der Wahl des Schlüssels sollte darüber hinaus eine möglichst geringe Restfehlerwahrscheinlichkeit gegeben sein, weshalb sich die Wahl größerer Werte unter Berücksichtigung der Code-Distanz empfiehlt. Bei der Wahl der Schlüssel in Abhängigkeit von der Restfehlerwahrscheinlichkeit ergibt sich allerdings eine Einschränkung, die in der Praxis zu beobachten ist: Diese Einschränkung betrifft insbesondere das Verhalten der Restfehlerwahrscheinlichkeit bei einer Überbelastung des Codes. Denn Hoffmann u. a. [24] stellen fest, dass sich einige Codes abhängig von der Wahl des Schlüssels entsprechend ihrer Restfehlerwahrscheinlichkeit verhalten, wohingegen andere deutlich höhere Fehlerzahlen verursachen. Daher unterteilen sie die Schlüssel in gutartig und bösartig.

Es finden sich in der Literatur weniger genaue Aussagen zur Wahl der Signaturen B als zur Wahl der Schlüssel [47, 55]. Hier lässt sich feststellen, dass die Wahl von $B = 0$ nicht sinnvoll ist, da auf diese Weise die ANB-Codierung zur AN-Codierung degeneriert, da $v_c = A \cdot v + 0 = A \cdot v$. Das hat eine schlechtere Fehlererkennung zur Folge. Damit gilt für die Wahl der Signaturen $B > 0$. [47]

Darüber hinaus rät U. Schiffel [47], alle Signaturen immer kleiner als den Schlüssel zu wählen sowie für eine ausgeglichene Verteilung der Signaturen zu sorgen. Wäre $B > A$, so wäre es möglich, dass in ungünstigen Konstellationen eine Gleichheit zu anderen gültigen Codewörtern mit anderen variablenspezifischen Signaturen entsteht. Durch die

Beschränkung auf $0 < B < A$ kann dies verhindert werden. Außerdem würde ein $B > A$ spätestens bei der Decodierung und bei der Überprüfung Probleme bereiten, da dort modulo A gerechnet wird

Des Weiteren empfiehlt sie, Signaturen im Verlauf längerer Berechnungen zu aktualisieren, um zufällige Gleichheiten der Signaturen während der Berechnungen zu verhindern.

P. Ulbrich [55] schränkt die Wahl der Signaturen etwas weiter ein. Er fordert, dass für die statischen Signaturen B_x, B_y und B_z von CoRed $B_x > B_y > B_z$ gilt, um negative dynamische Signaturen zu verhindern. Außerdem wählt er die Signaturen so, dass sie dieselbe Hamming-Distanz wie das gewählte A haben.

Die Implementierung von CoRed erfordert, dass die Distanzen zwischen den Signaturen unterschiedlich sind, d. h. es gilt $\forall i, j, k : |B_i - B_j| = |B_k - B_j| \implies i = k$.

Somit ergibt sich der folgende Regelsatz für die Parametrisierung der ANB-Codierung im CoRed-Ansatz:

R1 Die Restfehlerwahrscheinlichkeit sollte möglichst klein sein.

R2 Die Code-Distanz von A sollte möglichst groß sein.

R3 Es sollte $0 < B < A$ für alle Signaturen gelten.

R4 Die Signaturen sollten sortiert sein, sodass $B_x > B_y > B_z$ gilt.

R5 Die Distanzen zwischen allen Signaturen müssen unterschiedlich sein: $\forall i, j, k : |B_i - B_j| = |B_k - B_j| \implies i = k$.

3.3 Problemanalyse

Um die Parametrisierung der ANB-Codierung besser verstehen und mögliche Probleme ausmachen zu können, ist eine Analyse der CoRed-Implementierung sowie der Ergebnisse von Hoffmann u. a. [24] erforderlich. Hoffmann u. a. untersuchen typische Fallstricke bei der Anwendung arithmetischer Codierung am Beispiel der CoRed-Implementierung. Sie identifizieren dabei insgesamt 3 Fallstricke [24]:

F1 Fallstrick 1: Darstellung von Codes im Binärsystem (engl. *„Pitfall 1: Mapping Code to Binary“*)

F2 Fallstrick 2: Zustand zwischen Instruktionen (engl. *„Pitfall 2: Inter-Instruction State“*)

F3 Fallstrick 3: undefinierte Ausführungsumgebung (engl. *„Pitfall 3: Undefined Execution Environment“*)

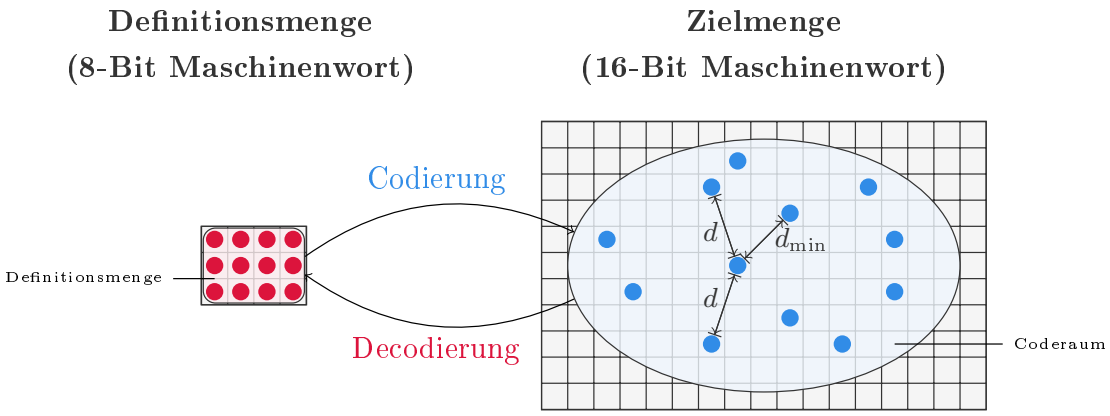


Abbildung 3.1: Unvollständige Ausnutzung des Maschinenwortes durch die Codierung. Die Codierung nutzt die Größe des verwendeten Maschinenwortes in der Binärdarstellung nicht vollständig aus. Dadurch kann es zu vermeintlich gültigen Codewörter kommen, die außerhalb des Coderaums aber innerhalb der Größe des Maschinenwortes liegen. In diesem Fall signalisiert die Hardware keinen Fehler und das Codewort ist rechnerisch korrekt, obwohl es zu einem Fehler gekommen und das Codewort semantisch falsch ist. Ohne eine weitere Überprüfung durch die Implementierung bleibt der Fehler unentdeckt. (Abbildung entnommen aus einer Arbeit von P. Ulbrich [55])

3.3.1 Fallstrick 1: Darstellung von Codes im Binärsystem

Im Zuge des Fallstricks **F1** stellen Hoffmann u. a. [24] fest, dass die Binärdarstellung des Codes im Fehlerfall durchaus zu Problemen führen kann.

Für die Darstellung der Codewörter in Binärform kann beispielsweise ein 16-Bit-Maschinenwort gewählt werden. Ein Teil n der 16-Bit ist für die Nutzdaten vorgesehen; der andere Teil k bleibt für die Prüfdaten zur Fehlererkennung. Nutzt man die verbleibenden k Bits vollständig aus und wählt den Schlüssel $A = 2^k$, so degeneriert die arithmetische Codierung zu einer bitweisen Verschiebung des zu codierenden Wertes. Die kleinsten Bits sind damit schlichtweg nutzlos. Daher wählt man ein $A < 2^k$. Dadurch werden allerdings nicht alle zur Verfügung stehenden Bits des Maschinenwortes ausgenutzt (vgl. Abbildung 3.1). [24]

Da die Prüfbits und die Datenbits bei der arithmetischen Codierung nicht streng voneinander getrennt sind (nicht-systematischer Code), sind auch die Lücken, die durch die nicht vollständige Ausnutzung entstehen, nicht systematisch. Bit-Fehler in den Codewörtern können so zu vermeintlich gültigen Codewörtern führen, die durch die Binärdarstellung des Coderaumes eigentlich nicht abgedeckt sind. Diese vermeintlich gültigen Codewörter sind allerdings immer größer als das Maximum der Nutzdaten. Diese Fehlermuster werden durch die Codierungstheorie nicht abgedeckt und bleiben daher unerkannt. [24]

Algorithmus 3.1: *decode()*-Funktion aus der CoRed-Implementierung

```
1 function decode(v_c, A, B)
2   if (v_c > v_c,max or (v_c mod A) ≠ B) then
3     SIGNAL_DUE()
4   end if
5   return (v_c mod A) ≠ B
6 end function
```

Algorithmus 3.1: *decode()*-Funktion aus der CoRed-Implementierung. Die *decode()*-Funktion bekommt einen codierten Wert v_c, den Schlüssel A und die variablenspezifische Signatur B als Parameter übergeben. Sie übernimmt eine Fehlerüberprüfung (Zeile 2) und gibt den decodierten Wert zurück. (Algorithmus entnommen aus einer Arbeit von Hoffmann u. a. [24])

Daher erweitern Hoffmann u. a. die Decodierungsfunktion um weitere Prüfmechanismen (vgl. Algorithmus 3.1, Zeile 2). Die *decode()*-Funktion bekommt einen codierten Wert v_c, den Schlüssel A und die variablenspezifische Signatur B als Parameter übergeben. Sie führt eine Fehlerüberprüfung durch (Zeile 2). Der erste Teil der Abfrage löst das zuvor beschriebene Problem der unsystematischen Lücken. Der zweite Teil prüft, ob es sich bei v_c um ein valides Codewort handelt. Die Funktion gibt den decodierten Wert v zurück Die Variable $v_{c,\max}$ in Algorithmus 3.1 wird in der Arbeit von Hoffmann u. a. [24] nicht definiert.

3.3.2 Fallstrick 2: Zustand zwischen Instruktionen

Hoffmann u. a. [24] stellen in ihrer Arbeit fest, dass Fehler, die zwischen zwei stark voneinander abhängigen Instruktionen auftreten, ggf. besonders schwere Auswirkungen haben können. Sie verdeutlichen dies anhand eines Beispiels. In den meisten Befehlssatzarchitekturen (engl. *Instruction Set Architecture (ISA)*), wie der verwendeten IA32-Architektur, existieren keine kombinierten Vergleichs- und Sprunginstruktionen. Für die Implementierung werden stattdessen die separaten Instruktionen für einen Vergleich (z. B. `cmp`) und einen Sprung (z. B. `jmp`) verwendet. [24] Die Vergleichsinstruktion speichert ihr Ergebnis im Nullbit (engl. *zero flag*) des Statusregisters. Die Sprunginstruktion entscheidet auf Basis des Nullbits, welcher Kontrollfluss ausgeführt werden soll.

Trifft ein transienter Fehler nun das Nullbit zwischen der Vergleichs- und der Sprunginstruktion, kann dies schwere Folgen haben, da es sich bei dem Nullbit um ein einzelnes, nicht redundantes Bit handelt. Der falsche Kontrollfluss wird ausgeführt und die implementierte Redundanzmaßnahme muss eine Lösung finden, solche Fehler zu erkennen. [24]

Das beschriebene Verhalten wird jedoch nur zu einem Problem, falls eines der Replikate fehlerhaft ist. Sind alle Replikate korrekt, kann der codierte Mehrheitsentscheider im

schlimmsten Fall signalisieren, dass aufgrund des Kontrollflussfehlers keine Entscheidung möglich ist (vgl. Algorithmus 2.1, Zeile 21 in Kapitel 2.5). Wiederherstellungsmaßnahmen, wie z. B. das erneute Durchführen des Entscheidungsprozesses, sind notwendig. Alle weiteren Kontrollflussfehler würden, falls alle Replikate fehlerfrei sind, lediglich dazu führen, dass fälschlicherweise von einer kleineren Konsensmenge ausgegangen wird. Solange keine weiteren Fehler auftreten, würde dennoch ein korrektes Ergebnis bestimmt werden.

Kritisch wird der Fehler erst, wenn eines der Replikate fehlerhaft ist und aufgrund eines Kontrollflussfehlers von einer falschen Konsensmenge ausgegangen wird. In diesem Fall weicht die Berechnung der dynamischen Signatur nach Hoffmann u. a. [24] um ein Vielfaches von A von ihrem eigentlichen, korrekten Wert ab: Für $v^i, v^j \in \{x, y, z\}$ und $i \neq j$ seien v_c^i bzw. v_c^j ihre codierten Formen und B_i, B_j ihre Signaturen mit $B_i > B_j$, da $B_x > B_y > B_z$ (vgl. **R4**). Dann errechnet sich die dynamische Signatur durch

$$\begin{aligned} v_c^i - v_c^j &= (A \cdot v^i + B_i) - (A \cdot v^j + B_j) \\ &= A \cdot v^i + B_i - A \cdot v^j - B_j \\ &= A \cdot (v^i - v^j) + B_i - B_j. \end{aligned}$$

Falls $v^i = v^j$, dann

$$\begin{aligned} A \cdot (v^i - v^j) + B_i - B_j &= B_i - B_j \\ &< A, \end{aligned}$$

weil $0 < B_j < B_i < A$, für alle $i \in x, y, z$ (vgl. **R3**).

Falls $v^i \neq v^j$, dann

$$\begin{aligned} A \cdot (v^i - v^j) + B_i - B_j &\geq A + B_i - B_j \\ &\geq A, \end{aligned}$$

da $B_i > B_j$ und $v^i \neq v^j$. Bei der Betrachtung dieser Ungleichung fällt allerdings auf, dass es einen Fall ($v^i < v^j$) gibt, für die sie nicht gilt. Da dieser Randfall erst zum Ende der Bearbeitungszeit dieser Arbeit aufgefallen ist, wird dieser erst in Abschnitt 7.2.1 diskutiert und in seiner Bedeutung für diese Arbeit eingeordnet.

Hoffmann u. a. [24] passen die *apply()*-Funktion von CoRed an, indem sie eine zusätzliche Abfrage ergänzen (vgl. Algorithmus 3.2, Zeile 2). Die ergänzte Abfrage betrachtet die dynamische Signatur B_{dyn} des CoRed-Mehrheitsentscheiders. Die dynamische Signatur wird zur Laufzeit im Mehrheitsentscheider aus den Signaturen der fehlerfreien Codewörter berechnet (vgl. Abschnitt 2.5) und gibt Auskunft darüber, ob ein Kontrollflussfehler aufgetreten ist. Ist B_{dyn} betragsmäßig größer oder gleich A, wurde also eine mathematisch gültige, aber semantisch falsche dynamische Signatur errechnet.

Algorithmus 3.2: *apply()*-Funktion aus der CoRed-Implementierung

```
1 function apply(v_c, A, B_dyn)
2   if (|B_dyn| ≥ A) then
3     SIGNAL_DUE()
4   end if
5   return v_c + B_dyn
6 end function
```

Algorithmus 3.2: *apply()*-Funktion aus der CoRed-Implementierung. Die *apply()*-Funktion erhält den codierten Wert v_c, den Schlüssel A und die dynamische Signatur aus dem CoRed-Mehrheitsentscheider als Parameter. Sie überprüft, ob aufgrund eines Fehlers falsche statische Signaturen zur Berechnung der dynamischen Signatur verwendet wurden. Zum Schluss addiert sie die dynamische Signatur auf das gewählte codierte Ergebnis und gibt die Summe zurück. (Algorithmus entnommen aus einer Arbeit von Hoffmann u. a. [24])

3.3.3 Fallstrick 3: undefinierte Ausführungsumgebung

Hoffmann u. a. [24] beschreiben in ihrer Arbeit, dass außerdem einige Bit-Fehler im Befehlszähler dazu führen können, dass es zu unvorhersehbaren Sprüngen innerhalb des Programmcodes kommt. Die meisten dieser Bit-Fehler werden durch eine Speicherisolation entdeckt und abgefangen. Einige wenige Sprünge enden allerdings in der Funktion, aus der sie ursprünglich kamen. Diese Sprünge bleiben folglich unentdeckt. So kann es passieren, dass auf veraltete Registerinhalte zugegriffen wird und dies zu komplexen Fehlermustern führt. Aus diesem Grund nullen sie zusätzlich die Register vor dem Mehrheitsentscheid.

3.4 Zusammenfassung

Dieses Kapitel beginnt mit der Definition des Fehlermodells, des Systemmodells und des Anwendungsmodell. Es beschreibt den aktuellen Stand der Forschung im Bereich der arithmetischen Codierung und identifiziert dabei die Regeln **R1** bis **R5**, welche für eine korrekte Parametrisierung eingehalten werden müssen.

Dieses Kapitel untersucht außerdem anhand der in der Arbeit von Hoffmann u. a. [24] aufgestellten Fallstricke mögliche Probleme im CoRed-Ansatz, die zu unentdeckten Datenfehlern führen. Dabei kann für den Fallstrick **F1** festgehalten werden, dass die Definition der Variablen $v_{c,\max}$ in der Arbeit von Hoffmann u. a. [24] fehlt. Diese wird zur Fehlerüberprüfung in der Decodierungsfunktion eingesetzt.

Darüber hinaus wird im Abschnitt zu Pitfall **F2** eine Möglichkeit identifiziert, eine Abfrage zur Fehlererkennung im Programmcode zu modifizieren, um potenziell die Fehlererkennungsleistung an dieser Stelle zu verbessern.

Damit können insgesamt zwei Punkte identifizert werden, für die im Zuge von Kapitel 4 Lösungen erarbeitet werden sollen:

- Es sollen mögliche Definitionen für $v_{c,\max}$ untersucht werden (vgl. Abschnitt 4.1).
- Es soll untersucht werden, ob eine strengere Abfrage zu einer verbesserten Fehlererkennungsleistung führen kann (vgl. Abschnitt 4.2).

Kapitel 4

Lösungsansätze

Im Folgenden sollen Lösungsansätze zu den in Kapitel 3 identifizierten Problemen erarbeitet werden, die im Verlauf der Arbeit mithilfe von Fehlersimulationen überprüft werden sollen. Im Zuge von Fallstrick **F3** wurden keine weiteren Probleme identifiziert, weshalb dieser im Folgenden nicht weiter betrachtet werden soll. Neben der Ausarbeitung der in Kapitel 3 angesprochenen Punkte soll außerdem ein neuer Lösungsansatz zur Reduktion der unentdeckten Datenfehler betrachtet werden. Dabei handelt es sich um die Betrachtung der Code-Distanz bei Verwendung der variablenspezifischen Signaturen.

4.1 Fallstrick 1: Darstellung von Codes im Binärsystem

Wie in Abschnitt 3.3.1 beschrieben, fehlt in der Arbeit von Hoffmann u. a. [24] eine Definition für die Variable $v_{c,\max}$. Diese Variable wird zur Erkennung von Fehlern in den unsystematischen Lücken des Codes eingesetzt und kann auf verschiedene Arten definiert werden. Drei Möglichkeiten sollen im Folgenden präsentiert werden. Dabei wird mit der einfachsten Möglichkeit begonnen und mit der umfangreichsten Definition geendet.

Für die Definitionen werden die Schlüssels A, die Signaturen B_i mit $i \in \{x, y, z\}$ sowie die maximalen Werte der Datentypen für uncodierte und codierte Werte benötigt. Sei im Folgenden INT_MAX der Maximalwert des Datentyps für uncodierte Werte, Schlüssel und Signaturen und INT_MAX_C der Maximalwert des Datentyps für codierte Werte. Für vorzeichenlose 8-Bit-Nutzdaten und vorzeichenlose 16-Bit-Codewörter wären beispielsweise INT_MAX $= 255 = 2^8 - 1$ und INT_MAX_C $= 65535 = 2^{16} - 1$.

Die wohl offensichtlichste Möglichkeit ist, $v_{c,\max}$ als INT_MAX_C zu definieren. Diese Einschränkung ist sehr lose und gleichzeitig kaum zielführend. Das Ziel der in Abschnitt 3.3.1 beschriebenen Überprüfung ist, Werte zu identifizieren, die größer als die maximal möglichen Nutzdaten sind. Dies ist mithilfe der Einschränkung $v_{c,\max} =$ INT_MAX_C offensichtlich nicht möglich, weil lediglich der technisch mögliche, maximale Wert betrachtet wird. Dieser kann ohnehin nicht ohne einen Überlauf der Variable überschritten werden.

Die zweite Alternative beschränkt $v_{c,\max}$ auf das rechnerisch größtmögliche Codewort. Hierzu müssen neben Annahmen über die Datentypen der uncodierten und codierten Werte auch Annahmen über die Datentypen von Schlüssel und Signatur getroffen werden. In dieser Arbeit wird angenommen, dass die Datentypen der Schlüssel und Signaturen dieselben sind wie die der uncodierten Werte. Mit dieser Annahme ist das rechnerisch größtmögliche Codewort $v_c = \text{INT\_MAX} \cdot \text{INT\_MAX} + \text{INT\_MAX} - 1 \stackrel{!}{=} v_{c,\max}$.

Dass diese Einschränkung allerdings ebenfalls zu großzügig ist, ist schnell zu sehen. Der Schlüssel wird in den seltensten Fällen als INT_MAX gewählt werden. Vielmehr wird die Auswahl abhängig von der Restfehlerwahrscheinlichkeit und der Code-Distanz der möglichen Schlüssel erfolgen. Damit ist $v_{c,\max} = \text{INT\_MAX} \cdot \text{INT\_MAX} + \text{INT\_MAX} - 1$ lediglich eine bessere obere Schranke.

Da allerdings sowohl der Schlüssel als auch die Signaturen bereits zur Übersetzungszeit bekannt sind, kann $v_{c,\max}$ weiter verfeinert werden. Für die Berechnung wird lediglich der Schlüssel sowie die größte variablenspezifische Signatur benötigt. Diese ist aufgrund von $B_x > B_y > B_z$ (vgl. **R4**) immer B_x. Dann ergibt sich $v_{c,\max} = A \cdot \text{INT\_MAX} + B_x$. Ohne weitere Informationen, wie die Größe des zu codierenden Werts, sind weitere Einschränkungen nicht möglich. Diese könnten lediglich zur Laufzeit gewonnen werden und würden keine verlässlichen Anhaltspunkte liefern, da bereits der Eingabewert von CoRed fehlerhaft sein könnte. Daher wird im Folgenden $v_{c,\max} = A \cdot \text{INT\_MAX} + B_x$ angenommen.

4.2 Fallstrick 2: Zustand zwischen Instruktionen

Die Abfrage in Zeile 2 in Algorithmus 3.2 stellt sicher, dass die dynamische Signatur, wie alle anderen Signaturen auch, nicht größer als A wird. Wäre eine der Signaturen größer als A, so könnten potenziell Fehler vor dem Dekodieren unerkannt bleiben, weil eine Überprüfung über $v_c \mod A \neq B$ ein gültiges Ergebnis erzielen würde.

Außerdem kann mithilfe der Abfrage in Zeile 2 eine besonders schwerwiegende Art der Kontrollflussfehler erkannt werden. Dabei geht es insbesondere um Fehler im Statusregister zwischen zwei voneinander abhängigen Instruktionen. Da die einzelnen Bits im Statusregister nicht redundant ausgelegt sind, können Bitfehler an dieser Stelle besonders schlimme Folgen haben. Sie können in der CoRed-Implementierung z. B. dazu führen, dass die dynamische Signatur falsch berechnet wird (vgl. Abschnitt 3.3.2).

Um solche Fehler zu erkennen, führen Hoffmann u. a. [24] eine zusätzliche Abfrage in der `apply()`-Funktion ein. Diese Abfrage zusammen mit den geforderten Regeln **R1** bis **R5** bedingen obere Schranken für die Signaturen B_{dyn} und B_E:

Algorithmus 4.1: *apply()*-Funktion von Hoffmann u. a. [24]

```
1 function apply(v_c, A, B_dyn)
2   if (|B_dyn| ≥ A) then
3     SIGNAL_DUE()
4   end if
5   return v_c + B_dyn
6 end function
```

Algorithmus 4.2: *apply()*-Funktion mit Anpassung

```
1 function apply(v_c, A, B_dyn)
2   if (|B_dyn| ≥ B_z) then
3     SIGNAL_DUE()
4   end if
5   return v_c + B_dyn
6 end function
```

Abbildung 4.1: Vergleich zweier `apply()`-Funktionen. Die `apply()`-Funktion (links) aus der Arbeit von Hoffmann u. a. [24] wird so abgeändert, dass der gültige Coderaum stärker eingeschränkt wird. (Abbildung nach einer Darstellung aus einer Arbeit von Hoffmann u. a. [24])

Im fehlerfreien Fall gilt $B_{\text{dyn}} = B_E$. Daher ist es ausreichend, im Folgenden B_E zu betrachten. Für B_E gilt nach P. Ulbrich [55]

$$B_E = \begin{cases} (B_x - B_y) + (B_x - B_z), & \text{falls } x = y = z \\ (B_x - B_y), & \text{falls } x = y \\ (B_x - B_z), & \text{falls } x = z \\ (B_y - B_z), & \text{falls } y = z \\ \text{false}, & \text{sonst.} \end{cases} \tag{4.1}$$

Im Allgemeinen ist ein $S \in \mathbb{N}$ eine obere Schranke von B_E genau dann, wenn S eine obere Schranke von $(B_x - B_y) + (B_x - B_z)$ ist. Die Richtung $B_E \leq S \implies (B_x - B_y) + (B_x - B_z) \leq S$ ist offensichtlich. Für die Rückrichtung wird die Regel **R4** genutzt. Falls $x = y$, dann ist

$$B_E = B_x - B_y \leq (B_x - B_y) + (B_x - B_z) \leq S,$$

denn $B_x - B_z \geq 0$. Der Fall $x = z$ verläuft analog.

Für den Fall $y = z$ betrachte

$$B_E = B_y - B_z = (B_x - B_z) - (B_x - B_y) \leq S - 2(B_x - B_y) \leq S,$$

wobei erneut $B_x - B_y \geq 0$ ausgenutzt wurde.

Insbesondere ist A eine obere Schranke, weil alle $|B_{\text{dyn}}| \geq A$ in der `apply()`-Funktion als entdeckte Fehler interpretiert werden. Es gilt also $(B_x - B_y) + (B_x - B_z) \leq A$. Dies kann zur Vereinfachung der Auswahl von Signaturen bei der Parametrisierung helfen, um sicherzustellen, dass **R3** für alle Signaturen, insbesondere B_E, erfüllt ist. Ist $B_E < A$ erfüllt, so ist auch $B_{\text{dyn}} < A$ sichergestellt, da im fehlerfreien Fall $B_E = B_{\text{dyn}}$ gilt.

Diese Arbeit soll untersuchen, ob eine strengere Formulierung der Abfrage $|B_{\text{dyn}}| \geq A$ als $|B_{\text{dyn}}| \geq B_z$ zu einer weiteren Reduktion unentdeckter Datenfehler führen kann.

Abbildung 4.1 zeigt die geplante Anpassung im Vergleich zur ursprünglichen Lösung. Die Idee hinter der Anpassung der Abfrage ist, dass die Signatur B_z, wie auch der Schlüssel A, für alle Parametrisierungen flexibel, aber bekannt ist. Die Änderung der Abfrage im Quellcode resultiert in einer neuen oberen Schranke $(B_x - B_y) + (B_x - B_z) \leq B_z$. Dies hat Einfluss auf die Parametrisierung, da bei der Auswahl der Signaturen nun $(B_x - B_y) + (B_x - B_z) \leq B_z$ gefordert werden muss. Diese Schranke drückt also die Distanz zwischen allen Signaturen.

Das hat einige Konsequenzen. Im Allgemeinen ist das kleinste Codewort $v_{c,\min} = B_z$ und das größte Codewort $v_{c,\max} = A \cdot \text{INT\_MAX} + B_x$. Damit liegen alle gültigen Codewörter im Intervall $[v_{c,\min}, v_{c,\max}]$. Wenn sich die Distanz zwischen den Signaturen verringert, wird das Intervall der möglichen Codewörter ebenfalls kleiner. Dadurch fallen Codewörter bei einem Bitfehler auf der einen Seite schneller aus dem Bereich der gültigen Codewörter. Auf der anderen Seite reduziert sich allerdings die Distanz zwischen den gültigen Codewörtern.

Die Reduktion von $(B_x - B_y) + (B_x - B_z) \leq A$ auf $(B_x - B_y) + (B_x - B_z) \leq B_z$ kann also potenziell die Distanz zwischen den Codewörtern zu sehr reduzieren und zu einer erhöhten Restfehlerwahrscheinlichkeit führen. Es ist allerdings auch denkbar, dass die Reduktion des Gültigkeitsbereichs in einem Maße geschieht, in welchem die Vorteile der Reduktion der Gefahr der erhöhten Restfehlerwahrscheinlichkeit überwiegen.

Darüber hinaus reduziert sich der Gültigkeitsbereich der dynamischen Signatur, weil im fehlerfreien Fall $B_E = B_{\text{dyn}}$ gilt. Eine Verfälschung der dynamischen Signatur, z. B. aufgrund eines Kontrollflussfehlers, kann so schneller auffallen.

Mithilfe einer Fehlersimulation soll deswegen experimentell überprüft werden, ob die Änderung der Abfrage in Kombination mit einer Anpassung der Parametrisierung die Anzahl unentdeckter Datenfehler reduziert.

4.3 Code-Distanz zwischen den Signaturen

Fehlersimulationen dieser Arbeit zeigen allerdings bereits, dass die zuvor beschriebenen Lösungsansätze das Auftreten unerkannter Datenfehler nicht vollständig lösen. Aus diesem Grund soll ein weiterer Lösungsansatz untersucht werden.

Da die Wahl der Schlüssel abhängig von der Hamming-Distanz die Fehlererkennungsleistung der AN-Codes verbessert, ist es naheliegend, die Fehlererkennungsleistung bei Berücksichtigung der Signaturen zu untersuchen. Hierzu sollen die Code-Distanzen von ANB-Codes berechnet werden. In einem weiteren Schritt sollen Signaturen, die zu besonders großen und besonders kleinen Code-Distanzen führen, als Parameter für die vorliegende CoRed-Implementierung verwendet werden.

Zu untersuchen ist an dieser Stelle auf der einen Seite die Fehlererkennungsleistung der angepassten Parametrisierung. Auf der anderen Seite können die errechneten Code-

Distanzen näher betrachtet werden. Hier kann der Frage nachgegangen werden, ob Regeln identifiziert werden können, welche Signatur-Kombinationen zu besonders großen oder besonders kleinen Code-Distanzen führen.

4.4 Zusammenfassung

Dieses Kapitel gibt mögliche Lösungsansätze, um die verbleibenden unentdeckten Fehler im CoRed-Ansatz zu beheben. In diesem Kapitel wird vorgeschlagen, eine der Abfragen im Programmcode von CoRed, die für die Fehlererkennung zuständig ist, strenger zu formulieren. Damit einher geht eine Anpassung der Parametrisierung, da die Auswahl der Signaturen wegen einer aus der strengeren Abfrage resultierenden oberen Schranke stärker kontrolliert werden muss.

Darüber hinaus wurde bei der Problemanalyse im Zuge von Fallstrick **F1** festgestellt, dass in der Arbeit von Hoffmann u. a. [24] die Definition der Variablen $v_{c,\max}$ fehlt. In diesem Kapitel wird eine Definition für diese Variable vorgeschlagen.

Zuletzt schlägt dieses Kapitel vor, die Auswahl der Signaturen in Abhängigkeit von der Code-Distanz zwischen den Signaturen zu treffen. Diese Anpassung der Signaturauswahl hat ebenfalls zum Ziel, die Fehlererkennungsleistung zu verbessern.

Anhand der Problemanalyse (vgl. Kapitel 3.3) und der in diesem Kapitel aufgezeigten Lösungsansätze lassen sich also drei Punkte identifizieren, die im Folgenden untersucht werden sollen:

N1 Es soll überprüft werden, ob eine Anpassung der `apply()`-Funktion und eine zusätzliche Berücksichtigung von $(B_x - B_y) + (B_x - B_z) < B_z$ die Anzahl unentdeckter Datenfehler reduziert.

N2 Es soll überprüft werden, ob eine Fehlererkennung für das Ergebnis v_c in der Decodierungsfunktion mithilfe der Variable $v_{c,\max} = A \cdot \text{INT\_MAX} + B_x$ wirksam ist.

N3 Es soll überprüft werden, ob die Erkennungsleistung des Codes durch die Code-Distanz der Codewörter unter Berücksichtigung der variablenspezifischen Signaturen beeinflusst werden kann.

Kapitel 5

Umsetzung und Versuchsaufbau

Die Umsetzung bzw. der Versuchsaufbau dieser Arbeit ist zweigeteilt. Um beurteilen zu können, ob die Wahl der Signaturen in Abhängigkeit der Code-Distanz die Menge der SDCs reduziert, müssen in einem vorbereitenden Schritt die Code-Distanzen berechnet werden.

Um eine Bewertung der in Kapitel 4 vorgeschlagenen Lösungsansätze durchführen zu können, werden in einem zweiten Schritt Fehlersimulationen mit verschiedenen Parametrisierungen und Implementierungen durchgeführt. Dieses Kapitel beschreibt die Vorgehensweise bei der Auswahl der Schlüssel und der Signaturen. Darüber hinaus wird der Aufbau der Fehlersimulationen skizziert. Für die Fehlerinjektion kommt das Werkzeug FAIL* zum Einsatz.

In dieser Arbeit werden 8-Bit Schlüssel und Signaturen untersucht, aufgrund der schnell ansteigenden Größe des Wertebereichs und damit einhergehender erheblich vergrößerter Rechenzeiten bei den Signatur-Experimenten wie auch bei der Fehlersimulation. Dadurch ergeben sich 16-Bit Codewörter bei 8-Bit großen Eingabewörtern.

5.1 Berechnung der Code-Distanzen

CoRed wird mit einem Schlüssel und drei Signaturen für die drei Ergebnisse der drei Replikate parametrisiert. Diese werden dann u. a. zur Codierung der Replikat-Ergebnisse während des Mehrheitsentscheids verwendet. Die Schlüssel und Signaturen sollen systematisch anhand von Regeln ausgewählt werden können.

Für die Schlüssel gibt es bereits etablierte Lösungen anhand der Restfehlerwahrscheinlichkeit sowie der bekannten Code-Distanz, die der entsprechende Schlüssel für jeweils zwei unterschiedliche Codewörter erzeugt. Die Signaturen verändern aber die Code-Distanz zwischen den Codewörtern, wenn sie bei der Codierung auf das Produkt aus Schlüssel und Signatur addiert werden. In diesem Kapitel soll beschrieben werden, wie die Code-Distanzen von jeweils zwei bzw. drei unterschiedlichen Codewörtern unter Berücksichtigung der Signaturen errechnet werden.

Die Berechnung der Code-Distanzen unter Berücksichtigung der Signaturen verfolgt dabei zwei Ziele. Zum einen sollen die errechneten Code-Distanzen verwendet werden, um mittels Fehlersimulationen prüfen zu können, ob die Code-Distanz dabei hilft, geeignete Signaturen auszuwählen. Dabei ist insbesondere zu prüfen, ob die Code-Distanz, die durch die Signaturen entsteht, die Anzahl der verbleibenden unentdeckten Datenfehler reduziert. Wie diese Frage genau untersucht werden soll, wird in Abschnitt 5.2.3 beschrieben. Im aktuellen Abschnitt soll es lediglich um die Bestimmung der Code-Distanzen an sich gehen.

Zum anderen soll die Verteilung der Code-Distanzen in Abhängigkeit der Signaturen untersucht werden, um eine geeignete Signatur-Auswahl mithilfe von Regeln unterstützen zu können. Dabei stellt sich insbesondere die Frage, wie die Code-Distanzen zwischen den Codewörtern in Abhängigkeit der Signaturen verteilt sind.

Die Berechnung der Code-Distanzen für einen festen 8-Bit-Schlüssel erfolgt in zwei Schritten. Als Erstes werden Code-Distanzen für jeweils zwei Codewörter und folglich auch zwei Signaturen berechnet. Ziel ist, gegebenenfalls hier schon erste Muster erkennen zu können.

In einem zweiten Schritt wird ein drittes Codewort und damit eine dritte Signatur hinzugenommen und entsprechend die Code-Distanz zwischen jeweils drei unterschiedlichen Codewörtern ermittelt. Dazu wird die Code-Distanz unter Verwendung von zwei bzw. drei Signaturen jeweils paarweise berechnet und das Minimum der paarweisen Code-Distanzen als Gesamt-Code-Distanz ermittelt.

Um Rechenzeit zu sparen, müssen die Kombinationen so gewählt werden, dass keine Kombination mehrfach berechnet werden kann. Außerdem muss vermieden werden, dass eine Signatur innerhalb der Berechnung einer Code-Distanz mehrfach vorkommt, da dies offensichtlich in einer Code-Distanz $d_C = 0$ resultieren würde. Daher werden die Signaturen immer absteigend sortiert ausgewählt.

Damit ist außerdem die Regel **R4**: $B_x > B_y > B_z$ aus der Literatur erfüllt. Darüber hinaus kann die Regel **R3**: $0 < B_i < A$, für alle $i \in x, y, z$ beachtet werden, da diese die Rechenzeit erheblich reduziert und aus fachlicher Sicht ohnehin notwendig ist (vgl. Abschnitt 3.2). Um eine Vorstellung davon zu erlangen, wie sich die Code-Distanz entwickelt, wenn Regel **R3** verletzt wird, werden allerdings bewusst auch solche Kombinationen gewählt, die **R3** nicht genügen.

Für einen festen Schlüssel und zwei beziehungsweise drei Signaturen werden dann alle möglichen Codewörter sowie die Hamming-Distanzen zwischen den Codewörtern berechnet. Wird eine Code-Distanz $d_C = 0$ berechnet, so wird die Berechnung an dieser Stelle für die Schlüssel-Signatur-Kombination frühzeitig abgebrochen, um Rechenzeit einzusparen. Die Schlüssel-Signatur-Kombination werden zusammen mit den ermittelten Code-Distanzen sowie weiteren zur Analyse hilfreichen Informationen in einer Ergebnis-Datei gespeichert.

Alternative Vorgehensweisen zur zuvor beschriebenen Berechnung der Code-Distanzen wurden untersucht. Dabei wurden insbesondere eine statische Analyse und der Einsatz

Schlüssel	d_C	Verhalten bei Überbelastung
113	$d_C = 2$	gutartig
185	$d_C = 4$	gutartig
233	$d_C = 4$	bösartig
241	$d_C = 2$	bösartig

Tabelle 5.1: Für die Versuche ausgewählte Schlüssel und ihre Eigenschaften. Es wurden insgesamt vier Schlüssel gewählt. Davon wurden zwei mit einer möglichst großen Code-Distanz gewählt, wobei einer der beiden gutartiges Verhalten ($A = 185$) und einer der beiden bösartiges Verhalten ($A = 233$) bei Überbelastung des Codes zeigt. Zwei weitere Schlüssel wurden so gewählt, dass sie die minimal mögliche Code-Distanz mit fehlererkennenden Eigenschaften bei Einbit-Fehlern ($d_C = 2$) erzeugen. Dabei ist $A = 113$ gutartig und $A = 241$ bösartig.

von LP-Solvern in Betracht gezogen. Es konnten allerdings keine passenden Werkzeuge identifiziert werden.

5.2 Vorgehen bei der Untersuchung der Lösungsansätze

Das Ziel ist, die vorgeschlagenen Regeln und Lösungsansätze **N1** bis **N3** auf ihre Wirksamkeit zu untersuchen. Zu diesem Zweck sollen Fehlersimulationen durchgeführt werden. Für diese Fehlersimulationen sind die Vorgehensweisen zu definieren. Interessant ist zum einen die Wahl der Schlüssel und der Signaturen, welche für die Fehlersimulationen verwendet werden sollen; zum anderen ist die Durchführung der Fehlersimulation selbst wichtig, da diese aufgrund relativ hoher Laufzeiten zu Problemen führen kann.

5.2.1 Auswahl der Schlüssel-Signatur-Kombinationen

Für die Parametrisierung der CoRed-Implementierung müssen geeignete Schlüssel identifiziert werden. Auf Basis der Schlüssel können in einem zweiten Schritt geeignete Signaturen gewählt werden (vgl. Tabelle 5.1). Anhand der Berechnungen von P. Ulbrich [55], der in seinen Arbeiten die Auswahl der Schlüssel anhand der Code-Distanz genauer untersucht und besonders gute Schlüssel identifiziert hat, bieten sich insbesondere $A = 185$ und $A = 233$ als Schlüssel an, weil sie beide eine Code-Distanz von vier aufweisen und diese für 8-Bit Schlüssel maximal ist. Dabei kann nach Hoffmann u. a. [24] $A = 185$ als gutartig und $A = 233$ als bösartig eingeordnet werden. Weiterhin werden Schlüssel mit geringerer Code-Distanz als Vergleichswerte benötigt. Hier werden der gutartige Schlüssel $A = 113$ sowie der bösartige Schlüssel $A = 241$ mit jeweils einer Code-Distanz $d_C = 2$ gewählt.

Mithilfe der gewählten Schlüssel können nun gültige Signatur-Kombinationen anhand der aus der Literatur bekannten Regeln ausgewählt werden (vgl. Abschnitt 3.2). Dazu wird für den gewählten Schlüssel ein Signatur-Tripel anhand der Regeln bestimmt. Um nicht

Schlüssel	Grundgesamtheit
113	168.202
185	758.596
233	1.528.912
241	1.693.740

Tabelle 5.2: Größe der statistischen Grundgesamtheit der in den Versuchen betrachteten Schlüssel. Die statistische Grundgesamtheit steigt mit steigendem Schlüssel. Das ist auf die Regeln **R3**, **R4** und **R5** (vgl. Abschnitt 3.2) zurückzuführen. Diese beschränken die möglichen Kombinationen in Abhängigkeit vom Schlüssel.

versehentlich eine besonders geeignete oder eine besonders ungeeignete Schlüssel-Signatur-Kombination mit einer sehr geringen bzw. sehr hohen Anzahl unentdeckter Fehler zu ziehen und die Ergebnisse so zu verfälschen, werden mehrere Schlüssel-Signatur-Kombinationen zufällig aus der Menge der möglichen Kombinationen gezogen. Für jede gewählte Schlüssel-Signatur-Kombination wird eine Fehlersimulation durchgeführt. Die Ergebnisse der Kampagnen werden untereinander verglichen.

Da der Begriff „Experiment“ aufgrund der Fehlersimulation bereits belegt ist, wird im Folgenden der Begriff „Versuch“ anstelle des Begriffs „Experiment“ im statistischen Kontext verwendet.

Die Grundgesamtheiten der verschiedenen Schlüssel, d.h. die Anzahl aller möglichen Signatur-Tripel für einen Schlüssel, sind in Tabelle 5.2 dargestellt. Die Stichprobengröße wird in dieser Arbeit auf 1.000 festgelegt. Das bedeutet, es werden 1.000 Schlüssel-Signatur-Kombinationen zufällig gezogen und daher 1.000 Kampagnen durchgeführt. Die Wahl der Stichprobengröße ergibt sich aus mehreren Anforderungen. Primär soll ein Bias in der Auswahl der Schlüssel-Signatur-Kombinationen verhindert werden, weshalb in jedem Fall mehrere Kombinationen zufällig gezogen werden müssen. Da in dieser Arbeit Mikroeffekte untersucht werden, die letztlich zu unentdeckten Datenfehlern führen können, sind sonderlich große Streuungen nicht zu erwarten. Dies reduziert die Größe der benötigten Stichprobe. Zuletzt benötigt eine Fehlerinjektionskampagne von CoRed auf dem verwendeten System etwa 100 s. Daher darf die Stichprobengröße ebenfalls nicht zu groß gewählt werden.

Für den Schlüssel $A = 185$ existiert z.B. eine Grundgesamtheit von 758.596 möglichen Kombinationen für B_x, B_y und B_z, woraus 1.000 Kombinationen zufällig gezogen werden[1]. Dabei gelten für die Grundgesamtheit lediglich die bisherigen Regeln aus der Literatur (vgl. 3.2). Es wird an dieser Stelle noch keine Filterung hinsichtlich der in Kapitel 4 vorgeschlagenen Regeln vorgenommen.

[1] Dazu wurde der Kommandozeilenbefehl `shuf -n 1000` verwendet.

Auf Basis der Stichprobengröße kann die Fehlerspanne ermittelt werden. Für einen z-Wert von $z = 1,96$[2], eine Populationsgröße von etwa $N = 168.202$ (für $A = 113$) bzw. $N = 1.693.740$ (für $A = 241$), eine Stichprobengröße von $n = 1.000$ und einem π-Wert $\pi = 0,5$[3] ergibt sich eine Fehlerspanne von etwa $E = z \cdot \sqrt{\frac{\pi(1-\pi)}{n}} \cdot \sqrt{\frac{N-n}{N-1}} = 0,030898 \approx 3,1\%$ für $N = 168.202$ bzw. $E = 0,030981 \approx 3,1\%$ für $N = 1.693.740$. Das heißt, dass eine Abweichung von etwa $3,1\%$ der mithilfe der Stichprobe ermittelten Werte von den tatsächlichen Werten möglich ist.

5.2.2 Implementierung und Ausführungsumgebung

Diese Arbeit baut auf einer bereits bestehenden C++-Implementierung des CoRed-Mehrheitsentscheiders auf. Mithilfe des GCC-Compilers in der Version 10.2.1 wird die Implementierung für die IA32 Architektur in der Optimierungsstufe `-O2` übersetzt. Als Ausführungsumgebung für die Implementierung wird das Echtzeitbetriebssystem *eCos*[4] eingesetzt. Dieses sorgt für die im Systemmodell (vgl. Abschnitt 3.1.2) geforderte räumliche Isolation.

Für die Fehlerinjektion wird das Werkzeug Fail* mit Bochs[5] als Simulator verwendet. Fail* ist in diesem Versuchsaufbau verantwortlich für die zeitliche Isolation der Anwendung, indem es einen Wächter implementiert, der bei Überschreitung einer festzulegenden maximalen Laufzeit eine Terminüberschreitung signalisiert.

Fail* injiziert Fehler auf Befehlssatzebene. Dazu wird in dieser Arbeit das vorgefertigte Experiment „generic-experiment“[6] verwendet, welches ein generalisiertes Fehlerinjektionsexperiment implementiert. Injiziert werden Einbit-Fehler in den Befehlszähler und Merker, den Speicher sowie frei verwendbare Register.

Ein einzelnes Experiment einer Fehlerinjektionskampagne kann anhand der Auswirkungen des injizierten Fehlers unterschiedlich kategorisiert werden. In dieser Arbeit werden dieselben Kategorien verwendet, die P. Ulbrich in seinen Arbeiten einsetzt [55, 57, 24].

Ein Fehler kann aufgrund seiner Auswirkungen in die Kategorien *Fehler ohne Auswirkungen*, *erkannter Fehler* und *unerkannter Datenfehler* eingeteilt werden. Bei einem Fehler ohne Auswirkungen ist aufgrund der Injektion ein Defekt entstanden, der durch Redundanzmaßnahmen verdeckt wird, und sich daher nicht weiter ausbreitet. [55]

Entdeckte Fehler sind Fehler, die auf unterschiedliche Arten erkannt und daher behandelt werden können. Sie lassen sich abhängig von der Art, auf die sie erkannt wurden, weiter einteilen in die Kategorien *CoRed*, *Terminüberschreitungen*, (engl. *Timeout*), *Hardwareausnahmen* (engl. *Trap*) und *Schutzverletzungen*. [55]

[2] Das entspricht einem Konfidenzniveau von 95%.

[3] Es wird $\pi = 0,5$ gewählt, weil über den Anteil der untersuchten Merkmale an der Grundgesamtheit nichts bekannt ist.

[4] eCos ist verfügbar unter https://ecos.sourceware.org/.

[5] Bochs ist verfügbar unter https://bochs.sourceforge.io/.

[6] Das vorgefertigte Experiment „generic-experiment“ ist verfügbar unter https://github.com/danceos/fail/tree/master/src/experiments/generic-experiment.

Die Kategorie *CoRed* beschreibt Fehler, die aufgrund der arithmetischen Codierung entdeckt wurden [55].

Bei einer *Terminüberschreitungen* sorgt der injizierte Fehler dafür, dass das Programm nicht in der vorgegebenen Zeit terminiert und daher zu lange Ressourcen blockiert. Solche Fehler werden durch die Hardware oder das Betriebssystem erkannt. [55]

Hardwareausnahmen werden - wie der Name bereits vermuten lässt - von der Hardware signalisiert. Dies geschieht, falls z.B. eine ungültige oder nicht zulässige Instruktion ausgeführt wird. [55]

Bei einer *Schutzverletzung* wird aufgrund des injizierten Fehlers auf Speicher oder Programmcode zugegriffen, der außerhalb des Zugriffs des gerade ausgeführten Programms liegt. Solche Verletzungen werden ebenfalls vom Betriebssystem oder der unterliegenden Hardware signalisiert. [55]

Sowohl *Terminüberschreitungen* als auch *Hardwareausnahmen* und *Schutzverletzungen* können zwar erkannt, allerdings nicht unmittelbar behoben werden. Stattdessen ist es möglich, den Mehrheitsentscheid erneut durchzuführen und so ein gültiges Ergebnis zu erreichen. [55]

Unerkannte Datenfehler sind Fehler, die nicht durch das Betriebssystem, die Hardware oder CoRed erkannt werden konnten. Sie können sich daher weiter ausbreiten und führen zu falschen Ergebnissen. [55]

5.2.3 Durchführung der Versuche

Im Folgenden soll der Erfolg der vorgeschlagenen Lösungsansätze (vgl. Kapitel 4) untersucht werden. Um die verschiedenen Ergebnisse bewerten zu können, wird eine Baseline als Vergleichsbasis benötigt. Daher wird zunächst beschrieben, wie die Baseline ermittelt wird. Anschließend wird der Versuchsaufbau zur Untersuchung der Lösungsansätze skizziert.

Bestimmung der Baseline

Da die Fehlererkennungsleistung von CoRed bei Einführung der neuen, vorgeschlagenen Regeln (vgl. Kapitel 4) bewertet werden soll, wird eine zufällige Stichprobe der Größe 1.000 aus der Grundgesamtheit für jeden Schlüssel als Baseline gezogen.

Da vier Schlüssel exemplarisch betrachtet werden, existieren also vier Stichproben der Größe 1.000 - eine für jeden Schlüssel mit unterschiedlichen Signatur-Tripeln. Es werden nicht für jede Stichprobe dieselben Signatur-Tripel verwendet, weil der Kombinationsraum des Schlüssels $A = 241$ erheblich größer ist als der des Schlüssels $A = 113$. Um mit der Stichprobe den Kombinationsraum vollständig ausnutzen zu können, müssen daher verschiedene Signatur-Tripel gewählt werden.

Für die Baseline werden weder die Anmerkungen **N1** oder **N2**, noch die Betrachtung der Code-Distanzen für **N3** berücksichtigt, um eine einheitliche Vergleichsgrundlage für alle

Algorithmus 5.1: *decode()* ohne $v_{c,\max}$

```
1 function decode(v_c, A, B)
2   if ((v_c mod A) ≠ B) then
3     SIGNAL_DUE()
4   end if
5   return (v_c mod A) ≠ B
6 end function
```

Algorithmus 5.2: *decode()* mit $v_{c,\max}$

```
1 function decode(v_c, A, B)
2   if (v_c > v_{c,max} or (v_c mod A) ≠ B) then
3     SIGNAL_DUE()
4   end if
5   return (v_c mod A) ≠ B
6 end function
```

Abbildung 5.1: Vergleich zweier `decode()`-Funktionen. Die `decode()`-Funktion (links), die in der Baseline verwendet wird, prüft in Zeile 2 lediglich, ob das Codewort offensichtlich verfälscht wurde. Die `decode()`-Funktion auf der rechten Seite wurde in Zeile 2 um eine zusätzliche Prüfung auf das maximal mögliche Codewort $v_{c,\max}$ erweitert (vgl. Abschnitt 3.3.1). (Abbildung nach einer Darstellung aus einer Arbeit von Hoffmann u. a. [24])

diese Änderungen zu schaffen. Das bedeutet insbesondere, dass die Baseline-Implementierung die Überprüfung $v_c > v_{c,\max}$ (vgl. Abschnitt 3.3.1) nicht umsetzt. Die `decode()`-Funktion führt lediglich eine reduzierte Überprüfung $(v_c \mod A) \neq B$ durch.

Mit den für die Baseline gezogenen Schlüssel-Signatur-Kombinationen werden Fehlerinjektionskampagnen durchgeführt. Die Ergebnisse der Kampagnen bilden die Baseline für alle weiteren Versuche.

Die Durchführung der Experimente unterscheidet sich etwas abhängig vom untersuchten Lösungsvorschlag. Daher wird an dieser Stelle erneut anhand der bisher verwendeten Kategorien differenziert.

Fallstrick 1: Darstellung von Codes im Binärsystem

In diesem Schritt soll es darum gehen, die Definition von $v_{c,\max} = A \cdot \mathrm{INT\_MAX} + B_x$ zu überprüfen (vgl. Abschnitte 3.3.1 und 4.1).

Dazu wird die Implementierung der `decode()`-Funktion entsprechend um eine Überprüfung auf $v_c > v_{c,\max}$ erweitert (vgl. Abbildung 5.1). Die geänderte Implementierung wird mit den Schlüssel-Signatur-Kombinationen aus den Stichproben parametrisiert. Daraufhin werden Fehler injiziert. Die Tabelle 5.3 vergleicht die Größe der $v_{c,\max}$-Implementierung mit der Größe der Baseline-Implementierung.

Die Ergebnisse dieser Injektionen werden mit den Fehlersimulationen der Baseline verglichen und ausgewertet. Dabei wird der Fokus insbesondere auf unentdeckte Datenfehler gelegt. Zu untersuchen ist, ob die Definition von $v_{c,\max}$ geschickt gewählt wurde. Ist dies der Fall, so reduziert sich die Anzahl der Schlüssel-Signatur-Kombinationen, bei denen unentdeckte Datenfehler zu beobachten sind, merklich.

Implementierung	Baseline	$v_{c,\max}$
Größe text	912 B	920 B
Größe bss	12 B	12 B
Größe data	0 B	0 B
Gesamtgröße	924 B	932 B
# Instruktionen	259	260

Tabelle 5.3: Größenvergleich der untersuchten Implementierungen. Die Tabelle zeigt die Größen der Segmente sowie die Anzahl der Instruktionen für die Baseline- und die $v_{c,\max}$-Implementierung im Vergleich.

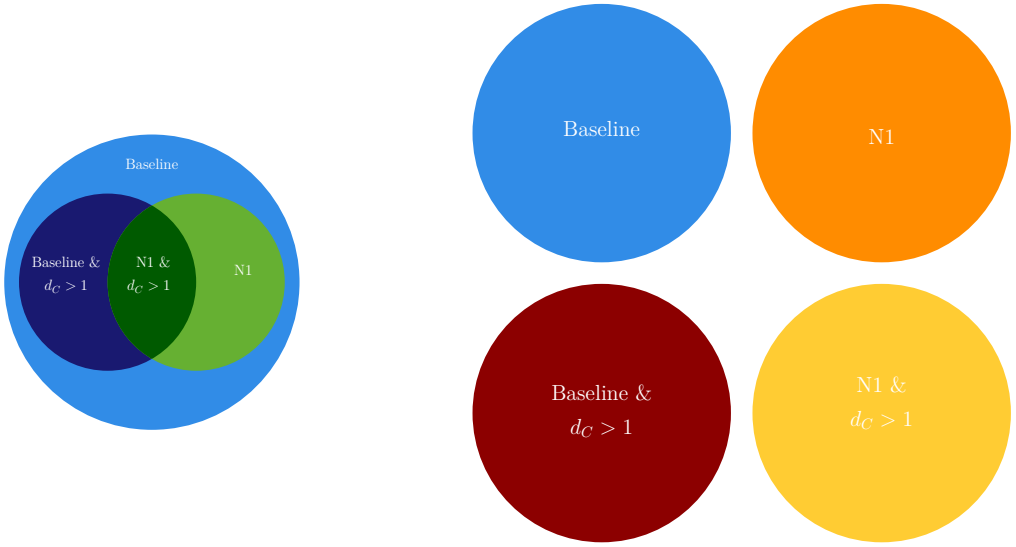
Fallstrick 2: Zustand zwischen Instruktionen

Im Zuge des Fallstricks **F2** wurde eine Anpassung der Abfrage in der `apply()`-Funktion in Kombination mit einer neuen Auswahlregel für Signaturen vorgeschlagen (vgl. Regel **N1**). Die Wirksamkeit der Änderung soll mithilfe der folgenden Versuche überprüft werden.

Stichprobenverfahren 1 (V1) Die Regel aus **N1** wird auf die Stichproben der Baseline angewendet und anhand der Regel $(B_x - B_y) + (B_x - B_z) < B_z$ gefiltert. Dies reduziert die Größe der Stichprobe. Die Schlüssel-Signatur-Kombinationen, welche nicht der Regel **N1** genügen, entfallen. Mit dieser reduzierten Stichprobe können erneut Fehlersimulationen durchgeführt werden bzw. können aus den Ergebnissen der Baseline die Ergebnisse für **N1** extrahiert werden. Es werden dabei beide Implementierungen aus Abschnitt 5.2.3 verwendet; sowohl in die Baseline-Implementierung als auch in die um $v_{c,\max}$ erweiterte Implementierung werden Fehler injiziert, weil die Baseline-Implementierung aufgrund der Vergleichbarkeit über alle Versuche hinweg als Vergleichsbasis herangezogen werden soll. Es wird allerdings davon ausgegangen, dass die $v_{c,\max}$-Implementierung die Fehlererkennungsleistung verbessert, weshalb auch diese Implementierung in die Auswertungen mit einbezogen werden soll.

Stichprobenverfahren 2 (V2) Dieses Verfahren soll dabei helfen, zu bewerten, ob die ermittelten Werte trotz der durch die Regeln verringerten Stichprobe realistisch sind. Bei der zweiten Versuchsart wird lediglich die Baseline-Implementierung injiziert, da die zusätzlichen Injektionen sehr zeitintensiv sind. Dafür wird eine neue Stichprobe der Größe 1.000 zufällig aus der Grundgesamtheit gezogen, bei der alle Schlüssel-Signatur-Kombinationen die Regel aus **N1** erfüllen. Die Ergebnisse der erneuten Injektion bei anderer Parametrisierung wird mit den Ergebnissen der ersten Versuchsart verglichen.

Anhand des Vergleichs der Ergebnisse der beiden Versuchsarten kann eine Vermutung darüber aufgestellt werden, ob die Ergebnisse realistisch sind. Weichen die Ergebnisse der



(a) Mengenbeziehungen im Stichprobenverfahren 1 (V1)

(b) Mengenbeziehungen im Stichprobenverfahren 2 (V2)

Abbildung 5.2: Mengenbeziehungen in den Stichprobenverfahren. Die Abbildung verdeutlicht die Beziehungen der Mengen, die für die beiden Stichprobenverfahren gezogen werden. Die Baseline-Menge des Stichprobenverfahren 1 (V1) entspricht der Baseline-Menge des Stichprobenverfahren 2 (V2). Aus den gezogenen Mengen ergeben sich die Versuche dieser Arbeit.

beiden Versuche nicht besonders stark voneinander ab, kann vermutet werden, dass die Ziehung der Stichprobe einigermaßen repräsentativ ist.

Die Ergebnisse sollen auf eine verbesserte Erkennung unentdeckter Datenfehler untersucht werden.

Code-Distanz zwischen den Signaturen

Vergleichbar zu dem Vorgehen in Abschnitt 5.2.3 soll an dieser Stelle untersucht werden, ob die Beachtung der Code-Distanz in Abhängigkeit der Signaturen zu einer Reduktion der unentdeckten Datenfehler führt.

Hierzu wird erneut die Stichprobe anhand von Regeln gefiltert; dabei werden die Regel-Kombinationen „Baseline und $d_C > 1$“ und „**N1** und $d_C > 1$“ betrachtet und mit den Ergebnissen der vorherigen Injektionsversuche verglichen. Die Forderung $d_C > 1$ deckt dabei die Regel **N3** ab. Für die Überprüfungen von **N3** werden die Zwischenergebnisse aus 5.1 verwendet. In diesem Schritt wurden bereits die Code-Distanzen für die verwendeten Schlüssel-Signatur-Kombinationen berechnet. Die Simulationen werden ebenfalls auf die in 5.2.3 beschriebenen Arten durchgeführt.

5.3 Zusammenfassung

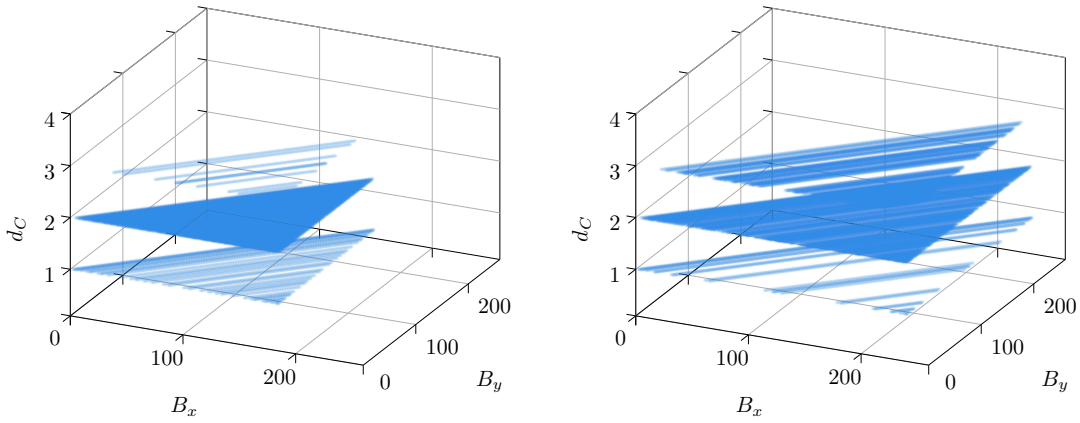
Dieses Kapitel beschreibt zunächst, wie die Code-Distanzen zwischen den Signaturen berechnet werden. Dazu wird erst die Code-Distanz zwischen jeweils zwei Signaturen berechnet. Auf diesen Zwischenergebnissen aufbauend werden die Code-Distanzen für jeweils drei Signaturen berechnet. Das Ergebnis sind Schlüssel-Signatur-Kombinationen, denen jeweils eine Code-Distanz zugeordnet wird. Diese Ergebnismenge ist nach Schlüsseln gruppiert. Die nach Schlüssel gruppierten Schlüssel-Signatur-Kombinationen dienen als Grundgesamtheiten für die folgenden Fehlerinjektionsexperimente.

Die Abbildung 5.2 zeigt die Mengenbeziehungen in den beiden Stichprobenverfahren der Fehlerinjektionsexperimente und gibt damit einen Überblick über die Menge der Versuche. Dabei wird das Stichprobenverfahren 1 (V1) für die Versuche der Baseline-Implementierung und der $v_{c,\max}$-Implementierung angewendet. Das Stichprobenverfahren 2 (V2) hingegen wird als vergleichendes Verfahren nur für die Baseline-Implementierung durchgeführt, weil es dazu dient, die Qualität der Stichproben einschätzen zu können. Die Baseline-Stichprobe ist im Verfahren **V1** und **V2** jeweils identisch. Die Mengen von **V2** sind mit einem Umfang von 1.000 Schlüssel-Signatur-Kombinationen gleich groß, wohingegen im Verfahren **V1** lediglich die Baseline 1.000 Kombinationen umfasst. Für alle weiteren Stichproben in **V1** wird die Baseline-Menge gefiltert. Alle Versuche werden jeweils für die Schlüssel $A = 113$, $A = 185$, $A = 233$ und $A = 241$ durchgeführt. Die Ergebnisse der Versuche werden den entsprechenden Themenbereichen der Auswertung zugeordnet.

Kapitel 6

Auswertung

Dieses Kapitel untersucht die in Kapitel 4 beschriebenen Lösungsansätze auf ihre Gültigkeit. Wie bereits das Kapitel 3 ist auch dieses Kapitel anhand der identifizierten Fallstricke und Lösungsansätze gegliedert. Zuvor wird allerdings die Verteilung der Code-Distanzen zwischen den Signaturen auf Regelmäßigkeiten untersucht. Zum Abschluss des Kapitels wird auf mögliche Einschränkungen der Validität aufgrund des Versuchsaufbaus der Abschnitte 6.2, 6.3 und 6.4 eingegangen.



(a) Verteilung der Code-Distanzen für zwei Signaturen für den Schlüssel $A = 185$.

(b) Verteilung der Code-Distanzen für zwei Signaturen für den Schlüssel $A = 241$.

Abbildung 6.1: Verteilung der Code-Distanzen für zwei Signaturen. Für jeweils einen festen Schlüssel werden Signatur-Kombinationen aus jeweils zwei Signaturen gebildet und die Code-Distanz für diese Kombinationen berechnet. Auf der x- und y-Achse sind die Signaturen B_x und B_y abgetragen. Auf der z-Achse ist die entstehende Code-Distanz abgebildet. Es bilden sich optisch Geraden im Raum. Liegen viele Geraden direkt nebeneinander, wirkt die jeweilige Ebene an dieser Stelle dichter und dunkler als an Stellen, an denen sich weniger Geraden in dieser Ebene befinden.

6.1 Muster in Code-Distanzen

Für die Muster-Analyse wurden, wie in Abschnitt 5.1 beschrieben, die Code-Distanzen in zwei Schritten ermittelt. Zunächst wurden die Code-Distanzen für zwei Signaturen berechnet. In einem zweiten Schritt wurde eine dritte Signatur hinzugezogen. Aus diesem Grund ist die Auswertung der Daten ebenfalls zweigeteilt. Im Folgenden sollen zunächst die Ergebnisse für die Berechnungen mit zwei Signaturen betrachtet werden. Darauf aufbauend werden die Ergebnisse für die Berechnungen mit drei Signaturen beschrieben. Dabei werden einige Gemeinsamkeiten deutlich.

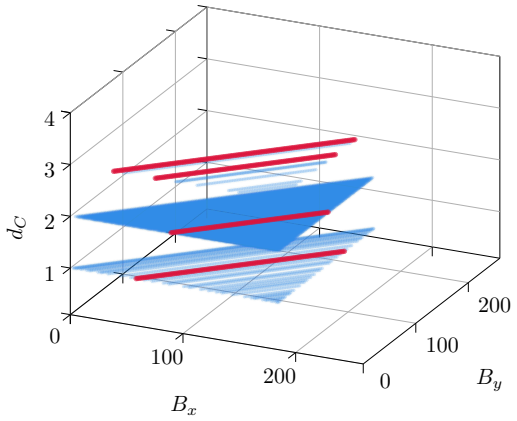
6.1.1 Verteilung der Code-Distanzen für zwei Signaturen

Abbildung 6.1 zeigt die ermittelten Code-Distanzen für zwei Signaturen in Abhängigkeit des Schlüssels A. Für die Schlüssel wurden zur Analyse die Werte $A = 113$, $A = 185$, $A = 233$ und $A = 241$ gewählt, da diese in den späteren Auswertungen ebenfalls eine Rolle spielen sollen (vgl. Abschnitt 5.2). Die Abbildungen für die verschiedenen Schlüssel fallen allerdings immer recht ähnlich aus. Daher werden an dieser Stelle lediglich die Schlüssel $A = 185$ und $A = 241$ untersucht, weil sie aufgrund der größeren Code-Distanz $d_c = 4$ interessanter sind. Die Auswertungen für die Schlüssel $A = 113$ und $A = 233$ sind ergänzend im Anhang aufgeführt (vgl. Abschnitt A.1).

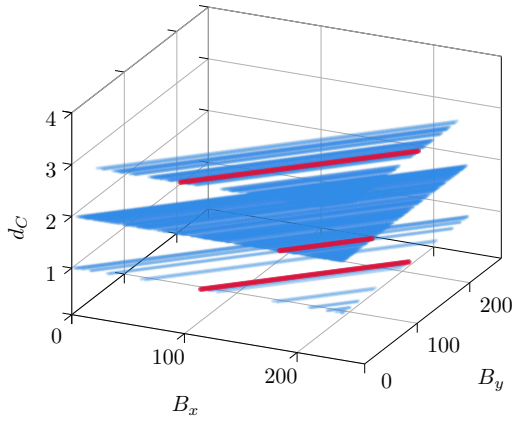
Die zusätzlichen Auswertungen zeigen, dass beobachtete Muster nicht zufällig für vereinzelte Schlüssel entstanden sind. Stattdessen existieren Regelmäßigkeiten, die im Folgenden beschrieben und durch die Abbildungen im Anhang weiter bestätigt werden. In der Abbildung 6.1 sind auf der x- und y-Achse die für die Berechnung der Code-Distanz verwendeten Signaturen abgetragen. Auf der z-Achse ist die resultierende Code-Distanz dargestellt.

Betrachtet man die Grafiken, fällt auf, dass die Code-Distanzen in Abhängigkeit der beiden Signaturen Geraden im Raum zu bilden scheinen. Darüber hinaus erwecken diese Geraden den Eindruck, als würden sie parallel zueinander verlaufen. Dies ist experimentell zu bestätigen:

Es soll eine Gerade g experimentell bestimmt werden. Dazu wird zunächst der Punkt P mit dem kleinsten x-Wert und dem kleinsten y-Wert einer Ebene gewählt. Stimmt die These, dass sich die Punkte auf Geraden verteilen und sich diese Geraden aufgrund der Parallelität nicht schneiden, dann liegt der Punkt Q mit dem größten x-Wert und dem größten y-Wert der Ebene vermutlich ebenfalls auf derselben Geraden. Seien p und q die Ortsvektoren von P und Q. Der Ortsvektor p dient als Stützvektor der Geraden. Aus den Vektoren p und q ist der Richtungsvektor r der Geraden zu errechnen. Die Parameterform von g ist dann $g = p + \lambda \cdot r$. Mithilfe von Punktproben kann nun bestimmt werden, ob die Punkte tatsächlich auf der ermittelten Gerade liegen.



(a) Beispiel für experimentell ermittelte Geraden im Raum für den Schlüssel $A = 185$.



(b) Beispiel für experimentell ermittelte Geraden im Raum für den Schlüssel $A = 241$.

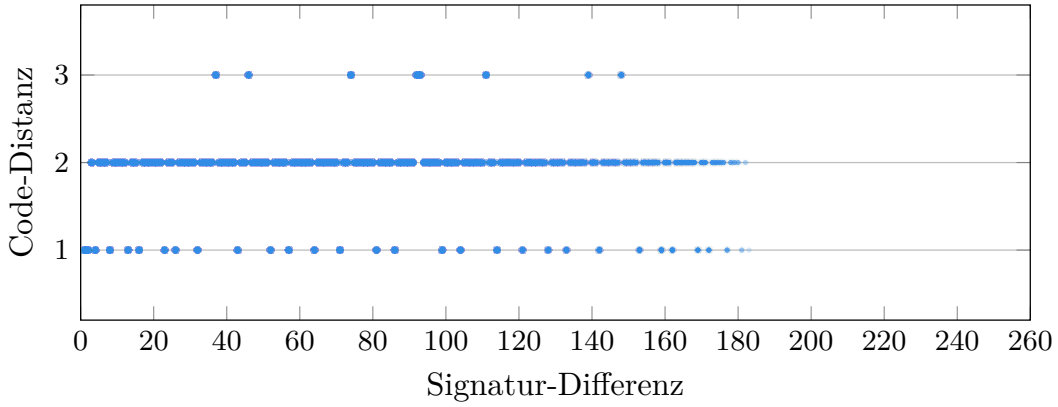
Abbildung 6.2: Beispiel für experimentell ermittelte Geraden im Raum unter Verwendung von zwei Signaturen. Die Abbildung basiert auf der Abbildung 6.1 und zeigt Beispiele für experimentell ermittelte Geraden (in rot). Auch hier werden wieder der Schlüssel $A = 185$ und $A = 241$ betrachtet, Signatur-Kombinationen aus jeweils zwei Signaturen gebildet und die Code-Distanz für diese Kombinationen berechnet. Alle Geraden nutzen dabei den Richtungsvektor $r = (1, 1, d_C)^T$. Lediglich die Stützvektoren unterscheiden sich.

Bei der experimentellen Bestimmung der Geraden fällt auf, dass der Richtungsvektor stets von der Form $r = (1, 1, d_C)^T$ ist. Die grafische Darstellung 6.2 der experimentell bestimmten Geraden lässt schnell erkennen, dass es sich in Abbildung 6.1 tatsächlich um parallele Geraden handeln muss.

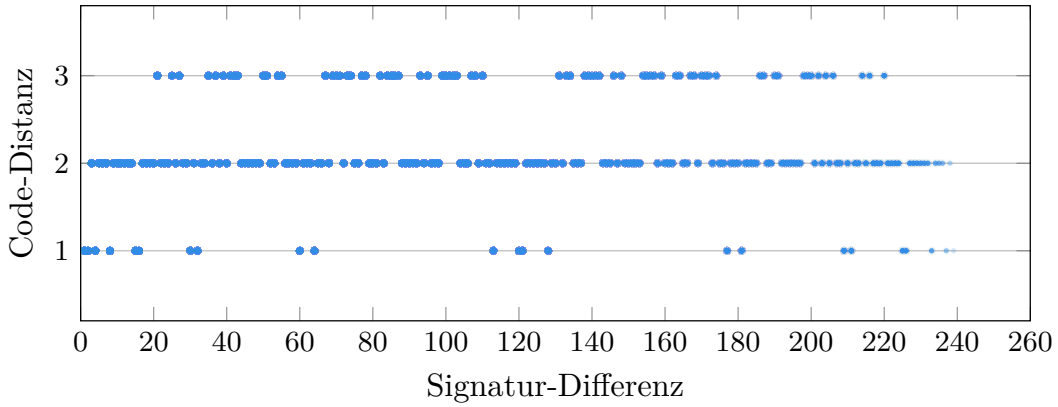
Aufgrund des Richtungsvektors kann vermutet werden, dass die Differenz zwischen den Signaturen eine Rolle bei der Musterbildung spielt. Abbildung 6.3 verdeutlicht die Beziehung zwischen Signatur-Differenzen und Code-Distanzen. Denn der Richtungsvektor $r = (1, 1, d_C)^T$ lässt vermuten, dass gleiche Differenzen stets zu derselben Code-Distanz führen - unabhängig davon, um welche Signaturen es sich dabei handelt.

Dies wird auch anhand der Abbildung deutlich. Die Differenz zwischen zwei Signaturen steht in direktem Zusammenhang mit der Code-Distanz zwischen den beiden Signaturen. Sind die Differenzen zwischen zwei Signatur-Paaren gleich, so sind auch die Code-Distanzen zwischen diesen Signatur-Paaren gleich: $B_1 - B_2 = B_3 - B_4 \implies d_C(B_1, B_2) = d_C(B_3, B_4)$, für $B_1, B_2, B_3, B_4 \in \{1, 2, ...255\}$, $B_1 > B_2$, $B_3 > B_4$ und $B_1 \neq B_2 \neq B_3 \neq B_4$.

Es kann daher vermutet werden, dass bestimmte Signatur-Differenzen abhängig vom Schlüssel robuste Bitmuster produzieren, welche eine große Code-Distanz zur Folge haben. Diese Vermutung unterstreicht die Bedeutung der Ergebnisse von Ulbrich [55]. Dieser stellte in seinen Arbeiten fest, dass abhängig vom Schlüssel besonders gute bzw. besonders schlechte Code-Distanzen entstehen, die die Fehlererkennungsleistung des Codes maßgeblich beeinflussen.



(a) Die Verteilung der Code-Distanz in Abhängigkeit der Signatur-Differenz für den Schlüssel $A = 185$.

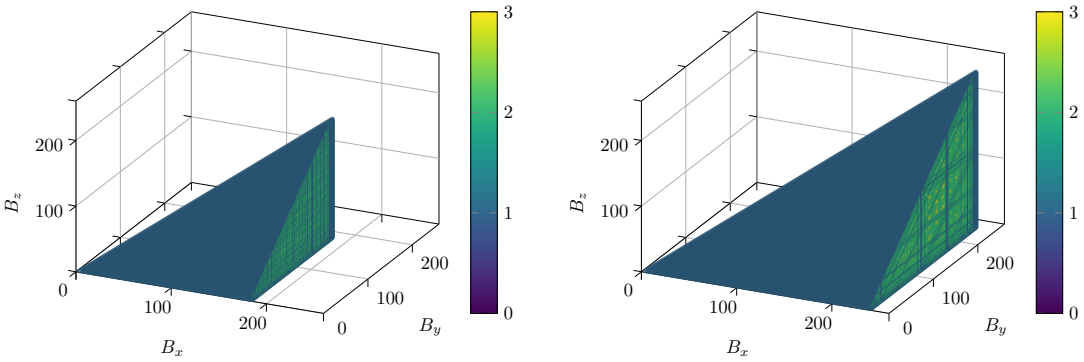


(b) Die Verteilung der Code-Distanz in Abhängigkeit der Signatur-Differenz für den Schlüssel $A = 241$.

Abbildung 6.3: Verteilung der Code-Distanz in Abhängigkeit der Signatur-Differenz. Die Grafiken zeigen die Code-Distanzen in Abhängigkeit von der Differenz zweier Signaturen für jeweils einen festen Schlüssel. Für die Differenzen zwischen den Signaturen eines Signatur-Paares und die Code-Distanz des Paares fällt dabei auf, dass eine Differenz stets nur eine bestimmte Code-Distanz erzeugt.

Eine explizite Rechenvorschrift für die Verteilung besonders guter Code-Distanzen konnte allerdings nicht ermittelt werden. Es kann jedoch festgehalten werden, dass für eine bekannte Code-Distanz für einen Schlüssel und zwei Signaturen aufgrund des immer gleichen Richtungsvektors der Geraden problemlos weitere Schlüssel-Signatur-Kombinationen mit derselben Code-Distanz ermittelt werden können. Somit kann zumindest eine Berechnung für größere Datentypen erheblich beschleunigt werden.

Die Vermutung, dass die Signatur-Differenz Einfluss auf die Code-Distanz nimmt, kann durch eine Beobachtung erhärtet werden. Anhand der Grafiken und der berechneten Daten ist zu erkennen, dass die Differenz der Signaturen keine Zweierpotenz sein darf. Dies



(a) Verteilung der Code-Distanzen für drei Signaturen für den Schlüssel $A = 185$.

(b) Verteilung der Code-Distanzen für drei Signaturen für den Schlüssel $A = 241$.

Abbildung 6.4: Verteilung der Code-Distanzen für drei Signaturen. Für jeweils einen festen Schlüssel werden Signatur-Kombinationen aus jeweils drei Signaturen gebildet und die Code-Distanz für diese Kombinationen berechnet. Auf der x-, y- und z-Achse sind die Signaturen B_x, B_y und B_z abgetragen. Die Code-Distanz wird in der Grafik farblich codiert als vierte Dimension dargestellt.

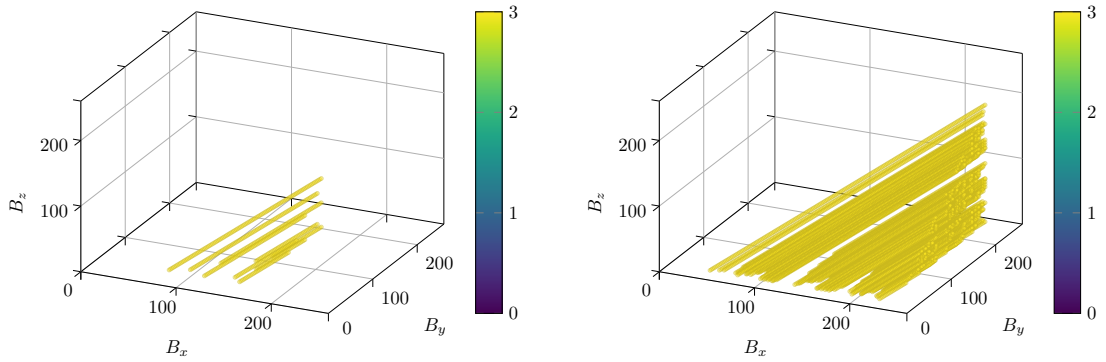
resultiert stets in $d_C = 1$ und unterstreicht damit bereits existierende Ergebnisse aus der Forschung (vgl. Abschnitt 3.2).

6.1.2 Verteilung der Code-Distanzen für drei Signaturen

Da für die Parametrisierung der CoRed-Implementierung allerdings drei Signaturen benötigt werden, wurde in einem zweiten Schritt eine dritte Signatur in den Berechnungen berücksichtigt. Als Schlüssel wurden erneut die Werte $A = 113$, $A = 185$, $A = 233$ und $A = 241$ gewählt. Auch hier sind die Ergebnisse für $A = 113$ und $A = 233$ dem Anhang zu entnehmen (vgl. Abschnitt A.1). In der Abbildung 6.4 sind auf der x-, der y- und der z-Achse die für die Berechnung der Code-Distanz verwendeten Signaturen abgetragen. Die resultierende Code-Distanz wird als vierte Dimension farblich dargestellt.

Die Signatur-Kombinationen für alle Grafiken in Abbildung 6.4 bilden Punktwolken in Form von schiefen, dreiseitigen Pyramiden. Die Mantelflächen der Pyramiden erscheinen vollständig dunkelblau, was eine Code-Distanz $d_C = 1$ bedeutet. Die Seitenflächen, die senkrecht zur xy-Ebene stehen, werden im Folgenden als Schnittflächen bezeichnet. Die Schnittflächen der Pyramiden sind grün gefärbt ($d_C = 2$) und mit dunkelblauen vertikalen und diagonalen Linien durchzogen ($d_C = 1$). Insgesamt erscheint die Wahl von Randwerten für Signaturen daher nicht sinnvoll. Darüber hinaus bedeutet die Wahl von „inneren“ Werten nicht notwendigerweise eine größere Code-Distanz.

Für die Schlüssel $A = 185$ und $A = 241$, die selbst eine Code-Distanz $d_C = 4$ aufweisen, sind vereinzelte gelbe ($d_C = 3$) Sprenkel in den Schnittflächen der Pyramiden zu erkennen. Für Abbildung 6.5 wird daher die Ergebnismenge der Signatur-Tripel und die resultierende



(a) Verteilung der Code-Distanzen für drei Signaturen gefiltert auf $d_C = 3$ für den Schlüssel $A = 185$.

(b) Verteilung der Code-Distanzen für drei Signaturen gefiltert auf $d_C = 3$ für den Schlüssel $A = 241$.

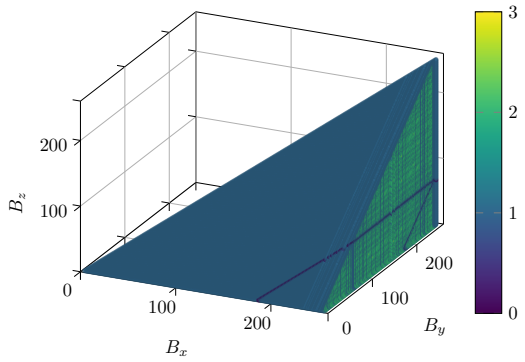
Abbildung 6.5: Verteilung der Code-Distanzen für drei Signaturen gefiltert auf $d_C = 3$. Für jeweils einen festen Schlüssel - links $A = 185$ und rechts $A = 241$ - werden Signatur-Kombinationen aus jeweils drei Signaturen gebildet und die Code-Distanz für diese Kombinationen berechnet. Das Ergebnis wird auf $d_C = 3$ gefiltert, um Muster erkennen zu können.

Code-Distanz auf $d_C = 3$ gefiltert, um ein genaueres Bild auf die Verteilung der besonders guten Code-Distanzen erhalten zu können.

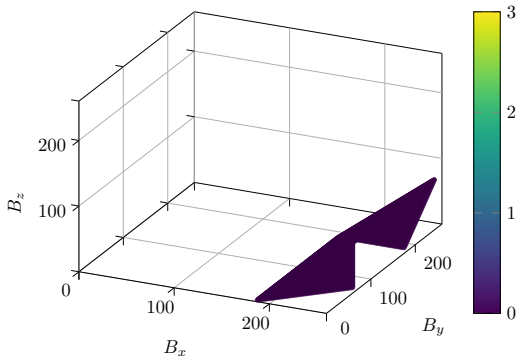
Nach der Filterung sind klare Regelmäßigkeiten zu erkennen. Die Code-Distanz in Abhängigkeit der drei Signaturen erzeugt Geraden im Raum, wie auch schon im Falle der zwei Signaturen. Auch diese Geraden sind parallel zueinander. Die Bestimmung der parallelen Geraden kann analog zu der Berechnung für zwei Signaturen erfolgen. Um den Richtungsvektor der längsten, am weitesten links gelegenen Gerade in Abbildung 6.5a ($A = 185$) ermitteln zu können, wird der Punkt mit dem kleinsten x-Wert sowie der Punkt mit dem größten z-Wert ausgewählt. Anhand dieser Punkte ergibt sich der Richtungsvektor $r = (1, 1, 1, d_C)^T$ für alle Geraden aufgrund der Parallelität. Die Geraden der xy-Komponenten und der xz-Komponenten lassen sich zusammensetzen zu einer Ebene im Raum. Es kann vermutet werden, dass sich der Ansatz auf dieselbe Art und Weise auch für n-fache Entscheider auf n-dimensionale Hyperebenen verallgemeinern lässt.

Dasselbe wird für die Schlüssel $A = 113$ und $A = 233$ mit der Code-Distanz $d_C = 2$ durchgeführt, welche für diese Schlüssel die maximale errechnete Code-Distanz ist. Diese Filterung zeigt allerdings, dass im Inneren der Pyramide lediglich eine sehr große grüne Punktwolke vorzufinden ist, die aufgrund ihrer Größe kaum Regelmäßigkeiten erkennen lässt. Aus diesem Grund sind diese Abbildungen lediglich im Anhang aufgeführt, da sie zu keiner zusätzlichen Erkenntnis führen (vgl. Abbildung A.4).

Letztlich handelt es sich bei den Signatur-Tripeln lediglich um eine Erweiterung der Signatur-Paare. Insbesondere ist die Code-Distanz des Tripels nichts anderes als das Minimum der Code-Distanzen der einzelnen Signatur-Paare des Tripels. Somit ist der Rich-



(a) Verteilung der Code-Distanzen für drei Signaturen für den Schlüssel $A = 185$ bei Missachtung der Regel $0 < B < A$ (vgl. **R3**)



(b) Verteilung der Code-Distanzen für drei Signaturen gefiltert auf $d_C = 0$ für den Schlüssel $A = 185$ bei Missachtung der Regel $0 < B < A$ (vgl. **R3**).

Abbildung 6.6: Verteilung der Code-Distanzen für drei Signaturen bei Missachtung der Regel $0 < B < A$ (vgl. R3). Für den Schlüssel $A = 185$ werden Signatur-Kombinationen aus jeweils drei Signaturen gebildet und die Code-Distanz für diese Kombinationen berechnet. Dabei wird auf die Einhaltung der Regel **R3** verzichtet. Das Ergebnis wird für die Grafik auf der rechten Seite zusätzlich auf $d_C = 0$ gefiltert, um die Verteilung von $d_C = 0$ besser erkennen zu können. Anhand der Grafiken wird deutlich, dass diese Regel auch für die Wahl der Signaturen von Relevanz ist.

tungsvektor $r = (1, 1, 1, d_C)^T$ sowie die Parallelität der Geraden nicht weiter überraschend. Wird die Differenz aus den Signatur-Paaren berechnet, die das Minimum der Code-Distanzen erzeugen, so existiert auch für Signatur-Tripel die zuvor beschriebene Beziehung zwischen Signatur-Differenz und Code-Distanz.

Zuletzt ist festzustellen, dass eine Code-Distanz $d_C = 0$ für keine gültige Schlüssel-Signatur-Kombination erreicht wird. Dies ist mit der Einhaltung der aus der Literatur bekannten Regeln zu erklären. Wird beispielsweise auf die Beachtung der Regel $0 < B < A$ (vgl. **R3**) verzichtet, so sind auch Kombinationen zu erkennen, welche zu einer Code-Distanz $d_C = 0$ führen. Dies ist immer dann der Fall, wenn die Differenz zwischen den Signaturen ein Vielfaches von A beträgt. Daher kann eine Code-Distanz $d_C = 0$ nicht unter Beachtung von **R3** auftreten. Diese Beobachtungen unterstützen daher die Ergebnisse von U. Schiffel [47], dass alle Signaturen so gewählt werden müssen, dass sie kleiner als der Schlüssel sind (vgl. Abschnitt 3.2). Darüber hinaus bekräftigt diese Beobachtung die Vermutung, dass die Differenz zwischen Signaturen für die Betrachtung der Code-Distanz zwischen den Signaturen eine wichtige Rolle spielt.

Insgesamt ist festzuhalten, dass aufgrund der Geraden-Bildung sowohl für Signatur-Paare als auch für Signatur-Tripel Muster zu erkennen sind. Eine Erklärung können potenziell die Distanzen zwischen den Signaturen der Signatur-Paare liefern. Rechenvorschriften zur einfachen Ermittlung besonders geeigneter Signaturen konnten über diese Feststellung

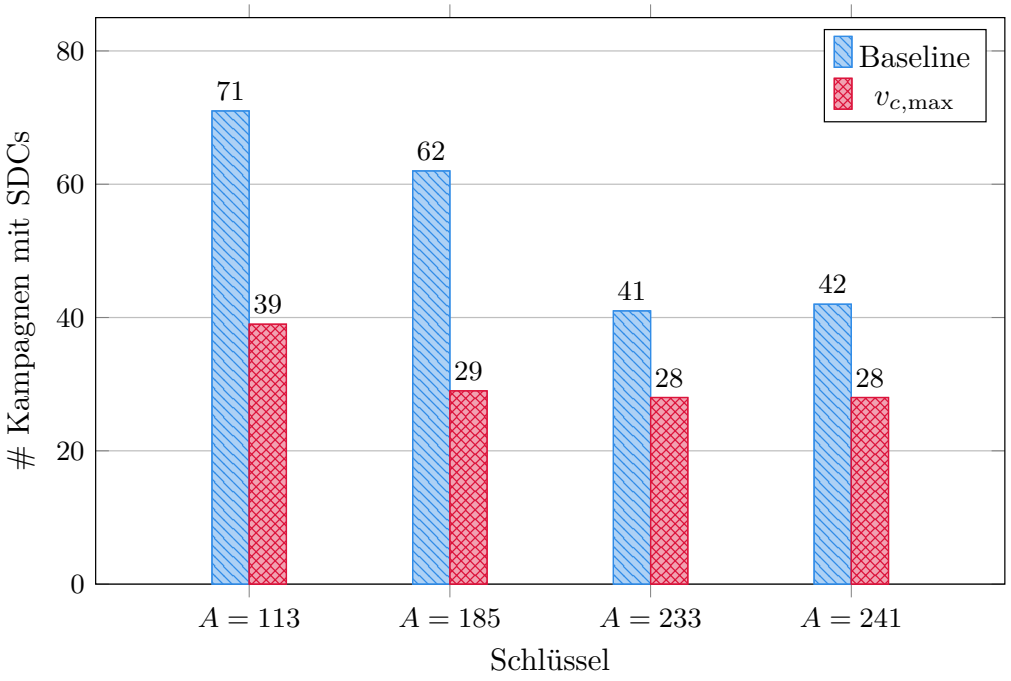


Abbildung 6.7: Reduktion unerkannter Datenfehler durch $v_{c,max}$. Das Säulendiagramm zeigt die Anzahl Schlüssel-Signatur-Kombinationen aus der Stichprobe, die zu unerkannten Datenfehlern führen, für die Baseline-Implementierung und die $v_{c,\max}$-Implementierung im Vergleich.

hinaus nicht abgeleitet werden. Falls die Code-Distanz einer Signatur-Kombination bekannt ist, können allerdings aufgrund der Muster weitere Signatur-Kombinationen mit derselben Code-Distanz einfach berechnet werden.

6.2 Fallstrick 1: Darstellung von Codes im Binärsystem

Da im Zuge des Fallstricks **F1** die Wirksamkeit der Definition von $v_{c,\max} = A \cdot \text{INT\_MAX} + B_x$ überprüft werden soll, wird die Baseline-Implementierung mit der $v_{c,\max}$-Implementierung verglichen. Dazu werden, wie in Abschnitt 5.2.3 beschrieben, Stichproben der Größe 1.000 für die Schlüssel $A = 113$, $A = 185$, $A = 233$ und $A = 241$ gezogen. Diese Stichproben enthalten Schlüssel-Signatur-Kombinationen, mit welchen die Implementierung parametrisiert wird. Diese parametrisierten Implementierungen werden dann mithilfe von FAIL* injiziert.

Die Ergebnisse der Fehlerinjektionen werden auf die Anzahl unentdeckter Datenfehler im Vergleich zur Baseline-Implementierung untersucht. Dabei soll betrachtet werden, wie viele Schlüssel-Signatur-Kombinationen zu unentdeckten Datenfehlern führen. Die Ergebnisse sind in Abbildung 6.7 dargestellt. Das Säulendiagramm zeigt die Anzahl der Kampagnen mit unentdeckten Datenfehlern in Abhängigkeit vom gewählten Schlüssel. Pro

Schlüssel werden die Baseline-Implementierung (blau gestrichelte Säule) und die $v_{c,\max}$-Implementierung (rot schraffierte Säule) verglichen.

Dabei wird deutlich, dass die $v_{c,\max}$-Implementierung die Anzahl der Kampagnen mit unentdeckten Datenfehlern deutlich reduziert. Für die beiden kleinere Schlüssel $A = 113$ und $A = 185$ wird sie um etwa 50% reduziert. Für die größeren Schlüssel $A = 233$ und $A = 241$ fällt die Anzahl der unentdeckten Datenfehler bereits in der Baseline-Implementierung geringer aus. Dadurch ist der Effekt hier nicht ganz so stark; mit einer Reduktion von etwa einem Drittel fällt er allerdings immer noch deutlich aus. Damit ist die Anpassung der Implementierung sowie die Definition von $v_{c,\max}$ anhand der Daten als wirksam zu bewerten.

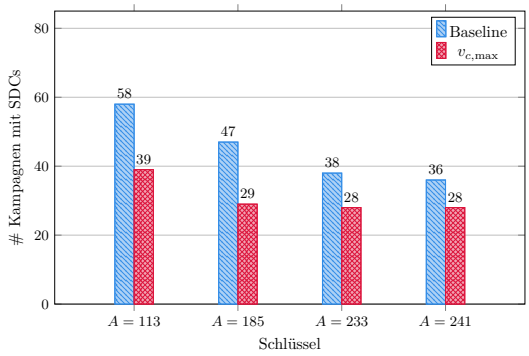
Aufgrund der geringen Größe der Stichprobe stellt sich allerdings die Frage, ob die Ergebnisse aussagekräftig genug sind. Dabei ist insbesondere anzumerken, dass die Unterschiede zwischen den Ergebnissen der beidem Implementierung deutlich größer als die für die Stichprobe errechnete Fehlerspanne von $3,1\%$ ausfallen. Auch wenn die tatsächlichen Werte der Baseline-Implementierung in etwa um die Fehlerspanne nach unten und die Werte der $v_{c,\max}$-Implementierung im gleichen Maße nach oben abweichen würden, wäre der Unterschied zwischen den beiden Implementierungen weiterhin deutlich erkennbar. Daher ist es sehr wahrscheinlich, dass sich die tatsächlichen Werte in einem ähnlichen Rahmen bewegen und die Änderung der Implementierung sinnvoll ist.

Aus diesem Grund bestätigen die Ergebnisse dieser Arbeit eindeutig die Ergebnisse von Hoffmann u. a. [24]. Dabei verbleiben für einige Schlüssel-Signatur-Kombinationen allerdings wenige unentdeckte Datenfehler.

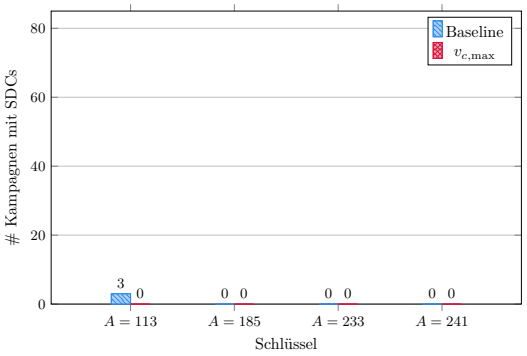
Bei einer genaueren Untersuchung der unentdeckten Datenfehler auf die Injektionsstellen wird deutlich, dass die meisten unentdeckten Datenfehler bei einer Injektion in den Befehlszähler oder in das Statusregister auftreten (vgl. Abbildung 6.8). Das größte verbleibende Problem sind also unentdeckte Kontrollflussfehler. Aufgrund der vergleichsweise geringen Anzahl von Instruktionen führen Kontrollflussfehler häufig aus dem Ausführungskontext heraus. In diesem Fall greifen Mechanismen des Betriebssystems oder der Hardware und signalisieren einen Fehler. Kontrollflussfehler, die für Sprünge innerhalb des Ausführungskontextes sorgen, sind besonders schwerwiegend und außerdem schwer zu entdecken.

Bei einer Injektion in den Speicher und die frei verwendbaren Register der Baseline-Implementierung sind ebenfalls Schlüssel-Signatur-Kombinationen mit unentdeckten Datenfehlern zu beobachten. Dies ist bei der $v_{c,\max}$-Implementierung nicht mehr der Fall. Eine Injektion in den Speicher oder die Register der $v_{c,\max}$-Implementierung führt bei keiner untersuchten Schlüssel-Signatur-Kombination zu unentdeckten Datenfehlern.

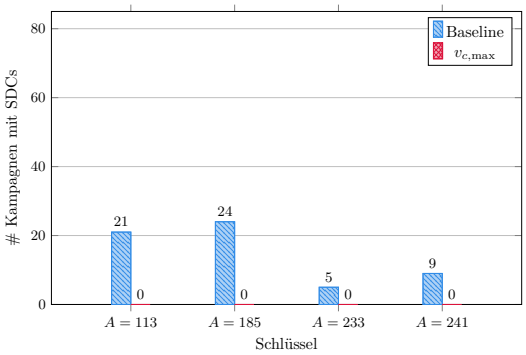
Untersucht man die unentdeckten Fehler weiter, fällt auf, dass die Anzahl der unentdeckten Kontrollflussfehler pro Kampagne selbst für die Baseline-Implementierung bereits relativ gering ausfällt (vgl. Tabelle 6.1). So bewegt sich die Anzahl der unentdeckten Kontrollflussfehler pro Kampagne im unteren einstelligen Bereich. Für einige Schlüssel kann



(a) Anzahl Schlüssel-Signatur-Kombinationen mit unerkannten Datenfehlern bei Injektion in den Befehlszähler.



(b) Anzahl Schlüssel-Signatur-Kombinationen mit unerkannten Datenfehlern bei Injektion in den Speicher.



(c) Anzahl Schlüssel-Signatur-Kombinationen mit unerkannten Datenfehlern bei Injektion in frei verwendbare Register.

Abbildung 6.8: Reduktion unerkannter Datenfehler durch $v_{c,\max}$ aufgeteilt nach Injektionsstelle der Fehlerinjektion. Das Säulendiagramm zeigt die Anzahl Schlüssel-Signatur-Kombinationen aus der Stichprobe, die zu unerkannten Datenfehler führen, für die Baseline-Implementierung und die $v_{c,\max}$-Implementierung im Vergleich. Dabei wird die Anzahl der unerkannten Datenfehler abhängig von der Injektionsstelle bei der Fehlerinjektion dargestellt.

die $v_{c,\max}$-Implementierung die Anzahl etwas weiter reduzieren. Treten überhaupt noch unentdeckte Kontrollflussfehler auf, sind es im schlimmsten Fall drei in einer Kampagne.

Die $v_{c,\max}$-Implementierung löst also das Problem unerkannter Fehler, die durch Injektionen in den Speicher sowie durch Injektionen in die Register auftreten. Für einige Schlüssel-Signatur-Kombinationen besteht jedoch noch ein Problem mit Kontrollflussfehlern. Die Anzahl der Schlüssel-Signatur-Kombinationen mit Kontrollflussfehlern sowie die Anzahl der Kontrollflussfehler pro Kampagne fallen in der $v_{c,\max}$-Implementierung allerdings bereits deutlich geringer aus als in der Baseline-Implementierung. Damit ist die Definition von $v_{c,\max}$ insgesamt als sinnvoll zu bewerten. Die Anzahl der Kontrollflussfeh-

A	Implementierung	min{ip}	max{ip}
113	Baseline	1	3
113	$v_{c,\max}$	1	3
185	Baseline	1	5
185	$v_{c,\max}$	1	2
233	Baseline	1	2
233	$v_{c,\max}$	1	2
241	Baseline	1	5
241	$v_{c,\max}$	1	2

Tabelle 6.1: Minimale und maximale Anzahl von Kontrollflussfehlern pro Kampagne der beiden Implementierungen im Vergleich. Die Tabelle zeigt die minimalen (Spalte $min\{ip\}$) und maximalen (Spalte $max\{ip\}$) Anzahlen von Kontrollflussfehlern in einer Kampagne, für die überhaupt noch unentdeckte Fehler existieren. Dabei wird abhängig vom verwendeten Schlüssel sowie der verwendeten Implementierung unterschieden. Die Baseline-Implementierung hat für die Schlüssel $A = 185$ und $A = 241$ eine etwas höhere Anzahl unentdeckter Kontrollflussfehler als die $v_{c,\max}$-Implementierung.

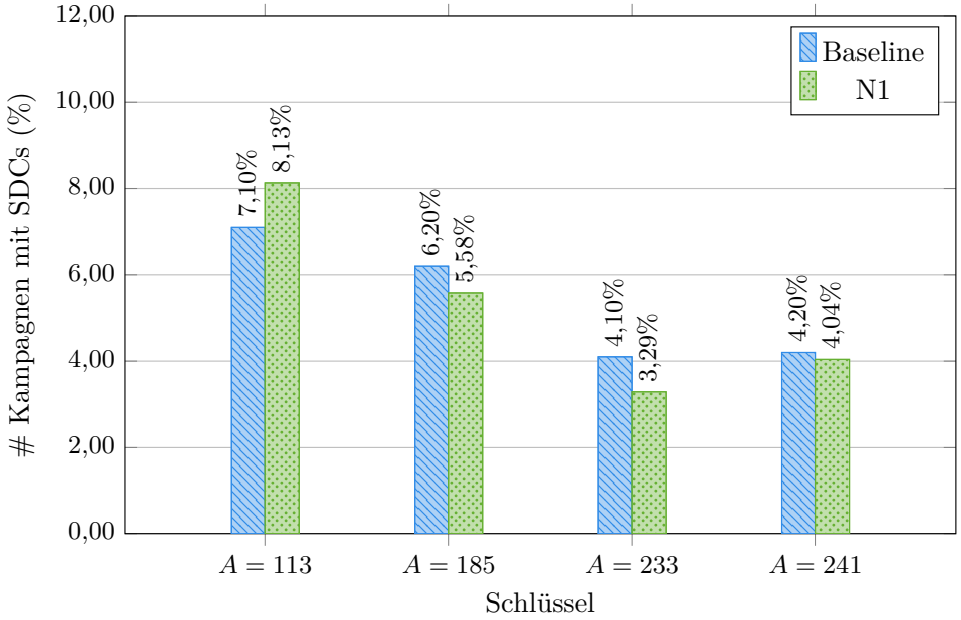
ler kann potenziell mithilfe der im Folgenden untersuchten Maßnahmen weiter reduziert werden.

6.3 Fallstrick 2: Zustand zwischen Instruktionen

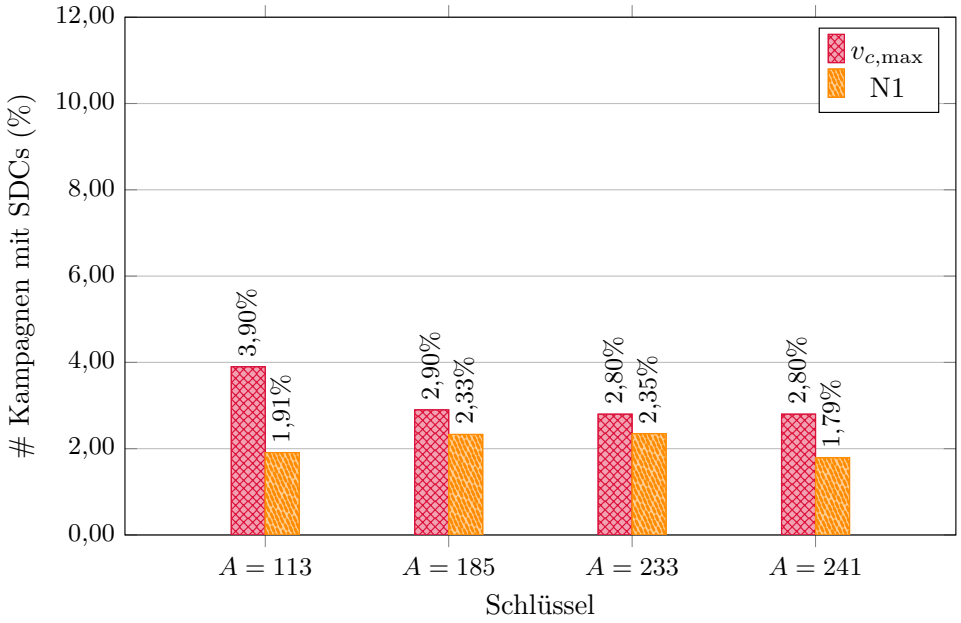
Um die Probleme des Fallstricks **F2** zu lösen, führen Hoffmann u. a. [24] eine zusätzliche Abfrage in der `apply()`-Funktion ein. In dieser Arbeit wird eine strengere Formulierung dieser Abfrage zusammen mit einer Anpassung der Parametrisierung (vgl. Abschnitt 4.2) untersucht. Dies wird unter der Regel **N1** zusammengefasst.

Wie bereits bei der Auswertung zum Fallstrick **F1** soll auch hier zunächst der Fokus darauf liegen, wie viele Schlüssel-Signatur-Kombinationen mindestens einen unentdeckten Datenfehler verursachen. Dazu werden im Rahmen des Versuchsaufbaus zwei Ansätze verfolgt (vgl. Abschnitt 5.2.3), um die Aussagekraft der Ergebnisse überprüfen zu können.

Im ersten Schritt wird eine Stichprobe pro Schlüssel verwendet. Diese Stichprobe wird mithilfe der Regel **N1** so reduziert, dass sie nur noch Schlüssel-Signatur-Kombinationen enthält, die die Signatur-Auswahl-Regel **N1** $((B_x - B_y) + (B_x - B_z) < B_z)$ erfüllen (vgl. Stichprobenverfahren 1 (V1)). Aufgrund des reduzierten Stichprobenumfangs der Versuche zu **N1** können an dieser Stelle keine absoluten Werte für die Anzahl der Kampagnen mit unerkannten Datenfehlern verwendet werden. Stattdessen wird die Anzahl der Kampagnen mit unerkannten Datenfehlern als prozentualer Anteil in Bezug auf die Größe der Stichprobe angegeben.



(a) Wirksamkeit der Einführung von **N1** für die Baseline-Implementierung.



(b) Wirksamkeit der Einführung von **N1** für die $v_{c,\max}$-Implementierung.

Abbildung 6.9: Wirksamkeit der Einführung von N1 (Stichprobenverfahren 1 (V1)). Das Säulendiagramm zeigt den Anteil der Schlüssel-Signatur-Kombinationen aus der Stichprobe, der zu unerkannten Datenfehlern führt, für die **Baseline-Implementierung** (oben) und die **$v_{c,\max}$-Implementierung** (unten) unter Berücksichtigung von **N1**. Es wurde das Stichprobenverfahren 1 (V1) angewendet.

Abbildung 6.9 zeigt die Auswirkungen der Änderung. Sowohl für die Baseline-Implementierung als auch für die $v_{c,\max}$-Implementierung ist nach der Einführung von **N1** eine Reduktion der Schlüssel-Signatur-Kombinationen mit unentdeckten Datenfehlern zu beobachten. Eine Ausnahme bildet lediglich der Schlüssel $A = 113$ in der Baseline-Implementierung.

Insbesondere unter Verwendung der $v_{c,\max}$-Implementierung ist die Reduktion der unentdeckten Datenfehler gut erkennbar. Das ist vermutlich dadurch zu erklären, dass in der Baseline-Implementierung die Wirksamkeit der Anpassung teilweise durch andere Fehler überdeckt wird.

Dabei ist zu berücksichtigen, dass alle Unterschiede zwischen den Baseline-Werten bzw. $v_{c,\max}$-Werten und den **N1**-Werten innerhalb der für den Stichprobenumfang berechneten Fehlerspanne $3,1\%$ liegen. Das bedeutet, dass der tatsächliche Wert jeweils um $3,1\%$ nach oben oder nach unten abweichen und somit zu einem anderen Ergebnis führen kann. Dennoch ist insgesamt eine klare Tendenz in den Daten erkennbar. Zudem ist die Anzahl der Schlüssel-Signatur-Kombinationen mit unentdeckten Datenfehlern für die Baseline bzw. die $v_{c,\max}$-Implementierung ohne Berücksichtigung von **N1** selbst bereits sehr gering, weshalb große Unterschiede kaum zu erwarten sind. Es kann daher vermutet werden, dass die Einführung von **N1** zu einer Verbesserung führt.

Andererseits ist ein gewisser Stichprobenfehler (engl. *Selection Bias*) nur schwer auszuschließen. Darüber hinaus kann die Reduzierung der Stichprobe um die Kombinationen, die nicht **N1** erfüllen, die Ergebnisse sowohl positiv als auch negativ beeinflussen. Zur Überprüfung der ermittelten Werte wird daher ein zweites Stichprobenverfahren angewendet (vgl. Stichprobenverfahren 2 (V2)). Dabei wird wiederum eine Stichprobe aus der Grundgesamtheit jedes Schlüssels gezogen. Es werden jedoch nur Schlüssel-Signatur-Kombinationen berücksichtigt, die **N1** erfüllen.

Die Ergebnisse des zweiten Verfahrens sind in Abbildung 6.10 dargestellt. Auch hier ist eine Tendenz in den Daten zu erkennen, die vermuten lässt, dass die Einführung von **N1** das Auftreten unentdeckter Datenfehler reduzieren kann. Die Werte sind wiederum in Prozent angegeben, um einen Vergleich mit Abbildung 6.9 zu erleichtern. Bei diesem Ansatz ist für den Schlüssel $A = 233$ eine Abweichung nach oben zu erkennen. Für den Schlüssel $A = 113$ fehlt sie hingegen.

Die Tabelle 6.2 vergleicht die beiden Stichprobenverfahren für die Versuche zu **N1** miteinander. In Spalte eins ist der verwendete Schlüssel aufgeführt. Spalte zwei enthält die verwendete Regel und in Klammern dahinter die verwendete Implementierung. In Spalte drei ist der Anteil der Schlüssel-Signatur-Kombinationen mit unentdeckten Datenfehlern für das Stichprobenverfahren 1 (V1) aufgelistet; in Spalte vier ist der Anteil für das Stichprobenverfahren 2 (V2) zu sehen. Die letzte Spalte enthält die Differenzen der beiden vorhergehenden Spalten in Prozentpunkten (%p).

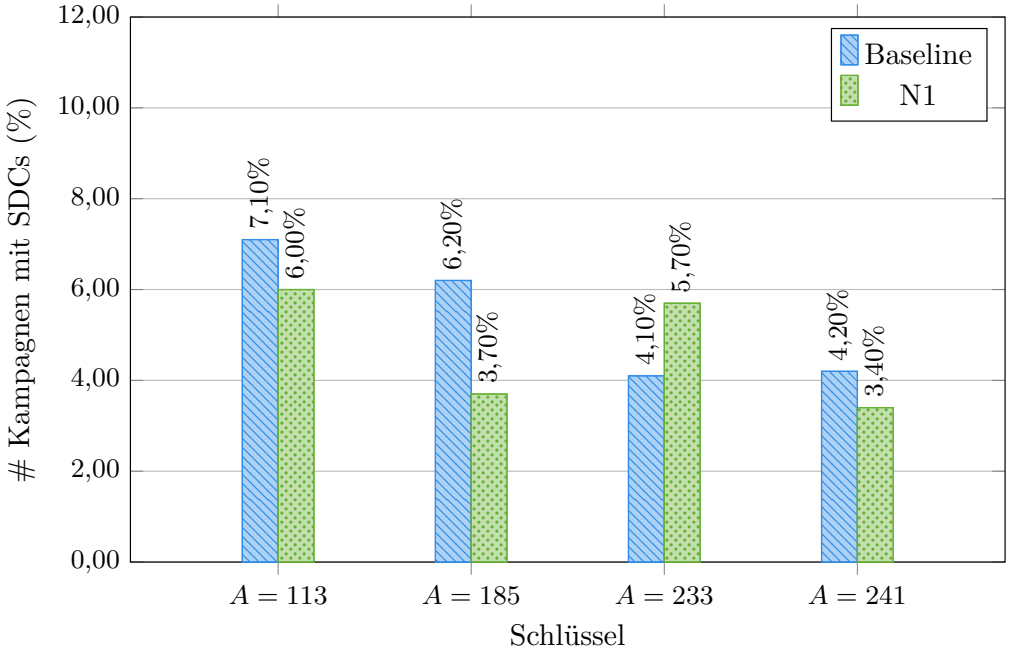


Abbildung 6.10: Wirksamkeit der Einführung von N1 (Stichprobenverfahren 2 (V2)). Das Säulendiagramm zeigt den Anteil der Schlüssel-Signatur-Kombinationen aus der Stichprobe, der zu unerkannten Datenfehler führt, für die **Baseline-Implementierung** unter Berücksichtigung von **N1**. Es wurde das Stichprobenverfahren 2 (V2) angewendet.

A	Regel	# SDCs (%) (**V1**)	# SDCs(%) (**V2**)	δ **V1** und **V2** (%p)
113	**N1** (Baseline)	8,13%	6,00%	2,13%p
185	**N1** (Baseline)	5,58%	3,70%	1,88%p
233	**N1** (Baseline)	3,29%	5,70%	2,41%p
241	**N1** (Baseline)	4,04%	3,40%	0,64%p

Tabelle 6.2: Vergleich zwischen V1 und V2 zur Bewertung der Wirksamkeit von N1. Die Tabelle vergleicht die beiden Stichprobenverfahren für die Versuche zu **N1** miteinander.

Vergleicht man die Abweichungen der beiden Stichprobenverfahren, so wird deutlich, dass die Differenzen immer kleiner als die Fehlerspanne 3,1% sind (vgl. Spalte δ ***V1 und V2 (%p)***). Prinzipiell ist es möglich, dass die Datenlage täuscht und die tatsächlichen Werte nicht auf eine Wirksamkeit von **N1** hinweisen. Da jedoch die Abweichungen zwischen den Stichprobenverfahren relativ gering sind und die Daten alle in eine ähnliche Richtung weisen, kann vermutet werden, dass die Stichproben trotz ihrer geringen Größe repräsentativ sind.

An dieser Stelle ist außerdem anzumerken, dass für die Schlüssel $A = 113$ und $A = 233$ vergleichsweise große Abweichungen zwischen den Stichprobenverfahren festgestellt wurden

(vgl. Tabelle 6.2). Die Abweichungen in Prozentpunkten (%p) bewegen sich dabei allerdings weiterhin in einem Bereich unterhalb der Fehlerspanne. Sie können allerdings dennoch eine Erklärung für die beiden Ausreißer in den Grafiken 6.9 und 6.10 sein.

Bei der Tabelle handelt es sich lediglich um einen Auszug der Daten. Die vollständigen Übersichten aller Fehlerinjektionsversuche können dem Anhang A.2 entnommen werden.

Untersucht man die unentdeckten Datenfehler anhand der Injektionsstelle, fällt auch hier wieder auf, dass für die Baseline-Implementierung noch Fehler aufgrund von Injektionen in den Speicher und in Register auftreten. Diese sind für die $v_{c,\max}$-Implementierung nicht mehr zu beobachten. Hier treten „nur“ noch unentdeckte Fehler bei einer Injektion in den Befehlszähler auf.

Die minimale und maximale Anzahl unentdeckter Datenfehler reduziert sich für die Baseline-Implementierung für den Schlüssel $A = 185$ von fünf auf drei und für $A = 241$ von fünf auf zwei. Für die v, cmax-Implementierung ist keine Veränderung zu erkennen. Aufgrund der geringen Effekte sind die Tabellen lediglich im Anhang aufgeführt (vgl. Tabellen A.15 und A.16).

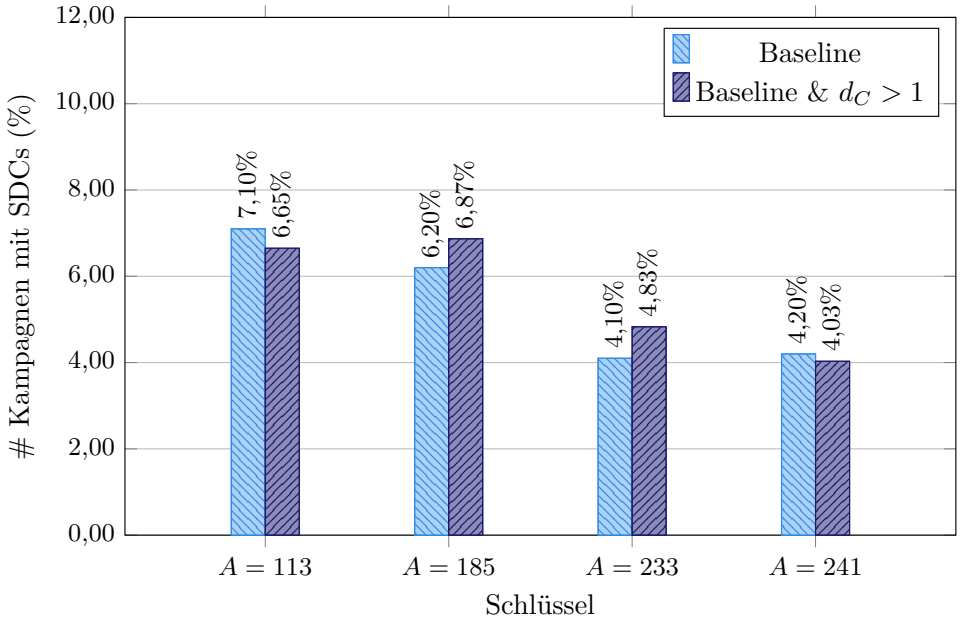
Beide Versuchsarten lassen insgesamt vermuten, dass die Berücksichtigung von **N1** zur Reduktion der unentdeckten Datenfehler beitragen kann. Effektiv verkleinert **N1** die Distanz zwischen den Signaturen und damit auch die gültigen Coderäume ein wenig. Dafür reduziert sich allerdings auch der Bereich, in dem die im Mehrheitsentscheid errechnete dynamische Signatur liegen darf. Eine Verfälschung der dynamischen Signatur aufgrund eines Kontrollflussfehlers kann so eher auffallen. Dieser Vorteil scheint den Nachteil des etwas kleineren Coderaums zu überwiegen.

Allerdings ist dieses Ergebnis aufgrund der geringen Unterschiede zwischen den beiden Benchmarks sowie der kleinen Stichprobengröße vorsichtig zu bewerten. Der Abschnitt 6.5 beschreibt sowohl die Probleme kleiner Stichprobengrößen sowie weitere Einschränkungen der Experimente genauer.

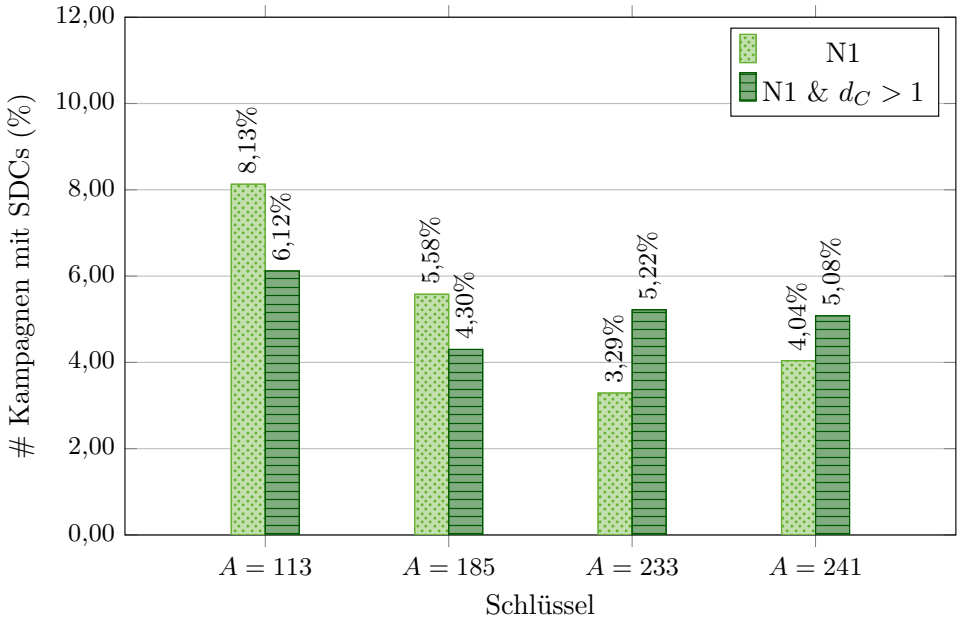
6.4 Code-Distanz zwischen den Signaturen

Mit den Erkenntnissen aus Abschnitt 6.1 ist nun noch zu klären, ob die Beachtung der Code-Distanz zwischen den Signaturen einen positiven Effekt auf die Anzahl der Kampagnen mit unentdeckten Datenfehler hat. Wie im vorherigen Abschnitt werden auch hier die Stichprobenverfahren **V1** und **V2** eingesetzt und verglichen, um eine bessere Vorstellung über die Aussagekraft der Ergebnisse bei vergleichsweise kleiner Stichprobengröße zu erhalten. Darüber hinaus wird die Regel **N1** mit in die Auswertung einbezogen, da im vorherigen Abschnitt eine leicht positive Tendenz in Bezug auf die Reduktion unentdeckter Datenfehler zu erkennen war.

Die Ergebnisse der Fehlerinjektion für das Stichprobenverfahren 1 (V1) sind in Abbildung 6.11 für die Baseline-Implementierung und in Abbildung 6.12 für die $v_{c,\max}$-Im-

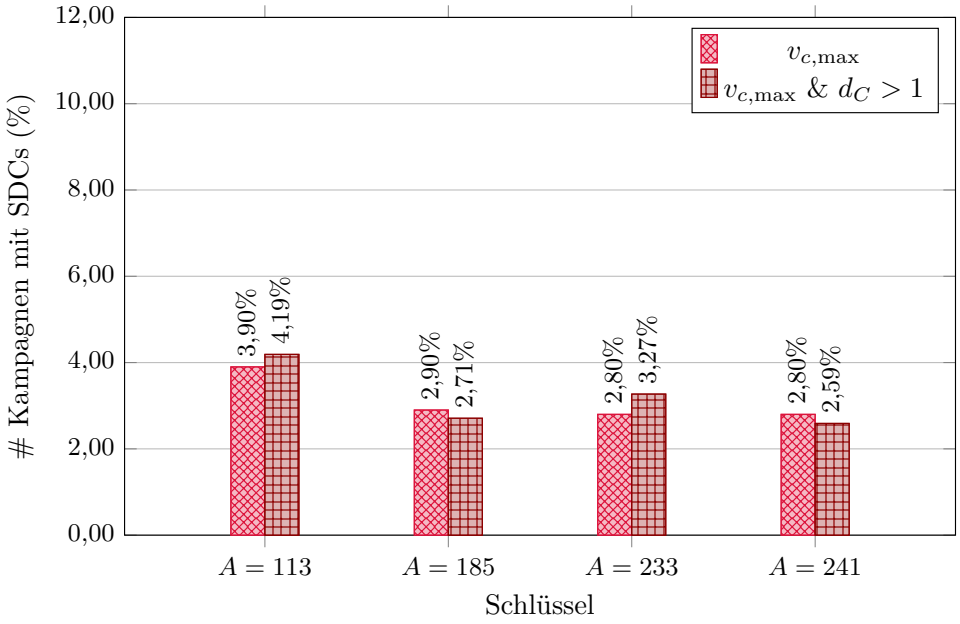


(a) Auswirkungen von Code-Distanzen zwischen den Signaturen auf Kampagnen mit SDCs in der Baseline-Implementierung.

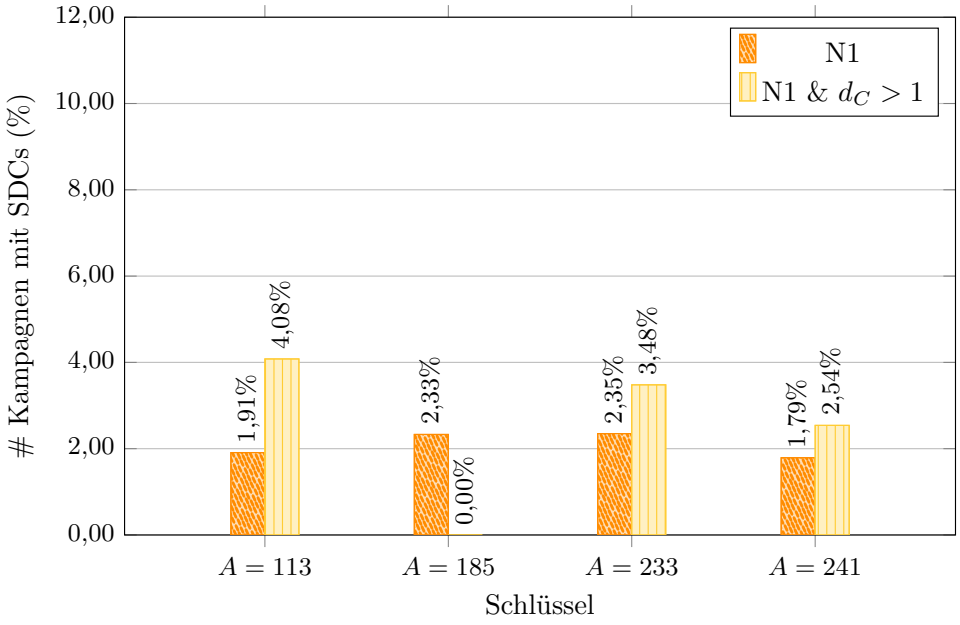


(b) Auswirkungen von Code-Distanzen zwischen den Signaturen auf Kampagnen mit SDCs in der Baseline-Implementierung unter Berücksichtigung von **N1**

Abbildung 6.11: Auswirkungen von Code-Distanzen zwischen den Signaturen auf Kampagnen mit SDCs (Stichprobenverfahren 1 (V1)) für die Baseline-Implementierung. Das Säulendiagramm zeigt den Anteil Schlüssel-Signatur-Kombinationen aus der Stichprobe, der zu unerkannten Datenfehlern führt. Dabei wird die **Baseline-Implementierung** einmal ohne (oben) und mit Beachtung von **N1** (unten) verwendet.



(a) Auswirkungen von Code-Distanzen zwischen den Signaturen auf SDCs in der $v_{c,\max}$-Implementierung



(b) Auswirkungen von Code-Distanzen zwischen den Signaturen auf Kampagnen mit SDCs in der $v_{c,\max}$-Implementierung unter Berücksichtigung von **N1**

Abbildung 6.12: Auswirkungen von Code-Distanzen zwischen den Signaturen auf Kampagnen mit SDCs (Stichprobenverfahren 1 (V1)) für die $v_{c,max}$-Implementierung. Das Säulendiagramm zeigt den Anteil Schlüssel-Signatur-Kombinationen aus der Stichprobe, der zu unerkannten Datenfehlern führt. Dabei wird die **$v_{c,max}$-Implementierung** einmal ohne (oben) und mit Beachtung von **N1** (unten) verwendet.

plementierung abgebildet. Zunächst sollen die Ergebnisse der Baseline-Implementierung betrachtet werden.

Insgesamt erscheinen die Auswirkungen unter Verwendung der Stichprobenverfahren 1 (V1) für die Baseline-Implementierung eher indifferent. Es kann keine klare Tendenz ausgemacht werden, ob die Einschränkung auf eine gute Code-Distanz zwischen den Signaturen eine insgesamt positive oder negative Veränderung in der Anzahl der Kampagnen mit unentdeckten Datenfehlern hervorruft. Stattdessen schwanken die Werte.

Dabei kann auch keine Regelmäßigkeit unter Berücksichtigung der Code-Distanz des Schlüssels erkannt werden. Während für den Schlüssel $A = 185$ ein leichter Anstieg von der Baseline hin zu $d_C > 1$ zu erkennen ist, fällt die Anzahl der Kampagnen mit SDCs für den Schlüssel $A = 241$ leicht ab. Beide Schlüssel haben eine Code-Distanz von $d_C = 4$. Für die Schlüssel $A = 113$ und $A = 231$ mit einer Code-Distanz von $d_C = 2$ sieht es ähnlich aus.

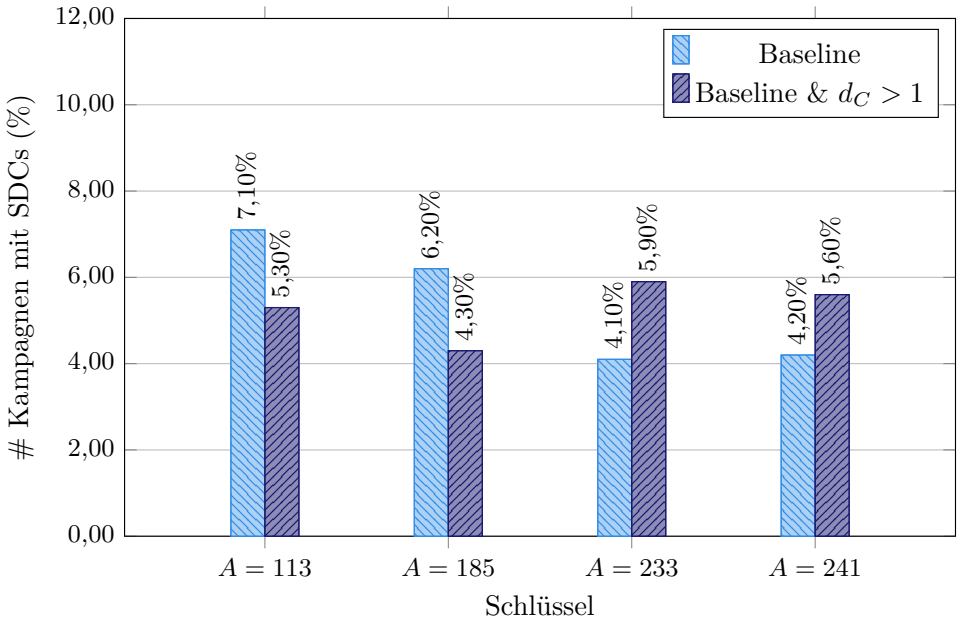
Unter Berücksichtigung von **N1** werden lediglich die Abstände zwischen den Säulen der jeweiligen Schlüssel etwas größer. Eine klare Tendenz, ob die Berücksichtigung der Distanz zwischen den Signaturen die Fehlererkennungsleistung verbessert, ist auch hier nicht zu erkennen. Ganz im Gegenteil: Während z.B. für den Schlüssel $A = 241$ noch ein leichter Abfall des Anteils der Kampagnen mit SDCs für Signatur-Kombinationen mit $d_C > 1$ zu erkennen war, steigt diese unter Berücksichtigung von **N1** leicht an.

Betrachtet man die Abbildungen für die $v_{c,\max}$-Implementierung (vgl. 6.12), ergibt sich ein ähnliches Bild. Auch hier ist weder für den Benchmark Baseline noch unter Berücksichtigung des Benchmarks **N1** eine klare Tendenz auszumachen. Allerdings steigt auch hier jeweils die Differenz zwischen den Säulen unter Berücksichtigung von **N1**.

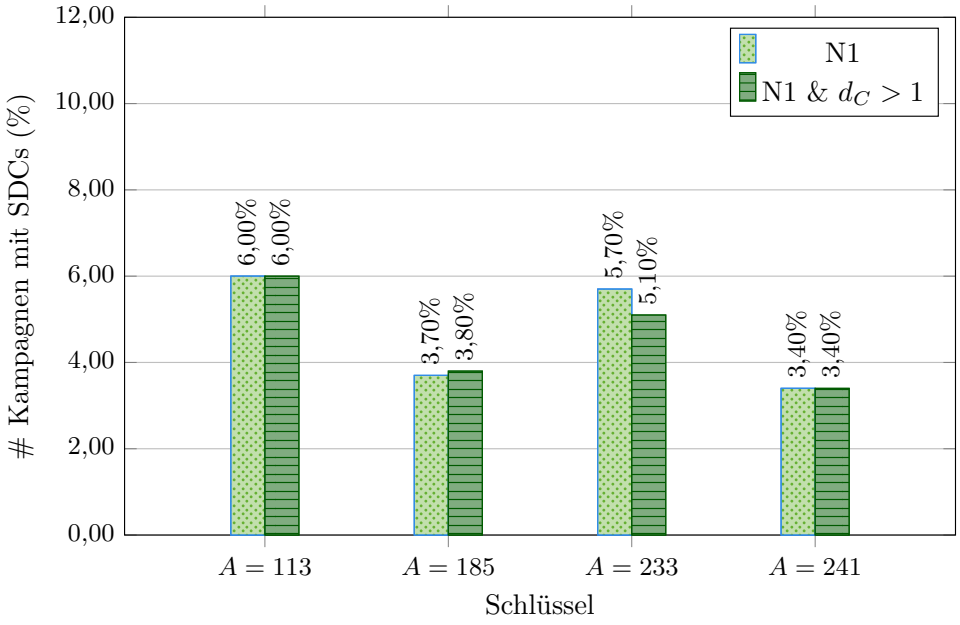
Für die Baseline-Implementierung im Stichprobenverfahren 2 (V2) ist die Situation vergleichbar. Betrachtet man Abbildung 6.13 für die Baseline-Implementierung genauer, so ist für die Schlüssel $A = 113$ und $A = 185$ eine eindeutige Verbesserung der Kampagnen mit unentdeckten Datenfehlern zu sehen. Für die Schlüssel $A = 233$ und $A = 241$ hingegen wird eine eindeutige Verschlechterung in gleichem Maße deutlich. Damit sind auch an dieser Stelle insbesondere keine Regelmäßigkeiten in Abhängigkeit von der Code-Distanz des Schlüssels zu erkennen. Diese liegt für $A = 113$ und $A = 233$ bei zwei; für $A = 185$ und $A = 241$ liegt sie bei vier. Für den Benchmark **N1** sind kaum noch Unterschiede zwischen den Säulen der verschiedenen Schlüssel zu erkennen. Somit ist das Ergebnis zu dieser Untersuchung divers.

Auch für diese Messungen sind weitere Informationen bezüglich der Abweichungen zwischen den Stichprobenverfahren im Anhang zu finden (vgl. Abschnitt A.2). Alle in diesem Abschnitt vorgestellten Werte haben Abweichungen, die kleiner als die Fehlerspanne sind.

Im Anhang sind weitere Tabellen aufgeführt. Diese beschreiben die minimale und maximale Anzahl unentdeckter Kontrollflussfehler für die Kampagnen der Stichprobe, die noch



(a) Auswirkungen von Code-Distanzen zwischen den Signaturen auf Kampagnen mit SDCs in der Baseline-Implementierung.



(b) Auswirkungen von Code-Distanzen zwischen den Signaturen auf Kampagnen mit SDCs in der Baseline-Implementierung unter Berücksichtigung von **N1**

Abbildung 6.13: Auswirkungen von Code-Distanzen zwischen den Signaturen auf Kampagnen mit SDCs (Stichprobenverfahren 2 (V2)) für die Baseline-Implementierung. Das Säulendiagramm zeigt den Anteil Schlüssel-Signatur-Kombinationen aus der Stichprobe, der zu unerkannten Datenfehlern führt. Dabei wird die **Baseline-Implementierung** einmal ohne (oben) und mit Beachtung von **N1** (unten) verwendet. Es wird das Stichprobenverfahren 2 (V2) eingesetzt.

unentdeckte Kontrollflussfehler aufweisen (vgl. Tabellen A.15 und A.16). Diese sind lediglich im Anhang aufgeführt, weil unter Berücksichtigung von $d_c > 1$ insbesondere für die $v_{c,\max}$-Implementierung keine klaren Verbesserungen festgestellt werden konnten.

Die Beobachtungen in diesem Abschnitt machen die Problematik hinter den kleinen Stichprobengrößen deutlich. Eine kleine Stichprobe bedeutet für eine große Grundgesamtheit stets eine große Fehlerspanne. Eine große Fehlerspanne hat zur Folge, dass der eigentliche Wert in einem relativ großen Intervall in Bezug auf die Stichprobe liegen kann. Denn es ist möglich, dass die Effekte aufgrund eines Bias in der Stichprobe verdeckt werden oder die Ergebnisse einen Zusammenhang herstellen, der in der Realität nicht existiert - ein gängiges Problem im Bereich der Statistik. Daher kann nicht bewiesen werden, dass kein Zusammenhang zwischen der Fehlererkennungsleistung und der Code-Distanz zwischen den Signaturen besteht. Die Ergebnisse lassen dies allerdings stark vermuten.

6.5 Einschränkungen der Validität

Sowohl bei der Durchführung als auch bei der Auswertung der Versuche sind eine Reihe von Einschränkungen zu beachten, die sich auf die Aussagekraft der Ergebnisse auswirken können. Diese Einschränkungen beziehen sich lediglich auf die Aussagekraft der Fehlersimulationsversuche (vgl. Abschnitte 6.2, 6.3 und 6.4). Die Aussagekraft der Muster in den Signaturen (vgl. Abschnitt 6.1) bleibt von diesen Einschränkungen unberührt.

Es ist wichtig zu beachten, dass nur eine Stichprobengröße von 1.000 bei einer Größe der Ursprungsmenge von z. B. 758.596 für $A = 185$ betrachtet wird. Dies kann dazu führen, dass die 1.000 Kombinationen zwar zufällig, aber dennoch ungünstig gezogen wurden. Dadurch ist es möglich, dass die Ergebnisse der 1.000 Kampagnen weniger aussagekräftig sind, weil sie aufgrund einer ungünstigen Ziehung besonders positiv oder besonders negativ ausfallen.

Darüber hinaus können die Ergebnisse aufgrund der Stichprobenreduktion anhand der Regeln zusätzlich verzerrt werden (engl. *Selection Bias*). Denn der Anteil potenzieller Fehler in der Auswahlmenge steigt mit sinkendem Stichprobenumfang. Um diesem Effekt entgegenzuwirken bzw. ihn bewertbar zu machen, werden in dieser Arbeit die beiden Stichprobenverfahren **V1** und **V2** eingesetzt und verglichen.

Die Entscheidung, eine Stichprobengröße von 1.000 zu wählen, ist auf den Umfang und die zeitlichen Beschränkungen dieser Arbeit zurückzuführen. Es wird eine Fehlerspanne $E \approx 3,1\%$ und ein Konfidenzniveau $z = 1,96 = 95\%$ betrachtet. Die Problematik dahinter lässt sich am besten anhand der Abbildung 6.14 verdeutlichen. Die Abbildung ist aus Abbildung 6.11a entstanden und um Fehlerbalken erweitert worden. Anhand der Darstellung der Fehlerbalken wird die Bedeutung der Fehlerspanne von $3,1\%$ schnell offensichtlich, weil sich die Fehlerbalken überschneiden. Die Fehlerspanne beschreibt die theoretische Abweichung zwischen dem tatsächlichen Wert und dem durch die Stichprobe ermittelten Wert. Insbesondere bei so kleinen Werten und so kleinen Abweichungen in den Vergleichen der

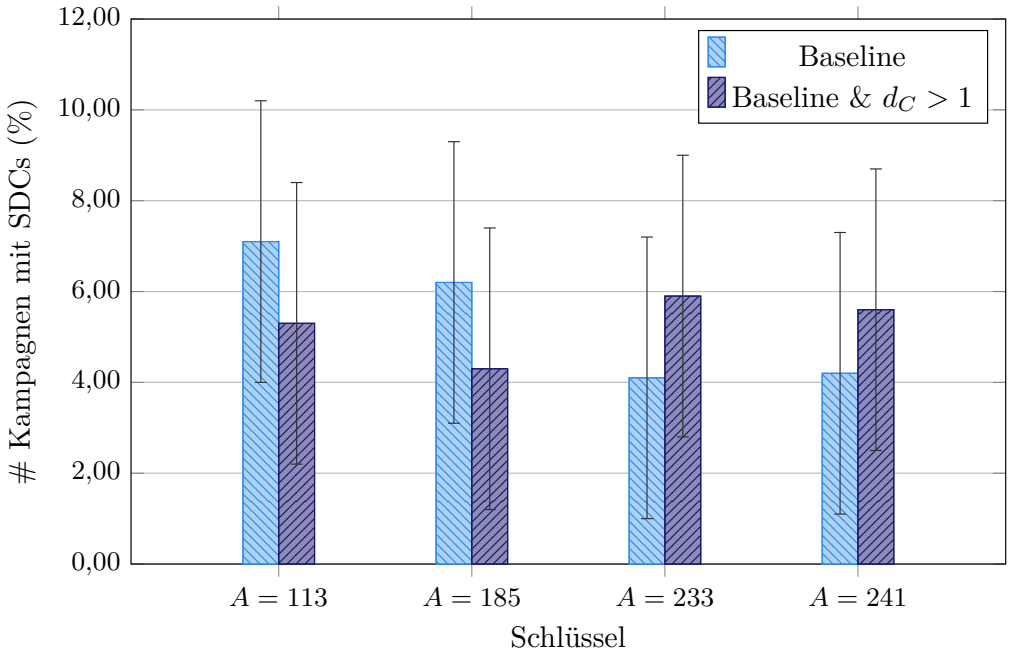


Abbildung 6.14: Auswirkungen der Fehlerrate von $3,1\%$ anhand der Abbildung 6.11a als Beispiel. Die vorliegende Abbildung erweitert die Abbildung 6.11a beispielhaft um Fehlerbalken. Anhand der Fehlerbalken wird deutlich, wie groß die Abweichung des realen Werts vom durch die Stichprobe ermittelten Wert sein kann.

Benchmarks, wie sie hier in den Versuchen ermittelt werden, kann eine Fehlerspanne von $3,1\%$ große Auswirkungen auf die Validität haben. Die Ergebnisse aus den Abschnitten 6.2 bis 6.4 dieser Arbeit sind folglich entsprechend einzuordnen.

Im günstigsten Fall hätte eine Fehlerspanne $E = 1\%$ und ein Konfidenzniveau $z = 2,58 = 99\%$ angenommen werden können. Dies erhöht jedoch den Stichprobenumfang n und damit den Rechenaufwand erheblich, wie im Folgenden gezeigt wird: Der π-Wert würde wie in Abschnitt 5.2.1 als $\pi = 0,5$ gewählt, da zu Beginn der Arbeit nichts über den Anteil der untersuchten Merkmale an der Grundgesamtheit bekannt wäre. Die Grundgesamtheit für $A = 113$ umfasst $N = 168.202$ Kombinationen und für $A = 241$ bereits $N = 1.693.740$. Für $A = 113$ gilt dann $n = \frac{2,58^2 \cdot \pi(1-\pi) \cdot N}{2,58^2 \cdot \pi(1-\pi) + E^2(N-1)} = 15.142,93 \approx 15.143$. Für $A = 241$ gilt $n = \frac{2,58^2 \cdot \pi(1-\pi) \cdot N}{2,58^2 \cdot \pi(1-\pi) + E^2(N-1)} = 16.479,1025 \approx 16.480$. Unter anderen (zeitlichen) Bedingungen wäre eine Stichprobengröße zwischen 15.200 (für $A = 113$) und 16.500 (für $A = 241$) also optimal gewesen.

Bei einer Laufzeit der Fehlersimulation pro Kampagne von etwa 100 s ergäbe sich somit ein Zeitaufwand t zwischen etwa $t = 100\,\text{s} \cdot 15.200 = 1.520.000\,\text{s} \approx 422,2\,\text{h}$ und $t = 100\,\text{s} \cdot 16.500 = 1.650.000\,\text{s} \approx 458,3\,\text{h}$ pro Versuch mit einem Benchmark und einem Schlüssel, was etwa 7,5 Tagen entspricht.

Eine weitere Einschränkung der Ergebnisse ergibt sich aus der Tatsache, dass aufgrund der zeitlichen Beschränkungen keine vollständige Zweigabdeckung erreicht werden konnte. Der Mehrheitsentscheider der CoRed-Implementierung entscheidet über die Ergebnisse der Replikate und folgt dabei je nach Konsensmenge einem anderen Pfad. Die Experimente wurden aus Zeitgründen nur für die Konstellation $x = y = z$ durchgeführt. Eine Aussage über das Verhalten des Entscheiders in Bezug auf die Lösungsvorschläge für eine andere Konstellation als $x = y = z$ kann daher nicht mit Sicherheit getroffen werden.

In der Diskussion wird darüber hinaus deutlich werden, dass auch die Wahl verschiedener Eingabewerte x, y und z neben der Zweigabdeckung entscheidend sein kann. So macht es z. B. einen Unterschied, ob in der Konstellation $x \neq z$ und $y = z$ Werte gewählt werden, sodass $x > z$ oder $x < z$, wenn ein Kontrollflussfehler auftritt (vgl. Abschnitt 7.2.1). Auch die Differenz zwischen x und z kann ausschlaggebend sein.

Weiterhin wäre eine Überprüfung der Ergebnisse auf das Verhalten bei Mehrbit-Fehlern sinnvoll gewesen. Diese Arbeit konzentriert sich aufgrund des Umfangs lediglich auf Einbit-Fehler. Die Fehlererkennungsleistung von Schlüssel-Signatur-Kombinationen, die zu höheren Code-Distanzen führen, kann nur so wirklich bewertet werden.

Die Untersuchungen und ihre Ergebnisse sind daher aufgrund der Begrenztheit der Experimente nicht abschließend. Aufgrund der dargestellten Ergebnisse kann jedoch davon ausgegangen werden, dass die Abweichungen der Ergebnisse vom tatsächlichen Wert nicht allzu groß sind.

Kapitel 7

Diskussion und Ausblick

In diesem Kapitel wird zunächst auf verwandte Arbeiten außerhalb des CoRedAnsatzes eingegangen. Es folgt eine Diskussion, in der insbesondere auf eine Lücke in der Umsetzung von CoRed eingegangen wird, die bei der Durchsicht dieser Arbeit festgestellt wurde. Es folgt eine Zusammenfassung und Diskussion der Ergebnisse dieser Arbeit, in der die wesentlichen Ergebnisse noch einmal in den Kontext der Arbeiten von P. Ulbrich [55] und Hoffmann u. a. [24] gestellt werden. Abschließend wird ein Ausblick auf mögliche weitere Forschungsthemen gegeben.

7.1 Verwandte Arbeiten

Neben den Arbeiten zum CoRed-Ansatz (vgl. Abschnitt 1.1) gibt es weitere verwandte Arbeiten, welche im Folgenden kategorisiert und umrissen werden sollen. Im Bereich der theoretischen Forschung zur arithmetischen Codierung und genauer zur AN-Codierung sind insbesondere die Arbeiten von D. T. Brown [12], A. Avižienis [4] und P. Forin [19] zu erwähnen.

D. T. Brown [12] stellt bereits im Jahr 1960 ANB-Codes vor. A. Avižienis [4] gibt im Jahr 1971 eine formale Definition für arithmetische Codes - genauer Residue Codes und AN-Codes. P. Forin [19] baut die AN-Codes über ANB-Codes weiter zu ANBD-Codes aus, womit eine vollständige Fehlererkennung im Sinne des Fehlermodells möglich wird.

Aus diesen theoretischen Grundlagen sind verschiedene praktische Anwendungen der arithmetischen Codierung entstanden. Eine der ersten war der Vital Coded Processor von P. Forin [19]. Dieser nutzt ANBD-Codierung, um die auszuführende Anwendung und ihre Daten zu codieren. Dadurch wird eine Erkennung von Berechnungsfehlern, falscher Operatoren, falscher Operationen sowie veralteter Werte ermöglicht [19]. Dafür muss allerdings der Datenfluss des zu codierenden Programmes vorab bekannt sein, um die Berechnung statischer Signaturen zu ermöglichen. Ist die Berechnung erfolgt, bietet der VCP allerdings eine große Fehlerabdeckung. [58]

ED^{4}I (Error Detection by Data Diversity and Duplicated Instructions) von Oh, Mitra und McCluskey [41] ist eine Technik zur Fehlererkennung auf Basis von AN-Codes. Das abzusichernde Programm wird dupliziert. Die zweite Version des Programms arbeitet allerdings nicht auf den originalen Werten des Programms, sondern auf AN-codierten Versionen dieser Werte. Diese beiden Programm-Versionen werden parallel ausgeführt und die Ergebnisse der beiden Programme verglichen. Der Vergleich überprüft, ob der codierte Wert ein Vielfaches des nicht-codierten Wertes ist. Ist dies nicht der Fall, ist ein Fehler aufgetreten und erkannt worden.

TRUMP (Triple Redundancy Using Multiplication Protection) nutzt ebenfalls AN-Codierung zur Absicherung eines Programmes [13]. Auch hier wird das Programm dupliziert und die duplizierte Version mit AN-Codierung abgesichert. Diese beiden Versionen werden an bestimmten Punkten miteinander verglichen. Kommt es zu einer Abweichung, werden Maßnahmen zur Fehlerbehandlung ausgeführt.

Der Unterschied zu ED^{4}I ist, dass TRUMP lediglich Register und nicht den Speicher codiert. Darüber hinaus werden nur solche arithmetischen Operationen codiert, die keine Korrekturmaßnahmen erfordern, wenn mit codierten Werten gerechnet wird. Dabei handelt es sich um Subtraktion und Addition (wie in Abschnitt 2.3.1 näher beschrieben). Diese beiden Aspekte haben eine höhere Anzahl unerkannter Datenfehler als im ED^{4}I-Ansatz zur Folge.

U. Schiffel [47] stellt in ihrer Dissertation zwei Ansätze zur softwarebasierten Codierung von Programmen vor. Dabei handelt es sich zum einen um *Software Encoded Processing*, bei welcher zur Laufzeit ANB-Codes angewendet werden, um mit einem nicht fehlertoleranten Binary sichere Ausführungen zu ermöglichen. Zum anderen diskutiert sie *Compiler Encoded Processing*. Mithilfe dieser Technik werden zur Übersetzungszeit arithmetische Codes auf sonst unsichere Programme angewendet. Dieser Ansatz wird für AN-Codierung genauer in den Arbeiten von Fetzer, Schiffel und Süßkraut [18] und Schiffel u. a. [48] erläutert bzw. vertieft.

Munk u. a. [38] stellen einen softwarebasierten Fehlertoleranz-Ansatz vor, welcher auf dem CoRed-Ansatz aufbaut und ihn erweitert. Dabei setzen sie anstelle eines codierten Entscheiders zwei *fail-silent* Entscheider ein, welche entweder einen korrekten Wert oder gar keinen Wert liefern. Die Entscheider überprüfen sich dabei gegenseitig, mit dem Ziel, dass ein Entscheider auftretende Fehler im anderen Entscheider beheben kann.

Darüber hinaus erweitern sie den CoRed-Ansatz von Dreifachredundanz auf N-fache Redundanz. Dabei wird das mehrfach redundant auszulegende Programm N mal repliziert. Die entstehenden Tasks werden auf verschiedenen Kernen des verwendeten Prozessors verteilt und parallel ausgeführt. Die Entscheider sind ebenfalls eigene Tasks, welche auf eigenen Kernen ausgeführt werden. Sie werden allerdings mit einem zeitlichen Versatz aktiviert.

S^{3}DES von Osinski und Mottok [42] baut ebenfalls auf dem CoRed-Ansatz auf. Dieser Ansatz verwendet wie der Ansatz von Munk u. a. [38] ebenfalls Mehrkern-Prozessoren mit

dem Ziel, die räumlichen Vorteile von Mehrkern-Prozessoren zur parallelen Ausführung zu nutzen. Die Entscheider werden, wie auch das Ausführungsprogramm, dreifach repliziert und überprüfen nach dem Entscheidungsprozess zunächst ihre eigenen Ergebnisse auf Kontrollfluss- und Datenfehler. Danach überprüfen sie gegenseitig ihre Ergebnisse und erzeugen eine Ausgabe. Der Entwurf ist dazu in der Lage, bis zu zwei Ausfälle oder Fehler von Entscheidern zu kompensieren.

Braun und Mottok [11] sammeln Irrtümer und Missverständnisse, die im Zusammenhang mit der Anwendung von AN-Codierung bisher in verschiedenen Arbeiten beobachtet wurden. Dazu gehört u.a. der Irrglaube, dass Schlüssel prim oder möglichst groß gewählt werden sollten. Hier zeigen viele Anwendungsbeispiele, dass in der Praxis weder eine Primzahl als Schlüssel noch ein größerer Schlüssel an sich zwangsläufig zu einer besseren Fehlererkennung führt.

Sie gehen in ihrer Arbeit, wie andere Arbeiten vor ihnen, im Bereich der Parametrisierung der Codierung ausschließlich auf die Wahl des Schlüssels und seine Bedeutung ein. Die weiteren Parameter wie variablenspezifische oder anwendungsspezifische Signaturen werden nicht betrachtet.

Um die Ergebnisse dieser Arbeit überprüfen zu können, wird das Verfahren der Fehlerinjektion (vgl. Abschnitt 2.6) eingesetzt. Auch in diesem Bereich existieren viele Arbeiten. Über einige dieser Arbeiten soll an dieser Stelle ein kurzer Überblick gegeben werden.

Die Arbeiten von Hsueh, Tsai und Iyer [25] und von Ziade, Ayoubi und Velazco [62] geben einen Überblick über Fehlerinjektionstechniken, ihre Vor- und Nachteile und mögliche Werkzeuge im entsprechenden Bereich. Beide unterscheiden dabei, wie in der Literatur üblich, zwischen hardwarebasierten und softwarebasierten Fehlerinjektionstechniken.

Im Bereich der hardwarebasierten Fehlerinjektion werden Fehler auf physikalischer Ebene direkt in Testhardware injiziert. Ein Beispiel für hardwarebasierte Fehlerinjektionswerkzeuge ist RIFLE von Madeira u.a. [32] aus dem Jahr 1994.

Kommen softwarebasierte Fehlerinjektionstechniken zum Einsatz, wird mithilfe von Werkzeugen versucht, Hardware-typische Fehlermuster auf Software-Ebene nachzustellen und in das System einzubringen. Ziade, Ayoubi und Velazco [62] nennen u.a. FERRARI von Kanawati, Kanawati und Abraham [26] als Beispiel für softwarebasierte Fehlerinjektionswerkzeuge.

Ziade, Ayoubi und Velazco erweitern außerdem die Unterscheidung um u.a. simulationsbasierte Fehlerinjektionstechniken. Simulationsbasierte Fehlerinjektion verwendet ein Modell des Systems, in welches Fehler injiziert werden sollen. Dieses System wird in einem Simulator ausgeführt, um in der Simulation Fehler zu injizieren.

Ein Beispiel für simulationsbasierte Fehlerinjektionswerkzeuge ist FAIL* (Fault Injection Leveraged) von H. Schirmeier [50, 49]. Dieses Werkzeug wurde u.a. in den bisherigen Arbeiten zum CORED-Ansatz verwendet und wurde auch in dieser Arbeit zur Bewertung der Ergebnisse eingesetzt.

7.2 Diskussion

Im Zuge der Problemanalyse ist eine Lücke in der CoRed-Implementierung aufgefallen, durch welche in einigen Randfällen eine Fehlererkennung nicht möglich ist (vgl. Abschnitt 3.3.2). In dieser Diskussion soll zunächst auf diesen Punkt eingegangen werden, bevor eine Zusammenfassung dieser Arbeit erfolgt.

7.2.1 Schwierigkeiten mit Randfällen bei der Fehlererkennung

Während der Korrektur dieser Arbeit ist im Zuge von Abschnitt 4.2 aufgefallen, dass die Fehlererkennung für einen Randfall nicht möglich ist. Hoffmann u. a. [24] schlagen in ihrer Arbeit die Erweiterung der `apply()`-Funktion vor, um schwerwiegende Kontrollflussfehler erkennen zu können. Dabei argumentieren sie, dass im Falle eines Kontrollflussfehlers, bei dem z. B. $x \neq z$ gilt, aber $B_{\text{dyn}} = x_c - z_c$ berechnet wird, die dynamischen Signatur stets um ein Vielfaches von A von ihrem eigentlichen, korrekten Wert abweicht [24].

Im Folgenden soll deutlich werden, dass diese Aussage nicht für alle Fälle stimmt. Wie in Abschnitt 3.3.2 beschrieben, muss der Fall $x = y = z$ an dieser Stelle nicht betrachtet werden, da der Mehrheitsentscheider bei einem Kontrollflussfehler lediglich von einer zu kleinen Konsensmenge ausgehen würde. Für den Fall, dass mindestens ein Replikat fehlerhaft ist, kann es schwerwiegende Konsequenzen haben, wenn von einer falschen Konsensmenge ausgegangen wird.

Seien erneut $v^i, v^j \in \{x, y, z\}$ und $i \neq j$ uncodierte Werte, v_c^i bzw. v_c^j ihre codierten Formen und B_i, B_j ihre Signaturen mit $B_i > B_j$, da $B_x > B_y > B_Z$ (vgl. **R4**). Dann errechnet sich die dynamische Signatur durch

$$\begin{aligned} v_c^i - v_c^j &= (A \cdot v^i + B_i) - (A \cdot v^j + B_j) \\ &= A \cdot v^i + B_i - A \cdot v^j - B_j \\ &= A \cdot (v^i - v^j) + B_i - B_j. \end{aligned}$$

Für den Fall $v^i = v^j$ treten keine Probleme auf (vgl. Abschnitt 3.3.2). Die Ungleichung

$$\begin{aligned} A \cdot (v^i - v^j) + B_i - B_j &= B_i - B_j \\ &< A \end{aligned}$$

gilt, weil $0 < B_j < B_i < A$, für alle $i \in x, y, z$ (vgl. **R3**).

Der Fall $v^i \neq v^j$ muss allerdings in $v^i > v^j$ und $v^i < v^j$ aufgeteilt werden. Für den Fall $v^i > v^j$ gilt, wie in Abschnitt 3.3.2 beschrieben,

$$\begin{aligned} A \cdot (v^i - v^j) + B_i - B_j &\geq A + B_i - B_j \\ &\geq A, \end{aligned}$$

da $B_i > B_j$ und $v^i > v^j$, weshalb $v^i - v^j$ positiv ist.

Betrachtet man den Fall $v^i < v^j$, wird die Differenz $v^i - v^j$ allerdings negativ und die Ungleichung gilt nicht. Für diesen Fall ist also zu zeigen, dass die dynamischen Signatur im Falle eines Kontrollflussfehlers nicht immer um ein Vielfaches von A von ihrem eigentlichen, korrekten Wert abweicht und der Fehler daher unerkannt bleibt. Dies soll anhand eines Gegenbeispiels erfolgen:

7.2.1 Beispiel. Seien $A = 113$, $B_x = 42$, $B_y = 31$ und $B_z = 15$. Die Forderung $0 < B < A$ (vgl. Regel **R3**) ist erfüllt, da $(42-31)+(42-15) = 38 \leq A$ gilt (vgl. Abschnitt 4.2). Seien $x = 5$ (fehlerhaft) und $y = z = 6$ (korrekt). Es trete ein Kontrollflussfehler auf, sodass $x \neq z$ gilt, aber $B_{\text{dyn}} = x_c - z_c$ berechnet wird. Dann gilt

$$\begin{aligned} |B_{\text{dyn}}| &= |x_c - z_c| \\ &= |(A \cdot x + B_x) - (A \cdot z + B_z)| \\ &= |(113 \cdot 5 + 42) - (113 \cdot 6 + 15)| \\ &= |-86| \\ &= 86 \\ &< A. \end{aligned}$$

Dieser Fehler wird aus mathematischer Sicht in der `apply()`-Funktion nicht entdeckt, da dort lediglich auf $|B_{\text{dyn}}| \geq A$ geprüft wird.

Darüber hinaus wird der Fehler auch in der `decode()`-Funktion nicht erkannt, weil die Prüfung $v_c \mod A \neq B$ nicht greift: Der Mehrheitsentscheider gibt im Beispiel 7.2.1 den Wert $x_c + B_{\text{dyn}}$ zurück. Vor dem Decodieren wird die Sprungsignatur von diesem Ergebnis abgezogen. Es ergibt sich als Eingabe v_c für die `decode()`-Funktion im Beispiel 7.2.1 also

$$\begin{aligned} v_c &= x_c + B_{\text{dyn}} - B_E \\ &= (A \cdot x + B_x) + (x_c - z_c) - (B_x - B_z) \\ &= (A \cdot x + B_x) + ((A \cdot x + B_x) - (A \cdot z + B_z)) - (B_x - B_z) \\ &= A \cdot x + B_x + A(x - z) + B_x - B_z - B_x + B_z \\ &= A \cdot x + B_x + A(x - z). \end{aligned}$$

Es verbleibt ein negatives Vielfaches von A als Rest aufgrund der Abweichung zwischen B_{dyn} und B_E. Dieser Fehler wird daher in der Berechnung $v_c \mod A \neq B$ nicht erkannt.

Insgesamt wird deutlich, dass aus mathematischer Sicht ein Fehler wie in Beispiel 7.2.1 nicht erkannt werden kann. Abhängig von der Implementierung kann es Konstellationen geben, in denen eine Erkennung aufgrund von Unterläufen dennoch möglich ist. Die Implementierung sollte sich für ein korrektes Verhalten im Fehlerfall allerdings nicht unbedingt auf Unterläufe verlassen.

Da negative dynamische Signaturen offensichtlich nur auftreten, wenn es zu einem schweren Fehler gekommen ist, kann die Implementierung so angepasst werden, dass in

der `apply()`-Funktion nicht nur geprüft wird, ob der Betrag von B_{dyn} größer als A ist. Vielmehr müssen auch negative Werte für B_{dyn} berücksichtigt und auf $B_{\text{dyn}} \leq 0$ geprüft werden, um die bisher potenziell unerkannten Fehler zu erkennen.

Alternativ kann die `decode()`-Funktion angepasst werden, sodass sie anhand eines Vergleichs der Sprungsignatur mit der dynamischen Signatur den Fehler erkennen kann. Zu diesem Zweck müsste allerdings auch die Sprungsignatur explizit durch den Mehrheitsentscheider weitergereicht werden. Die zuvor beschriebene Anpassung erscheint daher weniger invasiv.

Die Versuche dieser Arbeit sind von diesem Fehlerfall nicht betroffen, da in den Versuchen lediglich der Fall $x = y = z$ untersucht wurde.

7.2.2 Zusammenfassung

Diese Arbeit behandelt eine Reihe aufeinander aufbauender Ziele im Bereich der Parametrisierung der arithmetischen Codierung. Zunächst wurde die Verteilung der Code-Distanzen zwischen den Signaturen untersucht. Dabei sollte unter anderem geklärt werden, ob Muster in der Verteilung der Code-Distanzen existieren und ggf. sogar Rechenvorschriften formuliert werden können, um die Muster in den Code-Distanzen zu beschreiben. Mit diesen Mustern bzw. Rechenvorschriften kann eine Wahl der Signaturen potenziell erleichtert werden, falls die Wahl der Signaturen in Abhängigkeit von der Code-Distanz zwischen den Signaturen eine verbesserte Fehlererkennungsleistung bedeutet. Dies sollte im Zuge dieser Arbeit ebenfalls untersucht werden. Zu diesem Zweck wurden die Code-Distanzen zwischen Signatur-Paaren und Signatur-Tripeln errechnet und später auf Muster untersucht.

Darüber hinaus war ein weiteres Ziel dieser Arbeit, mithilfe der Parametrisierung der arithmetischen Codierung die Anzahl der verbleibenden unentdeckten Datenfehler zu reduzieren. Daher sollten bestehende Umsetzungen und Wege zur Reduktion der unerkannten Datenfehler betrachtet werden. Geprüft wurde dabei, ob ggf. Abfragen anders oder strenger gestaltet werden können, um die Erkennungsleistung zu verbessern.

Bei dieser Untersuchung fiel auf, dass in einem Lösungsvorschlag der Arbeit von Hoffmann u. a. [24] eine Variablen $v_{c,\max}$ zwar eingeführt, allerdings ihr Wert an keiner Stelle definiert wird. In dieser Arbeit wurde daher eine mögliche Definition für $v_{c,\max}$ vorgeschlagen.

Der eigentliche Teil dieser Untersuchung bestand allerdings darin, die von Hoffmann u. a. [24] vorgeschlagene Abfrage in der `apply()`-Funktion der CoRed-Implementierung so umzuformulieren und eine zusätzliche Regel für die Wahl der Signaturen einzuführen, dass eine Verfälschung der Codewörter schneller aus dem Bereich der gültigen Codewörter fällt und Fehler somit besser erkannt werden können. Die Anpassung der Abfrage und die Einführung einer Parameterregel wurden als **N1** bezeichnet. Da die Einführung der zu-

sätzlichen Regel allerdings den Coderaum potenziell etwas verkleinert, war zu überprüfen, ob die Anpassung tatsächlich eine verbesserte Fehlererkennung bedeutet.

Außerdem sollte aufbauend auf den ersten Ergebnissen untersucht werden, ob eine höhere Code-Distanz zwischen den Signaturen positive Auswirkungen auf die Erkennungsleistung des Codes hat. Dazu wurden unter anderem auch die Berechnungen und Erkenntnisse aus der Muster-Analyse verwendet.

Da die Lösungsvorschläge das Problem der unentdeckten Datenfehler nicht vollständig lösen konnten, wurden statistische Versuche mit zwei Stichprobenverfahren durchgeführt. Damit sollte sichergestellt werden, dass nicht versehentlich eine besonders gute bzw. eine besonders schlechte Parameterwahl getroffen wird und dies die Auswertungen verfälscht.

Die Ergebnisse diese Arbeit zeigen, dass Muster in der Verteilung der Code-Distanzen zwischen den Signaturen existieren. Diese Muster können durch parallele Geraden beschrieben werden. Für ein Signatur-Paar bzw. ein Signatur-Tripel mit einer bekannten minimalen Code-Distanz zwischen diesen Signaturen ist es mithilfe des experimentell ermittelten Richtungsvektors möglich, weitere Signatur-Paare bzw. Tripel mit derselben Code-Distanz zu bestimmen.

Für die Beantwortung der weiteren Fragen wurden Fehlersimulationen für verschiedene Schlüssel und Stichproben durchgeführt. Die Wirksamkeit der Definition von $v_{c,\max}$ kann mithilfe einer Fehlersimulation bestätigt werden. Damit stimmen die Ergebnisse dieser Arbeit mit denen von Hoffmann u. a. [24] überein und bestätigen diese somit.

Die Ergebnisse der Versuche zu **N1** zeigen eine leichte Tendenz einer verbesserten Fehlererkennungsleistung, die vermuten lässt, dass die Anpassung ebenfalls wirksam ist.

Die Ergebnisse, ob eine größere Code-Distanz zwischen den Signaturen die Fehlererkennungsleistung verbessert, fallen indifferent aus. Es wurde aufgrund der zeitlichen Beschränkung der Arbeit für die Fehlersimulation eine relativ kleine Stichprobe in der Fehlersimulation gewählt, welche die Aussagekraft der Ergebnisse etwas einschränken kann. Da das Bild der Untersuchung von **N1** allerdings eine leichte Tendenz zeigt und diese in der Untersuchung zu Abschnitt 6.1 fehlt, ist es sehr wahrscheinlich, dass die Berücksichtigung der Code-Distanz zwischen den Signaturen keinen Einfluss auf die Fehlererkennungsleistung des Codes hat.

Insgesamt wird damit anhand der Ergebnisse deutlich, dass die Wahl der Schlüssel viel entscheidender ist als eine Optimierung der Signaturen anhand ihrer Code-Distanzen. Somit kann diese Arbeit die Ergebnisse von P. Ulbrich [55] hinsichtlich der Bedeutung der Schlüsselwahl bestätigen. Darüber hinaus ist der Einschätzung von P. Ulbrich [55] zuzustimmen, dass die Wahl der Signaturen von untergeordneter Bedeutung ist.

Unabhängig von den in Abschnitt 6.5 beschriebenen Einschränkungen der Ergebnisse ist offensichtlich, dass weiterhin vereinzelte unentdeckte Kontrollflussfehler für einige Schlüssel-Signatur-Kombinationen existieren können. Darüber hinaus zeigen die Beobachtungen dieser Diskussion in Abschnitt 7.2.1, dass weiterhin spezielle Randfälle existieren,

in denen eine Fehlererkennung aus mathematischer Sicht nicht möglich ist. Diese treten lediglich bei einer bestimmten Parametrisierung und bei bestimmten Sensorwerten auf und sind daher auch in der Fehlersimulation nur unter ganz bestimmten Umständen oder sehr breit angelegten Versuchen überhaupt zu beobachten.

An dieser Stelle wird die große Herausforderung des Testens im Vergleich zur Verifikation im Allgemeinen deutlich. Um die Abwesenheit von Fehlern zeigen zu können, müsste für alle Parameter-Kombinationen sowie für alle Sensorwerte die Abwesenheit von Fehlern gezeigt werden können. Obwohl FAIL* ein sehr vielseitiges und aufgrund der Fehlerraumreduktion auch schnelles Werkzeug ist, ist dies in einer realistischen Zeit kaum durchführbar.

Damit kann diese Arbeit insgesamt die Ergebnisse von P. Ulbrich [55] im Hinblick auf die Wichtigkeit der guten Auswahl der Schlüssel bestätigen. Darüber hinaus kann diese Arbeit Lücken in der Fehlererkennung des CoRed-Ansatzes identifizieren und mögliche Lösungen zur Behebung dieser Lücken nennen. Auch wenn der Ansatz, die Wahl der Signaturen von der Code-Distanz zwischen den Signaturen abhängig zu machen, die Fehlererkennungsleistung von CoRed nicht wesentlich verbessert, ist mit diesen Ergebnissen dennoch ein weiterer Schritt zum besseren Verständnis der Parametrisierung gelungen.

7.3 Ausblick

Die identifizierten Einschränkungen der Ergebnisse bieten Möglichkeiten für zukünftige Erweiterungen und Verbesserungen.

Eine denkbare Richtung könnte die Erweiterung der Stichprobengröße sein, um eine größere Repräsentativität zu gewährleisten. Mit einer größeren Stichprobe würde außerdem eine kleinere Fehlerspanne und ein höheres Konfidenzniveau einhergehen, um die Ergebnisse weiter zu präzisieren. Eine erhöhte Stichprobengröße könnte dazu beitragen, eine potenzielle Verzerrung der Ergebnisse zu minimieren und eine zuverlässigere Basis für Schlussfolgerungen zu schaffen. Da allerdings die Ergebnisse der Code-Distanz zwischen den Signaturen in keine klare Richtung wiesen, ist es eher unwahrscheinlich, dass eine Wirksamkeit der Code-Distanz zwischen den Signaturen übersehen wurde.

Da die hier durchgeführten Versuche keine vollständige Zweigabdeckung des Mehrheitsentscheiders erreichen konnten, ist es denkbar, die durchgeführten Experimente mit anderen Eingaben und damit anderen Zweigen zu wiederholen. Die Untersuchung von anderen Kombinationen von Parametern könnte ein umfassenderes Verständnis für das Verhalten und die Fehlererkennungsleistung der vorgeschlagenen Anpassungen liefern.

Eine weitere, interessante Möglichkeit zur Untersuchung wäre die Einbeziehung von Mehrbit-Fehlern in die Analyse. Dies würde eine breitere Abdeckung aller möglichen Fehlerszenarien ermöglichen und kann ggf. potenzielle Schwachstellen bei Kombinationen mit kleineren Code-Distanzen aufdecken.

Neben möglichen, weiteren Versuchen im Bereich der Fehlerinjektionsexperimente besteht darüber hinaus die Möglichkeit, dass es tatsächlich Berechnungsvorschriften zur Ermittlung der Code-Abstände zwischen den Signaturen gibt. Sollte sich wider Erwarten in größer angelegten Fehlersimulationsexperimenten eine Verbesserung der Fehlererkennungsleistung in Abhängigkeit von den Code-Distanzen zwischen den Signaturen feststellen lassen, lohnt sich auch hier eine genauere Betrachtung.

Unabhängig von den in dieser Arbeit durchgeführten Versuchen bleiben einige wenige unentdeckte Fehler im CoRed-Ansatz, die abhängig von der Parametrisierung auftreten. Dies kann verschiedene Ursachen haben. Zum einen ist es möglich, dass seltene Randfälle in den bisherigen Untersuchungen übersehen wurden. Wahrscheinlicher ist jedoch, dass eine weitere regelbasierte Einschränkung der Parametrisierung zum Erfolg führt, da unentdeckte Fehler nur noch für bestimmte Schlüssel-Signatur-Kombinationen auftreten. Hier ist eine genauere Untersuchung auf mögliche Gemeinsamkeiten zwischen den Schlüssel-Signatur-Kombinationen, die zu unentdeckten Fehlern führen, sinnvoll.

Weiterhin sollten die in der Diskussion beschriebenen Anpassungen zur Behebung der Lücke in einer Fehlersimulation untersucht werden, um die Wirksamkeit der Anpassung zeigen oder widerlegen zu können.

Schließlich können Randfälle, wie der in der Diskussion beschriebene (vgl. Abschnitt 7.2.1), auch in weiteren Arbeiten wie denen von P. Munk [37] und L. Osinski und J. Mottok [42] enthalten sein. Dies könnte ebenfalls untersucht werden.

Anhang A

Anhang

In diesem Anhang werden die Analysen aus Abschnitt 6.1 für weitere Schlüssel fortgesetzt. Anhand der Ergebnisse wird deutlich, dass die beschriebenen Muster systematisch für alle Schlüssel sind und nicht bloß vereinzelt auftreten.

Darüber hinaus sind in Abschnitt A.2 Tabellen aufgeführt, auf denen die Grafiken in Abschnitt 6.4 basieren. Diese Tabellen geben einen vollständigen Überblick über die Abweichungen zwischen den verschiedenen Stichprobenverfahren.

A.1 Muster in Code-Distanzen

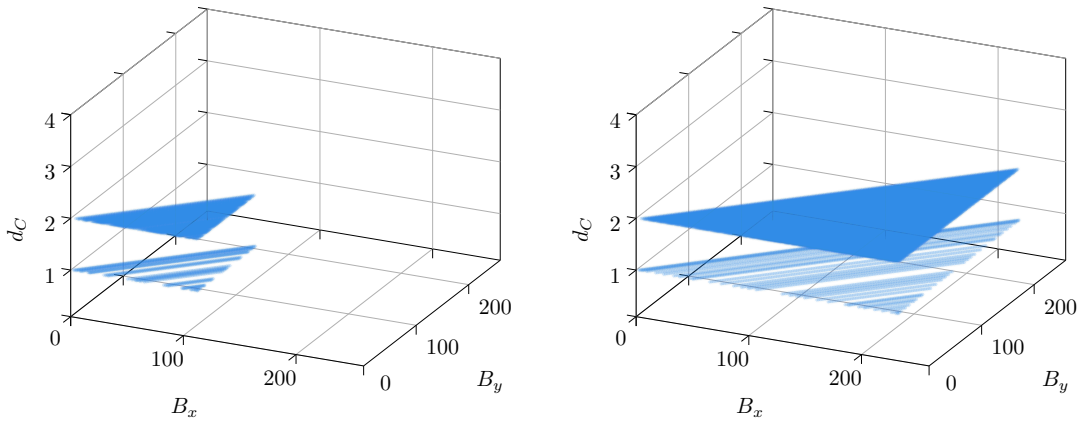
An dieser Stelle werden die Analysen aus Abschnitt 6.1 für die Schlüssel $A = 113$ und $A = 233$ präsentiert. Dabei wird genau so vorgegangen wie in Abschnitt 6.1.

Abbildung A.1 zeigt die ermittelten Code-Distanzen für zwei Signaturen in Abhängigkeit der Schlüssel $A = 113$ und $A = 233$. Auch hier ist zu erkennen, dass die Beziehung Geraden im Raum entstehen lässt. Diese können mit der in Abschnitt 6.1 beschriebenen Methodik experimentell ermittelt werden. Dabei kann auch hier der Richtungsvektor $r = (1, 1, d_C)^T$ verwendet werden.

Die Beziehung zwischen Signatur-Differenzen und Code-Distanzen existiert auch für die Schlüssel $A = 113$ und $A = 233$ und verdeutlicht, dass hier eine Regelmäßigkeit besteht (vgl. Abbildung 6.3 und A.2).

Die Verteilung der Code-Distanzen für drei Signaturen und die festen Schlüssel $A = 113$ und $A = 233$ ist in Abbilung A.3 zu sehen. Auch hier sind die Verteilungen vergleichbar zu denen in Abbilung 6.4. Lediglich die Anzahl der möglichen Signatur-Kombinationen und die Code-Distanz variiert in Abhängigkeit vom gewählten Schlüssel.

In Abbildung A.4 ist die Verteilung der Code-Distanz $d_C = 2$ für $A = 113$ und $A = 233$ in Abhängigkeit der drei Signaturen B_x, B_y und B_z abgebildet. Diese Darstellungen ergänzen die Abbildung 6.5 aus Abschnitt 6.1. Für die Schlüssel $A = 113$ und $A = 233$ ist die Code-Distanzen $d_C = 2$ maximal. Gleichzeitig existieren allerdings sehr viele Schlüssel-

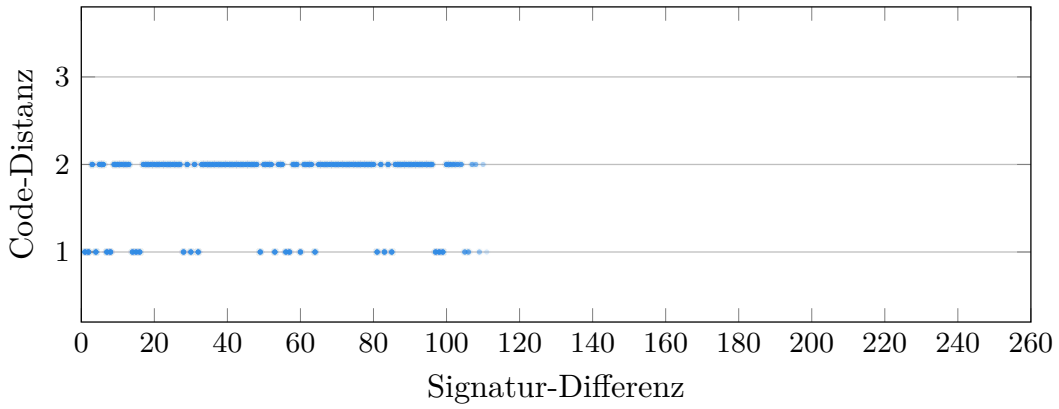


(a) Verteilung der Code-Distanzen für zwei Signaturen für den Schlüssel $A = 113$.

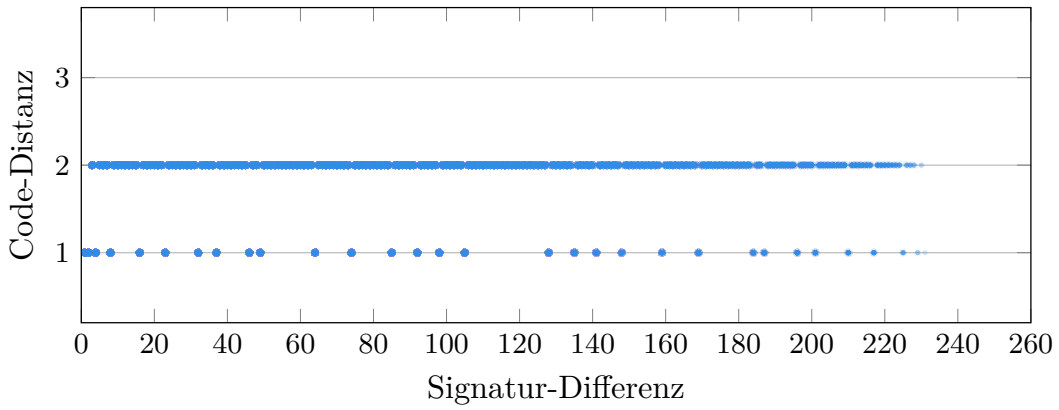
(b) Verteilung der Code-Distanzen für zwei Signaturen für den Schlüssel $A = 233$.

Abbildung A.1: Verteilung der Code-Distanzen für zwei Signaturen (Schlüssel A = 113 und A = 233). Für jeweils einen festen Schlüssel werden Signatur-Kombinationen aus jeweils zwei Signaturen gebildet und die Code-Distanz für diese Kombinationen berechnet. Auf der x- und y-Achse sind die Signaturen B_x und B_y abgetragen. Auf der z-Achse ist die entstehende Code-Distanz abgebildet.

Signatur-Kombinationen, für die die Bedingung $d_C = 2$ greift. Die Abbildungen lassen deshalb lediglich eine große Punktwolke erkennen, in welchen existierende Muster untergehen. Sie sind daher nicht Teil der Auswertung in Kapitel 6.

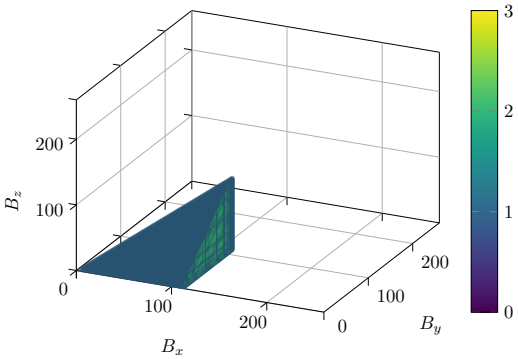


(a) Die Verteilung der Code-Distanz in Abhängigkeit der Signatur-Differenz für den Schlüssel $A = 113$.

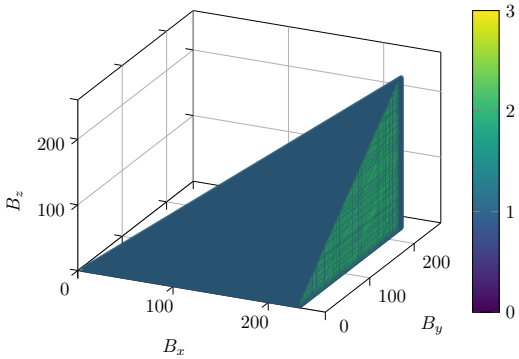


(b) Die Verteilung der Code-Distanz in Abhängigkeit der Signatur-Differenz für den Schlüssel $A = 233$.

Abbildung A.2: Verteilung der Code-Distanz in Abhängigkeit der Signatur-Differenz (Schlüssel $A = 113$ und $A = 233$). Die Grafiken zeigen die Code-Distanzen in Abhängigkeit von der Differenz zweier Signaturen für jeweils einen festen Schlüssel ($A = 113$ links und $A = 233$ rechts). Für die Differenzen zwischen den Signaturen eines Signatur-Paares und die Code-Distanz des Paares fällt dabei auf, dass eine Differenz stets nur eine bestimmte Code-Distanz erzeugt.

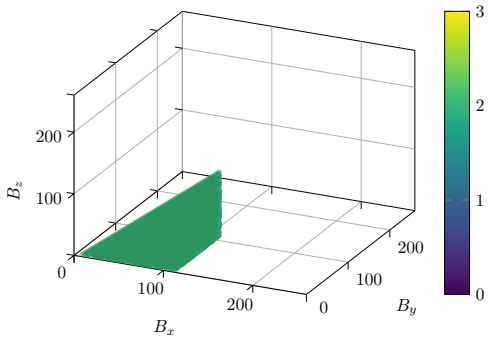


(a) Verteilung der Code-Distanzen für drei Signaturen für den Schlüssel $A = 113$.

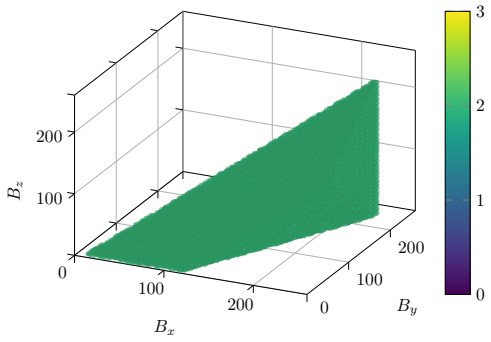


(b) Verteilung der Code-Distanzen für drei Signaturen für den Schlüssel $A = 233$.

Abbildung A.3: Verteilung der Code-Distanzen für drei Signaturen (Schlüssel A = 113 und A = 233). Für jeweils einen festen Schlüssel werden Signatur-Kombinationen aus jeweils drei Signaturen gebildet und die Code-Distanz für diese Kombinationen berechnet. Auf der x-, y- und z-Achse sind die Signaturen B_x, B_y und B_z abgetragen. Die Code-Distanz wird in der Grafik farblich codiert als vierte Dimension dargestellt.



(a) Verteilung der Code-Distanzen für drei Signaturen gefiltert auf $d_C = 2$ für den Schlüssel $A = 113$.



(b) Verteilung der Code-Distanzen für drei Signaturen gefiltert auf $d_C = 2$ für den Schlüssel $A = 233$.

Abbildung A.4: Verteilung der Code-Distanzen für drei Signaturen gefiltert auf $\mathbf{d_C = 2}$. Für jeweils einen festen Schlüssel - links $A = 113$ und rechts $A = 233$ - werden Signatur-Kombinationen variiert und die Code-Distanz berechnet, die sie erzeugen. Das Ergebnis wird auf $d_C = 2$ gefiltert, um potenziell Muster erkennen zu können

A.2 Ergebnistabellen der Fehlerinjektionsversuche

Die ersten Tabellen in diesem Anhang zeigen einen Überblick über die Ergebnisse der Fehlerinjektionsversuche in Bezug auf die Anzahl der Kampagnen mit unentdeckten Datenfehlern für die Baseline-Implementierung. Der Fokus liegt an dieser Stelle auf den Abweichungen der verschiedenen Stichprobenverfahren untereinander. Sind die Abweichungen besonders groß, muss angenommen werden, dass eine der Stichproben ungünstig gezogen wurde und die Versuche werden mit einer weiteren Stichprobe erneut durchgeführt.

Eine Tabelle beschreibt jeweils die Ergebnisse eines Schlüssels. Für jeden Schlüssel werden zwei Stichprobenverfahren angewendet. Daher existieren für jeden Schlüssel mindestens zwei Tabellen. Diese werden jeweils auf einer Seite gruppiert. Für den Fall $A = 113$ existieren sogar drei Tabellen, weil die Ergebnisse der ersten Versuche erkennen ließen, dass vermutlich eine ungünstige Stichprobe zufällig gezogen wurde. Hängen Daten aus einer Tabelle von Daten aus einer anderen Tabelle ab, wird dieser Bezug durch eine entsprechende Referenz verdeutlicht.

Diese Tabellen sind daher alle identisch aufgebaut. Das verwendete Stichprobenverfahren ist in der Beschreibung der jeweiligen Tabelle vermerkt. Spalte eins der Tabellen enthält stets den Schlüssel, Spalte zwei die jeweilige Regel, die für diese Zeile gilt. Die jeweilige Stichprobengröße ist in der Spalte *Stichprobengröße* zu sehen. Die Abkürzung *# SDCs* in der vierten Spalte steht für die Anzahl der Kampagnen mit verschiedenen Schlüssel-Signatur-Kombinationen aus der Stichprobe, für die bei der Fehlerinjektion unentdeckte Datenfehler auftreten. In der fünften Spalte *# SDCs (%)* ist die Anzahl der Kampagnen mit verschiedenen Schlüssel-Signatur-Kombinationen, die zu SDCs führen, in Prozent in Abhängigkeit von der Stichprobengröße dargestellt. Die Spalte *δ <Tabellenreferenz> (%p)* zeigt die Differenz in Prozentpunkten (%p) zwischen den Werten aus der Spalte *# SDCs (%)* und den Werten aus der Spalte *# SDCs (%)* der Tabelle <Tabellenreferenz>.

Die zweite Tabellenart (Tabelle A.10 bis A.14) präsentiert die Ergebnisse der Fehlerinjektionsversuche in Bezug auf die Anzahl der Kampagnen mit unentdeckten Datenfehlern für die $v_{c,\max}$-Implementierung. Hier wurde lediglich das Stichprobenverfahren 1 (V1) eingesetzt, da eine erste Beurteilung der Stichprobe bereits durch die Versuche zur Baseline-Implementierung erfolgt sind und es zeitliche Einschränkungen in Bezug auf den Umfang der Arbeit gab. Aus diesem Grund sind die Tabellen für die $v_{c,\max}$-Implementierung etwas anders aufgebaut. Die Spalten eins bis fünf sind identisch. Die letzte, sechste Spalte entfällt.

Es folgt eine dritte Tabellenart, welche die minimale und maximale Anzahl von Kontrollflussfehlern pro Kampagne untersucht. Die Tabelle A.15 betrachtet dabei die Baseline-Implementierung; die Tabelle A.16 zeigt die Ergebnisse der $v_{c,\max}$-Implementierung. Beide Tabellen enthalten in der ersten Spalte den betrachteten Schlüssel. Die zweite Spalte gibt den verwendeten Benchmark (Baseline, Baseline & $d_C > 1$, **N1**, **N1** & $d_C > 1$) an. In

Spalte drei ($min\{ip\}$) ist die minimale Anzahl der Kontrollflussfehler aller Kampagnen der Stichprobe aufgeführt, die noch unentdeckte Kontrollflussfehler aufweisen; in Spalte vier ($max\{ip\}$) die maximale Anzahl unentdeckter Kontrollflussfehler aller Kampagnen.

A	Regel	Stichprobengröße	# SDCs	# SDCs (%)	δ A.3 (%p)
113	Baseline	1.000	46	$4,60\%$	$0,00\%p$
113	Baseline und $d_C > 1$	388	8	$2,06\%$	$3,24\%p$
113	**N1**	196	5	$2,55\%$	$3,45\%p$
113	**N1** und $d_C > 1$	41	0	$0,00\%$	$6,00\%p$

Tabelle A.1: Anzahl Schlüssel-Signatur-Kombinationen mit SDCs für A = 113 bei Aktivierung verschiedener Regeln und ungünstiger Stichprobenwahl. Es wird die Baseline-Implementierung injiziert. Für die Daten dieser Tabelle wurde das Stichprobenverfahren 1 (V1) eingesetzt. Es fällt auf, dass die Differenzen zwischen den ermittelten Raten (Spalte δ *A.3*) allesamt größer als die errechnete Fehlerspanne $3,1\%$ ist. Die betroffenen Felder sind rot markiert. Es ist insbesondere zu erkennen, dass die Abweichungen zu den Ergebnissen der Tabelle A.3 von Zeile zu Zeile größer werden. Daher kann vermutet werden, dass die hier verwendete Basisstichprobe ungünstig gezogen wurde. Aus diesem Grund wurde eine weitere Stichprobe gezogen und dieser Versuch erneut durchgeführt. Die Ergebnisse sind in Tabelle A.2 dargestellt.

A	Regel	Stichprobengröße	# SDCs	# SDCs (%)	δ A.3 (%p)
113	Baseline	1.000	71	$7,10\%$	$0,00\%p$
113	Baseline und $d_C > 1$	406	27	$6,65\%$	$1,35\%p$
113	**N1**	209	17	$8,13\%$	$2,13\%p$
113	**N1** und $d_C > 1$	49	3	$6,12\%$	$0,12\%p$

Tabelle A.2: Anzahl Schlüssel-Signatur-Kombinationen mit SDCs für A = 113 bei Aktivierung verschiedener Regeln. Es wird die Baseline-Implementierung injiziert. Für die Daten dieser Tabelle wird das Stichprobenverfahren 1 (V1) eingesetzt. Die Werte der Spalte δ *A.3* sind alle nicht größer als die errechnete Fehlerspanne $3,1\%$ - im Gegensatz zu den Werten in Tabelle A.1. Daher kann vermutet werden, dass die Ergebnisse dieser Tabelle repräsentativer sind als die Ergebnisse aus Tabelle A.1. Die Daten dieser Tabelle sind in die Auswertung eingeflossen.

A	Regel	Stichprobengröße	# SDCs	# SDCs (%)	δ A.2 (%p)
113	Baseline	1.000	71	$7,10\%$	$0,00\%p$
113	Baseline und $d_C > 1$	1.000	53	$5,30\%$	$1,35\%p$
113	**N1**	1.000	60	$6,00\%$	$2,13\%p$
113	**N1** und $d_C > 1$	1.000	60	$6,00\%$	$0,12\%p$

Tabelle A.3: Anzahl Schlüssel-Signatur-Kombinationen mit SDCs für A = 113 bei Aktivierung verschiedener Regeln. Es wird die Baseline-Implementierung injiziert. Für die Daten dieser Tabelle wird das Stichprobenverfahren 2 (V2) eingesetzt.

A	Regel	Stichprobengröße	# SDCs	# SDCs (%)	δ A.5 (%p)
185	Baseline	1.000	62	$6,20\%$	$0,00\%p$
185	Baseline und $d_C > 1$	553	38	$6,87\%$	$2,57\%p$
185	**N1**	215	12	$5,58\%$	$1,88\%p$
185	**N1** und $d_C > 1$	93	4	$4,30\%$	$0,50\%p$

Tabelle A.4: Anzahl Schlüssel-Signatur-Kombinationen mit SDCs für A = 185 bei Aktivierung verschiedener Regeln. Es wird die Baseline-Implementierung injiziert. Für die Daten dieser Tabelle wird das Stichprobenverfahren 1 (V1) eingesetzt. Die Abweichungen zu den Ergebnissen der Tabelle A.5 sind alle geringer als die Fehlerspanne mit $3,1\%$, obgleich die Abweichung in Zeile zwei der Tabelle mit $2,57\%p$ bereits relativ groß ist.

A	Regel	Stichprobengröße	# SDCs	# SDCs (%)	δ A.4 (%p)
185	Baseline	1.000	62	$6,20\%$	$0,00\%p$
185	Baseline und $d_C > 1$	1.000	43	$4,30\%$	$2,57\%p$
185	**N1**	1.000	37	$3,70\%$	$1,88\%p$
185	**N1** und $d_C > 1$	1.000	38	$3,80\%$	$0,50\%p$

Tabelle A.5: Anzahl Schlüssel-Signatur-Kombinationen mit SDCs für A = 185 bei Aktivierung verschiedener Regeln. Es wird die Baseline-Implementierung injiziert. Für die Daten dieser Tabelle wird das Stichprobenverfahren 2 (V2) eingesetzt. Die Abweichungen zu den Ergebnissen der Tabelle A.4 sind alle geringer als die Fehlerspanne mit $3,1\%$, obgleich die Abweichung in Zeile zwei der Tabelle mit $2,57\%p$ bereits relativ groß ist.

A	Regel	Stichprobengröße	# SDCs	# SDCs (%)	δ A.7 (%p)
233	Baseline	1.000	41	$4,10\%$	$0,00\%p$
233	Baseline und $d_C > 1$	642	31	$4,83\%$	$1,07\%p$
233	**N1**	213	7	$3,29\%$	$2,41\%p$
233	**N1** und $d_C > 1$	115	6	$5,22\%$	$0,12\%p$

Tabelle A.6: Anzahl Schlüssel-Signatur-Kombinationen mit SDCs für A = 233 bei Aktivierung verschiedener Regeln. Es wird die Baseline-Implementierung injiziert. Für die Daten dieser Tabelle wird das Stichprobenverfahren 1 (V1) eingesetzt. Die Abweichungen in Spalte fünf sind bis auf jene für die Regel **N1** relativ gering. Alle Abweichungen sind geringer als die Fehlerspanne.

A	Regel	Stichprobengröße	# SDCs	# SDCs (%)	δ A.6 (%p)
233	Baseline	1.000	41	$4,10\%$	$0,00\%p$
233	Baseline und $d_C > 1$	1.000	59	$5,90\%$	$1,07\%p$
233	**N1**	1.000	57	$5,70\%$	$2,41\%p$
233	**N1** und $d_C > 1$	1.000	51	$5,10\%$	$0,12\%p$

Tabelle A.7: Anzahl Schlüssel-Signatur-Kombinationen mit SDCs für A = 233 bei Aktivierung verschiedener Regeln. Es wird die Baseline-Implementierung injiziert. Für die Daten dieser Tabelle wird das Stichprobenverfahren 2 (V2) eingesetzt. Die Abweichungen in Spalte fünf sind bis auf jene für die Regel **N1** in Zeile drei relativ gering. Alle Abweichungen sind geringer als die Fehlerspanne.

A	Regel	Stichprobengröße	# SDCs	# SDCs (%)	δ A.9 (%p)
241	Baseline	1.000	42	$4,20\%$	$0,00\%p$
241	Baseline und $d_C > 1$	695	28	$4,03\%$	$1,57\%p$
241	**N1**	223	9	$4,04\%$	$0,64\%p$
241	**N1** und $d_C > 1$	118	6	$5,08\%$	$1,68\%p$

Tabelle A.8: Anzahl Schlüssel-Signatur-Kombinationen mit SDCs für A = 241 bei Aktivierung verschiedener Regeln. Es wird die Baseline-Implementierung injiziert. Für die Daten dieser Tabelle wird das Stichprobenverfahren 1 (V1) eingesetzt. Die Abweichungen in der Spalte δ *A.9 (%p)* sind alle relativ gering.

A	Regel	Stichprobengröße	# SDCs	# SDCs (%)	δ A.8 (%p)
241	Baseline	1.000	42	$4,20\%$	$0,00\%p$
241	Baseline und $d_C > 1$	1.000	56	$5,60\%$	$1,57\%p$
241	**N1**	1.000	34	$3,40\%$	$0,64\%p$
241	**N1** und $d_C > 1$	1.000	34	$3,40\%$	$1,68\%p$

Tabelle A.9: Anzahl Schlüssel-Signatur-Kombinationen mit SDCs für A = 241 bei Aktivierung verschiedener Regeln. Es wird die Baseline-Implementierung injiziert. Für die Daten dieser Tabelle wird das Stichprobenverfahren 2 (V2) eingesetzt. Die Abweichungen in der Spalte δ *A.8 (%p)* sind alle relativ gering.

A	Regel	Stichprobengröße	# SDCs	# SDCs (%)
113	Baseline	1.000	27	2,70%
113	Baseline und $d_C > 1$	388	5	1,29%
113	**N1**	196	0	0,00%
113	**N1** und $d_C > 1$	41	0	0,00%

Tabelle A.10: Anzahl Schlüssel-Signatur-Kombinationen mit SDCs für A = 113 bei Aktivierung verschiedener Regeln und ungünstiger Stichprobenwahl. Es wird die $v_{c,\max}$-Implementierung injiziert. Für die Daten dieser Tabelle wird das Stichprobenverfahren 1 (V1) eingesetzt. Im Zuge der Versuche für die Tabellen A.1 und A.3 wurde festgestellt, dass die Stichprobenwahl ungünstig ist aufgrund großer Abweichungen zwischen den Versuchen für die Baseline-Implementierung. Daher wurde auch hier die Messung für eine weitere Stichprobe wiederholt (vgl. Tabelle A.11).

A	Regel	Stichprobengröße	# SDCs	# SDCs (%)
113	Baseline	1.000	39	3,90%
113	Baseline und $d_C > 1$	406	17	4,19%
113	**N1**	209	4	1,91%
113	**N1** und $d_C > 1$	49	2	4,08%

Tabelle A.11: Anzahl Schlüssel-Signatur-Kombinationen mit SDCs für A = 113 bei Aktivierung verschiedener Regeln. Es wird die $v_{c,\max}$-Implementierung injiziert. Für die Daten dieser Tabelle wird das Stichprobenverfahren 1 (V1) eingesetzt.

A	Regel	Stichprobengröße	# SDCs	# SDCs (%)
185	Baseline	1.000	29	2,90%
185	Baseline und $d_C > 1$	553	15	2,71%
185	**N1**	215	5	2,33%
185	**N1** und $d_C > 1$	93	0	0,00%

Tabelle A.12: Anzahl Schlüssel-Signatur-Kombinationen mit SDCs für A = 185 bei Aktivierung verschiedener Regeln. Es wird die $v_{c,\max}$-Implementierung injiziert. Für die Daten dieser Tabelle wird das Stichprobenverfahren 1 (V1) eingesetzt.

A	Regel	Stichprobengröße	# SDCs	# SDCs (%)
233	Baseline	1.000	28	2,80%
233	Baseline und $d_C > 1$	642	21	3,27%
233	**N1**	213	5	2,35%
233	**N1** und $d_C > 1$	115	4	3,48%

Tabelle A.13: Anzahl Schlüssel-Signatur-Kombinationen mit SDCs für A = 233 bei Aktivierung verschiedener Regeln. Es wird die $v_{c,\max}$-Implementierung injiziert. Für die Daten dieser Tabelle wird das Stichprobenverfahren 1 (V1) eingesetzt.

A	Regel	Stichprobengröße	# SDCs	# SDCs (%)
241	Baseline	1.000	28	2,80%
241	Baseline und $d_C > 1$	695	18	2,59%
241	**N1**	223	4	1,79%
241	**N1** und $d_C > 1$	118	3	2,54%

Tabelle A.14: Anzahl Schlüssel-Signatur-Kombinationen mit SDCs für A = 241 bei Aktivierung verschiedener Regeln. Es wird die $v_{c,\max}$-Implementierung injiziert. Für die Daten dieser Tabelle wird das Stichprobenverfahren 1 (V1) eingesetzt.

Schlüssel	Regel	min{ip}	max{ip}
113	Baseline	1	3
113	Baseline & $d_C > 1$	1	3
113	**N1**	1	2
113	**N1** & $d_C > 1$	1	1
185	Baseline	1	5
185	Baseline & $d_C > 1$	1	5
185	**N1**	1	3
185	**N1** & $d_C > 1$	1	3
233	Baseline	1	2
233	Baseline & $d_C > 1$	1	2
233	**N1**	1	3
233	**N1** & $d_C > 1$	1	3
241	Baseline	1	5
241	Baseline & $d_C > 1$	1	5
241	**N1**	1	2
241	**N1** & $d_C > 1$	1	2

Tabelle A.15: Minimale und maximale Anzahl von Kontrollflussfehlern pro Kampagne für alle Benchmarks im Vergleich. Es wird die Baseline-Implementierung injiziert. Für die Daten dieser Tabelle wird das Stichprobenverfahren 1 (V1) eingesetzt. Es werden lediglich Kontrollflussfehler betrachtet.

Schlüssel	Regel	min{ip}	max{ip}
113	Baseline	1	3
113	Baseline & $d_C > 1$	1	3
113	**N1**	1	2
113	**N1** & $d_C > 1$	1	1
185	Baseline	1	2
185	Baseline & $d_C > 1$	1	2
185	**N1**	1	2
185	**N1** & $d_C > 1$	0	0
233	Baseline	1	2
233	Baseline & $d_C > 1$	1	2
233	**N1**	1	3
233	**N1** & $d_C > 1$	1	3
241	Baseline	1	2
241	Baseline & $d_C > 1$	1	2
241	**N1**	1	2
241	**N1** & $d_C > 1$	1	2

Tabelle A.16: Minimale und maximale Anzahl von Kontrollflussfehlern pro Kampagne für alle Benchmarks im Vergleich. Es wird die $v_{c,\max}$-Implementierung injiziert. Für die Daten dieser Tabelle wird das Stichprobenverfahren 1 (V1) eingesetzt. Es werden lediglich Kontrollflussfehler betrachtet.

Abkürzungsverzeichnis

ALU	Arithmetisch-logische Einheit
CoRed	Combined Redundancy
DRAM	Dynamic Random Access Memory
DUE	Detected Unrecoverable Error
ECC	Error Correcting Code
Fail*	Fault Injection Leveraged
ISBN	internationale Standardbuchnummer
MPU	Memory Protection Unit
RAM	Random Access Memory
SDC	Silent Data Corruption
SPoF	Single Point of Failure
TMR	Triple Modular Redundancy

Abbildungsverzeichnis

Algorithmenverzeichnis

Tabellenverzeichnis

Literatur

[1] J. B. Almeida u. a. *Rigorous Software Development : An Introduction to Program Verification*. Hrsg. von I. Mackie. London [u.a.]: Springer London, 2011. ISBN: 978-0-85729-017-5. DOI: 10.1007/978-0-85729-018-2.

[2] J. Arlat, Y. Crouzet und J.-C. Laprie. "Fault injection for dependability validation of fault-tolerant computing systems". In: *The Nineteenth International Symposium on Fault-Tolerant Computing*. 1989, S. 348–355. DOI: 10.1109/FTCS.1989.105591.

[3] J. Arlat u. a. "Fault Injection for Dependability Validation: A Methodology and Some Applications". In: *IEEE Transactions on Software Engineering* 16.2 (1990), S. 166–182. DOI: 10.1109/32.44380.

[4] A. Avižienis. "Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design". In: *IEEE Transactions on Computers* C-20.11 (1971), S. 1322–1331. DOI: 10.1109/T-C.1971.223134.

[5] A. Avižienis u. a. "Basic Concepts and Taxonomy of Dependable and Secure Computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), S. 11–33. DOI: 10.1109/TDSC.2004.2.

[6] R. Baumann. "Soft Errors in Advanced Computer Systems". In: *IEEE Design & Test of Computers* 22.3 (2005), S. 258–266. DOI: 10.1109/MDT.2005.69.

[7] F. Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. 2005, S. 41–46.

[8] D. Binder, E. C. Smith und A. B. Holman. "Satellite Anomalies from Galactic Cosmic Rays". In: *IEEE Transactions on Nuclear Science* 22.6 (1975), S. 2675–2680. DOI: 10.1109/TNS.1975.4328188.

[9] N. Binkert u. a. "The Gem5 Simulator". In: *ACM SIGARCH Computer Architecture News* 39.2 (Aug. 2011), S. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718.

[10] P. Bishop. "Software Fault Tolerance by Design Diversity". In: *Software Fault Tolerance*. Hrsg. von M. R. Lyu. John Wiley & Sons Ltd., 1995, S. 211–230. ISBN: 04-719-5068-8. URL: https://www.cse.cuhk.edu.hk/~lyu/book/sft/index.html.

[11] J. Braun und J. Mottok. "The Myths of Coded Processing". In: *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. 2015, S. 1637–1644. DOI: 10.1109/HPCC-CSS-ICESS.2015.24.

[12] D. T. Brown. "Error Detecting and Correcting Binary Codes for Arithmetic Operations". In: *IRE Transactions on Electronic Computers* EC-9.3 (1960), S. 333–337. DOI: 10.1109/TEC.1960.5219855.

[13] J. Chang, G. A. Reis und D. I. August. "Automatic Instruction-Level Software-Only Recovery". In: *International Conference on Dependable Systems and Networks (DSN'06)*. 2006, S. 83–92. DOI: 10.1109/DSN.2006.15.

[14] C. Constantinescu. "Trends and challenges in VLSI circuit reliability". In: *IEEE Micro* 23.4 (2003), S. 14–19. DOI: 10.1109/MM.2003.1225959.

[15] E. Dubrova. *Fault-Tolerant Design*. Springer New York, NY, 2013. ISBN: 978-1-4939-0240-8. DOI: 10.1007/978-1-4614-2113-9.

[16] K. Echtle. *Fehlertoleranzverfahren*. Springer Berlin Heidelberg, 1990. ISBN: 978-3-6427-5765-5. DOI: 10.1007/978-3-642-75765-5.

[17] S. Edwards u. a. "Design of Embedded Systems: Formal Models, Validation, and Synthesis". In: *Proceedings of the IEEE* 85.3 (1997), S. 366–390. DOI: 10.1109/5.558710.

[18] C. Fetzer, U. Schiffel und M. Süßkraut. "AN-Encoding Compiler: Building Safety-Critical Systems with Commodity Hardware". In: *Computer Safety, Reliability, and Security*. Hrsg. von B. Buth, G. Rabe und T. Seyfarth. Springer Berlin Heidelberg, 2009, S. 283–296. ISBN: 978-3-6420-4468-7. DOI: 10.1007/978-3-642-04468-7_23.

[19] P. Forin. "Vital Coded Microprocessor Principles and Application for Various Transit Systems". In: *Proceedings of the IFAC IFIP/IFORS Symposium on Control, Computers, Communications in Transportation* (1989), S. 79–84. DOI: 10.1016/S1474-6670(17)52653-1.

[20] R. A. Frohwerk. "Signature analysis: A new digital field service method". In: *Hewlett-Packard Journal* 28.9 (1977), S. 2–8.

[21] O. Goloubeva u. a. *Software-Implemented Hardware Fault Tolerance*. Springer, 2006. ISBN: 978-0-3872-6060-0. DOI: 10.1007/0-387-32937-4.

[22] R. W. Hamming. "Error detecting and error correcting codes". In: *The Bell System Technical Journal* 29.2 (1950), S. 147–160. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

[23] J. L. Hennessy und D. A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Hrsg. von S. Merken und N. McFadden. Sixth. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN: 01-281-1905-5.

[24] M. Hoffmann u. a. "A Practitioner's Guide to Software-Based Soft-Error Mitigation Using AN-Codes". In: *IEEE International Symposim on High Assurance Systems Engineering*. 2014, S. 33–40. DOI: 10.1109/HASE.2014.14.

[25] M.-C. Hsueh, T. K. Tsai und R. K. Iyer. "Fault injection techniques and tools". In: *Computer* 30.4 (1997), S. 75–82. DOI: 10.1109/2.585157.

[26] G. A. Kanawati, N. A. Kanawati und J. A. Abraham. "FERRARI: A Flexible Software-Based Fault and Error Injection System". In: *IEEE Transactions on Computers* 44.2 (1995), S. 248–260. DOI: 10.1109/12.364536.

[27] H. Kopetz und W. Steiner. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Third. Springer Cham, 2022. ISBN: 978-3-0311-1992-7. DOI: 10.1007/978-3-031-11992-7.

[28] I. Koren und C. M. Krishna. *Fault-Tolerant Systems*. Hrsg. von I. Koren und C. M. Krishna. Second. San Francisco (CA): Morgan Kaufmann, 2021, S. 161–202. ISBN: 978-0-1281-8105-8. DOI: 10.1016/B978-0-12-818105-8.00015-2.

[29] L. Lamport, R. Shostak und M. Pease. "The Byzantine Generals Problem". In: *ACM Transactions on Programming Languages and Systems* (Juli 1982), S. 382–401. DOI: 10.1145/357172.357176.

[30] K. P. Lawton. "Bochs: A Portable PC Emulator for Unix/X". In: *Linux Journal* 1996.29es (Sep. 1996), S. 7. ISSN: 1075-3583.

[31] J. H. van Lint. *Introduction to Coding Theory*. Hrsg. von F. W. Gehring, P. R. Halmos und C. C. Moore. Springer Berlin, Heidelberg, 1982. ISBN: 978-3-6620-7998-0. DOI: 10.1007/978-3-662-07998-0.

[32] H. Madeira u. a. "RIFLE: A General Purpose Pin-Level Fault Injector". In: *Dependable Computing - EDCC-1*. Hrsg. von K. Echtle, D. Hammer und D. Powell. Springer Berlin Heidelberg, 1994, S. 197–216. ISBN: 978-3-5404-8785-2. DOI: 10.1007/3-540-58426-9_132.

[33] J. Maiz u. a. "Characterization of Multi-bit Soft Error events in advanced SRAMs". In: *IEEE International Electron Devices Meeting*. 2003. DOI: 10.1109/IEDM.2003.1269335.

[34] M. M. Mano und M. Ciletti. *Digital Design: With an Introduction to the Verilog HDL*. Hrsg. von A. Gilfillan. Fifth. Pearson Deutschland, 2012. ISBN: 978-0-1327-7420-8.

[35] T. C. May und M. H. Woods. "Alpha-particle-induced soft errors in dynamic memories". In: *IEEE Transactions on Electron Devices* 26.1 (1979), S. 2–9. DOI: 10.1109/T-ED.1979.19370.

[36] S. Mukherjee. *Architecture Design for Soft Errors*. Hrsg. von S. Mukherjee. San Francisco, CA, USA: Morgan Kaufmann, 2008. ISBN: 978-0-1236-9529-1. DOI: 10.1016/B978-0-12-369529-1.X5001-0.

[37] P. Munk. "A software fault-tolerance mechanism for mixed-critical real-time applications on consumer-grade many-core processors". Diss. Technische Universität Berlin, 2016.

[38] P. Munk u. a. "A software fault-tolerance mechanism for real-time applications on many-core processors". In: *The Workshop on Highly-Reliable Power-Efficient Embedded Designs*. 2016.

[39] E. B. Nightingale, J. R. Douceur und V. Orgovan. "Cycles, Cells and Platters: An Empirical Analysis of Hardware Failures on a Million Consumer PCs". In: *Proceedings of the ACM SIGOPS / EuroSys European Conference on Computer Systems*. 2011, S. 343–356. DOI: 10.1145/1966445.1966477.

[40] T. J. O'Gorman. "The Effect of Cosmic Rays on the Soft Error Rate of a DRAM at Ground Level". In: *IEEE Transactions on Electron Devices* 41.4 (1994), S. 553–557. DOI: 10.1109/16.278509.

[41] N. Oh, S. Mitra und E. J. McCluskey. "ED^4I: Error Detection by Diverse Data and Duplicated Instructions". In: *IEEE Transactions on Computers* 51.2 (2002), S. 180–199. DOI: 10.1109/12.980007.

[42] L. Osinski und J. Mottok. "S^3DES - Scalable Software Support for Dependable Embedded Systems". In: *Architecture of Computing Systems*. Hrsg. von M. Schoeberl u. a. Cham: Springer International Publishing, 2019, S. 15–27. ISBN: 978-3-0301-8656-2.

[43] D. A. Patterson und J. L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Fifth. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. ISBN: 01-240-7726-9.

[44] W. W. Peterson und E. J. Weldon Jr. *Error-Correcting Codes*. Second. Cambridge, MA, USA: MIT Press, März 1972. ISBN: 978-0-2625-2731-6.

[45] S. Poledna. "Replica Determinism in Distributed Real-TimeSystems: A Brief Survey". In: *Real-Time Systems* 6.3 (Mai 1994), S. 289–316. ISSN: 1573-1383. DOI: 10.1007/BF01088629.

[46] T. R. N. Rao. *Error Coding for Arithmetic Processors*. Hrsg. von H. G. Booker und N. DeClaris. London, United Kingdom: Academic Press, 1974. ISBN: 978-0-1239-4228-9.

[47] U. Schiffel. "Hardware Error Detection Using AN-Codes". Diss. Technische Universität Dresden, 2011.

[48] U. Schiffel u. a. "ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software". In: *Computer Safety, Reliability, and Security*. Hrsg. von E. Schoitsch. Springer Berlin Heidelberg, 2010, S. 169–182. ISBN: 978-3-6421-5651-9. DOI: 10.1007/978-3-642-15651-9_13.

[49] H. Schirmeier. "Efficient Fault-Injection-based Assessment of Software-Implemented Hardware Fault Tolerance". Diss. Technische Universität Dortmund, 2016.

[50] H. Schirmeier u. a. "FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance". In: *2015 11th European Dependable Computing Conference (EDCC)*. 2015, S. 245–255. DOI: 10.1109/EDCC.2015.28.

[51] B. Schroeder, E. Pinheiro und W.-D. Weber. "DRAM Errors in the Wild: A Large-Scale Field Study". In: *Communications of the ACM* 54.2 (Feb. 2011), S. 100–107. ISSN: 0001-0782. DOI: 10.1145/1897816.1897844.

[52] W. Schütz. "Fundamental issues in testing distributed real-time systems". In: *Real-Time Systems* 7.2 (Sep. 1994), S. 129–157. ISSN: 1573-1383. DOI: 10.1007/BF01088802.

[53] M. L. Shooman. *Reliability of Computer Systems and Networks: Fault Tolerance, Analysis, and Design*. John Wiley & Sons, Ltd, 2002. ISBN: 978-0-4712-2460-0. DOI: 10.1002/047122460X.ch4.

[54] H. Stichtenoth. *Algebraic Function Fields and Codes*. Hrsg. von S. Axler und K. A. Ribet. Second. Springer Publishing Company, Incorporated, 2008. ISBN: 35-407-6877-7. DOI: 10.1007/978-3-540-76878-4.

[55] P. Ulbrich. "Ganzheitliche Fehlertoleranz in eingebetteten Softwaresystemen". Diss. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2014.

[56] P. Ulbrich. *Vorlesung Verlässliche Systemsoftware (VSS) - Kapitel 6 Fehlerinjektion*. Vorlesungsfolien Modul „Verlässliche Systemsoftware" TU Dortmund. Wintersemester 2020. URL: https://sys-old.cs.tu-dortmund.de/Teaching/WS2020/VSS/Vorlesung/.

[57] P. Ulbrich u. a. "Eliminating Single Points of Failure in Software-Based Redundancy". In: *Ninth European Dependable Computing Conference*. 2012, S. 49–60. DOI: 10.1109/EDCC.2012.21.

[58] U. Wappler und M. Müller. "Software Protection Mechanisms for Dependable Systems". In: *Proceedings of the Conference on Design, Automation and Test in Europe*. Association for Computing Machinery, 2008, S. 947–952. ISBN: 978-3-9810-8013-1. DOI: 10.1145/1403375.1403604.

[59] R. Wattenhofer. *Blockchain Science: Distributed Ledger Technology*. Inverted Forest Publishing, 2019. ISBN: 978-1-7934-7173-4.

[60] C. Weaver u. a. "Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor". In: *Proceedings. 31st Annual International Symposium on Computer Architecture*. 2004, S. 264–275. DOI: 10.1109/ISCA.2004.1310780.

[61] Y. C. Yeh. "Triple-triple redundant 777 primary flight computer". In: *IEEE Aerospace Applications Conference*. Bd. 1. 1996, S. 293–307. DOI: 10.1109/AERO.1996.495891.

[62] H. Ziade, R. A. Ayoubi und R. Velazco. "A Survey on Fault Injection Techniques". In: *The International Arab Journal of Information Technology* 1 (2004), S. 171–186.