Funktionale Sicherheit in der Medizintechnik: Was kann man mit Software erreichen?

Martin Süßkraut[*], Jörg Kaienburg[†] und Christof Fetzer[‡]

4. Oktober 2015

1 Einführung

In der Medizintechnik hat die funktionale Sicherheit zum Ziel, medizinische Geräte derart auszulegen, dass jeder anzunehmende Defekt rechtzeitig erkannt wird, sodass davon keine Gefährdung für den Patienten ausgeht. Nicht zuletzt aufgrund der Relevanz für das menschliche Leben wurde mit der Normenreihe EN 60601-X ein regulatives Rahmenwerk geschaffen, das die Sicherheits- und ergonomischen Anforderungen an medizinische elektrische Geräte und in medizinischen Systemen definiert.

In anderen Industriesektoren werden ebenfalls besondere Anforderungen an die Produkte und Systeme gestellt, die über jeweils geltende Industriestandards festgesetzt werden. Beispielsweise definiert die internationale Norm IEC 61508 die Anforderungen an die Entwicklung von elektrischen, elektronischen und programmierbaren elektronischen Systemen, die eine Sicherheitsfunktion ausführen.

Trotz der Unterschiede zwischen den Industriesektoren und den darin zum Einsatz kommenden Produkten steht eine Frage immer wieder im Raum und muss mit einer hinreichend hohen Zuverlässigkeit beantwortet werden: Kann ein (funktionaler) Defekt eines sicherheitsrelevanten Produktes rechtzeitig erkannt und eine Gefährdung von Personen oder der Umwelt verlässlich verhindert werden?

Dieser Artikel widmet sich Software-Verfahren, die zum Ziel haben, sog. Hardware-Ausführungsfehler zu erkennen, die zu einem Fehlverhalten von Produkten führen. Die

[*]SIListra Systems GmbH, Königsbrückerstr. 124, 01099, Dresden, martin.suesskraut@silistra-systems.com
[†]SIListra Systems GmbH, Königsbrückerstr. 124, 01099, Dresden, joerg.kaienburg@silistra-systems.com
[‡]Technische Universität Dresden, Fakultät Informatik, Institut für Systemarchitekture, 01089, Dresden, christof.fetzer@tu-dresden.de

vorgestellten Verfahren erkennen Defekte und verhindern Gefährdungen, z. B. von Patienten, die während einer Operation von medizinischen Geräten versorgt werden. In dem Artikel wird von einem exemplarischen Medizingerät ausgegangen, das eine sicherheitskritische Funktion ausübt: die Versorgung eines Patienten (Medikament, Sauerstoff). Dieses Medizingerät ist teilweise aus Consumer-Komponenten aufgebaut, die nicht für den Einsatz in sicherheitsgerichteten Medizinanwendungen entwickelt worden sind. Es wird gezeigt, dass man trotzdem mit solchen Komponenten funktional sichere Produkte entwickeln kann. Dazu bedarf es geeigneter Maßnahmen, die Fehlverhalten erkennen und offenbaren.

Den Fokus legt der Artikel auf das Verfahren Diversified Encoding. Bei diesem Verfahren handelt es sich um ein innovatives Erkennungsverfahren, das vollständig in Software umgesetzt und demzufolge komplett unabhängig von der zugrunde liegenden Hardware ist. Es erkennt transiente, permanente und systematische Fehler und erreicht Erkennungsraten der höchsten Sicherheitsanforderungen, z. B. SIL 4. In diesem Artikel werden das Prinzip des Verfahrens, die grundlegenden Annahmen, eine Möglichkeit zur Anwendung vorgestellt. Die Vorteile von Diversified Encoding liegen darin, dass:

- die originäre Funktion eines damit geschützten Programms nicht geändert wird,
- das Verfahren automatisiert mit Hilfe eines Entwicklungswerkzeuges in das zu schützende Programm eingebaut wird,
- der Schutz kontinuierlich während der gesamten Laufzeit wirkt, verschleißfrei und komplett Hardware-unabhängig ist.

Insbesondere der zuletzt genannte Vorteil bedeutet, dass Entwicklern die Möglichkeit gegeben wird, sogar mit sog. COTS[1]-Hardware sicherheitskritische Medizingeräte entwickeln zu können.

2 Beispiel

Die exemplarische Sicherheitsfunktion aus diesem Abschnitt wird in den folgenden Abschnitten verwendet, um Diversified Encoding, seine Eigenschaften und Vorteile zu erklären. Das Beispiel konzentriert sich auf die Anwendung von Diversified Encoding mittels Code-Transformation.

Die Sicherheitsfunktion ist ein abgesicherter Zähler – beispielsweise für die abgegebene Wirkstoffmenge. Die Funktion hat die folgenden vereinfachte Spezifikation:

- Der Zähler wird durch eine 32 Bit vorzeichenlose Ganzzahl repräsentiert.
- Der Startwert des Zählers ist 1.
- Der Zähler hat eine Schnittstelle zum Abfrage und Erhöhen des Zählers um einen vorzeichenlosen Ganzzahlwert.
- Ausführungsfehler beim Verändern des Zählers müssen aufgedeckt werden.

Der Software-Architekt entwirft eine C-Schnittstelle zu dieser Spezifikation:
`uint32_t incSafeCounter(uint32_t incValue)`.

Die Implementierung enthält keinerlei Maßnahmen, um Ausführungsfehler aufzudecken. Abschnitte 3 und 4 demonstrieren, wie der nötige Quellcode für die Fähigkeit zur Auf-

[1] COTS = commercial of the shelf

deckung von Ausführungsfehler generiert wird. Besonders Abschnitt 4.2 geht auf die notwendigen Änderungen in der Schnittstelle des Zählers ein, um aufgedeckte Ausführungsfehler zu behandeln. Der folgende Quellcode zeigt eine mögliche Implementierung des Zählers (mit C99 fest-breiten Ganzahltypen[2]):

```
1  /* Standard C99 fest-breiten Ganzzahltypen */
2  #include <stdint.h>
3
4  /* Sicherheitskritische Zählervariable */
5  static uint32_t safeCounter = 1;
6
7  /* Implementierung der Schnittstelle */
8  uint32_t incSafeCounter(uint32_t incValue) {
9    safeCounter += incValue;
10   return safeCounter;
11 }
```

3 Software Coded Processing

Software Coded Processing (SCP) baut auf der Coded Processing Technologie von Forin [For89] auf. SCP fügt Informationsredundanz zu einem Software-Programm hinzu, um Ausführungsfehler aufzudecken. Um SCP in ein Software-Programm zu integrieren, muss das Programm entweder manuell umgeschrieben werden oder es kann ein automatiert arbeitendes Code-Transformationswerkzeug genutzt werden. SCP wird auf den gesamten sicherheitskritischen Datenfluss eines Programmes angewendet: alle Konstanten, Variablen und Operation müssen kodiert werden, um SCP verwenden zu können.

In der Literatur werden verschiedene Kodierungen beschrieben [For89, SSF09, SSSF10]. Einer der verbreitesten Kodierungen ist die AN-Kodierung[3]: Jeder Wert im Programm wird mit einer Konstante A vervielfacht [SSF09]. Werte, die kein Vielfaches von A sind, werden als ungültig gewertet. Mit SCP müssen alle Operationen mit diesen kodierten Werten arbeiten. Ein Ausführungsfehler erzeugt ungültig kodierte Werte.

Ein hilfreiches Beispiel ist die Addition $2 + 3$. Kodiert man mit $A = 7$ muss die Berechnung zu $14 + 21$ umgeschrieben werden. Ohne einen Ausführungsfehler ist das Ergebnis 35: ein gültiger Wert, weil er ein Vielfaches von 7 ist.

Zwei prinzipielle Möglichkeiten für Ausführungsfehler können das Ergebnis beeinflussen:

- Einer der beiden Operanden ist bereits ein ungültiger Wert. Dieser ungültiger Wert ist entweder das Resultat eines Bit-Fehlers im Speicher oder er wurde durch eine fehlerhafte Berechnung erzeugt. Sobald die 14 in eine 13 geändert wird, ist das Ergebis 34 kein Vielfaches von 7 und somit kein gültiger Wert.
- Die Additionsoperation selbst kann fehlerhaft ausgeführt werden. Zum Beispiel

[2] Der Datentyp `uint32_t` ist ein 32 Bit vorzeichenloser Ganzzahltyp. Der Datentype `uint64_t` ist ein 64 Bit vorzeichenloser Ganzzahltyp.

[3] AN ist der Name der Kodierung und keine Abkürzung.

kann die Operation zusätzlich 2 zum Ergebnis hinzuaddieren. Dann ist das Ergebnis 37 kein gültiger Wert.

Ein kodiertes Programm könnte zur Laufzeit jeden Wert zu jeder Zeit auf Gültigkeit überprüfen. Es ist jedoch vorteilhafter, nur die Ausgaben zu überprüfen [SSK15]. Zwischenüberprüfungen sind nicht nötig, weil Fehler durch den Datenfluss bis zu den Ergebnissen propagieren.

Der folgenden Quellcode zeigt das Beispiel aus Abschnitt 2 mit einer vereinfachten AN-Kodierung mit $A = 7$:

```
1  /* Standard C99 fest-breiten Ganzzahltypen */
2  #include <stdint.h>
3
4  /* 1 is AN-kodiert 7 */
5  static uint64_t safeCounter = 7;
6
7  uint64_t incSafeCounter_encoded(uint64_t incValue)
8  {
9    safeCounter = add_encoded(safeCounter, incValue);
10   return safeCounter;
11 }
```

Die wesentlichen Unterschiede zwischen der originalen, nativen Zählerimplementierung und der kodierten Zählerimplementierung sind:

Datentypen Die Datentypen wurden von `uint32_t` auf `uint64_t` geändert. SCP benötigt zusätzliche Bits für die Informationsredundanz.

Konstanten Alle Konstanten müssen AN-kodiert werden. Im Beispiel ist der Initialisierungswert von `safeCounter` nun 7.

Operationen Kodierte Operationen ersetzen alle nativen Operationen. Kodierte Operationen arbeiten ausschließlich auf kodierten Werten, während die native Operationen auf nativen Werten arbeiten. Zum Beispiel, liefert die Operation `add_encoded` die kodierte Summe ihrer beiden Operanden zurück.

Die Sicherheit der AN-Kodierung hängt von der Konstante A ab. Die Größe von A und die Hammingdistanz der resultierenden Kodierung beeinflussen die Sicherheit der AN-Kodierung [SSK15].

4 Diversified Encoding

4.1 Überblick

Diversified Encoding basiert auf zwei unterschiedlichen Ausführungen derselben Sicherheitsfunktion. Diese zwei Ausführungen sind:

Native Ausführung Die native Ausführung entspricht der Ausführung der originalen Sicherheitsfunktion ohne Kodierung. Der Quellcode für die originale Sicherheitsfunktion bildet dabei den Quellcode der nativen Ausführung. Die native Ausführung arbeitet auf nativen (originalen) Eingabewerten und dem nativen Zustand. Sie ändert nur den nativen Zustand. Das Ergebnis der nativen Ausführung ist die native Ausgabe.

Kodierte Ausführung Die kodierte Ausführung basiert auf der kodierten Variante der Sicherheitsfunktion. Sie arbeitet auf kodierten Eingabewerten und dem kodierten Zustand. Das Ergebnis ist die kodierte Ausgabe.

Beide Ausführungen sind vollkommen unabhängige Berechnungen, die jedoch auf den selben Werten operieren. Die kodierten Eingabewerte sind die kodierten Varianten der nativen Eingabewerte. Der Quellcode der nativen Sicherheitsfunktion wird verwendet, um den Quellcode der kodierten Ausführung zu erstellen. Die Erstellung kann entweder manuell oder – empfohlen für die Reproduzierbarkeit – mit einem Werkzeug erfolgen.

Das Diversity Framework verwaltet beide Ausführungen. Das Code-Transformierungswerkzeug erzeugt auch den Quellcode des Diversity Frameworks aus dem Quellcode der originalen Sicherheitsfunktion.

Die Komponente, die die Sicherheitsfunktion aufruft, erkennt und behandelt Ausführunsgfehler mit der Hilfe von Checksummen. O. B. d. A wird eine solche Komponente Aufrufer genannt. Das Diversity Frameworks erzeugt zwei Checksummen: eine über die nativen Ausgabewerte und eine über die kodierten Ausgabewerte. Die Aufruferkomponente arbeitet ausschließlich mit den nativen Ein- und Ausgaben. Nachdem die Sicherheitsfunktion ausgeführt wurde, muss die Aufruferkomponente die Checksumme über die nativen Ausgaben mit der Checksumme über die kodierten Ausgaben vergleichen. Sind die Checksummen verschieden, wurde ein Ausführungsfehler aufgedeckt und muss behandelt werden.

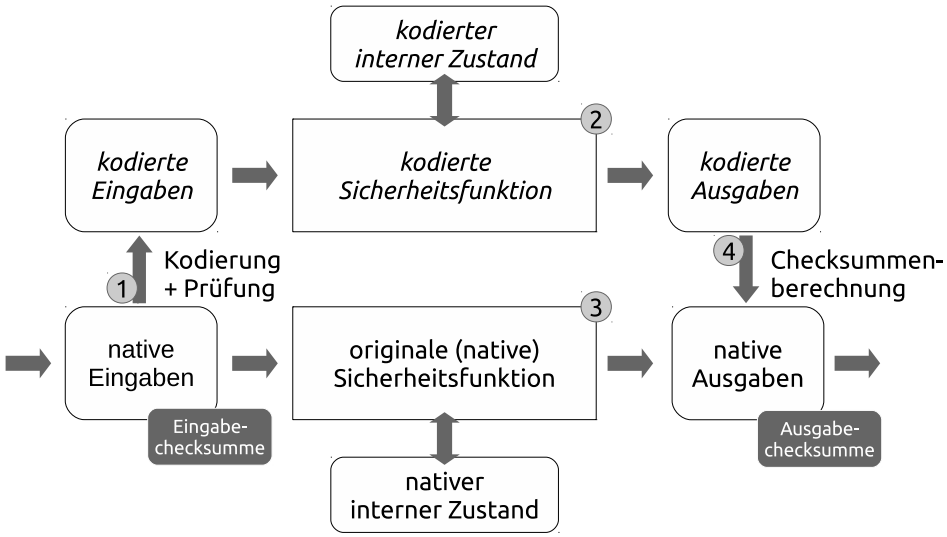

Abbildung 1: Das Datenflussmodell von Diversified Encoding.

Der Quellcode für die Schritte ①, ② und ④ aus Abb. 1 kann von einem Code-Transformationswerkzeug automatisch generiert werden. Schritte ① und ④ sind Teil des Diversity Frameworks und Schritt ② ist die kodierte Sicherheitsfunktion. Der Quellcode von Schritt ③ ist der originale Quellcode der nativen Sicherheitsfunktion.

Die Sicherheitsargumentation für Diversified Encoding baut auf den zwei zueinander diversitären Ausführungen der Sicherheitsfunktion auf [SSK15].

4.2 Zähler mit Diversified Encoding

Dieser Abschnitt illustriert die Benutzung von Diversified Encoding mit dem Beispiel aus Abschnitt 2. Das Code-Transformationswerkzeug generiert einen Einstiegspunkt – die C Funktion `incSafeCounterSafe` für die C Funktion `incSafeCounter`. Dieser Einstiegspunkt ist Teil des generierten Diversity Frameworks für den Zähler und enthält die vier Schritte aus Abb. 1:

1. Die Generierung der kodierten Eingabewerte aus den nativen Eingabewerten.
2. Die Ausführung der kodierten Sicherheitsfunktion.
3. Die Ausführung der nativen Sicherheitsfunktion.
4. Die Berechnung der Checksumme über die kodierten Ausgaben.

Der neue Einstiegspunkt hat die folgenden Schnittstelle:
`uint32_t incSafeCounterSafe(uint32_t* checksum, uint32_t incValue)`.
Der Rückgabewert und der Parameter `incValue` haben die selbe Semantik wie in der Implementierung von `incSafeCounter`. Die Aufruferkomponente muss einen Zeiger auf eine Variable zum Speichern der Ausgabechecksumme zur Verfügung stellen. Im folgenden ist der Einfachheit halber, wenn von der Checksumme gesprochen wird, immer die Ausgabechecksumme gemeint. Einstiegspunkt `incSafeCounterSafe` speichert diese Checksumme in der Adresse auf die `checksum` zeigt.

Um die Ausführung auf Ausführungsfehler hin zu überprüfen, z. B. wenn der Rückgabewert von `incSafeCounterSafe` verwendet werden soll, muss die Aufruferkomponente den Wert der berechneten Checksumme mit der Checksumme über die nativen Ausgaben vergleichen. Das Code-Transformationswerkzeug erzeugt für die Berechnung der Checksumme über die nativen Ausgaben die Funktion:
`uint32_t SIListra_diversity_output_checksum(uint32_t returnValue)`.
Das Argument `returnValue` muss der Rückgabewert sein, den `incSafeCounterSafe` zurückgegeben hat.

5 Code-Transformation

Das Code-Transformationswerkzeug generiert automatisch den Quellcode der kodierten Sicherheitsfunktion und des Diversity Frameworks. Die automatische Generierung hat die folgenden Vorteile verglichen mit einer manuellen Implementierung:

Entwicklungszeit Änderungen am Quellcode der nativen Sicherheitsfunktion können zügig und ohne manuelle Schritte in die kodierte Sicherheitsfunktion und das Diversity Framework übernommen werden.

Flexibilität Anforderungen und Transformationseinstellungen können flexibel geändert werden und erfordern lediglich eine erneute Ausführung des Code-Transformierungswerkzeuges. Ohne Werkzeugsupport können Änderungen, z. B. das Ein- oder Ausschalten der Kontrollflussüberprüfungen oder die Änderungen am Kodierungsparameter A, komplette manuelle Neuerstellung von großen Teilen der kodierten Sicherheitsfunktion und des Diversity Frameworks erfordern.

Korrektheit und Reproduziertbarkeit Ein Code-Transformationswerkzeug macht in der

Regel weniger Fehler als ein Entwickler. Diversified Encoding reduziert zusätzlich das Risiko eines Fehlers des Code-Transformationswerkzeuges. Für den Fall, dass sich die vom Code-Transformationswerkzeug erstellte kodierte Sicherheitsfunktion nicht so verhält, wie die originale Sicherheitsfunktion, wird dieser Fehler als Ausführungsfehler aufgedeckt.

Das Code-Transformationswerkzeug arbeitet wie ein C-Compiler. Es führt die folgenden Schritte in der gezeigten Reihenfolge durch:

Vorverarbeitung (durch C-Präprozessor) Das Werkzeug muss den Quellcode aller eingebunden Header-Dateien vorverarbeiten. Hierfür müssen dieselben C-Macros wie für den Compileraufruf definiert werden.

Parsen Das Werkzeug erzeugt einen abstrakten Syntaxbaum vom vorverarbeiteten Quellcode und führt dabei eine semantische Analyse durch. Beispielsweise wird eine Typanalyse benötigt, um jeder Variable und jedem Ausdruck einen Datentypen gemäß der Regel des C-Standards zuweisen zu können. Zu diesem Zeitpunkt kann das Werkzeug auch die Benutzung nicht-unterstützter C-Sprachmerkmale erkennen und melden.

Kodierung Nach dem Parsen liegen ausreichend semantische Informationen vor, um die Kodierung durchzuführen. Das Werkzeug ersetzt alle Konstanten, Variablen und Operationen mit ihren jeweiligen kodierten Varianten.

Code Generierung Die Kodierung liefert ein Zwischenergebnis. Als letzter Schritt wird aus diesem Zwischenergebnis wieder C-Code erzeugt.

Der aktuelle Stand der Technik erlaubt die Kodierung von fast allen Merkmalen des C99 Sprachstandards:

- Jede Ganzzahlarithmetik wird unterstützt inklusive Bit-weiser logischer Operationen und Vergleiche.
- Alle Kontrollflussanweisung von C99, d. h. `if-else`, inklusive Schleifen, Funktionsaufrufe, `switch`, `break` und `continue`. Indirekter Kontrollfluss sowie `setjmp` und `longjmp` werden aktuell noch nicht unterstützt.
- Komplexe Datenstrukturen basierend auf Feldern (Arrays) und `structs` und Zeigerarithmetik werden unterstützt.

Für jede native C-Übersetzungseinheit[4], die das Werkzeug als Eingabe erhält, erzeugt es eine kodierte Übersetzungseinheit als C-Code. Der generierte C-Code muss durch einen C-Compiler weiter verarbeitet werden.

Optimierungen im C-Compiler können die Kodierung nicht entfernen. Der Fakt, dass alle Werte Vielfache eines Kodierungsparameters A sind, ist für den C-Compiler nicht ableitbar, weil der C-Compiler jede Übersetzungseinheit individuell behandelt. Selbst mit Optimierungen im Linker (Link-Time-Optimizations), welche dem Optimierer alle Übersetzungseinheiten auf einmal sichtbar machen, können Optimierungen die Kodierung nicht entfernen.

[4]Englisch: translation unit.

6 Diskussion und Fazit

6.1 Umfang der Fehleraufdeckung

Diversified Encoding bietet eine sehr hohe Wahrscheinlichkeit Ausführungsfehler aufzudecken. Die Sicherheit der Lösung basiert auf Software Coded Processing. In Bezug auf die IEC 61508 und die ISO 26262 haben Experimente und Analysen gezeigt, dass Diversified Encoding eine ausreichend hohe Safe-Failure-Fraction für den höchsten Diagnosedeckungsgrad erreichen kann, was eine Voraussetzung für die höchsten Sicherheitslevel der Normen ist. Beispielsweise beschreibt [GKSF15] ein Fehlerinjektionsexperiment bei dem die gemessene Wahrscheinlichkeit, dass Diversified Encoding einen injizierten Fehler nicht aufdeckt, bei nur 0.002 % liegt. Für dieses Experiment wurden über 300 Millionen Fehler injiziert.

Diversified Encoding umfasst die Aufdeckung von Ausführungsfehlern in allen Berechnungen einer Sicherheitsfunktion, d. h. im Datenfluss und Zustand[5] der Sicherheitsfunktion. Eingaben und Ausgaben können mit den Lösungen aus [SSK15] abgesichert werden.

Fehler im Zeitverhalten, z. B. zu langsame Ausführung oder ein Absturz der Sicherheitsfunktion, deckt Diversified Encoding in Zusammenarbeit mit einem Watchdog auf (siehe [SSK15]).

Systematische Software-Fehler, d. h. Software-Bugs, werden von Diversified Encoding nicht aufgedeckt. Die für die Aufdeckung solcher Software-Fehler benötigten Informationen – vor allem die Spezifikation – fließen nicht in das Diversified Encoding ein.

Direkte Hardware-Zugriffe, z. B. über Assemblercode, sind außerhalb des Umfangs eines automatischen Code-Transformationswerkzeuges. Es ist aber möglich, Assemblercode manuell zu kodieren. In solchen Fällen ist es sinnvoll, automatische Code-Transformation und manuelle Kodierung zu kombinieren.

6.2 Vorteile von Diversified Encoding

Die Entscheidung, Diversified Encoding für die Aufdeckung von Ausführungsfehler einzusetzen, ist eine Design-Entscheidung, die sicherheitskritische Teile der Architektur und den Entwicklungsprozess betreffen. Für das beste Ergebnis sollte Diversified Encoding so früh wie möglich im Entscheidungsprozess eingeplant werden.

Diversified Encoding ist Hardware-unabhängig. Das Verfahren macht keine Annahmen über die zugrunde liegende Hardware. Die einzige Anforderung ist die Verfügbarkeit eines Standard-konformen C-Compilers für die Zielhardware. Die Zielhardware selbst muss nicht zertifiziert sein oder speziell für sicherheitskritische Systeme entwickelt worden sein. Somit ist es möglich, mit Consumer-Hardware und Diversified Encoding sicherheitskritische Systeme zu entwickeln.

Weil Diversified Encoding eine Software-Lösung ist, ist sie flexibeler als Hardware-Lösungen. Der Einsatz von Diversified Encoding kann auf die sicherheitskritischen Teile eines Systems beschränkt werden und stellt keinerlei Anforderungen an die nicht-

[5] Der Zustand entspricht dem Speicher in dem Variablen der Sicherheitsfunktion liegen.

sicherheitskritischen Teile. Ein Code-Transformationswerkzeug erhöht die Flexibilität und macht Entwicklerresourcen für die Entwicklung der eigentlichen Funktionalität der Anwendung frei (siehe Abschnitt 5).

Software Coded Processing ist die Basis von Diversified Encoding. Es ist möglich, Software Coded Processing ohne Diversified Encoding zu nutzen. Allerdings hat Diversified Encoding zwei entscheidende Vorteile gegenüber dem alleinigen Einsatz von Software Coded Processing:

- Diversified Encoding mit der AN-Kodierung deckt Fehler auf, die nur von komplexeren Kodierungen, wie ANB und ANBD alleine aufgedeckt werden können [SSSF10]. Wegen ihrer Komplexität stellen ANB und ANBD höhere Leistungsanforderungen an die Hardware als Diversified Encoding mit AN. Somit benötigt Diversified Encoding weniger Hardware-Resourcen als Software Coded Processing alleine bei vergleichbarer Sicherheit.
- Diversified Encoding deckt Fehler des Code-Transformationswerkzeuges auf. Wegen des Vergleiches der originalen Ausführung mit der automatisch generierten Ausführung werden Generierungsfehler des Code-Transformationswerkzeug aufgedeckt. Somit hat ein Code-Transformationswerkzeug für Diversified Encoding eine geringere Kritikalität als ein Compiler.

Besonders wenn ein Code-Transformationswerkzeug verwendet wird, reduziert Diversified Encoding den Entwicklungsaufwand für das Gesamtsystem. Zusätzlich macht Diversified Encoding durch die kontinuierlich Überwachung der Berechnung regelmäßige Speicherprüfungen und Befehlssatztests zur Laufzeit überflüssig. Die inverse Ablage von Größen innerhalb des Zustandes der Sicherheitsfunktion ist nicht mehr notwendig. Durch die Vermeidung solcher defensiver Entwicklungsmaßnahmen werden durch Diversified Encoding weniger Entwicklungsresourcen benötigt.

Zusammenfassend löst Diversified Encoding mit Software Coded Processing das Problem, dass keine leistungsstarke, mit modernen Funktionen ausgestatte Hardware für sicherheitskritische Anwendungen zertifiziert ist. Diversified Encoding ermöglicht den Einsatz moderner Hardware, z. B. leistungsstarke ARM-Prozessoren mit GPU-Unterstützung in sicherheitskritischen Anwendungen. Die Möglichkeit, solche moderne Hardware in sicherheitskritischen Systemen einzusetzen, ermöglicht die Entwicklung neuer Generationen sicherheitskritischer Systeme, die intelligent, leistungsstark und sicher sind.

Literaturverzeichnis

[For89] P. Forin. Vital coded microprocessor principles and application for various transit systems. In *IFA-GCCT*, pages 79–84, Sept 1989.

[GKSF15] Majdi Ghadhab, Jörg Kaienburg, Martin Süßkraut, and Christof Fetzer. Is Software Coded Processing an Answer to the Execution Integrity Challenge of Current and Future Automotive Software-Intensive Applications? In *Proceedings of AMAA 2015, 19th International Conference on Advanced Microsystems for Automotive Applications*, July 2015.

[SSF09] Ute Schiffel, Martin Süßkraut, and Christof Fetzer. An-encoding compiler: Building safety-critical systems with commodity hardware. In *SAFECOMP '09: Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security*, pages 283–296, Berlin, Heidelberg, 2009. Springer-Verlag.

[SSK15] Martin Süßkraut, André Schmitt, and Jörg Kaienburg. Safe Program Execution with Diversified Encoding. In *Proceedings of the 13th embedded world conference 2015*, February 2015.

[SSSF10] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software. In Erwin Schoitsch, editor, *Computer Safety, Reliability, and Security*, volume 6351 of *Lecture Notes in Computer Science*, pages 169–182. Springer Berlin / Heidelberg, 2010.