Sichere Programmausführung mit Diversified Encoding

Martin Süßkraut* und Jörg Kaienburg†

29. April 2017

Aktuell ist Hardware, die speziell für sicherheitskritische Systeme entworfen und zertifiziert wurde, ein wichtiger Baustein sicherheitskritischer Anwendungen. Solche Hardware stellt die Aufdeckung von Ausführungsfehlern zur Verfügung. Allerdings erfordern viele zukünftige Anwendungen, z. B. das automatisierte Fahren, neue Funktionen und Ausführungsgeschwindigkeiten, für die keine zertifizierte sicherheitskritische Hardware verfügbar ist. Die Lösung für dieses Problem ist die Nutzung von Hardware, die nicht für den Einsatz in sicherheitskritischen Systemen zertifiziert wurde, jedoch die Anforderungen an Funktionen und Ausführungsgeschwindigkeit erfüllt, beispielsweise Consumer-Hardware. In einer solchen Architektur übernimmt eine Software-Lösung die Aufdeckung der Ausführungsfehler.

Dieser Artikel führt die Software-Lösung „Diversified Encoding mit Software Coded Processing“ ein. Wegen ihrer Hardware-Unabhängigkeit bietet diese Lösung die Flexibität, sicherheitskritische Systeme mit nicht-sicherheitskritischer Hardware zu bauen. Die Flexibiltät der Lösung wird durch ihre Automatisierung mittels eines Code-Transformationswerkzeuges erhöht.

Schlüsselwörter: funktionale Sicherheit, IEC 61508, ISO 26262, Software Coded Processing, Software Diversifikation

1 Einführung

Traditionell, wird sicherheitskritische Hardware verwendet, um Ausführungsfehler aufzudecken. Aktuelle nicht-sicherheitskritische Hardware bzw. Consumer-Hardware ist typischer weise nicht in der Lage, Ausführungsfehler aufzudecken, stellt dafür jedoch eine

*SIListra Systems GmbH, Königsbrückerstr. 124, 01099, Dresden, martin.suesskraut@silistra-systems.com

†SIListra Systems GmbH, Königsbrückerstr. 124, 01099, Dresden, joerg.kaienburg@silistra-systems.com

signifikant höhere Leistung zur Verfügung. Deswegen benötigen sicherheitskritische Systeme, die hohe Ausführungsgeschwindigkeit oder spezielle Funktionen brauchen, eine alternative Lösung zur Aufdeckung von Ausführungsfehlern.

Dieser Artikel führt in die Software-basierte Lösung „Diversified Encoding mit Software Coded Processing“ ein: Eine Lösung, um kontinuierlich während der Laufzeit Ausführungsfehler aufzudecken und die damit ermöglicht, jedes System von sich aus sicher zu machen. Weil Diversified Encoding Hardware-unabhängig ist, kann die Lösung auch mit Consumer-Hardware verwendet werden. Nach der Einführung in Abschnitt 1 führt Abschnitt 2 ein Beispiel ein, das durch den ganzen Artikel hindurch wiederverwendet wird. Abschnitte 3 und 4 führen Software Coded Processing und Diversified Encoding ein.

Die ISO-Norm 26262 listet Coded Processing in Band 5 unter D 2.3.6 als eine Technologie, um Hardware-Ausführungsfehler aufzudecken [ISO11]. In der IEC-Norm 61508 wird Coded Processing in Band 2 in A.4 [65A10] als eine Lösungsmöglichkeit gelistet. Coded Processing in Kombination mit Diversified Encoding ist eine flexible Lösung zur Aufdeckung von Ausführungsfehlern mit dem Potenzial, die Sicherheitslevel ASIL-D und SIL-4[1] zu erreichen. Die Verwendung eines automatisierten Code-Transformationswerkzeuges erlaubt die effiziente Anwendung von Diversified Encoding in verschiedenen Industrien, beispielsweise in der Automobil-, Medizin-, Automatisierungs-, Bahntechnik sowie Raum- und Luftfahrttechnik.

Abschnitt 5 führt ein Code-Transformationswerkzeug für die Generierung des Quellcodes der Diversified Encoding Lösung ein. Dafür liest das Werkzeug den Quellcode der Sicherheitsfunktion ein und erzeugt den Quellcode für Diversified Encoding als Ausgabe. Der erzeugte Quellcode hat dieselbe Funktionalität wie der Originale-Quellcode und deckt zusätzlich Ausführungsfehler auf. Ein Alleinstellungsmerkmal des vorgestellten Werkzeuges ist die breite Unterstützung der Programmiersprache C.

Der Artikel schließt in Abschnitt 6 mit einer Diskussion der Vorteile der Diversified Encoding Lösung: Hardware-Unabhängigkeit, Flexibilität bei der Software-Umsetzung und Werkzeugunterstützung für die Programmiersprache C. Diese Vorteile ermöglichen kurze Entwicklungs- und Innovationszyklen. Ein weiterer beabsichtigter Effekt ist die Entkopplung von Funktion und Hardware-Fehleraufdeckung durch Diversified Encoding. Die vorgestellte Lösung erlaubt Systemarchitekturen, die Sicherheitsfunktionen auf nicht-zertifizierter Hardware ausführen.

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.

[1] Um sicherheitskritische Systeme auf SIL-4 zu heben, schreibt die IEC 61508 unabhängig von der Lösung zur Aufdeckung von Ausführungsfehlern Hardware-Redundanz vor: HW-Fehlertoleranz muss mind. 1 betragen.

Die Sicherheitsfunktion ist ein abgesicherter Zähler. Sie 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 Aufdeckung 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.

[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.

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 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 (siehe Abschnitt 4.3). 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 folgenden Eigenschaften von A beeinflussen die Sicherheit der AN-Kodierung:

Größe Je größer A gewählt wird, um so sicherer fällt die Kodierung aus. Zur Veranschaulichung sei n die Anzahl der benötigten Bits, um alle nativen Werte darzustellen (z. B. 32 Bit) und k sei die Anzahl der benötigten Bits, um A darzustellen (z. B. 3 Bits für $A = 7$). Dann werden $n + k$ Bits benötigt, um alle kodierten Werte

repräsentieren zu können. Die Anzahl der gültigen kodierten Werte ist dieselbe wie die Anzahl der nativen Werte: 2^n. Nur diese 2^n Werte von allen möglichen Werten, die mit $n + k$ Bits darstellbar sind, sind gültig. Alle anderen Werte sind ungültig. Ein Fehler wird nur dann nicht erkannt, wenn der Fehler einen gültigen Wert in einen anderen gültigen Wert ändert. Die Wahrscheinlichkeit für einen zufälligen Fehler, einen gültigen Wert in einen anderen gültigen Wert zu ändern, ist annähernd: $\frac{2^n}{2^{n+k}} = \frac{1}{2^k}$. Somit hängt die Wahrscheinlichkeit $\frac{1}{2^k}$ für einen unerkannten zufälligen Fehler nur von der Größe von A ab.

Hamming-Distanz Weil Computer Werte als binäre Worte darstellen, ist auch die Hamming-Distanz der AN-Kodierung wichtig. Die Hamming-Distanz hängt von A ab. Beispielsweise sollte A keine Zweierpotenz sein, weil die resultierende AN-Kodierung die Hamming-Distanz 0 hat. Der Grund dafür ist, dass eine Multiplikation mit einer Zweierpotenz gleichwertig zu einem Schieben der Bits aller Werte um eine fixe Distanz hin zu den höherwertigen Bits ist.

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. Abschnitt 4.3 zeigt ein Beispiel für einen sicheren Vergleich zwischen beiden Checksummen.

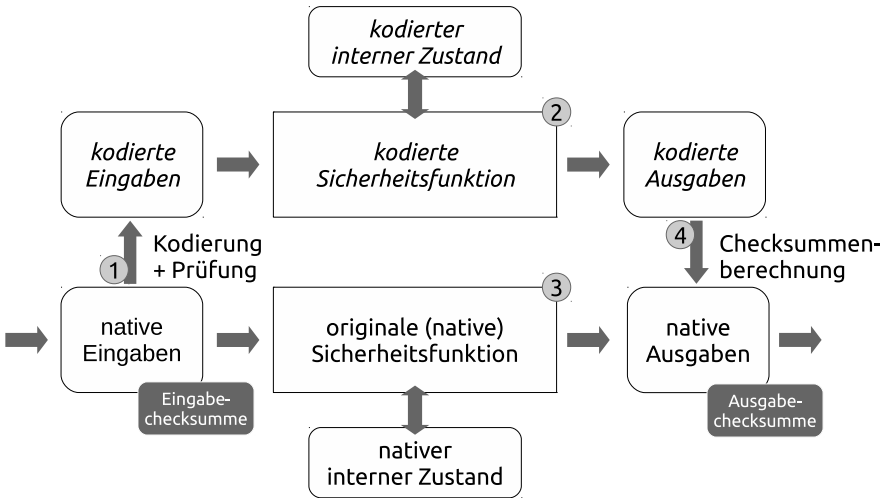

Abbildung 1: Das Datenflussmodell von Diversified Encoding.

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. Der nächste Abschnitt zeigt ein Beispiel für einen sicheren Vergleich mit Hilfe eines Watchdogs.

4.3 Integration mit einem Watchdog

Dieser Abschnitt illustriert wie Diversifed Encoding an einen Challange-Response-Watchdog (CRWD) angebunden werden kann. Ein CRWD überwacht ein Sicherheitssystem. Er sendet in regelmäßigen Abständen eine zufällige Frage (Challange) an das Sicherheitssystem und erwartet die korrekte Antwort (Response) vom System in einem vorher definiertem Zeitfenster. Falls die Antwort inkorrekt ist oder nicht im erwarteten Zeitfenster eintrifft, initiert der CRWD eine vorab definierte Sicherheitsreaktion für das System, z. B. Ausgangssignale Abschalten, Reset oder Umschalten auf ein Reservesystem. Solche CRWDs sind Stand der Technik in der Industrie.

Fragen und Antworten sind im Allgemeinen Ganzzahlwerte. Für jede mögliche Frage, wird die korrekte Antwort vorherberechnet und als Teil der Konfiguration des Systems im CRWD gespeichert.

Der Inhalt dieses Abschnittes besteht darin, wie die Anbindung so umgesetzt werden kann, dass der CRWD überprüft, dass (1) die Sicherheitsfunktion regelmäßig ausgeführt wird und (2) die Ausführung der Sicherheitsfunktion frei von Ausführungsfehlern ist. Der folgende Quellcode zeigt wie Diversified Encoding in die Berechnung der Antwort mit einbezogen werden kann. Der CRWD schreibt die Frage in die Variable `challange` wenn er eine neue Frage stellt. Zusätzlich löscht der CRWD die Variable `response`, die die Antwort speichert, z. B. indem sie auf einen ungültigen Wert für eine Antwort gesetzt wird. Wenn der CRWD eine Antwort erwartet, liest er die Variable `response` und vergleicht den Wert mit der erwarteten Antwort. In der Praxis kann die Antwort von der Sicherheitsfunktion mit anderen Teilantworten zu einer gemeinsamen Antwort für das ganze System sicher zusammengefasst werden.

```
1 /* Konstante zur Berechnung der Antwort für den Zähler */
2 #define SAFE_COUNTER_SIG   (12u)
3
4 /* speichert die aktuelle Frage des CRWD */
5 extern uint32_t challenge;
6 /* speichert die aktuelle Antwort für den CRWD */
7 extern uint32_t response;
8
```

```
9  /* Checksummenvariable für Diversified Encoding */
10 uint32_t checksum = 0;
11
12 /* erhöhe den Zähler um 1 */
13 uint32_t currentCounterValue = incSafeCounterSafe(&checksum, 1u);
14
15 /* berechne die Antwort für den CRWD aus der Frage und
16    der Checksumme vom Diversified Encoding */
17 response = challenge + checksum + SAFE_COUNTER_SIG -
18     SIListra_diversity_output_checksum(currentCounterValue);
19 /* ... */
```

Wenn kein Ausführungsfehler die Berechnung von `currentCounterValue` beeinflusst hat, haben `SIListra_diversity_output_checksum(currentCounterValue)` und die Variable `checksum` den gleichen Wert. Die erwartete Antwort ist demnach:
`challenge + SAFE_COUNTER_SIG`.

Stimmen jedoch `SIListra_diversity_output_checksum(currentCounterValue)` und `checksum` nicht überein, dann wurde ein Ausführungsfehler aufgedeckt. In diesem Fall wird die berechnete Antwort von der erwarteten Antwort abweichen und der CRWD kann geeignet auf diesen Fehler reagieren. Falls die Sicherheitsfunktion nicht mindestens einmal zwischen Schreiben der Frage und Lesen der Antwort aufgerufen wird, enthält `response` eine ungültige Antwort. Damit überprüft der CRWD, dass die Sicherheitsfunktion regelmäßig ausgeführt wird und die Ausführung nicht von Ausführungsfehlern verfälscht wurde.

Die vorgestellte Lösung überschreibt immer die Variable `response`. Damit kann eine fehlerfreie Ausführung, die direkt auf eine fehlerhafte Ausführung folgt, den Fehler der fehlerhaften Ausführung maskieren, wenn der CRWD die Antwort nicht zwischen den beiden Ausführungen, sondern erst nach der zweiten Ausführung überprüft. Man kann diese Lösung so anpassen, dass ein einmal aufgetretener Fehler in die nächste Antwort propagiert wird, um die Maskierung zu verhindern.

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 i. d. R. 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, Funtionaufrufe, `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 Abschnitt 3 wurden die Eigenschaften des Kodierungsparameters A besprochen, die zu einer möglichst hohen Aufdeckungsrate beitragen: Größe und Hammingdistanz. 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 den Abschnitte 4.1 und 4.3 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 Abschnitt 4.3).

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.

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

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-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

[65A10] IEC SC 65A. Functional safety of electrical/electronic/programmable electronic safety-related systems. Technical Report IEC 61508, The International Electrotechnical Commission, 3, rue de Varembé, Case postale 131, CH-1211 Genève 20, Switzerland, 2010.

[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.

[ISO11] ISO TC22/SC3/WG16. ISO/DIS 26262 - Road vehicles – Functional safety. Technical report, Geneva, Switzerland, July 2011.

[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.