Bild: Pixabay

Software ist in der Medizin nicht nur Teil eines Produkts, sondern kann auch als eigenständiges Medizinprodukt gelten. Damit einher gehen erhöhte Anforderungen an deren Sicherheit.

Von der Vision zur Wirklichkeit

Aktuelle Sicherheitsnormen fordern den Nachweis, dass die funktionalen Anforderungen erfüllt und sicherheitsrelevante nicht-funktionale Softwarefehler ausgeschlossen sind. Hierzu zählen zum Beispiel Stacküberläufe, Laufzeitfehler sowie Compiler-Fehler. Deren Abwesenheit kann durch moderne statische Analyse- und Compilertechniken bewiesen werden.

Daniel Kästner
CTO bei AbsInt

Das Gebiet der Software für Medizinprodukte ist umwälzenden und sich zunehmend beschleunigenden Änderungen unterworfen. Wurde Medizinsoftware vor einigen Jahren lediglich als kleiner Zusatz zur elektronischen Hardware wahrgenommen, werden heutzutage drei große Bereiche unterschieden: Software als integraler Bestandteil eines Medizinprodukts (zum Beispiel zur Steuerung von Infusionspumpen oder Dialysegeräten), Software als eigenständiges Medizinprodukt (zum Beispiel Bildanalyseverfahren für Kernspintomographen) und Gesundheitssoftware (zum Beispiel mobile Applikationen auf Mehrzweckgeräten) [1]. Tiefgreifende Änderungen gibt es dabei auch innerhalb des klassischen Bereichs der softwaregesteuerten eingebetteten medizinischen Systeme.

Wie auch in anderen Industriebereichen wird immer mehr sicherheitskritische Funktionalität durch Software realisiert. Eine Fehlfunktion der Steuerungssoftware kann im Extremfall Menschenleben gefährden, auch in weniger kritischen Systemen hohe Kosten verursachen – nicht zuletzt durch Rückrufaktionen. Allein im Zusammenhang mit Infusionspumpen wurden nach Aussage der FDA zwischen 2005 und 2009 mehr als 500 Tote registriert. Im gleichen Zeitraum wurden aus Sicherheitsgründen 87 Rückrufe von Infusionspumpen durchgeführt. Eine Analyse dieser Ereignisse durch die FDA hat Softwarefehler als eine der Hauptursachen identifiziert [2]. Seitdem ist die Zahl weiter gestiegen: Aufgrund von Software-Defekten wurden zwischen 2011 und 2015 durch die FDA 627 Software-Geräte zurückgerufen, zwölf davon mit der höchsten Gefährdungsstufe [3]. Zusätzlich steigt durch die zunehmende Vernetzung elektronischer Geräte die Gefahr von Cybersecurity-Risiken dramatisch an – nicht nur im Bereich von Gesundheitssoftware, sondern auch im Bereich von eingebetteten Medizinprodukten.

Dies unterstreicht die zentrale Bedeutung einer systematischen Verifikation und Validierung von Medizinsoftware: Das Ziel der Software-Verifikation liegt in der Bereitstellung eines objektiven Nachweises, dass die relevanten Anforderungen an die Software erfüllt worden sind und keine Sicherheitsverletzungen auftreten können. Die Validierung prüft zusätzlich, ob die gestellten Anforderungen konsistent und vollständig sind und das gewünschte Verhalten abbilden [4].

Normenlandschaft

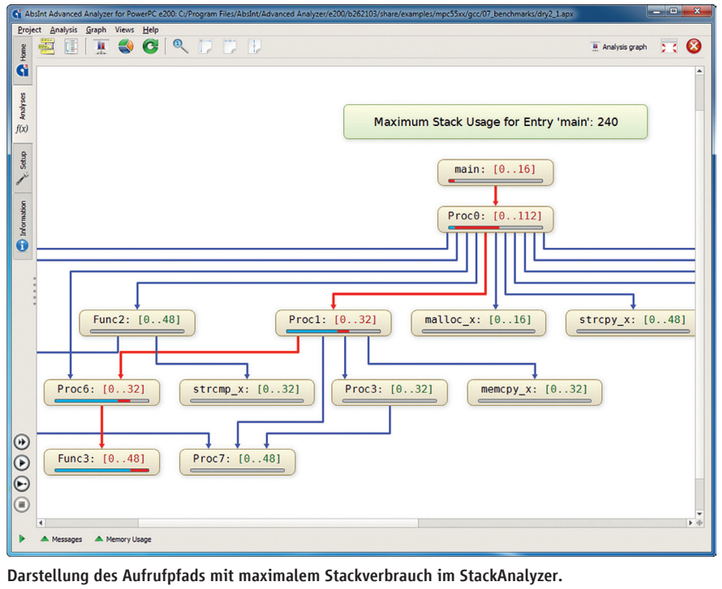
Der Software-Entwicklungsprozess wird durch nationale und internationale Normen und Vorschriften reguliert. Bei Medizinprodukten ist in Deutschland der Nachweis der Übereinstimmung mit dem Medizinproduktegesetz (MPG) erforderlich [5]. Darin wurde die Richtlinie 2007/47/EG in deutsches Recht übernommen, die in Absatz (20) explizit fordert, Software für Medizinprodukte – sowohl als eigenständiges Element wie auch als Bestandteil eines Medizinproduktes – in Übereinstimmung mit dem Stand der Technik zu validieren. Grundsätzlich unterliegen alle elektrischen/elektronischen Systeme der IEC 61508 [6], von der im Automobilsektor die branchenspezifische Norm ISO 26262 [7] oder im Bahnsektor die Norm EN 50128 [8] abgeleitet wurden. Die höchsten Anforderungen werden mit der Norm DO-178C [9] im Luftfahrtbereich gestellt; auch die Zertifizierungsprozesse sind dort deutlich strikter als im Medizintechnikbereich.

ÜBER ABSINT

Die AbsInt GmbH liefert Werkzeuge zur Validierung, Verifikation, Optimierung und Zertifizierung von sicherheitskritischer Software. Zu den Kernprodukten zählen die Quellcode-Analysatoren RuleChecker zum Überprüfen von Codierrichtlinien und Astrée zum Nachweis der Abwesenheit von Laufzeitfehlern und Datenwettläufen. Die statischen Analysatoren StackAnalyzer (Nachweis der Abwesenheit von Stacküberläufen), aiT WCET Analyzer (Berechnung von Laufzeitgarantien für Echtzeitsoftware) und TimingProfiler (Abschätzung der Programm-Ausführungszeit in frühen Entwicklungsphasen) arbeiten auf Binärcode-Ebene. TimeWeaver verbindet statische Pfadanalyse mit Echtzeit-Tracing auf Instruktionsebene, um maximale Ausführungszeiten auf Hochleistungs-Multicoreprozessoren abzuschätzen. Durch Einsatz des formal verifizierten Compilers CompCert kann Fehlcompilierung sicher ausgeschlossen werden.

AbsInts Qualification Support Kits und Qualification Software Life Cycle Data-Berichte ermöglichen eine automatische Toolqualifizierung nach allen gängigen Sicherheitsnormen. AbsInt wurde im Jahr 1998 als Spin-Off der Universität des Saarlandes gegründet und beliefert Kunden aus mehr als 40 Ländern in aller Welt.

Im Medizinbereich formulieren EN 60601-1 [10] und IEC 61010-1 [11] allgemeine Anforderungen für die Erstellung funktional sicherer Medizingeräte. Der Schwerpunkt der IEC 62304 [12, 1] liegt auf Software für Medizingeräte und definiert insbesondere einen Lebenszyklus für Software-Entwicklung mit Fokus auf komponentenorientierte Software-Architekturen und Softwarewartung. Die Vorgaben für Verifikation und Validierung sind jedoch sehr allgemein gehalten und im Detailgrad nicht mit IEC 61508, ISO 26262 oder EN 50128 vergleichbar. EN 45502-1 [13] und ISO 14708-1 [14] geben spezifische Leitlinien für aktive implantierte Geräte und verweisen dabei bezüglich Softwareanforderungen auf die IEC 62304. Mit dem Risikomanagement für Medizingeräte befasst sich die ISO 14971 [15]. Die Norm ISO 13485 [16] betrachtet allgemeines Qualitätsmanagement. Zu den relevanten nationalen Richtlinien im Bereich Verifikation und Validierung zählen insbesondere auch die US Quality System Regulations [17] und die FDA General Principles of Software Validation [4].



Bilder: AbsInt

Darstellung des Aufrufpfads mit maximalem Stackverbrauch im StackAnalyzer.

Anforderungen an Verifikation und Validierung

Gemeinsam ist allen Safety-Normen und Regulierungen im Software-Bereich, dass sie die Identifikation funktionaler und nicht-funktionaler Gefahrenstellen fordern sowie den Nachweis, dass die Software die relevanten Sicherheitsziele erfüllt. Dies gilt ohne Einschränkung auch für den Bereich der Medizinsoftware, auch wenn der Detailgrad der Anforderungsbeschreibung deutlich hinter anderen Industrienormen liegt. Es müssen die Codierrichtlinien eingehalten werden, die das Risiko von Programmierfehlern minimieren. Verbreitete Standards sind insbesondere MISRA C [18], SEI CERT C [19], ISO/TS 17951 [20] und CWE (Common Weakness Enumeration) [21]. Eine automatisierte Prüfung durch Softwarewerkzeuge (zum Beispiel AbsInt RuleChecker [22]) hat sich in sicherheitskritischen Industriebereichen als fester Teil des Entwicklungsprozesses durchgesetzt. Zum Korrektheitsnachweis für funktionale Programmeigenschaften haben sich automatische und modellbasierte Testverfahren neben formalen Methoden wie Model Checking etabliert. Bei der Systemkorrektheit spielen jedoch auch sicherheitsrelevante nicht-funktionale Qualitätseigenschaften eine wichtige Rolle. Es ist offensichtlich, dass ein Programm nicht wegen Speicher-

fehlern abstürzen darf, und dass die Reaktionen von Echtzeitsoftware innerhalb der vorgegebenen Zeit erfolgen müssen. Die Folgen nicht-funktionaler Fehler können schwerwiegend sein – bekannte Beispiele sind die Explosion der Ariane 5 [23], die unbeabsichtigte Beschleunigung im Toyota Camry [24] und die fehlerhafte Strahlendosisberechnung beim Therac-25-Unfall [25].

Die Überprüfung nicht-funktionaler Programmeigenschaften durch Testverfahren ist problematisch. Tests und Messungen können prinzipiell nicht die Abwesenheit von Fehlern beweisen. Erschwerend kommt hinzu, dass keine spezifischen Testfälle zur Stimulation der maximalen Laufzeit oder des maximalen Stackverbrauchs zur Verfügung stehen. Auch die Bestimmung sinnvoller Testende-Kriterien für Eigenschaften wie Timing, Stackverbrauch oder das Auftreten von Laufzeitfehlern wie Division durch Null oder arithmetische Überläufe stellt ein ungelöstes Problem dar. Der Testprozesss für nicht-funktionale Software-Anforderungen ist dementsprechend nicht fokussiert, der erforderliche Testaufwand hoch und die Testabdeckung unvollständig.

Statische Analyse durch »Abstrakte Interpretation«

In den letzten Jahren haben sich statische Analysatoren auf Basis der »Abstrakten Interpretation« zum Stand der Technik bei der Verifikation nicht-funktionaler Eigenschaften entwickelt. Ein statischer Analysator ist ein Softwarewerkzeug, das zur Ermittlung der Analyseergebnisse keine Ausführung des untersuchten Programmes benötigt.

Die sogenannten sicheren statischen Analysatoren, die auf der Theorie der Abstrakten Interpretation basieren, zählen zu den formalen Verifikationsmethoden. Abstrakte Interpretation ist eine semantikbasierte Methodik für Programmanalysen [26]. Sie ermöglicht volle Kontroll- und Datenabdeckung und kann einen formalen Nachweis von Programmeigenschaften erbringen. Die sogenannten False Negatives, also übersehene Fehler, sind ausgeschlossen. Statische Analysen können leicht automatisiert und in den Entwicklungsprozess integriert werden, und sie ermöglichen es, Fehler in frühen Entwicklungsphasen zu entdecken.

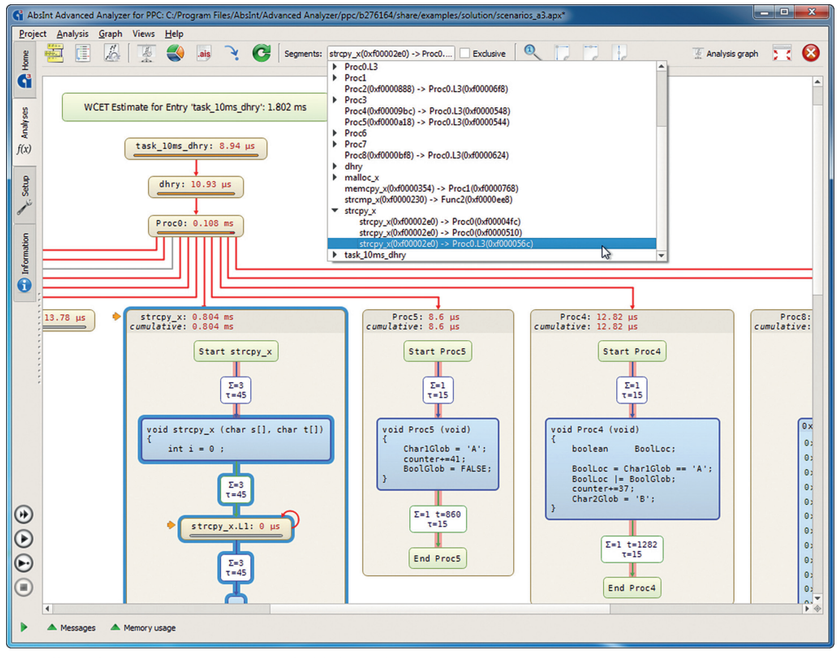
Moderne statische Analysatoren skalieren gut und erlauben die Analyse vollständiger sicherheitskritischer Industrieanwendungen [27]. Abstrakte Interpretation eignet sich sehr gut, um nicht-funktionale Softwarefehler auszuschließen. Beispiele sind sichere statische Analysatoren zum Nachweis der maximalen Ausführungszeit und des maximalen Stackverbrauchs von Tasks sowie zum Ausschluss des Auftretens von Laufzeitfehlern und Datenwettläufen [28].

Stacküberläufe ausschließen

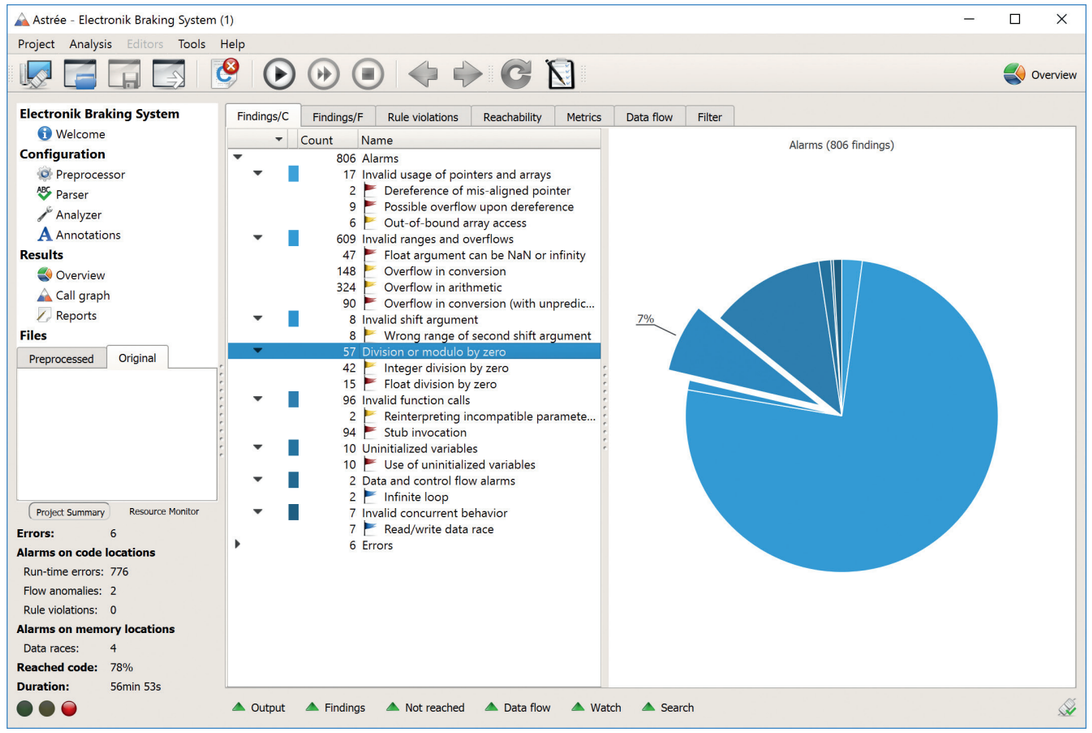
Stacküberläufe können schwerwiegende Auswirkungen haben, die zu fehlerhaften Programmausführungen oder zu Abstürzen führen können. Sie können die Datenintegrität zerstören und damit unerwünschte Interferenzen zwischen Softwarekomponenten verursachen. Das Tool StackAnalyzer [29] von AbsInt berechnet zum Beispiel, wie die Stackhöhe entlang der verschiedenen Kontrollpfade zu- und abnimmt. Diese Information wird dazu verwendet, den maximalen Stackverbrauch einer Task zu berechnen, womit sich die Abwesenheit von Stacküberläufen beweisen lässt [30].

Nachweis der Echtzeitfähigkeit

Viele Tasks in sicherheitskritischen eingebetteten Systemen haben harte Echtzeitanforderungen. Das heißt, die Einhaltung ihrer Deadlines ist zum korrekten Funktionieren des Systems erforderlich. Statische Analysatoren auf Basis der Abstrakten Interpretation können garantierte und präzise obere Schranken der maximalen Ausführungszeit berücksichtigen [31, 32]. Sie sind für komplexe Prozessoren mit Caches, komplexen Pipelines, Single-Core und Multi-Core Prozessoren verfügbar. Eine Voraussetzung ist jedoch, dass das Zeitverhalten des Prozessors deterministisch vorhersagbar ist. Dies ist bei der jüngsten Generation ursprünglich für den Consumer-Bereich entwickelter High-Performance Multicore System-On-Chips oft nicht mehr erfüllt. Der Hauptgrund



Darstellung der Trace-Segmente mit Laufzeitinformationen im TimeWeaver.



Ergebnisdarstellung mit Übersicht der Laufzeitfehler-Kategorien in Astrée.

liegt in Interferenzen zwischen den einzelnen Cores der betroffenen Multicore-Architekturen, die sich nicht vermeiden oder begrenzen lassen [33]. Hier bieten sich hybride WCET-Analysatoren an, die auf Basis nicht-invasiver Echtzeit-Traces das Zeitverhalten der ausführbaren Codestücke unter Berücksichtigung typischer Interferenzen messen und daraus einen kritischen Ausführungspfad konstruieren [34, 35].

Ausschluss von Laufzeitfehlern

Eine weitere Klasse kritischer Fehler sind Programmierfehler aufgrund undefinierten oder nicht spezifizierten Verhaltens der verwendeten Programmiersprache, die sogenannten Laufzeitfehler. Hierzu zählen arithmetische Überläufe, Feldgrenzenverletzungen, Pufferüberläufe, aber auch Datenwettläufe oder Deadlocks. Solche Fehler können die Datenintegrität zerstören, falsche Systemreaktionen verursachen und zu einem Softwareabsturz führen. Sie sind gleichzeitig die kritischsten Einfalltore für Cybersecurity-Attacken auf Code-Ebene [36].

Ein Beispiel für einen statischen Laufzeitfehler-Analysator auf Basis der Abstrakten Interpretation ist das Tool Astrée [37], das alle potentiellen Laufzeitfehler in C-Programmen entdeckt und damit den sicheren Nachweis der Abwesenheit solcher Fehler ermöglicht. Astrée verfügt über eine leistungsstarken Analysekern, der eine hohe Analysepräzision erreicht, aber dennoch für komplexe Industrie-Anwendungen skaliert. Beispielsweise konnte eine vollständige kommerzielle Flugzeugsteuerung von über 500.000 Zeilen Code ohne jeden Fehlalarm analysiert werden [38]; das bislang größte analysierte Programm besteht aus 110 Millionen Zeilen präprozessierten Codes, für die knapp zwei Tage Analysezeit erforderlich waren [39].

Compiler-Fehler

Stacküberläufe und Laufzeitfehler sind zwei Hauptursachen software-verursachter Speicherkorruption. Die dritte Hauptursache sind Compiler-Fehler: Der Compiler erzeugt fehlerhaften Code für ein korrektes Eingabeprogramm. Durch einen formal bewiesenen Compiler kann Fehlcompilierung ausgeschlossen werden; somit können alle drei Hauptursachen software-verursachter Speicherkorruption eliminiert werden [40, 41].

Hohe Zuverlässigkeit ist möglich

Sicherheitsorientierte Codierrichtlinien ermöglichen einen auf Minimierung von Programmierfehlern ausgerichteten Programmierstil. Funktionale Korrektheit kann durch formale Verifikationsverfahren bewiesen oder durch automatisierte Testverfahren geprüft werden. Die Abwesenheit von sicherheitskritischen nicht-funktionalen Fehlern, darunter Stacküberläufe, Zeitüberschreitungen, Laufzeitfehler und Datenwettläufe, kann durch statische Analysatoren auf Basis der Abstrakten Interpretation bewiesen werden. Formal verifizierte Compiler stehen zur Verfügung, bei denen mathematische Beweise vorliegen, dass es keine Fehlcompilierung geben kann. Den Entwicklern steht somit ein mächtiger Baukasten von Software-Werkzeugen zur Verfügung, die einen hohen Zuverlässigkeitsgrad sicherheitskritischer Software ermöglichen. ■

LITERATURVERZEICHNIS

[1] IEC/DIS 62304 – Draft International Standard. Health software – Software life cycle processes. 2018

[2] White Paper: Infusion Pump Improvement Initiative. U.S. Food and Drug Administration, Center for Devices and Radiological Health. Silver Spring 2010

[3] J. G. Ronquillo and D. M. Zuckerman: Software-Related Recalls of Health Information Technology and Ohter Medical Devices: Implications for FDA Regulation of Digital Health. In: The Milbank Quaterly. Vol. 95. Seite 535–553. Wiley Periodicals Inc. Hoboken 2017

[4] General Principles of Software Validation; Final Guidance for Industry and FDA staff. U.S. Food and Drug Administration, Center for Devices and Radiological Health, Center for Biologics Evaluation and Research. Silver Spring 2002

[5] Medizinproduktegesetz in der Fassung der Bekanntmachung vom 7. August 2002 (BGBl. I S. 3146); zuletzt geändert durch Artikel 12 des Gesetzes vom 24. Juli 2010 (BGBl. I S. 983).

[6] IEC 61508 — Functional safety of electrical/electronic/programmable electronic safety-related systems. 2010

[7] ISO/FDIS 26262 – Road vehicles – Functional safety. 2018

[8] CENELEC EN 50128 — Railway applications – Communication, signalling and processing systems – Software for railway control and protection systems. 2011.

[9] RTCA DO-178C — Software Considerations in Airborne Systems and Equipment Certification. 2011.

[10] IEC 60601-1:2005/AMD1:2012 — Medical electrical equipment – Part 1: General requirements for basic safety and essential performance. 2012

[11] IEC 61010-1:2010 — Safety requirements for electrical equipment for measurement, control, and laboratory use – Part 1: General requirements. 2010

[12] DIN EN 62304; VDE 0750-101:2016-10 — Medical device software – Software life cycle processes. 2016

[13] EN 45502-1:2015 — Implants for surgery. Active implantable medical devices. General requirements for safety, marketing and for information to be provided by the manufacturer. 2015

[14] ISO 14708-1:2014 — Implants for surgery – Active implantable medical devices. Part 1: General requirements for safety, marketing and for information to be provided by the manufacture. 2014

[15] ISO 14971:2007-03 — Medical devices - Application of risk management to medical devices. 2007

[16] ISO 13485:2016 — Medical devices – Quality management systems – Requirements for regulatory purposes. 2016

[17] FDA: CFR - Code of Federal Regulations Title 21 (1. April 2018), https://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfcfr/CFRSearch.cfm?CFRPart=820 (Stand 14. Januar 2019)

[18] MISRA-C:2012 Guidelines for the use of the C language in critical systems. 2013

[19] Software Engineering Institute SEI: SEI CERT C Coding Standard – Rules for Developing Safe, Reliable, and Secure Systems. Carnegie Mellon University. Pittsburgh 2016.

[20] ISO/IEC — Information Technology – Programming Languages, Their Environments and System Software Interfaces – Secure Coding Rules (ISO/IEC TS 17961). 2013

[21] Mitre Corporation: CWE – Common Weakness Enumeration (o.J.), https://cwe.mitre.org (Stand 14. Januar 2019)

[22] AbsInt GmbH: RuleChecker (o.J.), https://www.absint.com/rulechecker/index.htm (Stand 14. Januar 2019)

[23] Lions, J.L.: ARIANE 5 Failure-Full Report. European Space Agency (1996), http://www-users.math.umn.edu/~arnold//disasters/ariane5rep.html (Stand 14. Januar 2019)

[24] M. Barr: Bookout V. Toyota – 2005 Camry software Analysis (2013) http://www.safetyresearch.net/Library/BarrSlides_FINAL_SCRUBBED.pdf (Stand 14. Januar 2019)

[25] N. G. Leveson and C. S. Turner: An investigation of the therac-25 accidents. In: IEEE Computer. Vol. 26. Seite 18–41, Washington D.C. 1993

[26] P. Cousot and R. Cousot: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. Seite 238-252. New York 1977

[27] D. Kästner, A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, A. Schmidt, H. Hille, S. Wilhelm, and C. Ferdinand: Finding All Potential Runtime Errors and Data Races in Automotive Software. SAE World Congress, Detroit 2017

[28] D. Kästner: Applying Abstract Interpretation to Demonstrate Functional Safety. In: Formal Methods Applied to Industrial Complex Systems (J.-L. Boulanger, ed.). UK: ISTE/Wiley. London 2014

[29] AbsInt GmbH: StackAnalyzer (o.J.), https://www.absint.com/stackanalyzer/index_de.htm (Stand 14. Januar 2019)

[30] D. Kästner and C. Ferdinand: Proving the Absence of Stack Overflows. In: SAFECOMP '14: Proceedings of the 33th International Conference on Computer Safety, Reliability and Security. Vol. 8666 of LNCS. Seite 202–213. Springer International Publishing. Basel 2014

[31] J. Souyris, E. Le Pavec, G. Himbert, V. Jégu, G. Borios, and R. Heckmann: Computing the worst case execution time of an avionics program by abstract interpretation. In: Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis. Seite 21–24. Springer. Berlin/Heidelberg 2005

[32] AbsInt GmbH: aiT WCET Analyzer (o.J.), https://www.absint.com/ait/index.htm (Stand 14. Januar 2019)

[33] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt: Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement. In: ECRTS 2014 – 26th Euromicro Conference on Real-Time Systems. The Institute of Electrical and Electronics Engineers. Madrid 2014

[34] AbsInt GmbH: TimeWeaver, https://www.absint.com/timeweaver/index.htm (Stand 14. Januar 2019)

[35] D. Kästner, M. Pister, C. Ferdinand, and S. Wegener: Obtaining Worst-Case Execution Time Bounds on Modern Microprocessors. Embedded World Congress, Nürnberg 2018

[36] D. Kästner, L. Mauborgne, and C. Ferdinand: Detecting Safety- and Security-Relevant Programming Defects by Sound Static Analysis. In: The Second International Conference on Cyber-Technologies and Cyber-Systems (CYBER 2017) (J.-C. B. Rainer Falk, Steve Chan, ed.). Vol. 2 of IARIA Conferences. Seite 26–31. IARIA XPS Press. Barcelona 2017

[37] AbsInt GmbH: Astrée (o.J.), https://www.absint.com/astree/index_de.htm (Stand 14. Januar 2019)

[38] A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, D. Kästner, S. Wilhelm, and C. Ferdinand: Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée. Embedded Real Time Software and Systems Congress, Toulouse 2016

[39] D. Kästner, M. Mauborgne, S. Wilhelm, B. Schmidt, M. Schlund, and C. Ferdinand: Analyze This! Sound Static Analysis for Integration Verification of Large-Scale Automotive Software. SAE World Congress, Detroit 2019.

[40] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand: CompCert - A Formally Verified Optimizing Compiler. Embedded Real Time Software and Systems Congress, Toulouse 2016

[41] D. Kästner, L. Mauborgne, and C. Ferdinand: Practical Experience on Qualifying a Formally Verified Compiler: Reducing V&V Effort with CompCert. Forum Safety and Security, Sindelfingen 2018