# Testen von Datensicherheit in vernetzten und automatisierten Fahrzeugen durch virtuelle Steuergeräte

Zur Erlangung des akademischen Grades eines

**DOKTOR-INGENIEURS (Dr.-Ing.)**

von der KIT-Fakultät für
Elektrotechnik und Informationstechnik,
des Karlsruher Instituts für Technologie (KIT)

genehmigte

**DISSERTATION**

von

**M. Sc. Andreas Florian Lauber**
geb. in Filderstadt

Tag der mündlichen Prüfung: 04.05.2020

Hauptreferent: Prof. Dr.-Ing. Eric Sax
Korreferent: Prof. Dr. rer. nat. Sabine Glesner

# Kurzfassung

In der Automobilindustrie sind in den vergangenen Jahren die zwei Trends Automatisierung und Vernetzung entstanden. Diese Trends sorgen für eine steigende Anzahl an Funktionen im Fahrzeug. Neben einer Erhöhung des Komforts nehmen jedoch auch die Risiken durch den unerlaubten Zugriff von außen zu. Das IT-Manipulationen bei Fahrzeugen keine Ausnahme bilden, zeigen bereits erste Beispiele. Besonders durch die langen Lebenszyklen in der Automobilindustrie und der Tatsache, dass 44 % aller Angriffe auf IT-Systeme durch bekannte Schwachstellen geschehen, müssen Fahrzeuge bereits in der Entwicklung abgesichert werden.

Bezogen auf die funktionale Sicherheit (engl. Safety) existieren in der Automobilentwicklung bereits eine Vielzahl an Testprozessen und -methoden. Eine Übertragbarkeit dieser auf die Datensicherheit (engl. Security) ist jedoch nicht gegeben, wodurch neue Methoden am Entstehen sind. Daher wird eine Testmethode mittels virtuellen Steuergeräten vorstellt. Hierfür wird aufgezeigt, welche Beobachtungspunkte und Überwachungsfunktionen für die Tests der Datensicherheit gegeben sein müssen, wie sich daraus eine Testmethodik ableiten lässt und wie diese Testmethodik anschließend in die Automobilentwicklung eingebunden werden kann.

Für die Testmethodik wurden die Bereiche Speicher-, Numerische-, Systematische-, Funktionale- und Anwendungsfehler identifiziert. Der Fokus wird dabei auf die ersten beiden Fehlerarten gelegt und daraus Kriterien für einen Test der Datensicherheit abgeleitet. Anhand der Kriterien werden Beobachtungspunkte in bestehenden Testsystemen analysiert und basierend auf einem virtuellen Steuergerät neue Beobachtungspunkte ergänzt. Hierbei werden nicht nur Schnittstellen des Steuergeräts berücksichtigt, sondern ebenfalls interne Zustände des steuernden Artefakts (ausgeführte Instruktionen, Variablen und Speicherbereiche) und des ausführenden Artefakts (Register, lokaler Busse und Recheneinheit). Eine Berücksichtigung der internen Zustände ist wichtig, da

Speicherfehler und numerische Fehler nicht zwangsläufig an die Schnittstellen propagieren und dadurch in der Umwelt sichtbar sind.

Anhand der Beobachtungspunkte in einem virtuellen Steuergerät wurde eine Gesamtsystemsimulation erstellt, die das Steuergerät mit Applikation, Prozessor und Peripherie simuliert. Eine Co-Simulation übernimmt die Erzeugung der Teststimuli. Durch die Beobachtungspunkte können Rückschlüsse auf das Verhalten und die Zustände innerhalb des Steuergeräts gezogen werden. Durch die zusätzlichen Beobachtungspunkte können Testmethoden wie Überwachung der Instruktionen und Register, Analyse der Eingaben, Markierung des genutzten Speichers und Analysemöglichkeiten der Codeabdeckung eingesetzt werden. Zusätzlich ergeben sich durch evolutionäre Verfahren die Möglichkeit der Maximierung des Variablen Wachstums und des Speicherzugriffs sowie die Minimierung des Abstands zwischen Heap und Stack.

Um die Testmethoden in einem Entwicklungsprozess einsetzten zu können werden die Ergebnisse auf eine vernetzte Funktion (Adaptive Cruise Control) skaliert und das Echtzeitverhalten beurteilt. Für die Simulation eines einzelnen Steuergeräts können dabei bis zu 62 Steuergeräte parallel simuliert werden, bevor die Simulation auf der realen Hardware schneller als der Ablauf in der Simulation ist. Durch die Ergänzung von Überwachungsfunktionen und Co-Simulationen sinkt das Echtzeitverhältnis jedoch exponentiell.

Abschließend wird die Testmethodik mit Angriffen aus der Automobilindustrie bewertet und aufgezeigt, welche Fehler erkannt worden wären. Die Testmethodik ist dabei jedoch nur so genau, wie die zugrundeliegenden Modelle. Eine Erhöhung der Genauigkeit bedeutet dabei höhere Kosten in der Entwicklung. Zudem muss der Quellcode für die Applikation als Source- oder Maschinencode bekannt sein. Wird die Testmethode als Ergänzung zu bereits bekannten Testverfahren eingesetzt, können jedoch Probleme in der Datensicherheit bereits der Entwicklung erkannt werden.

# Abstract

In recent years, two trends have emerged in the automotive industry: automation and connectivity. These trends have caused an increasing number of functions in the vehicles. In addition to enhancing comfort, the risks of unauthorized external access have increased. First examples show that vehicles are no exception when it comes to IT manipulations. Especially the long life cycles in the automotive industry and the fact that 44 % of all attacks on IT systems occur through known vulnerabilities require vehicles to be secured during development.

With respect to functional safety, a large number of test processes and methods are already in place for automotive development. However, a transposition of these methods to data security is not given. As a result new methods are emerging. This work will present a test method based virtual electronic control units (ECUs). It will show which observation points and monitoring functions must be given for the security tests, how test methodology can be derived and how this test methodology can subsequently be integrated in automotive development.

The following areas were identified for the test methodology: memory errors, numerical, systematic, functional and application errors. However, the focus is on the first two types of error. Criteria for testing data security are derived from these fields. On the basis of the criteria, observation points in existing test systems have been analyzed and new observation points have been added based on a virtual ECU. In this context, not only interfaces of the ECU are considered, but also internal states of the controlling (executed instructions, variables, and memory areas) and the executing artifact (registers, local buses, and arithmetic unit) are considered. Considering the internal states is important, since memory errors and numerical errors do not necessarily propagate to interfaces and therefore are not visible in the ECU environment.

Based on the observation points in a virtual control unit, an overall system simulation has been created which simulates the control unit with the application, processor and peripherals. A co-simulation generates test stimuli. The observation points can be used to determine the behavior and states within the ECU. The additional observation points allow the use of test methods to monitor instructions and registers, analyze inputs, mark the used memory and analyze code coverage. Furthermore, evolutionary techniques maximize variable growth and memory access and minimize the distance between heap and stack.

To use these test methods in development processes, the results were scaled to a connected function (Adaptive Cruise Control) and the real-time behavior was evaluated. For the simulation of a stand-alone ECU, up to 62 controllers may be simulated in parallel before the simulation on the real hardware is faster compared to the simulation. Adding monitoring functions and co-simulations, reduces the real-time ratio exponentially.

To conclude, the test methodology is evaluated with attacks from the automotive industry and it is shown which errors could have been detected. Nevertheless, the test methodology is only as accurate as the simulated models. Increasing the accuracy results in higher development costs. Furthermore, the application must be known as source code or machine code. However, if the test method is used as a supplement to already known test methods, data security problems can be detected during development.

# Danksagung

Die vorliegende Arbeit entstand während meiner Zeit als wissenschaftlicher Mitarbeiter am Institut für Technik der Informationsverarbeitung (ITIV) des Karlsruher Institut für Technologie (KIT). Ich möchte mich bei allen bedanken, die mich in dieser lehrreichen und wundervollen Zeit begleitet und unterstützt haben.

Mein besonderer Dank gilt meinem Doktorvater Prof. Eric Sax für die herausragende Unterstützung im Verlauf dieser Arbeit und die stets ausführlichen und sehr hilfreichen Rückmeldungen. Ebenfalls bedanken möchte ich mich bei Prof. Sabine Glesner für die Übernahme des Korreferats und bei der weiteren Prüfungskommission, bestehend aus Prof. Ivan Peric, Prof. Olaf Dössel und Prof. Martin Doppelbauer.

Ganz besonders bedanken möchte ich mich bei allen Kollegen und Studenten am ITIV, mit denen ich zusammenarbeiten durfte, für die großartige Zusammenarbeit und die vielen inspirierenden Gespräche. Ohne die diese Arbeit nicht möglich gewesen wäre.

Ein spezieller Dank gilt auch meiner Familie und meinen Freunden, die mich in allen Phasen dieser Arbeit begleitet und mir Rückhalt gegeben haben. An dieser Stelle vielen Dank an meine Eltern, Christine und Hans-Peter Lauber, meine Brüder, Thomas und Tobias, und meine Freundin Susanna Pitzer unter anderem für den bedingungslosen Rückhalt und das Verständnis.

Karlsruhe, im August 2020 *Andreas Florian Lauber*

# Inhaltsverzeichnis

# 1 Einleitung

## 1.1 Motivation zum Testen der Datensicherheit in vernetzten und automatisierten Fahrzeugen

“The car is the ultimate mobile device“ sagte Jeff Williams (leitender Geschäftsführer bei Apple Inc.) bereits im Mai 2015 im Interview auf der "Re/-code’s Code conference"[123]. Seither hat sich im Bereich der Vernetzung von Fahrzeug und der Integration neuer Dienste viel getan.

### 1.1.1 Mobilitätstrends

Im Bereich der Automobilindustrie entstanden die vier Mobilitätstrends Elektrifizierung, Vernetzung, Automatisierung und Mobilität als Service [6]. Bei diesen Trends ist die Elektrifizierung besonders auf eine nachhaltige Lebensweise und gleichzeitig die Mobilität als Dienstleistung (Mobility as a Service (MaaS)) auf die Individualisierung der Mobilität zurückzuführen. MaaS bedeutet dabei, dass die Menschen oder Güter ins Zentrum der Mobilität gestellt werden, um diesen durch einen maßgeschneiderten Zugang das passende Verkehrsmittel anzubieten [38].

Zusätzlich wird durch die Digitalisierung die Möglichkeit geschaffen, Fahrzeuge zum einen miteinander zu vernetzen, zum anderen Fahrzeugfunktionen zu automatisieren [6]. Bei der Vernetzung von Fahrzeugen untereinander und der Vernetzung von Fahrzeugen mit der Infrastruktur (Car-to-X) werden neue Mobilitätskonzepte ermöglicht, die einen besseren Verkehrsfluss schaffen und damit ebenfalls die Nachhaltigkeit des Verkehrs steigern. So können beispielsweise Ampeln frühzeitig auf ankommende Fahrzeuge reagieren, um die Dauer der Lichtphasen anzupassen oder Fahrzeuge ihre Geschwindigkeit an die Rotphase der Ampel anzugleichen. Weiter ermöglicht die Vernetzung neue

Betriebskonzepte wie das automatisierte Fahren. Dabei wird der Fahrer stufenweise entlastet, bis die Fahrfunktion ganzheitlich vom Fahrzeug, ohne Eingriff des Fahrers, realisiert ist (siehe Kapitel 3.1.1). Kombiniert man die Automatisierung mit der Mobilität als Service kann laut OECD in mitteleuropäischen Städten das Verkehrsaufkommen auf 10 % reduziert werden [80].

### 1.1.2 Steigende Funktionalität bei kürzeren Entwicklungszyklen

Die Nachfrage nach zunehmender Automatisierung, immer mehr Komfort, Integration der sozialen Medien und Vernetzung der Fahrzeuge untereinander sowie mit der Infrastruktur sorgt im Fahrzeug für eine zunehmende Zahl an intelligenten und vernetzten Steuergeräten. Hierzu zählt besonders die Vernetzung der Fahrzeuge und Infrastruktur zur Erhöhung der Fahrsicherheit. Aber auch die Zahl der Systeme zum Schutz von Verkehrsteilnehmern stieg in den letzten Jahren deutlich an (siehe Abbildung 1.1). Wo früher nur Gurtstraffer die Insassen schützten, unterstützen heute eine Vielzahl an elektronischen Funktionen, um die Folgen des Aufpralls zu reduzieren.

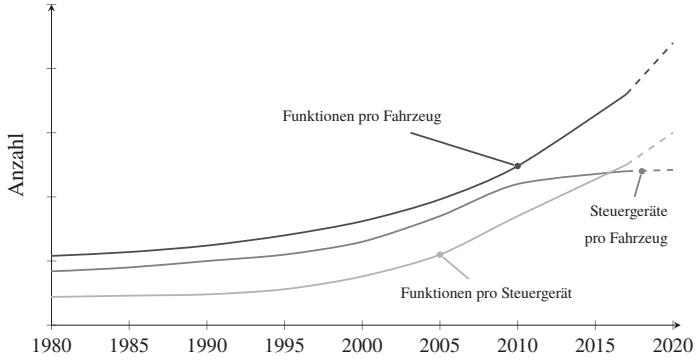



Abbildung 1.1: Steigerung der Funktionalität im Fahrzeug zwischen 1980 und 2020 [92]

Durch einen begrenzten Bauraum im Fahrzeug ist auch der Platz für Steuergeräte begrenzt, wodurch die Zahl dieser im Vergleich zur Zahl der Funktionen in den letzten Jahren nur sehr langsam stieg. Als Konsequenz nimmt die Anzahl

an Funktionen pro Steuergerät zu (siehe Abbildung 1.1). Durch die wachsende Funktionalität und gleichzeitige Vernetzung steigt das Zusammenspiel der verschiedenen Funktionen untereinander und die damit verbundenen Wechselwirkungen an. Beides zusammen führt zu einer höheren Komplexität und potentiellem Fehlverhalten der Funktionen im Fahrzeug.

Zusammen mit der Anzahl der Funktionen wächst die Zahl der Codezeilen (Line of Codes) im Fahrzeug (siehe Abbildung 1.2). Ein modernes Fahrzeug hat heutzutage etwa 100 Millionen Codezeilen [50, 67]. Im Vergleich hierzu hat das Betriebssystem Windows 7 mit rund 40 Millionen Codezeilen weniger als die Hälfte und moderne Flugzeuge mit rund 6,5 Millionen Codezeilen nur einen Bruchteil [18, 108]. Die steigende Anzahl an Codezeilen und die Zunahme der Funktionalität im Fahrzeug führt dazu, dass die Software immer komplexer wird, wodurch der Aufwand zur Absicherung der Funktionen steigt, da jede Codezeile während der Entwicklung verifiziert werden muss.

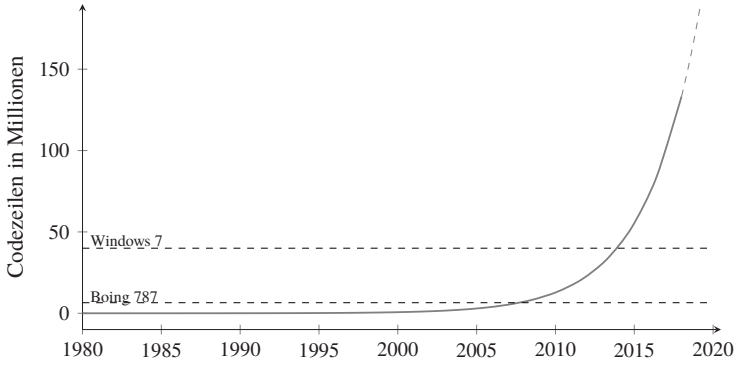



Abbildung 1.2: Anstieg der Codezeilen im Fahrzeug zwischen 1980 und 2020 [67, 50]

In der allgemeinen Softwareentwicklung wird davon ausgegangen, dass 15 - 50 Fehler pro 1000 ausgelieferten Codezeilen existieren. Durch ein erfolgreiches Testen kann dies zwar auf 0,5 Fehler je 1000 Codezeilen reduziert werden [68]. Ein fehlerfreies Programm wird jedoch auch dadurch nicht erreicht. Zudem stellen die Fehler nicht ausschließlich funktionale Probleme dar. Zwischen 1 - 10 % der Fehler haben einen Einfluss auf die Datensicherheit [126].

Neben den Fahrsicherheitsfunktionen werden Smartphones und mobile Geräte im Fahrzeug sowie Fahrzeuge und Infrastruktur außerhalb des Fahrzeugs für

eine Steigerung des Komforts angebunden. Durch eine Kommunikation mit diesen Internet of Things (IoT) Geräten – wie beispielsweise Infrastruktur, Gebäuden, Parkuhren, Kameras, etc. –, wird auch zukünftig die Funktionalität und damit die Komplexität im Fahrzeug weiter ansteigen. Dieser Trend ist in Abbildung 1.1 und Abbildung 1.2 als gestrichelte Linie dargestellt. Seit 2014 führt Daimler beispielsweise sukzessive eine Datenübertragung über das Internet in allen Fahrzeugklassen ein. Dies soll dem Kunden serienmäßig die Möglichkeit geben aktiv auf das Fahrzeug zuzugreifen und einwirken zu können. Unter anderem werden Zentralverriegelung, Standheizung, Fahrzeugortung und Diagnosedaten angebunden [118].

Um die Trends aus dem Sektor der mobilen Geräte und des IoT im Fahrzeug zu integrieren, sinken die Produktlebens- und Entwicklungszyklen. Dieser Effekt ist in Abbildung 1.3 anhand dreier Beispiele (Mercedes S-Klasse als Oberklasse Fahrzeug [149], BWM 5er als Beispiel der Mittelklasse [133] und VW Golf für die Unterklasse [159]) dargestellt. Die Markierungen stehen jeweils für die Einführung eines neuen Modells. Bei dieser Auflistung sind keine Modellüberarbeitungen oder sogenannte “Face-Lifts“ eingeschlossen. Diese reduzieren die Produktlebenszyklen noch weiter.

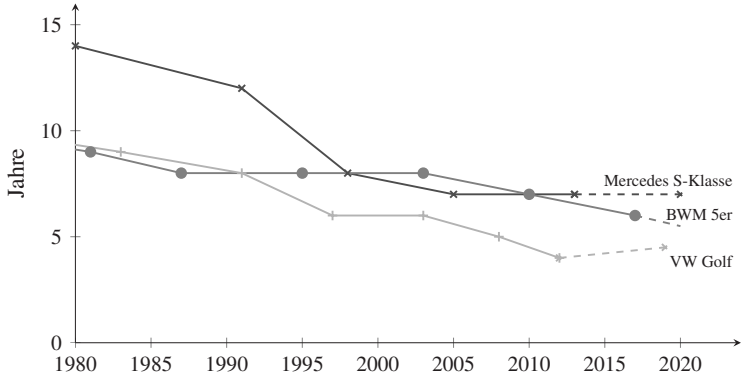



Abbildung 1.3: Produktlebenszyklen der Fahrzeuge zwischen 1980 und 2020 [133, 149, 159]

War der Produktlebenszyklus einer Mercedes-Benz S-Klasse im Jahr 1979 noch 14 Jahre (Modell W 140), so ist der Lebenszyklus 2017 auf 7 Jahre (Modell W222) [149] gesunken. Beim VW-Golf haben sich die Produktlebenszyklen von 10 Jahren (Golf 1) auf 4 Jahre (Golf 7) mehr als halbiert

[159]. Für das Modell W223 von Mercedes-Benz (vermutliche Einführung im Jahr 2020) und den VW Golf 8 (vermutliche Einführung ab 2019/2020), wird es sich ähnlich verhalten. Betrachtet man diese Entwicklung, so hat sich der Produktlebenszyklus von Fahrzeugen innerhalb 40 Jahren halbiert.

Analog zu den Produktlebenszyklen verkürzten sich die Entwicklungszeiten in den vergangenen Jahren [96]. Um dies bei der zunehmenden Funktionalität zu gewährleisten, müssen die Entwicklungsmethoden angepasst werden. Ein aktueller Trend geht hierbei zum Einsatz virtueller Entwicklungsmethoden, die ein paralleles Verifizieren und Validieren der entwickelten Produkte ermöglicht [96].

### 1.1.3 Steigende Anzahl von IT-Manipulationen

Auch wenn die Vielzahl der Steuergeräte im Fahrzeug derzeit von Hackern nur selten angegriffen oder manipuliert werden, so verhalten sich diese System dennoch analog zur Entwicklung in der IT-Industrie.

**Definition 1** (Hacker)**:**
*Hacker sind hoch ausgebildete Experten die sich durch Ausnutzung von Schwachstellen und Programmierfehlern unberechtigt Zugang zu Computersystemen verschaffen. Hacker werden im weiteren auch als Angreifer auf das System bezeichnet. [100]*

**Definition 2** (Angriff)**:**
*Ein Angriff ist das Vorgehen eines Hackers mit dem er Zugriff auf ein IT-System erlangt [33]. Ziel eines Angriffs ist dabei die Manipulation von Daten, das Erlangen von Wissen bzw. übergeordneten Rechten oder das Stören eines zur Verfügung gestellten Dienstes.*

Je mehr Schnittstellen für die Anbindung an externe Funktionen in ein Fahrzeug integriert werden, umso ähnlicher werden die Kommunikationspfade in Steuergeräten den Systemen aus der PC-Welt. In vielen Ober- und Mittelklassewagen stecken heute schon Recheneinheiten, die mit den Schnittstellen und der Rechenleistung einem Desktop-PC ähneln. Schnittstellen wie Bluetooth, WLAN und Mobilfunk integrieren dabei schon heute das Smartphone und eine Internetanbindung in das Fahrzeug. Die Vielzahl an Schnittstellen bieten dabei

die Möglichkeit über Angriffsvektoren (Viren, Würmer, Zufallseingaben, etc,) die Software über Schwachstellen zu manipulieren (siehe Abbildung 1.4).

**Definition 3** (Angriffsvektor)**:**
*Ein Angriffsvektor beschreibt den Weg, mit dem ein Angreifer Zugang zu einem System erhält, um dort eine speziell entwickelte Schadsoftware auszuführen [62].*

**Definition 4** (Schwachstelle)**:**
*Eine Schwachstelle ist eine kausaler Zusammenhang, bei der eine potenzielle oder tatsächliche Abweichung von einem gewünschten Zustand auftritt [114]. Die Begriffe Schwäche (engl. weakness) und Verletzlichkeit (engl. vulnerabilty) werden in der Literatur häufig synonym verwendet [53].*

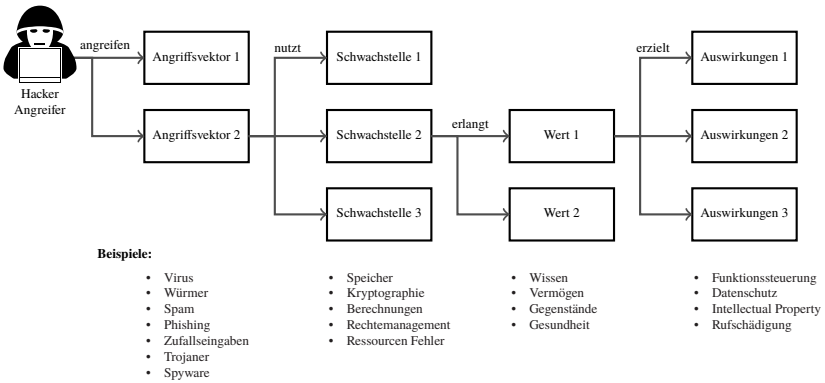



Abbildung 1.4: Zusammenhang zwischen Angriffen, Schwachstellen und Auswirkung

Ein Angreifer führt dabei Angriffe über die Vektoren aus, indem er beispielsweise über die Schnittstellen (Bluetooth, WLAN, Mobilfunk) Viren in das System einbringt, die anschließend Schwachstellen ausnutzen (siehe Kapitel 4 für eine Kategorisierung der Schwachstellen). Dabei werden Fehler in der Software – beispielsweise Fehler im Speicher, falsche Berechnungen, fehlerhafte Sicherheitsmechanismen – verwendet, um an Werte (Vermögen, Wissen, Funktionen) zu gelangen. Anhand der Werte versucht der Angreifer dann das System zu manipulieren und ein fehlerhaftes Verhalten herbeizuführen (Aus-

wirkung des Angriffs). Über die Auswirkungen können Schäden am System verursacht werden. Ein solcher Schaden kann im Automobil der Eingriff in die Fahrdynamik (Längs- und Querregelung), Zugriff auf Komfortfunktionen oder das Erlangen von personenbezogenen Daten (z. B. Ort, Route, Fahrer) sein.

**Definition 5** (Schaden)**:**
*Ein Schaden entsteht, wenn ein Wert in einer negativen, nicht wünschenswerten Weise verändert wird. Dies beinhaltet die teilweise Zerstörung oder Beschädigung der Werte [24].*

Beachtet man zudem, dass weltweit etwa 1 Millionen Hacker-Angriffe pro Tag auf Systeme der IT-Branche und IoT stattfinden [66], ist es nur eine Frage der Zeit, bis die Schnittstellen im Fahrzeug für Angriffe genutzt werden. Dies spiegeln auch die Zahlen des Deutschen Instituts für Wirtschaftsforschung wieder. Demzufolge gab es im Jahr 2015 alleine 14,7 Millionen Fälle von Internetkriminalität in Deutschland. Durch die zunehmende Vernetzung von IoT-Geräten und Fahrzeugen ist anzunehmen, dass diese Zahl weiter steigen wird, da neue Möglichkeiten für Angriffe auf zum Teil schlecht oder ungesicherten Systeme entstehen. Die steigende Gefahr durch Hacker bei vernetzten Geräten zeigen auch die Angriffe durch IoT-Geräte. Hier ist das Bot-Netz Mirai besonders zu erwähnen, dass im Jahr 2016 500.000 IoT-Geräte manipuliert hat [135], eigenen schadhaften Code auf diesen Geräten ausführte und dadurch für Angriffe auf Internetdienste wie Google, Twitter und Telekom-Router verantwortlich war [151].

Dass moderne Fahrzeuge hier keine Ausnahme bilden, zeigen Miller und Valasek [69]. In dieser Veröffentlichung wurden 20 Fahrzeuge weltweiter Hersteller aus den Jahren 2006 bis 2014 untersucht und nach deren Angriffspotential bewertet. Später wurde gezeigt, wie eines dieser Fahrzeuge über Remote-Zugriff angegriffen und ferngesteuert werden kann [70]. Weitere Angriffe über Telematik, Mobilfunk und Bluetooth fasst [121] zusammen. Angriffe auf Fahrzeuge beinhalten dabei nicht nur die Steuerung des Radios, Abblendlichts oder der Scheibenwischer, die bereits zu gefährlichen Situationen führen können, sondern auch direkte sicherheitskritische Funktionen wie Beschleunigung, Bremse und Lenkung. Wenn ein Angreifer die Schutzmechanismen im Fahrzeug überwindet, ist es nicht ausgeschlossen, dass dieser die Kontrolle über das gesamte Fahrzeug übernimmt. Dies kann sowohl das Bremsen aus jeder Geschwindig-

keit, das Beschleunigen, den Eingriff in die Lenkung und viele weiter Aktionen beinhaltet beinhalten [85].

Obwohl Angriffe dieser Art bisher nur im rein akademischen Umfeld gezeigt wurden, ist nicht auszuschließen, dass durch die zunehmende Vernetzung der Fahrzeuge gewollte oder ungewollte Manipulationen von Fahrfunktionen entstehen. Wichtig ist außerdem, dass es für die Sicherheitsanforderungen keinen Unterschied macht, ob direkte Bedrohungen auf die Fahrsicherheit (Bremsen, Lenken, etc.) oder indirekte Bedrohungen durch Ablenkung des Fahrers (Lautstärke des Radios, Scheibenwischer, Beleuchtung, etc.) entstehen [85]. Daher müssen beide Fälle (direkte und indirekte Bedrohungen) gleichermaßen abgesichert sein. Zudem ist es für die Sicherheit des Fahrzeugs irrelevant, ob diese Bedrohungen durch ein Fehlverhalten im Steuergerät (Software oder Hardware) oder durch eine Manipulation eines Angreifers erfolgt.

Für die Datensicherheit ist es zudem keine Frage ob Steuergeräte angriffen werden, sondern wann [85]. Da die Funktionalität in Systemen ansteigt und diese zunehmend vernetzter werden, entstehen mehr Schnittstellen die in der Entwicklung abgesichert werden müssen. Das Verhalten des Systems wird dabei über diese Schnittstellen im Regelbetrieb beeinflusst. Werden nun die Schnittstellen manipuliert, so wird ebenfalls das Verhalten des Systems beeinflusst. Weiter ist zu beachten, dass sich die Art der Angriffe auf ein System weiterentwickeln. Das heißt: Systeme oder Algorithmen, die heute als sicher gelten, können bereits morgen Schwachstellen enthalten. Als Beispiel ist hierfür die Heartbleed-Schwachstelle im Security-Protokoll “Secure Sockets Layer (SSL)“ zu benennen [17]. Dabei konnten über das Verschlüsselungsprotokoll Speicherbereiche eines an das Internet angeschlossenen Gerätes unbefugt ausgelesen werden, obwohl dies jahrelang als sicher galt.

Beim Ausführen eines Angriffs auf ein System werden Angriffsvektoren ausgeführt um Schwachstellen auszunutzen. Nachdem ein solcher Angriff erkannt wird, muss die entsprechende Schwachstelle im System behoben werden. Zusätzlich werden die Angriffe in Verbindung mit der Schwachstelle in Datenbanken veröffentlicht.

**Definition 6** (bekannte Schwachstelle)**:**
*Sind Schwachstellen über Angriffsvektoren verknüpft und wurden bereits ausgenutzt, so werden diese veröffentlicht und zählen als bekannte Schwachstellen.*

Besonders unter dem Gesichtspunkt, dass 44 % aller Angriffe auf IT-Systeme über bekannte Schwachstellen geschehen [160], ist es wichtig Sicherheitsmechanismen nach dem aktuellen Stand der Technik zu implementieren. Weiter zeigen Analysen, dass viele der bekannten Schwachstellen von Programmierfehlern abstammen [82].

Eine weitere wichtige Ursachen für das Eindringen in Systeme ist der Verlust oder Diebstahl von Passwörtern, Zufälle und die Weitergabe von Insiderwissen [103]. Nur ein sehr geringer Teil der Angriffe auf vernetzte Systeme wird durch, bis zum Angriffszeitpunkt, unbekannte Schwachstellen ausgeführt. Die Verteilung der klassifizierten Ursachen für Angriffe ist in Abbildung 1.5 dargestellt.

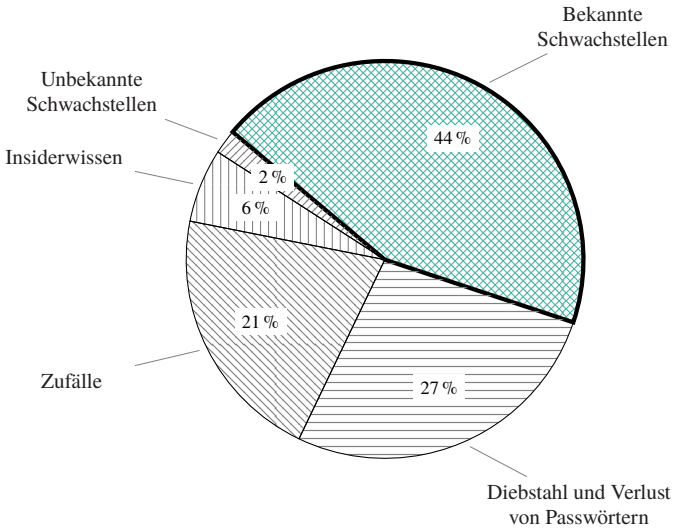



Abbildung 1.5: Klassifizierung der Ursachen für Angriffe auf IT-Systeme nach [103], [160]

Besonders durch die große Anzahl an Angriffen über bekannten Schwachstellen, ist es wichtig, neue Software während der Entwicklung auf diese zu prüfen. Um dies sicher zu stellen, muss eine Software während des Entwicklungszyklus regelmäßig gegen die bekannten Schwachstellen getestet werden. In der IT-Industrie werden die Software-Komponenten isoliert betrachtet und auf Schwachstellen untersucht. Die umgebenden Komponenten (z. B. Betriebs-

system) werden dabei vernachlässigt. Für automobile Steuergeräte ist dieser Ansatz jedoch nicht sinnvoll. Manche Steuergeräte benötigen elektrische Signale oder die Kommunikation mit weiteren Steuergeräten, um eine Funktion korrekt darstellen zu können [55]. Daher müssen für die Prüfung einer Funktion häufig mehrere Steuergeräte berücksichtigt werden.

Obwohl der Verlust und Diebstahl von Passwörtern einen weiteren großen Teil der Angriffe auf Systeme ausmacht, wird dies im Weiteren nur am Rande diskutiert. Zum einen ist der Schutz gegen gestohlene Passwörter nicht möglich, zum anderen sind diese im Fahrzeug typischerweise vom Hersteller vergeben und dem Benutzer nicht bekannt. Verfahren zur sicheren Speicherung von Passwörtern werden nicht untersucht. Ebenso wird das Thema Insiderwissen nicht weiter betrachtet, da dies über Testprozesse während der Entwicklung nicht beeinflusst werden kann.

## 1.2 Ziele und eigener Beitrag

In dieser Arbeit wird eine Methodik vorgestellt, um die Datensicherheit in der Entwicklung von Steuergerätesoftware sicherzustellen. In dieser Methode werden neben externen Schnittstellen auch interne Signale des Steuergeräts, die Peripherie sowie der Prozessor überwacht. Dies ermöglicht einen tieferen Überwachungsgrad, als bei bisherigen Testmethoden. Bisherige Testmethoden, welche das gesamte Steuergerät betrachten, nutzen die Ein- und Ausgabeschnittstellen zum Stimulieren der Applikation und Verifizieren des Verhaltens. Durch die vorgestellte Testmethode wird zusätzlich zu diesen Schnittstellen ein Einblick in das Steuergerät auf Prozessorebene ermöglicht, wodurch das Verhalten während den Tests detaillierter betrachtet werden kann.

Zu Beginn werden hierfür bekannte Schwachstellen bestehender IT-Systeme analysiert und daraus Anforderungen an den Test der Datensicherheit gestellt. Diese Anforderungen werden anschließend auf die Übertragbarkeit in die Automotive-Domäne untersucht und Kriterien ermittelt, um bestehende Testmethoden sowie die vorgestellte Testmethode zu bewerten.

Zur Bewertung der bestehenden Testmethoden in Bezug auf neue Anforderungen der Datensicherheit wird untersucht, welche Methoden die zuvor hergeleiteten Anforderungen erfüllen und wie diese durch zusätzliche Beobach-

tungspunkte für die Datensicherheit eingesetzt werden können oder adaptiert werden müssen.

Basierend auf den Erkenntnissen der Bewertung bestehender Testmethoden wird eine Vorgehensweise zum Test der Datensicherheit vorgestellt. Da Angriffe nur selten auf Schwächen von einzelnen Schlüsseln oder Algorithmen zurückzuführen sind, wird das Testen der Datensicherheit in eingebetteten Systemen als Gesamtsystem betrachtet.

Da das Gesamtsystem während der Entwicklung jedoch noch nicht zwingend als prototypische Plattform zur Verfügung steht, wird das System virtualisiert und die Software auf einem virtuellen Steuergerät getestet. Zusätzlich entsteht dadurch eine Möglichkeit, die internen Signale und Schnittstellen zu beobachten und damit Rückschlüsse auf das Verhalten des Prozessors zu ziehen.

Abschließend wird eine Skalierbarkeitsuntersuchung durchgeführt, um die Testmethode auf einen Steuergeräteverbund zu übertragen. Zudem wird die Testmethode in den Entwicklungsprozess der Automobilindustrie eingebunden.

## 1.3 Aufbau der Arbeit

Es wird eine Testmethodik vorgestellt, die Datensicherheit im Fahrzeug auf Prozessorebene testbar macht. Hierfür werden in Kapitel 2 die Grundlagen vernetzter Fahrzeugsysteme, Datensicherheit, virtueller Plattformen und virtueller Steuergeräte sowie der Test von Steuergeräten in der Automobilindustrie vorgestellt.

Anschließend werden in Kapitel 3 der aktuelle Stand von Wissenschaft und Technik zur vernetzten Mobilität, dem Test der funktionalen Sicherheit und Datensicherheit beschrieben. Aufbauend auf dem Stand der Wissenschaft und Technik werden Lücken identifiziert.

Um die Bedeutung der Datensicherheit im Fahrzeug herauszustellen, werden in Kapitel 4 die Schwachstellen aus IT-Systemen untersucht und auf Automotive-Steuergeräte übertragen. Hierbei wird ebenfalls der Unterschied zu eingebetteten Systemen aus weiteren Domänen betrachtet.

Aufbauend auf Kapitel 3 und Kapitel 4 werden in Kapitel 5 Anforderungen an ein Testsystem für Datensicherheit hergeleitet, bestehende Testsysteme analysiert und bewertet. Hierbei wird ein Fokus auf die Überwachung der internen Signale eines Steuergeräts gelegt.

Mit den Erkenntnissen aus Kapitel 5 wird in Kapitel 6 eine Testmethodik zur Verifikation der Datensicherheit mittels virtuellen Steuergeräten vorgestellt. Hierfür werden Überwachungsfunktionen erstellt und die Applikation des Steuergeräts untersucht.

Die vorgestellte Testmethodik wird in Kapitel 7 in bestehende Testprozesse eingebunden und Erweiterungen der Testprozesse aus dem Stand der Wissenschaft und Technik aufgezeigt.

In Kapitel 8 wird die vorgestellte Methodik auf ihre Skalierbarkeit für einzelne Steuergeräte und Fahrzeugsysteme untersucht sowie eine Übertragbarkeit für die parallele Ausführung der Tests vorgenommen.

Grenzen der Testmethodik zur Überwachung verschiedener Artefakte werden in Kapitel 9 bewertet sowie eine Übertragbarkeit auf Black-Box-Tests vorgenommen.

Abschließend wird die vorgestellte Testmethodik in Kapitel 10 zusammengefasst.

# 2 Grundlagen

Im folgenden werden die benötigen Begriffe und Grundlagen eingeführt. Diese Grundlagen umfassen dabei einen Überblick über die Themen vernetzte Fahrzeugsysteme, Datensicherheit, virtuelle Plattformen und die Entwicklung von eingebetteten Systemen im Fahrzeug.

## 2.1 Vernetzte Fahrzeugsysteme

Die Funktionalität wird im Fahrzeug mit sogenannten Steuergeräten bereitgestellt. Wird die Funktionalität von mehreren Steuergeräten beeinflusst, spricht man von einem vernetzten System.

### 2.1.1 Steuergeräte

In modernen Fahrzeugen steigt die Anzahl an Assistenzsystemen, um im Fahrzeug sowohl mehr Komfort als auch mehr Verkehrssicherheit zu erreichen. Durch Systeme wie Abstandsregeltempomat, Spurhalteassistent oder Notbremsassistent wird die Sicherheit der Verkehrsteilnehmer erhöht und Unfälle werden reduziert. Gleichzeitig steigt der Komfort durch Verkehrszeichen- oder Müdigkeitserkennung, vernetzte Infotainmentsysteme[1], Assistenten zum automatisierten Ein- und Ausparken, das Halten und Wechseln der Fahrspur oder das automatische Abblendlicht. Zusätzlich wird zukünftig die künstlicher Intelligenz den Komfort weiter erhöhen, um beispielsweise intelligente Temperaturregelungen oder bedarfsgerechte Sitzanpassungen zu ermöglichen. Alle diese Funktionen werden im Fahrzeug auf Steuergeräten ausgeführt.

[1] Z. B. Vernetzung des Infotainment mit Kombiinstrument, Musik-Player, Smartphone oder Spracherkennung

**Definition 7** (Steuergerät)**:**
*Ein Steuergerät (engl.: Electronic Control Unit (ECU)) ist die Umsetzung einer Funktion in einem elektronischen System. Es stellt die Kontrolleinheit eines mechatronischen Systems dar [94].*

Je nach Art der Funktion existieren unterschiedliche Anforderungen an die Steuergeräte. So werden für intelligente Sensoren (z. B. Tag-Nacht Erkennung) und Aktoren (z. B. elektrische Sitzverstellung) teilweise Steuergeräte mit einer bare-metal[2] Programmierung eingesetzt, da Umfang und Komplexität meistens gering sind [116]. Für Funktionen des Automobilantriebsstrangs, der Motorsteuerung, des Fahrwerks und des Kombiinstruments werden Betriebssysteme verwendet, um die Hardware zu abstrahieren und Aufgaben (engl. Tasks) parallel auszuführen.

Zur Abstraktion der Hardware wird auf Steuergeräten Automotive Open Systems Architecture (AUTOSAR) als Betriebssystem genutzt. Neben einem Echtzeitbetriebssystem (Basissoftware) spezifiziert AUTOSAR weitere Software-Module, wie eine Laufzeitumgebung (Runtime Environment) zur Ausführung der Applikationssoftware [86]. Damit bildet AUTOSAR nicht nur das Betriebssystem, sondern eine komplette Software-Architektur ab (siehe Abbildung 2.1). Innerhalb der Applikationsschicht werden die Funktionen in Form einzelner Applikationen ausgeführt, die über die Laufzeitumgebung auf die Basissoftware zugreifen.

In der Basissoftware werden Funktionen zur Hardwareabstraktion, Steuergeräteabstraktion (Kommunikation, Ein-/Ausgaben, Speicher), komplexe Treiber (zur Ausführung von Zugriffen auf die Hardware ohne Hardwareabstraktion) und Dienste (Speicherservice, Kommunikationsservice) ausgeführt. Um die Darstellung im weiteren Verlauf zu vereinfachen, wird der Bereich des Prozessor/Hardware auch als ausführendes Artefakt und die Applikation, Laufzeitumgebung und Basissoftware (Betriebssystem) als steuerndes Artefakt bezeichnet.

Das steuernde Artefakt bezieht sich dabei immer auf eine Software (bare-metal oder mit Betriebssystem), um das ausführende Artefakt und damit die Umwelt

[2] Bei bare-metal wird das Programm direkt auf dem Mikrocontroller ausgeführt, ohne dass ein Betriebssystem vorhanden ist.

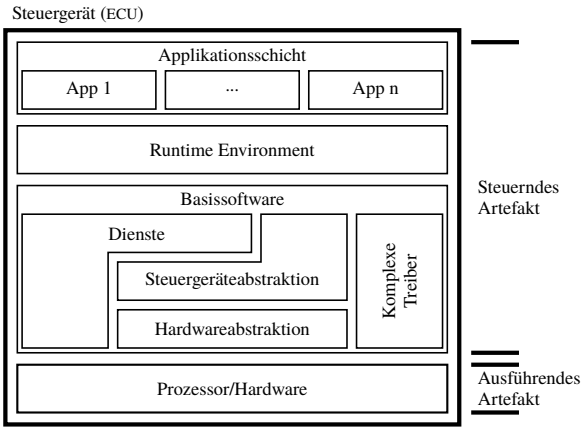



Abbildung 2.1: Aufbau eines AUTOSAR Steuergeräts nach [86]

zu beeinflussen. Das ausführende Artefakt (siehe Abbildung 2.2) ist eine physikalische Einheit (ein sogenanntes Prozessorsystem), auf der eine Software ausgeführt wird. Diese besteht aus mindestens einer Ausführungseinheit (z. B. Arithmetic Logic Unit (ALU)), einer Debug-Schnittstelle sowie mindestens einem Speicher. Über Ein-/Ausgänge (z. B. Sensoren, Aktoren oder Bussysteme) greift das Steuergerät auf die Umwelt zu.

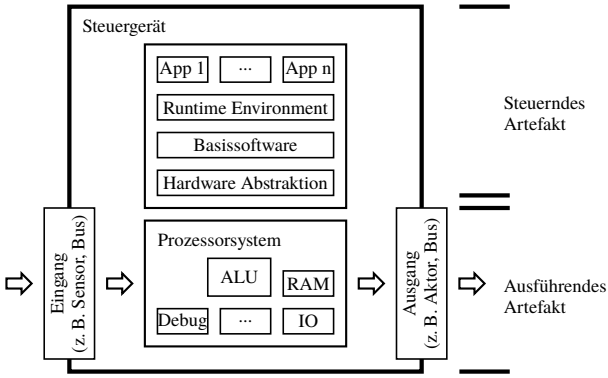



Abbildung 2.2: Aufbau eines Steuergeräts mit steuerndem und ausführendem Artefakt

**Definition 8** (Prozessorsystem)**:**
*Das Prozessorsystem führt die Applikation in einer Recheneinheit aus und enthält Module für Timer, Interruptcontroller, Ein- und Ausgaben sowie Speicher [90].*

Die benötigte Leistung eines Steuergeräts hängt von der Funktionalität ab und reicht von 8-Bit Mikroprozessoren mit 30 MHz bis 16-Bit Mikroprozessoren mit 300 MHz für intelligente Sensoren und Aktoren [148]. Um die Anforderungen bei komplexen Fahrassistentsfunktionen oder der sogenannten Sensorfusion zu erfüllen, werden derzeit 32-Bit Prozessoren mit bis zu sechs Kernen (je 300 Mhz) eingesetzt [139]. Beim Infotainment werden für Audio- und Videoverarbeitung ebenfalls hohe Rechenleistungen gefordert, sodass leistungsstarke Prozessoren mit Grafikbeschleuniger enthalten sein können.

### 2.1.2 Elektrik/Elektronik-Architektur

Durch eine Steigerung der Funktionalität (siehe Kapitel 1.1.2) werden häufig Daten zwischen Steuergeräten ausgetauscht [116]. Eine Punkt-zu-Punkt Verbindung der Steuergeräte ergäbe jedoch hohe Kosten und Gewicht [116]. Daher wurden die Steuergeräte über sogenannte Bussysteme (kurz Bus) in eine Elektrik-Elektronik (E/E)-Architektur integriert.

Zudem werden Fahrzeugfunktionen nicht mehr auf einem einzelnen Steuergerät ausgeführt, sondern die Berechnung läuft verteilt auf einem Steuergeräteverbund. Für einen Spurhalteassistent nimmt eine Kamera Bilder auf, um die Fahrbahnmarkierung zu detektieren. Diese werden an ein weiteres Steuergerät übermittelt, das die detektierten Spuren mit der Position des Fahrzeugs vergleicht, eine Trajektorie berechnet und Befehle an die Lenkung versendet, um mittig in der Spur zu fahren. Kommen weitere Funktionen zum automatisierten Fahren hinzu, so nimmt auch die Vernetzung der Steuergeräte weiter zu.

Um die Kommunikation im Fahrzeug herzustellen werden Bussysteme wie CAN, LIN, FlexRay, MOST oder Ethernet im Fahrzeug verbaut [131]. Die Verknüpfung der Steuergeräte bildet dann eine sogenannte E/E-Architektur, die typischerweise in verschiedene Bereiche (Domänen) unterteilt ist (siehe Abbildung 2.3).

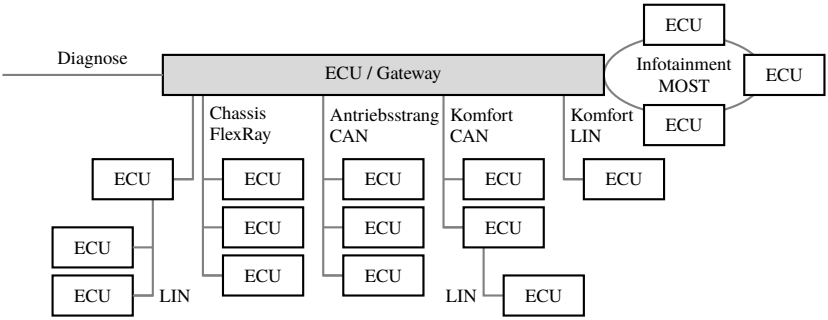



Abbildung 2.3: Exemplarischer Aufbau einer E/E-Architektur nach [116]

Weiter Formen der E/E-Architektur sind beispielsweise domänenbasiert oder mit einem Zentralrechner [116]. Die Kommunikation der Teilfunktionen über ein entsprechendes Bussystem haben jedoch alle Architekturen gemeinsam. Daher wird im weiteren eine beispielhafte E/E-Architektur aus [116] verwendet (siehe Abbildung 2.3).

### 2.1.3 Vernetztes Fahrzeug

Neben der Vernetzung im Fahrzeug, um Fahrassistenzfunktionen abzubilden und ein automatisiertes Fahren zu ermöglichen, steigt auch die Vernetzung des Fahrzeugs mit der Umwelt über eine sogenannte Kommunikationsschnittstelle (siehe Abbildung 2.4).

**Definition 9** (Vernetztes Fahrzeug)**:**
*Ein vernetztes Fahrzeug verbindet Fahrzeugeigenschaften (Funktionen, Sensoren, Aktoren, Speicher) mit Geräten (Fahrzeugen oder Infrastruktur) die über ein Netzwerk außerhalb des Fahrzeugs verbunden sind [34].*

Die Vernetzung der Fahrzeuge wird zum einen genutzt, um den Verkehrsfluss zu erhöhen, vor Rettungsfahrzeugen zu warnen oder Fahrzeuge mit dem Flottenmanagement und der Infrastruktur zu verknüpfen (Fahrzeug zu Fahrzeug und Fahrzeug zu Infrastruktur Kommunikation (Car2X)). Zum anderen werden Verbindungen zu Smartphones und dem Internet hergestellt, um bei-

spielsweise Navigationsziele zu teilen oder von der Ferne den Ladezustand des Fahrzeugs zu überwachen. Hierzu werden schon heute drahtlose Schnittstellen wie Bluetooth, WLAN und Internet integriert. Seit 2014 führt Daimler beispielsweise sukzessive eine Datenübertragung in allen Fahrzeugklassen ein, um Kunden serienmäßig die Möglichkeit zu geben aktiv auf das Fahrzeug zuzugreifen [118]. Unter anderem werden Zentralverriegelung, Standheizung, Fahrzeugortung und Diagnosedaten angebunden [118]. Auch eine Anbindung an Backend-Systeme und Cloud-Dienste ist in zukünftigen Fahrzeugen denkbar.

Darüber hinaus ist seit dem 1. April 2018 das sogenannte eCall-System europaweit für neu zugelassene Fahrzeuge Pflicht [26], das über eine integrierte Internetverbindung (Global System for Mobile Communications (GSM)) im Notfall mit der Rettungsleitstelle kommuniziert.

Zukünftig werden Fahrzeuge Diagnosedaten über das Internet an Werkstätten senden und eine Intervallwartung[3] in eine prädiktive Wartung[4] ändern. Weiter können Software-Updates über das Internet auf Fahrzeugen installiert werden (Software-Over-The-Air (SOTA)), ohne dies in der Werkstatt über die Onboard Diagnose (OBD) Schnittstelle durchführen zu müssen.

Wird die Vernetzung des Fahrzeugs mit der E/E-Architektur aus Abbildung 2.3 verknüpft, ergibt sich ein System, dass intern über Bussysteme kommuniziert und Informationen an Schnittstellen zur Umwelt bereit stellt (siehe Abbildung 2.4).

Durch die Anbindung von externen Geräten in die E/E-Architektur des Fahrzeugs besteht eine indirekte Verbindung zwischen Umwelt (Infrastruktur, Smartphones und Internet) und den Fahrfunktionen. Durch die Vernetzung mit der Umwelt können nicht nur Daten an die Infrastruktur, Smartphones, etc. gesendet und von diesen empfangen, sondern ebenfalls von Angreifern manipuliert werden. Dadurch kann ein Angreifer zum einen Informationen aus dem Fahrzeug erhalten und zum anderen manipulierte Daten an das Fahrzeug

[3] Intervallwartungen bezeichnen das periodische Ausführen von Wartungsarbeiten. z. B. alle 2 Jahre oder 30.000 km

[4] Prädiktive Wartung beschreibt einen vorausschauenden Ansatz, bei dem Bauteile proaktiv auf Basis erhobener Daten gewartet werden, bevor diese ausfallen, ohne feste Wartungsintervalle vorzugeben.

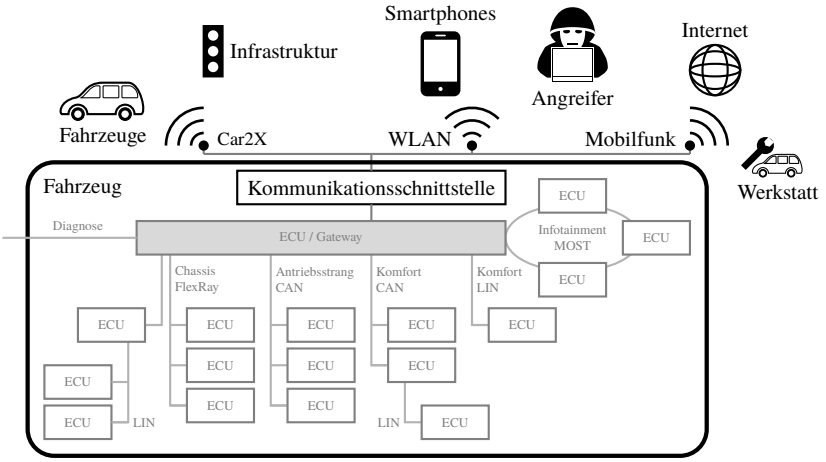



Abbildung 2.4: Exemplarische Vernetzung der E/E-Architektur aus Abbildung 2.3 mit der Umwelt

senden, um das funktionale Verhalten im Fahrzeug zu beeinflussen. Die Funktionen werden zwar gekapselt und mit Firewalls versehen, um Einflüsse von außen zu verhindern, jedoch können durch Fehlfunktionen die Fahrfunktionen manipuliert oder Wissen über die Fahrzeug interne Kommunikation erhalten werden.

## 2.2 Rolle von Hersteller und Zulieferern bei der Entwicklung von Automobilen

Bei der Entwicklung eines Fahrzeugs werden Steuergeräte und funktionale Komponenten nicht ausschließlich vom Hersteller (Original Equipment Manufacturer (OEM)) des Fahrzeugs entwickelt und produziert. Der Einbau einer Zentralverriegelung in der Fahrzeugtür, erfordert in der Endmontage des OEM, eine Software von Softwarelieferanten und das Türsystem von einem Modullieferanten (Tier 1). Der Tier 1 bezieht dabei einzelne Komponenten (Fenster, Verkleidung, Griffe, Türsteuerung, etc.) von einem Komponentenlieferanten (Tier 2), der wiederum Einzelteile von einem Teilelieferanten (Tier 3) bezieht. Die jeweiligen Anforderungen der Komponenten und Einzelteile werden dabei

vom Auftraggeber im sogenannten Lastenheft festgehalten und vom Auftragnehmer im sogenannten Pflichtenheft mit der umzusetzenden Lösungsmöglichkeit beschrieben.

Da die Zulieferer in der Zulieferkette mit jeder Ebene mehr werden, wird die Kette auch als Lieferpyramide bezeichnet, um die Struktur der Lieferanten (Tier 1 - Tier 3) bis hin zum Produzenten des Endproduktes (OEM) abzubilden. Der OEM steht dabei an der Spitze der Pyramide und bekommt von Lieferenten Software, Module, Komponenten oder einzelne Teile geliefert (siehe Abbildung 2.5) [113]. Lieferanten auf den unteren Stufen können Stufen überspringen und andere Lieferanten oder sogar den OEM direkt beliefern.

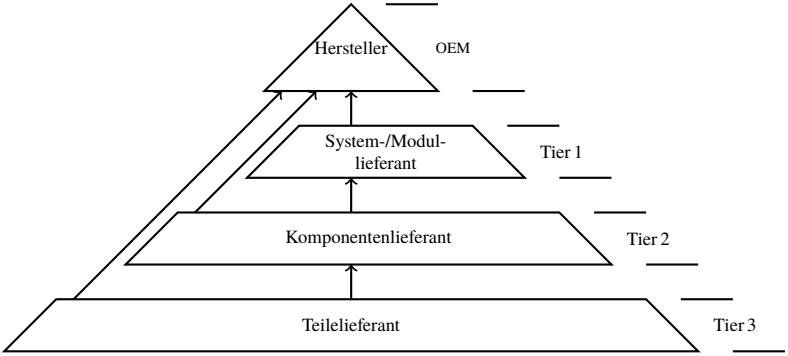



Abbildung 2.5: Symbolische Darstellung der Zulieferkette als Pyramide [113]

Für den Test der Datensicherheit werden Komponenten und Systeme mit Softwareanteil, nicht jedoch Einzelteile, die ein Tier 3 bereitgestellt, betrachtet. Daher bezieht sich der Begriff Zulieferer im weiteren Verlauf auf Lieferanten, die Software für Systeme oder Komponenten herstellen. Betrachtet man dabei ein Steuergerät, so spezifiziert der Hersteller das funktionale Verhalten und kauft hierfür Systeme oder Komponenten bei den Zulieferern ein. Der jeweilige System- oder Komponentenlieferant implementiert die geforderte Funktionalität. Hierfür wählt oder entwickelt dieser das Steuergerät, um die Funktion auszuführen. Werden Software und Hardware von verschiedenen Lieferanten angefertigt, so muss der Hersteller die Schnittstellen zwischen Software und Hardware (inkl. Prozessortyp) entsprechend spezifizieren. Nach dem Kauf al-

ler Komponenten und Module kann der Hersteller ein Gesamtsystem aufbauen und die Funktionalität validieren.

## 2.3 Funktionale Sicherheit

In der Automobilindustrie ist derzeit die funktionale Sicherheit ein maßgeblicher Faktor für die Entwicklung neuer Fahrzeuge. Hierbei wird ein besonderer Fokus darauf gelegt, dass das elektronische System beim Auftreten eines Fehlers mit gefahrbringender Wirkung in einem sicheren Zustand verbleibt, bzw. einen sichern Zustand herstellen kann [44].

**Definition 10** (Funktionale Sicherheit)**:**
*Die funktionale Sicherheit (engl. Safety) ist ein Teil der Gesamtsicherheit, die von der korrekten und zuverlässigen Funktion des Systems abhängt. Bei fehlerhafter Funktion können Gefahren für Personen oder Güter entstehen. [44]*

Für die Einhaltung der funktionalen Sicherheit muss ein System demnach so implementiert werden, dass Fehler nur mit "hinreichendem Restrisiko" [44] vorkommen und das Fahrzeug jederzeit sicher geführt werden kann.

In der Automobilindustrie wird die Klassifizierung des auftretenden Risikos anhand Automotive Safety Integrity Level (ASIL) vorgenommen. Um das Risiko auf ein "gesellschaftlich akzeptiertes Maß" zu reduzieren schreiben Normen, wie ISO 26262 [44], das Vorgehen zur Bewertung vor. Im Vorgehen werden dabei die Gefahr (engl. Exposure), Schwere (engl. Severity) und Beherrschbarkeit (engl. Controllability) bewertet und zu einem Gesamtrisiko verrechnet. Als Ergebnis werden die Funktionen in ASIL-Werte zwischen ASIL A bis ASIL D eingestuft, wobei ASIL A das geringste und ASIL D das höchste Risiko darstellt. Für Funktionen, deren Gesamtrisiko als gering eingestuft wird, sind Verfahren des Qualitätsmanagements (QM) ausreichend.

Während der Entwicklung eines Steuergeräts müssen die Komponenten dann anhand der in der Norm festgeschriebenen Methoden erstellt werden, wobei die Anforderungen mit höherem ASIL ansteigen. Neben den Methoden der Entwicklung enthält die Norm ISO 26262 zudem eine Definition der Rollen,

Verantwortlichkeiten und Tätigkeiten für das funktionale Sicherheitsmanagement sowie Maßnahmen für die Verifikation und Validierung [44].

## 2.4 Datensicherheit

Obwohl Angriffe auf Fahrzeuge derzeit seltener als im IT-Bereich vorkommen (vgl. Kapitel 3.4 und [66]), ist nicht auszuschließen, dass durch die zunehmende Vernetzung der Fahrzeuge gewollte oder ungewollte Manipulationen von Fahrfunktionen zunehmen. Daher wird die Datensicherheit zunehmend wichtiger. Um unerlaubten Zugriff auf ein System zu erlangen, werden grundlegende Fehler im Design, der Architektur oder Implementierung (weakness) als Schwachstelle ausgenutzt. Aufbauend auf diesen Schwachstellen entstehen Sicherheitslücken. Diese beschreiben das Phänomen jedoch nicht allgemeingültig, sondern zielen auf genaue Beispiele und Angriffe ab.

Um die Datensicherheit gegen weitere Arten der Sicherheit abzugrenzen, wird sie zunächst definiert und anschließend gegen weitere Arten der Sicherheit abgegrenzt.

**Definition 11** (Datensicherheit)**:**
*Datensicherheit ist die Sicherheit von Daten vor dem Zugriff Unbefugter [57].*

Der Schutz der Daten vor dem Zugriff Unbefugter beinhaltet dabei die Vertraulichkeit (nur autorisierte Benutzer haben Zugriff auf die Daten), die Integrität (Schutz der Daten vor beabsichtigtem oder unbeabsichtigtem Verändern) und die Verfügbarkeit (Gewährleistung des Zugriffs auf die Daten). Die Datensicherheit wird in der Literatur häufig auch als IT-Sicherheit oder Cyber-Security (kurz Security) bezeichnet. Im weiteren werden die Begriffe daher synonym verwendet.

### 2.4.1 Unterschied Datensicherheit und funktionale Sicherheit

Der Unterschied zwischen funktionaler Sicherheit und Datensicherheit besteht darin, dass es sich bei funktionaler Sicherheit um den Umgang mit Fehlfunk-

tionen handelt, die durch unvorhergesehene Fehler verursacht werden [44]. Datensicherheitsbedrohungen werden von einem Angreifer verursacht, der das System absichtlich verändern, beteiligten Personen Schaden zufügen oder Informationen sammeln will [87].

**Definition 12** (Angriff auf Fahrzeug)**:**
*Überträgt man Definition 2 und Definition 11 auf ein Fahrzeug, ergibt sich, dass ein Angriff auf ein Fahrzeug die mutwillige Beeinflussung der Funktionalität oder die Erlangung von Daten ist.*

Bezogen auf ein Fahrzeug ist das Erlangen von Wissen besonders für den Datenschutz wichtig. Durch einen Angreifer dürfen keine persönlichen Daten freigegeben werden. Hierzu zählen neben Fahrzeugposition (GPS) auch Telematikdaten (Geschwindigkeit oder Fahrprofil). Die Manipulation von Daten, das Erlangen von unbefugten Berechtigungen oder das Stören eines Dienstes, kann im Fahrzeug die Funktionalität von Fahr- oder Komfortfunktionen beeinflussen.

Ein weiterer Unterschied der Datensicherheit ist, dass die Datensicherheit im Gegensatz zur funktionalen Sicherheit zu den nicht funktionalen Anforderungen eines Systems gehört [3].

### 2.4.2 Unterschied Datensicherheit und Datenschutz

In der IT-Sicherheit wird zwischen Datensicherheit und Datenschutz unterschieden, da zum einen Daten vor Manipulationen geschützt sein müssen und zum anderen Rückschlüsse auf Personen durch Daten nicht möglich sein dürfen.

**Definition 13** (Datenschutz)**:**
*Beim Datenschutz handelt es sich um den Schutz von personenbezogenen oder personenbeziehbaren Daten. [27]*

Beim Datenschutz haben die zu schützenden Daten direkten Bezug auf eine natürliche Person (Namen, Adressen, etc.) und dienen meist der Analyse des Verhaltens der entsprechenden Person. Während der Datenschutze nur auf per-

sonenbezogene Daten fokussiert ist, kennt die Datensicherheit keine Trennung zwischen personenbezogenen und nicht personenbezogenen Daten [57]. Bei der Datensicherheit sind zusätzlich relevante Daten für das ausführende System betroffen (Variablen, Betriebssystem, Applikations-Code, kryptografische Schlüssel, etc.). Beim Datenschutz steht der unbefugte Zugriff auf die Daten im Vordergrund, ohne eine ausreichende Datensicherheit ist dies jedoch nicht umsetzbar.

Der weitere Fokus ist die Datensicherheit. Daher wird im Folgenden nicht weiter zwischen Daten mit und ohne Personenbezug unterschieden. Für beides wird im weiteren Verlauf der abstrahierte Begriff “Daten“ verwendet.

Im weiteren Verlauf wird unter Sicherheit des Systems und Security immer die Datensicherheit verstanden. Der Datenschutz und die Verarbeitung personenbezogener oder geschützter Daten ist im weiteren nicht explizit im Fokus, wird jedoch in der Implementierung auf dem Steuergerät über die Datensicherheit abgedeckt.

### 2.4.3 Angriffsklassifizierung auf IT-Systeme

Auf IT-Systeme werden Angriffe in der Regel ausgeführt um unautorisierten Zugriff auf Daten zu bekommen (Spoofing), Daten zu manipulieren (Tampering), Handeln mit falscher Identität (Reputiation), Daten zu stehlen (Inforamtion Disclosure), Benutzer zu stören, eine Funktionalität einzuschränken (Denial of Service) oder höhere Benutzerrechte zu erlangen (Elevation of Privileges). Diese Klassifizierung basiert auf dem von Microsoft vorgestellten STRIDE-Modell [40]. Der Zugriff auf unautorisierte Daten umfasst dabei nicht nur den in Kapitel 2.4.2 vorgestellten Datenschutz und den Schutz der Funktionalität, sondern auch den Schutz von Intellectual Property (IP). Hierunter versteht man den Schutz des geistigen Eigentums, wie den Quellcode und anderes Wissen zum Herstellen eines Produkts.

Da die Datensicherheit in der IT-Industrie bereits weit verbreitet ist, wird auf den Erkenntnissen der IT-Industrie aufgebaut und diese auf die Fahrzeugindustrie übertragen. Eine Gegenüberstellung und Übertragung der Domänen folgt in Kapitel 4.

### 2.4.4 Klassifizierung der Angreifergruppen und deren Motivation

Ein wichtiger Grundsatz der IT-Sicherheit ist, dass jedes Sicherheitssystem von einem ausreichend motivierten, qualifizierten und finanzierten Angreifer durchbrochen werden kann. Es ist davon auszugehen, dass irgendwann in der Zukunft eine Schwachstelle entdeckt wird, die dem Angreifer den Zugriff auf das System erlaubt [85]. Da ein System nie zu 100 % sicher gestaltet werden kann, ist es für die Betrachtung der Datensicherheit wichtig, welche Gruppen das System angreifen. Eine Kategorisierung der Angreifergruppen kann anhand der Motivation, dem zeitlichen und finanziellen Budget sowie dem vorhandenen Wissen vorgenommen werden [14]. In Abhängigkeit dieser vier Faktoren, ist zu unterscheiden, welches Potential ein Angreifer besitzt, um Schwachstellen auszunutzen. Überträgt man dies auf das Automobil, so entstehen folgende Angreifergruppen:

**Fahrer** zeichnen sich durch kein Spezialwissen aus. Die Motivation eines Fahrers zur Manipulation seines Fahrzeugs ist die Freischaltung von Fahrfunktionen oder Services ohne Bezahlung. Ein Weiterer Grund für die Manipulation ist die Erhöhung des Wiederverkaufswerts. Alleine durch die Manipulation des Kilometerzählers entsteht in Deutschland ein jährlicher Schaden von 6 Millionen Euro [105]. Da ein Fahrer kein Spezialwissen besitzt, kann dieser keine Schwachstellen direkt ausnutzen, sondern benötigt eine fertige Software, um die Fahrfunktionen zu manipulieren.

**Skript-Kiddies** sind Angreifer mit wenig Budget und geringem Know-How. Bei dieser Angreifergruppe wird das eindringen und manipulieren der System aus Langeweile als “Sport“ betrieben. Skript-Kiddies haben ein Basiswissen und wissen, wie Schwachstellen beeinflusst werden, benötigen aber spezielle Software für die Angriffsvektoren, um auf die Schwachstellen zugreifen zu können.

**Diebe** stehlen ganze Fahrzeuge durch Manipulation der Fahrzeugelektronik (Fahrzeugschlüssel, Zentralverriegelung oder Wegfahrsperre) oder manipulieren Fahrzeugteile um diese nach einem Diebstahl weiter zu verkaufen [39]. Das Wissen dieser Klasse ist divergent. Das Wissen der Diebe startet beim Wissen der Skript-Kiddies, die spezielle Software für die Angriffe benötigen und reicht bis zu Expertenwissen, um Schwachstellen

direkt ausnutzen zu können. Hierfür wird eigene Software geschrieben, um Zugriff auf bekannte Schwachstellen zu bekommen.

**Konzerne** können ein großes Know-How und Budget einsetzen, um die Konkurrenz gezielt zu schwächen oder auszuspionieren. Durch das Spezialwissen kann eigene Software geschrieben werden, um Zugriff auf bekannte Schwachstellen zu bekommen. Zusätzlich ist es möglich, dass durch das vorhandene Spezialwissen neue Schwachstellen im System entdeckt werden.

**Regierungen** haben ein nahezu unbegrenztes Budget und können beliebig viel Know-How engagieren. Ziel ist es die Wirtschaft oder Regierung anderer Länder zu schwächen. Diese Klasse setzt das Spezialwissen ein, um noch unbekannte Schwachstellen zu identifizieren. Dadurch wird die Schwachstelle möglichst lange ausgenutzt, bevor diese durch ein Softwareupdate behoben wird.

Besonders die drei Gruppen Fahrer, Skript-Kiddies und Diebe setzten in der Regel nur einfache Mittel zum Angreifen oder Manipulieren der Fahrzeuge ein. Durch eine sichere Implementierung und manipulationssichere Kommunikation werden diese drei Gruppen bereits an Angriffen auf die Fahrzeuge gehindert oder deren Zugriff zumindest erschwert. Um eine sichere Implementierung und Kommunikation zu erreichen, ist es nötig, dass keine Schwachstellen im System vorhanden sind. Da eine vollständige Abdeckung aller Sicherheitslücken in komplexen Systemen aus Zeitgründen nicht möglich ist [55], müssen während der Entwicklung zumindest bekannte Schwachstellen geschlossen werden. Dadurch kann bereits ein Großteil der Angriffe verhindert werden (siehe Abbildung 1.5).

### 2.4.5 Overflows in Buffern, Heap und Stack

In der Informationssicherheit und Programmierung wird eine Anomalie, bei der ein Programm beim Schreiben von Daten in einen Speicher die Speichergrenze überschreitet und benachbarte Speicherplätze überschreibt, als Overflow bezeichnet (siehe Abbildung 2.6). [30]

Overflows werden oft durch fehlerhafte Eingaben ausgelöst, wenn davon ausgegangen wird, dass alle Eingaben kleiner als eine bestimmte Größe sind. Schreibvorgänge, die mehr Daten erzeugen, führen dazu, dass über die Gren-

zen des Speichers hinaus geschrieben wird. Wenn dadurch benachbarte Daten oder ausführbarer Code überschrieben werden, kann dies zu einem unregelmäßigen Programmverhalten führen, einschließlich Speicherzugriffsfehlern, falschen Ergebnissen und Abstürzen. [30]

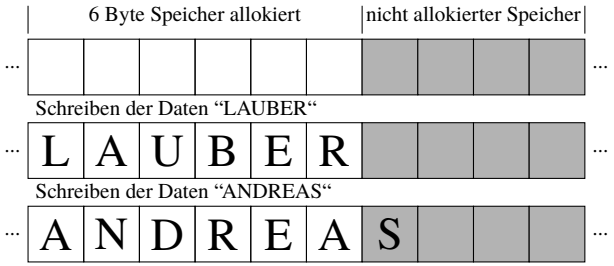



Abbildung 2.6: Erzeugung eines Buffer-Overflows durch überschreiben von nicht allokiertem Speicher

Wird beispielsweise ein Buffer mit 6 Byte im Speicher allokiert, so kann das Datenwort “LAUBER“ geschrieben werden, ohne dass die Speichergrenzen verletzt werden (siehe Abbildung 2.6). Das Schreiben des Datenworts “ANDREAS“ umfasst 7 Byte und überschreibt daher eine nicht allokierte Speicherzelle. Wird die überschriebene Speicherzelle bereits verwendet, so werden die darin enthaltenen Werte überschrieben und damit verfälscht.

Besondere Bereiche stellen bei Overflows der Heap und Stack dar. Stack wird in einem Programm für die statische Speicherzuweisung und Heap für die dynamische Speicherzuweisung verwendet, die beide im RAM des Computers gespeichert sind. Jedoch ist zu beachten, dass eine Speicherzuweisung im Stack den unteren Bereich des Speichers vergrößert. Eine Zuweisung im Heap vergrößert den oberen Bereich des Speichers (siehe Abbildung 2.7). Dadurch wachsen beide Bereiche aufeinander zu [30]. Werden nun viele Daten in Heap und Stack geschrieben, vergrößern sich die Bereiche, bis diese aneinander angrenzen oder in einander überlaufen. Bei Overflows auf dem Heap und Stack spricht man, je nach Richtung des Overflows, von einem Heap- oder Stack-Overflow. Ein Zeiger (engl. Pointer) gibt dabei die oberste bzw. unterste genutzte Adresse auf dem Heap und Stack (Heap- und Stack-Pointer) an. Ein Base-Pointer gibt die unterste mögliche Grenze zwischen Speicher für Variablen und Programmspeicher an. Im Programmspeicher unterhalb des

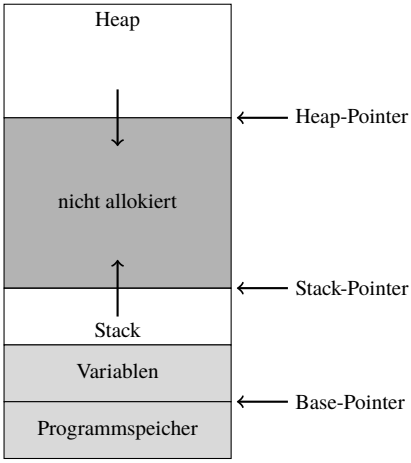



Abbildung 2.7: Heap/Stack-Overflows durch überschreiben von nicht allokiertem Speicher

Base-Pointers wird das auszuführende Programm abgespeichert, im Bereich über dem Base-Pointer werden alle Variablen abgelegt.

Ein Zwischenschritt zum Heap-/Stack-Overflow ist der sogenannte “Memory Exhaustion“. Dabei wird versucht den Speicher durch Vergrößern von Heap oder Stack maximal zu allokieren, bis kein freier Speicher bleibt. Dadurch wird verhindert, dass andere Funktionen Speicher allokieren und die Funktionen an ihrer Ausführung gehindert.

### 2.4.6 Injections zum Ausführen von Code

Injection ist die Ausnutzung eines Overflows, der durch die Verarbeitung ungültiger Daten verursacht wird. Die Injektion wird von einem Angreifer verwendet, um Code in ein anfälliges Computerprogramm einzubringen (oder zu “injizieren“) und den Ablauf der Ausführung zu verändern [30].

Nutzt man einen Overflow (siehe Kapitel 2.4.5) aus, um Speicherbereiche zu manipulieren, so können die überschriebenen Speicherzellen nicht nur mit Werten, sondern ebenfalls mit ausführbarem Code beschrieben werden. Überschreibt der Overflow genug Speicher, so wird ebenfalls der Programmspei-

cher überschrieben (siehe Abbildung 2.7). Daten oder Code, die durch einen solchen Overflow geschrieben wurden, können im weiteren Ablauf des Programms aufgerufen und damit ausgeführt werden, wodurch der Ablauf des Programms verändert (manipuliert) wird.

## 2.5 Virtuelle Modelle, Plattformen und Steuergeräte

Wenn Software für eingebettete Systeme entwickelt wird, stehen in den frühen Phasen der Entwicklung keine Prototypen für das Steuergerät zur Verfügung. Daher werden die späteren Steuergeräte als Modell nachgebaut und das Verhalten der Software auf diesen simuliert (siehe Kapitel 2.5.1). Somit kann bereits frühzeitig mit der Entwicklung der Software begonnen werden.

**Definition 14** (Modell)**:**
*Ein Modell ist die Abbildung eines Systems, das aufgrund der Anwendung bekannter Gesetzmäßigkeiten, einer Identifikation oder auch getroffener Annahmen gewonnen wird. Das System wird bezüglich ausgewählter Fragestellungen hinreichend genau abbildet [1].*

Eine hinreichend genaue Abbildung des Systems stellt stets eine vereinfachte Form des realen Systems dar, die nur für einen speziellen Zweck entwickelt ist [41]. Daher ist das Modell nur für einen spezifischen Zweck bzw. die ausgewählte Fragestellung gültig. Da die Kosten der Modellerstellung exponentiell mit der Gültigkeit des Modells steigen, ist eine möglichst genaue Abbildung nicht erstrebenswert. Der eingeschränkte Gültigkeitsbereich ist jedoch für das Modell nicht hinderlich, solange damit die ausgewählte Fragestellung gelöst werden kann [41].

Um eine Software auch in frühen Entwicklungsphasen zu testen, wird ein Modell des Systems aufgebaut und die Funktionalität des zu entwickelnden Steuergeräts (engl. Electronic Control Unit (ECU)) simuliert. Die Modelle des Steuergeräts können mit Hochleistungsrechnern, Mikroprozessoren oder rein

virtuell auf einem Computer[5] bestehen. Durch die Nutzung virtueller Modelle während der Entwicklung muss keine reale Hardware eingesetzt werden und die Funktionalität des Systems (z.B. Steuergeräts) kann auf beliebigen Computern ausgeführt werden.

**Definition 15** (Gesamtsystem)**:**
*Ein Gesamtsystem ist ein Verbund, dass aus verschiedenen teils heterogenen Subsystemen besteht [98].*

Bezogen auf die Funktionalität eines Fahrzeug wird im folgenden das Gesamtsystem auf eine spezifische Funktionalität bezogen und als Modell betrachtet. Bezogen auf die E/E-Architektur aus Kapitel 2.1.2 können Funktionen über mehrere Steuergeräte verteilt sein. D. h. das Modell eines Gesamtsystems besteht aus mindestens einer Plattform (Steuergerät, Sensor, Aktor, etc.) mit den darin befindlichen Komponenten und der Kommunikation zu anderen Systemen.

**Definition 16** (Plattform)**:**
*Die Plattform bezeichnet in der Informationstechnik eine einheitliche Grundlage, auf der Anwendungsprogramme ausgeführt oder entwickelt werden [24].*

Im weiteren Verlauf wird beim Begriff Plattform kein Unterschied gemacht, unabhängig, ob es sich um ein Steuergerät, Sensor oder Aktor handelt. Die Plattform bezeichnet dabei immer das Modell des realen Geräts. Wird die Anwendung auf einer realen Ausführungseinheit (Ziel-Plattform) betrieben, so wird von Steuergerät, Sensor oder Aktor gesprochen.

**Definition 17** (Ziel-Hardware)**:**
*Die Ziel-Hardware bezieht sich auf die reale Ausführungseinheit (z. B. Prozessor) die im Steuergerät des Fahrzeugs verbaut wird.*

Um die Recheneinheit innerhalb des Systems abzubilden wird zusätzlich der Begriff des Prozessormodells eingeführt.

[5] Der Begriff Computer bezieht sich auf Desktop-PCs (Personal Computer) und Server (Netzwerkressource zum Verarbeiten eines Programms).

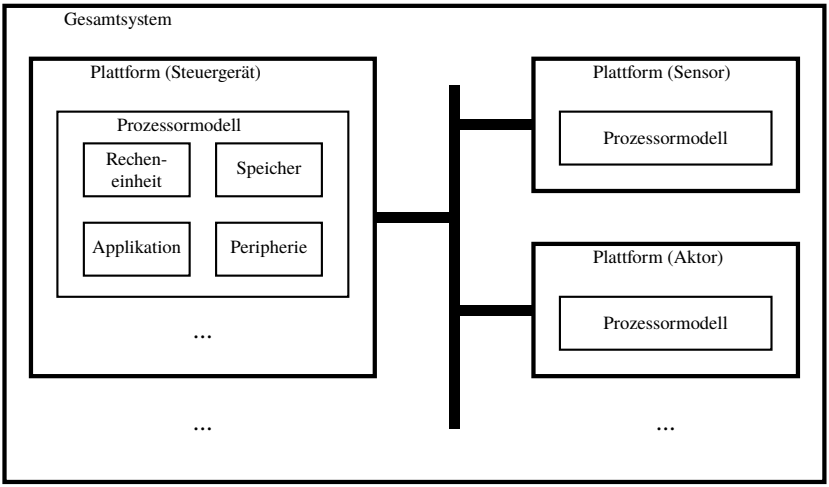



Abbildung 2.8: Hierarchie und Zusammenhang der verschiedenen virtuellen Modelle

**Definition 18** (Prozessormodell)**:**
*Ein Prozessormodell ist ein mathematisches Modell, dass alle Komponenten eines Prozessors beinhaltet. Hierzu zählen die Recheneinheit, Speicher, Peripherie und die verbindenden Busstrukturen innerhalb des Prozessors (lokale Busse).*

Eine Übersicht der Hierarchiestufen und Zusammenhänge der verschiedenen Modelle ist in Abbildung 2.8 schematisch dargestellt.

Ein Gesamtsystem oder eine Plattform kann als reale Hardware und als Modell vorhanden sein. Durch eine entsprechende Simulation können Anwendungsprogramme darauf ausgeführt werden. Im Unterschied zum Prozessormodell, das nur simuliert werden kann, bezeichnet die Plattform die Ausführungseinheit in der Simulation und in der realen Welt. Besteht das Modell oder die Plattform nur auf dem Computer, wird von einem virtuellen Modell beziehungsweise einer virtuellen Plattform gesprochen.

### 2.5.1 Simulation virtueller Modelle und virtueller Plattformen

Da ein virtuelles Modell und eine virtuelle Plattform nur auf dem Computer existieren, ist es nötig, diese auf einem Computer (Host-PC) auszuführen. Eine solche Ausführung wird als Simulation oder Emulation der Plattform oder des Modells bezeichnet.

**Definition 19** (Emulation)**:**
*Emulation ist der Prozess der Nachahmung des nach außen hin sichtbaren Verhaltens [93]. Der innere Zustand des Emulationsmechanismus muss nicht dem inneren Zustand der realen Einheit entsprechen.*

Bei einer Emulation ist nicht nötig das System im Detail zu modellieren, um das gewünschte Verhalten zu erhalten. Lediglich Ein- und Ausgaben müssen mit dem realen System identisch sein.

**Definition 20** (Simulation)**:**
*Bei einer Simulation handelt es sich um eine abstrakte, aber möglichst realitätsnahe Nachbildung des Geschehens der Wirklichkeit. D.h. die Simulation ist eine Ausführung der Applikation ohne reales System, welche die Abläufe der Software mithilfe eines Modells nachstellt [1].*

Das Ziel der Simulation ist es in der Plattform so weit wie möglich jedes Detail zu modellieren, um darzustellen, was in der Realität geschieht. Durch die Simulation eines Modells können Parameter und Schnittstellen beobachtet werden, die in der realen Umgebung nicht zugänglich sind. Grund hierfür ist, dass nicht auf reale, physikalische Schnittstellen zugegriffen werden muss, sondern jeder Simulationsschritt unterbrochen und die Zustände der Simulation beobachtet werden können. Das Modell muss hierfür jedoch die Beobachtungspunkte hinreichend genau abbilden.

Im folgenden sind die internen Zustände des zu beobachtenden Systems von Bedeutung. Daher wird im weiteren Verlauf der Fokus auf einer Simulation liegen. Dennoch ist es möglich Teile des Modells die nicht beobachtet werden zu emulieren. Führt man diese Teile des Modells in einer separaten Emulation

oder Simulation aus, so wird von einer Co-Emulation oder Co-Simulation gesprochen.

**Definition 21** (Co-Simulation)**:**
*Die Co-Simulation ist die gemeinsame Ausführung mehrerer verschiedenen Simulatoren. Diese können verschiedene Domänen mit unterschiedlichen Zeitschritten berücksichtigen [102].*

Während der Co-Simulation tauschen die jeweiligen Subsysteme Daten über dedizierte Schnittstellen in den Simulatoren aus.

Bei der Simulation eines Modells wird zusätzlich zwischen zwei Stufen, der Paravirtualisierung und der Gesamtsystemsimulation, unterschieden.

**Definition 22** (Paravirtualisierung)**:**
*Paravirtualisierung ist eine Technologie zur Virtualisierung, die Schnittstellen bereitstellt, welche der Hardware ähnlich aber nicht identisch sind [122].*

Durch das Bereitstellen ähnlicher Schnittstellen, kann die Anwendung des Systems unverändert bleiben, der sogenannte Hardware Abstraction Layer (HAL) muss jedoch angepasst werden. Diese Anpassung übernimmt bei der Paravirtualisierung ein sogenannter Hypervisor. Daher ist es bei einer Paravirtualisierung nicht möglich, die gesamte Applikation (inkl. HAL) im virtuellen Modell und auf der realen Hardware ohne Anpassung auszuführen. Einsatz findet die Paravirtualisierung beim funktionalen Test von Algorithmen, da hierfür die Hardwareschnittstellen nicht benötigt werden.

Eine Erweiterung zur Paravirtualisierung bildet die Gesamtsystemsimulation. Diese ermöglicht es, eine Applikation ohne Änderung in der Simulation und auf der realen Hardware auszuführen. Für eine Gesamtsystemsimulation muss das virtuelle Modell hinreichend genau nachgebildet sein, um Fehlverhalten durch Simulation des Modells auszuschließen.

**Definition 23** (Gesamtsystemsimulation)**:**
*Ein Gesamtsystemsimulator ist ein Architektursimulator, der in einer solchen Detailtiefe simuliert, dass die Software aus dem realen Systemen ohne Modifikation im Simulator ausgeführt werden kann [63].*

Eine Gesamtsystemsimulation stellt effektiv virtuelle Hardware bereit, die unabhängig von der Art des Host-PCs ist. Das Gesamtsystem-Modell zur Simulation eines Prozessors muss Prozessorkerne, Peripheriegeräte, Speicher, Verbindungsbusse und Netzwerkverbindungen umfassen.

### 2.5.2 Aufbau einer Gesamtsystemsimulation

Um eine Applikation in einer Gesamtsystemsimulation auszuführen, ist zunächst eine virtuelle Plattform mit Prozessormodell (Prozessor, Speicher und der dazugehörigen Peripherie) nötig. Diese Plattform kann um ein Modell des Gesamtsystems erweitert werden. Zusätzlich muss eine Applikation existieren, die für die entsprechende Ziel-Hardware erzeugt (Cross-Kompiliert) wurde. Durch das Cross-Kimpilieren kann die Applikation auf der realen und virtuellen Plattform ausgeführt werden. Dadurch liegt die Applikation im Maschinencode spezifisch für die verwendete Plattform vor. Diese Maschinenbefehle werden von der Gesamtsystemsimulation eingelesen und interpretiert, sodass die Applikation unter Berücksichtigung der Plattform spezifischen Randbedingungen ausgeführt wird.

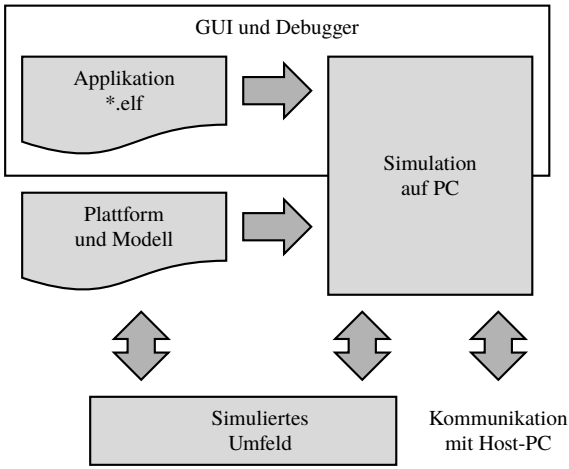



Abbildung 2.9: Aufbau einer Gesamtsystemsimulation

Durch eine Benutzeroberfläche (Graphical User Interface (GUI)) kann ein Programmierer Einfluss auf die Applikation und Simulation nehmen und beides steuern. Für die Ausführung der Simulation werden durch ein simuliertes Umfeld sowie über eine Kommunikation zum Host-PC Daten eingegeben und Ausgaben protokolliert.

### 2.5.3 Zyklen- und instruktionsakkurate Simulation

Bei der Gesamtsystemsimulation von virtuellen Plattformen wird zwischen einer zyklenakkuraten und instruktionsakkuraten Simulation unterschieden.

**Definition 24** (Zyklenakkurate Simulation)**:**
*Eine zyklenakkurate Simulation simuliert die Architektur zyklisch und bildet damit das exakte Zeitverhalten nach [58].*

Durch die Nachbildung des Zeitverhaltens ist es möglich Rückschlüsse auf Bearbeitungsdauern zu erhalten. Dieses Wissen wird genutzt, um Deadlocks oder Probleme durch unterschiedliches Zeitverhalten zu identifizieren. Die ausgeführten Instruktionen werden bei dieser Art der Simulation abstrahiert, um das exakte Zeitverhalten sicherzustellen. Ein instruktionsgenaues Verhalten ist bei dieser Simulation nicht möglich. Hierzu dient die sogenannte instruktionsakkurate Simulation.

**Definition 25** (Instruktionsakkurate Simulation)**:**
*Eine instruktionsakkurate Simulation (Instruction Set Simulation (ISS)) ist ein Simulationsmodell, das Anweisungen, interne Variablen und den Zugriff auf die Register des Prozessors befehlsgenau abbildet [56].*

Durch eine instruktionsakkurate Simulation wird das Verhalten eines Prozessors imitiert. Hierfür werden die prozessorspezifischen Befehle im Host-PC interpretiert und nachgebildet. Da sich der Befehlssatz des Prozessors und des Host-PCs unterschieden, können Simulationszeit und reale Ausführungszeit verschieden sein. Das Zeitverhalten auf der realen Hardware kann jedoch durch die bekannte Ausführungszeiten pro Instruktion zurückgerechnet werden [58]. Die Anpassung der Simulationsgeschwindigkeit je Prozessor ermöglicht

es dann die heterogene Simulation verschiedener Prozessoren, Befehlssätze und Prozessorgeschwindigkeiten zyklengenau nachzubilden.

Zur Betrachtung der Datensicherheit sind sowohl zyklenakkurate Simulation als auch instruktionsakkurate Simulation wichtig. Da durch eine zyklenakkurate Simulation das Zeitverhalten ermittelt wird, kann hiermit auf Angriffe über sogenannte Seitenkanäle geschlossen werden. Bei diesen Angriffen werden physikalische Effekte wie das Zeitverhalten oder die Stromaufnahme beobachtet. Im Gegensatz hierzu wird bei einer instruktionsakkuraten Simulation auf die genaue Abbildung der Funktion geachtet. Hierdurch können fehlerhafte Implementierungen erkannt werden. Im weiteren werden Seitenkanäle nicht berücksichtigt, da diese Einflüssen nicht nur von den gewählten Algorithmen, sondern auch von der gewählten Hardware abhängen. Daher können Seitenkanalangriffe von einer korrekten Implementieren der Software nicht oder nur gering beeinflusst werden. Im weiteren Verlauf wird daher der Fokus auf eine instruktionsakkurate Simulation gelegt.

## 2.6 Entwicklung von eingebetteten Systemen in der Automobilindustrie

Die steigende Anzahl an Funktionalität im Fahrzeug (siehe Kapitel 1) wird besonders durch den Einsatz sogenannter eingebetteter Systeme vorangetrieben. Für die Gewährleistung der funktionalen Sicherheit sind in der Automobilentwicklung Prozesse und Testmethoden etabliert. Die wichtigsten Prozesse und Testmethoden werden im folgenden vorgestellt.

### 2.6.1 Eingebettete Echtzeitsysteme

Eingebettete Systeme sind Computersysteme, die als Bestandteil eines größeren (technischen) Systems informationsverarbeitende Aufgaben wahrnehmen [74]. In diesem Zusammenhang sind eingebettete Systeme in Haushaltsgeräten, Smartphones und Medizintechnik anzutreffen. Neben genannten Bereichen werden eingebettete Systeme auch in der Automobilindustrie in Form von Steuergeräten eingesetzt und kommen zum Einsatz, wenn mindestens eine der vier folgenden Kategorien benötigt wird [3]:

1. Steuerung, Überwachung und Regelung
2. Daten- und Signalverarbeitung
3. Interaktion mit Benutzern
4. Systemmanagement, Konfiguration und Diagnose

Bei der Steuerung, Überwachung und Regelung werden Sensorwerte ausgelesen und zur Verfügung gestellt. Die Sensorwerte werden in der Daten- und Signalverarbeitung aufbereitet (z. B. gefiltert). Mithilfe der Daten- und Signalverarbeitung werden die Regelgrößen ermittelt und an die Aktoren zur Regelung weitergegeben. Falls für das System Benutzereingaben erforderlich sind, stellt das eingebettete System die Schnittstellen (z. B. Knöpfe und Anzeigen) zur Verfügung und wertet diese aus. Durch das Systemmanagement und die Konfiguration werden Parameter des eingebetteten System während des Starts und zur Laufzeit gesetzt und durch die Diagnose überwacht. Durch eine Kommunikation können eingebettete Systeme Daten mit weiteren Systemen austauschen.

Das eingebettete System stellt dabei die Schnittstellen zwischen der Regelungstechnik in Software und der Umwelt des technischen Systems dar (siehe Abbildung 2.10). Die Schnittstellen zur Umwelt werden durch Sensoren und Aktoren (auch Peripherie genannt) realisiert.

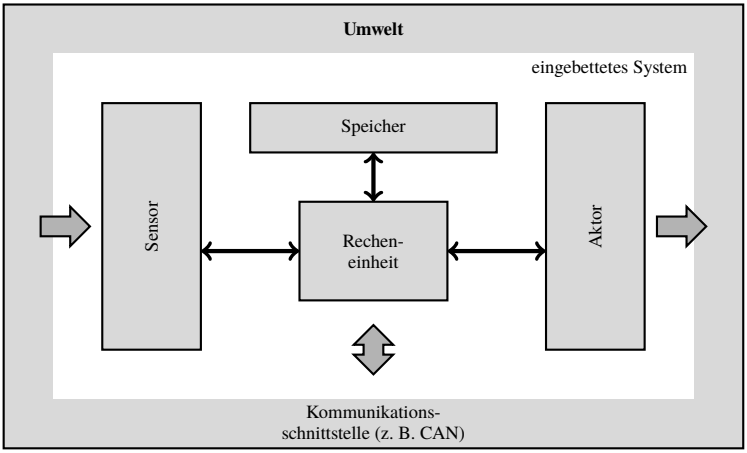



Abbildung 2.10: Eingebettetes System mit Schnittstellen zur Umwelt

Das eingebettete System umfasst neben den Schnittstellen zur Umwelt mindestens eine Recheneinheit und einen Speicher. Die Recheneinheit wird durch die Software gesteuert und erledigt die gestellten Aufgaben. Innerhalb des Speichers werden Ergebnisse und die auszuführende Software abgespeichert.

Besonders für sicherheitskritische Funktionen (wie Bremse oder ABS) müssen eingebettete Systeme in der Automobilindustrie echtzeitfähig sein, um auf Ereignisse reagieren zu können und Personen nicht zu gefährden. Dazu müssen Ergebnisse innerhalb einer begrenzten Zeit abgearbeitet werden.

Der Begriff eingebettetes System und Steuergerät wird im weiteren synonym verwendet. Ein Steuergeräteverbund stellt eine Einheit aus mehreren Steuergeräten dar, die über gemeinsame Kommunikationsschnittstellen (z.B. CAN, FlexRay oder Ethernet) verbunden sind.

### 2.6.2 Entwicklungsmethoden für eingebettete Systeme in der Automobilindustrie

Eingebettete Systeme werden typischerweise anhand verschiedener Methoden entwickelt, welche die Entwicklungsschritte festlegen. Als klassische Entwicklungsprozesse sind das Wasserfallmodell, Spiralmodell und das V-Modell zu nennen. In der Automotive Entwicklung wird verstärkt auf das V-Modell gesetzt, daher wird an dieser Stelle nur das V-Modell betrachtet. Für einen Überblick der weiteren Entwicklungsmodelle wird auf die Literatur verwiesen [59].

Das V-Modell ist ein Vorgehensmodell zum Planen und Durchführen von Entwicklungsprojekten über den gesamten Lebenszyklus eines Produkts [111]. Es legt dabei die Vorgehensweise, die Art der Ergebnisse sowie die Kommunikation zwischen Auftraggeber und Auftragsnehmer fest. In der Softwareentwicklung wird das V-Modell derzeit in der Automobilentwicklung eingesetzt und ist Pflicht für alle Entwicklungsprojekte des Bundes [111]. Dieses Vorgehensmodell basiert auf den vier Punkten:

- Systemerstellung
- Qualitätsmanagement
- Konfigurationsmanagement
- Projektmanagement

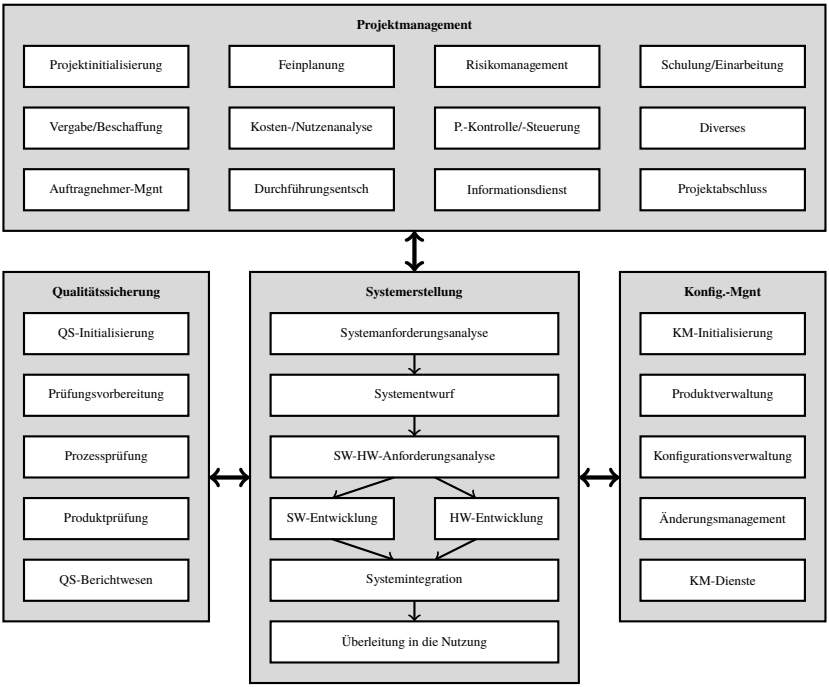



Abbildung 2.11: Submodelle des V-Modells aus [111]

Die Zusammenhänge der einzelnen Modelle sind in Abbildung 2.11 gezeigt. Für eine ausführliche Beschreibung des Qualitätsmanagement, Konfigurationsmanagement und Projektmanagement wird auf die Literatur verwiesen [92, 111].

### 2.6.3 Systemerstellung im V-Modell

Für die Entwicklung der Software eines Steuergeräts wird innerhalb des V-Modells das Submodell der Systemerstellung genutzt. Im V-förmig angeordneten Submodell ist das Vorgehen von der Anforderungsanalyse über den Entwurf und die Implementierung, bis zum Test in einzelnen Schritten definiert. Einen Überblick über das Vorgehen gibt Abbildung 2.12.

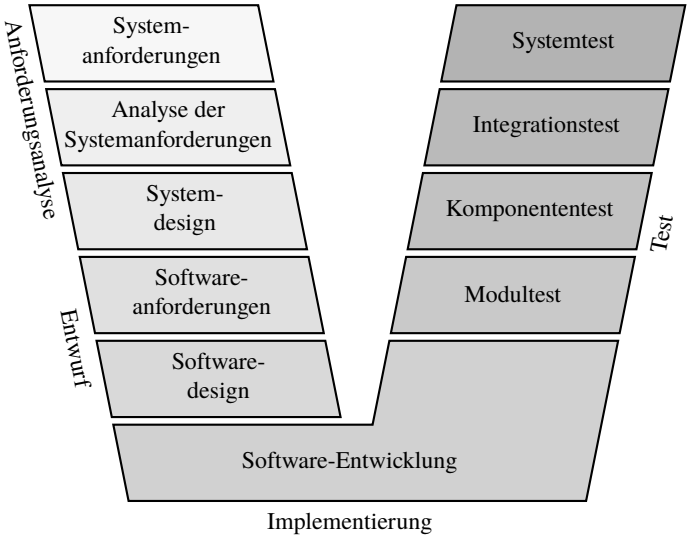



Abbildung 2.12: Kurzform der Systemerstellung des V-Modells [73, 91]

Für die Systemerstellung werden auf der linken Seite Anforderungen und die Software als Grob- und Feinentwurf erstellt. Im unteren Bereich der Systemerstellung wird die Software implementiert, bevor sie auf der rechten Seite integriert und getestet wird. Durch eine Verknüpfung der rechten und linken Seite wird überprüft, ob die Anforderungen der linken Seite korrekt entwickelt wurden. Die Verifikation stellt dabei sicher, ob das Produkt seiner zuvor erstellten Spezifikation entspricht.

Die Bezeichnung V-Modell bezieht sich im folgenden immer auf das Submodell der Systemerstellung, da diese im Fokus steht. Für einen Überblick der weiteren Submodelle wird auf die Literatur verwiesen [111].

### Anforderungsanalyse

Während der Analyse (siehe Abbildung 2.12) werden die Anforderungen an das System in formaler Sprache aufgestellt. In diesem Bereich werden Funktionsumfänge festgelegt und die Basis für das späteren Systementwurf und den Systemtest gelegt. Zudem werden die Randbedingungen an die technische und organisatorische Umgebung betrachtet und Risiken bewertet [91].

### Entwurf

Im Entwurf (siehe Abbildung 2.12) werden Software und Hardware des Systems in Form eines "Top-Down"-Ansatzes entworfen. Hierfür wird das Design der Funktionalität immer weiter verfeinert bis schließlich ein detaillierter Entwurf des Systems, der Software und der Hardware vorhanden ist.

### Implementierung

Basierend auf dem Entwurf werden anschließend (siehe Abbildung 2.12) einzelne Module für die Software durch klassische Programmiersprachen implementiert und die Hardware erstellt [59].

### Testen

Die implementierten Software-Module werden zuerst einzeln im Modultest auf ihre Funktionalität getestet (siehe Abbildung 2.12) und anschließend durch die Integration zu größeren Modulen zusammen gefasst ("Bottom-Up"). Während der Integration wird durch Integrationstests sichergestellt, dass durch das Zusammenwirken der Module keine Fehler in das System integriert werden. Durch eine zunehmende Integration wird das System immer vollständiger, sodass abschließend das gesamte System auf Fehler geprüft werden kann [59].

Nach Abschluss aller Testschritte wird das System in die Nutzung überführt.

### 2.6.4 Musterphasen während der Systemerstellung

Während der Entwicklung werden Muster des Systems angefertigt, um das Verhalten frühzeitig zu verifizieren. Diese Muster werden dabei in die Kategorien A - D eingeteilt und geben Aufschluss über den Reifegrad des jeweiligen Musters [73].

A-Muster ist ein bedingt fahrtaugliches Funktionsmuster mit Einschränkungen der spezifizieren Eigenschaften. Geeignet ist das A-Muster für Voruntersuchungen (z. B. Simulation auf einem Computer, siehe Kapitel 2.6.5).

B-Muster ist ein funktionsfähiges, fahrtaugliches Muster, das mit komplettem Funktionsumfang erstellt wurde und für den Einsatz im Fahrzeug genutzt werden kann (z. B. Ausführung auf leistungsstarker Hardware, siehe Kapitel 2.6.5).

C-Muster ist unter seriennahen Bedingungen mit Serienwerkzeugen gefertigt und besteht aus Serienmaterial (z. B. Ausführung auf Ziel-Hardware, siehe Kapitel 2.6.5). Abweichungen von der Spezifikation sind erlaubt und werden dokumentiert.

D-Muster ist unter Serienbedingungen mit Serienwerkzeugen gefertigt und besteht aus Serienmaterial. Abweichungen aus C-Mustern sind korrigiert.

Um Fahrzeugfunktionen zu testen, können die verschiedenen Muster eingesetzt werden. A-Muster und B-Muster werden dabei genutzt, ohne dass eine Zielplattform vorhanden ist. Diese werden auf Prototyping Plattformen umgesetzt und werden daher im weiteren als Prototypen abgekürzt.

### 2.6.5 Methoden der Funktionsprüfung von eingebetteten Systemen

Bei der Funktionsprüfung wird zwischen den zwei Vorgehen Debuggen und Testen unterschieden.

**Definition 26** (Debuggen)**:**
*Das Debuggen dient in der Entwicklung zur Lokalisierung von Fehlern [59].*

*Hierzu wird die Anwendung zum Überprüfen des aktuellen Zustands der Software angehalten und später fortgesetzt.*

Das Debuggen dient bereits während der Implementierung der Fehlerbehebung. Im Gegensatz zum Testen steht beim Debuggen die Fehlerlokalisierung in der Software im Vordergrund. Für das Debuggen müssen entsprechende Schnittstellen im Code und der Hardware verfügbar sein. Anhand dieser Schnittstellen wird ein tiefer Einblick in das System gegeben, wodurch direkte Rückschlüsse auf das ausgeführte Verhalten gezogen werden können (siehe Kapitel 5).

**Definition 27** (Testen)**:**
*Beim Testen werden Eingabeparameter (Stimuli) über Schnittstellen in die Anwendung gegeben und die Ausgaben beobachtet. Entsprechen die Ausgaben dem erwarteten Verhalten, so ist der Test fehlerfrei [91].*

Das Testen erfordert eine systematische Herangehensweise mit Vorbedingungen, Stimuli und erwarteten Ausgaben. Das Testen dient dazu Fehler in der Implementierung zu erkennen. Bei einem fehlerfreien Durchlauf der Testfälle wird angenommen, dass die Applikation richtig funktioniert, da die erwarteten Ausgaben generiert wurden. Interne Zustände werden nicht überwacht, sondern nur Ein- und Ausgaben. Die Wiederhol- und Reproduzierbarkeit sind wesentliche Bedingungen zum Testen [91].

Während den Tests durchläuft der Testgegenstand verschiedene Phasen. Der jeweilige Test orientiert sich dabei am Zeitpunkt der Entwicklung und dem aktuellen Stand im V-Modell. In den frühen Phasen der Entwicklung werden Modelle erstellt und durch sogenannte Model-in-the-Loop (MiL)-Tests verifiziert. Die Simulation eines Modells erfolgt in der Entwicklung bereits frühzeitig anhand grafischer Beschreibungen. Durch diese grafischen Beschreibungen wird anschließend der Quellcode in einer Entwicklungsumgebung (Integrated Development Environment (IDE)) generiert oder geschrieben und mit Hilfe einer Laufzeitumgebung (Runtime Environment (RTE)) zum Testen ausgeführt (Software-in-the-Loop (SiL)). Durch das Kompilieren wird der Maschienencode erzeugt, der auf einem Prozessor ausgeführt werden kann. Dieser Code wird auf eine reale Hardware geschrieben (engl. Flashen) und innerhalb eines Hardware-in-the-Loops (HiLs) ausgeführt. Nach den erfolgreichen Tests im HiL

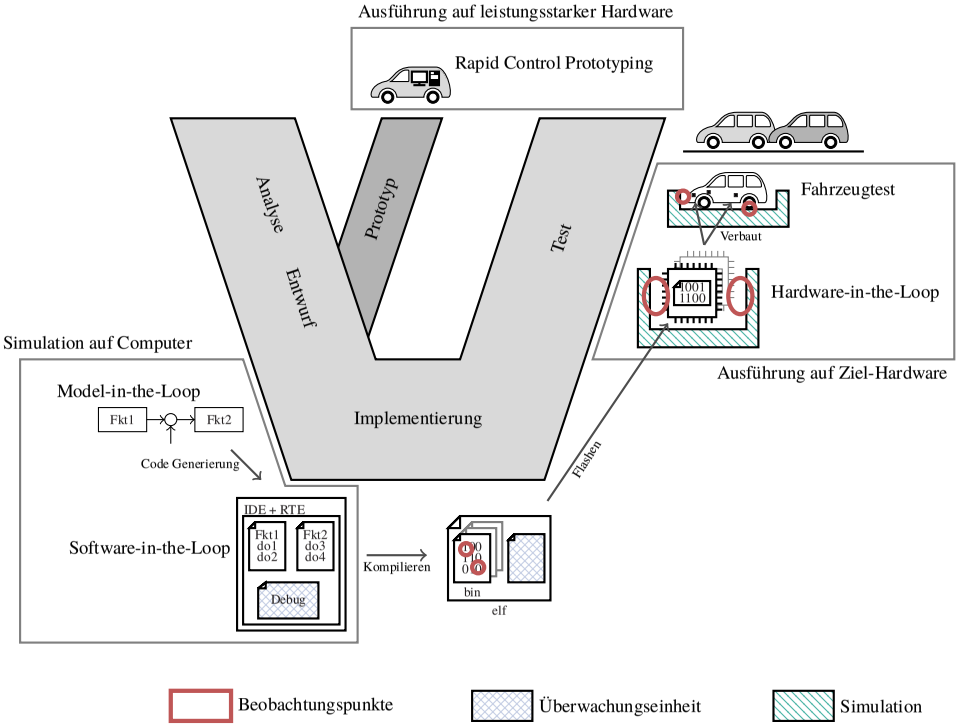



Abbildung 2.13: Einordnung und Kategorisierung der Tests im V-Modell

wird die Hardware in Form eines Steuergeräts im Fahrzeug verbaut, dieses auf dem Fahrzeugprüfstand und später auf der Straße getestet.

Alternativ lassen sich Funktionen bereits frühzeitig als Prototypen in ein Gesamtsystem integrieren, um die Funktionalität unter realen Bedingungen validieren zu können. Hierfür wird die Applikation auf spezielle Rapid Control Prototyping (RCP) Systeme aufgebracht [77]. Diese Systeme bestehen aus einer leistungsstarken Hardware, um Funktionen auszuführen, bevor diese für ein Steuergerät optimiert sind.

Die verschiedenen Tests lassen sich anhand der Entwicklungsphasen in das V-Modells aus Abbildung 2.12 einordnen (siehe Abbildung 2.13) und werden im folgenden als klassische Tests bezeichnet.

Zusätzlich unterscheiden sich die Tests aus Abbildung 2.13 dadurch, welche Art des Codes eingesetzt wird. Grundsätzlich lassen sich dabei die vier verschiedene Arten grafische Beschreibung, Quellcode, Assemblercode und Maschienencode unterscheiden (siehe Abbildung 2.14).

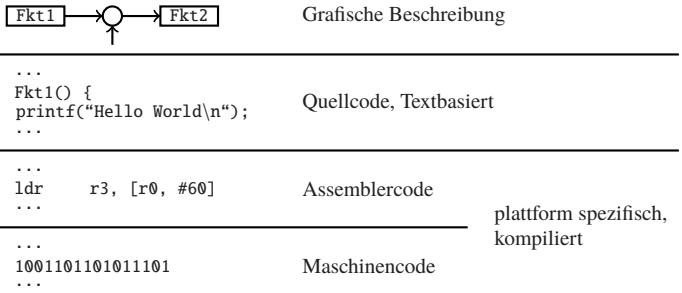



Abbildung 2.14: Vergleich der verschiedenen Arten von Code

**Definition 28** (Grafische Beschreibung)**:**
*Die grafische Beschreibung ist eine Abbildung von Formen und Blöcken, die den Ablauf einer Anwendung repräsentiert [22].*

**Definition 29** (Quellcode)**:**
*Bei sogenanntem Quellcode (engl. Sourcecode) handelt es sich um das in einer von Menschen lesbaren Programmiersprache (auch Hochsprache genannt) geschriebene Programm [24].*

**Definition 30** (Assemblercode)**:**
*Bei Assemblercode handelt es sich um eine von Menschen lesbare Darstellung des Maschinencodes [83].*

**Definition 31** (Maschinencode)**:**
*Bei Maschinencode (auch Binärcode) handelt es sich um die Aneinanderreihung der binären Symbole für die verschiedenen Operationen des Prozessors, sowie verschiedener Daten (z.B. Konstanten oder Sprungtabellen) [83]. Der Maschinencode wird vom Compiler aus dem Quellcode oder der grafischen Beschreibung generiert und ist plattformabhängig.*

Neben der Codeart lassen sich die Tests zusätzlich nach der Ausführungsplattform gruppieren (siehe Abbildung 2.13):

A) Simulation auf einem Computer
B) Ausführung auf leistungsstarker Hardware
C) Ausführung auf Ziel-Hardware
D) Simulation mit virtuellen Plattformen

Die vier Gruppen unterscheiden sich jeweils anhand der Ausführungseinheit, dem steuernden Artefakt, der Realitätsnähe, dem Testsystem und der Möglichkeit für Echtzeitbetrieb (siehe Abbildung 2.15). Bei der Entwicklung nach dem beschriebenen V-Modell können alle Methoden nacheinander in den verschiedenen Phasen eingesetzt werden.

#### A) Simulation auf einem Computer

Eine Möglichkeit Anwendungen frühzeitig zu testen besteht in der Simulation auf dem Computer. Hierfür werden grafische Modelle der Algorithmen erzeugt oder diese textbasiert beschrieben. Diese Modelle werden anschließend auf einem Computer als Model-in-the-Loop (MiL) oder Software-in-the-Loop (SiL) ausgeführt. Bei MiL und SiL werden die Modelle und deren Umgebung ohne physikalische Hardwarekomponenten simuliert. Ein Beispiel für die Simulation auf dem Computer mittels SiL ist die Ausführung der Softwaretests direkt in der Entwicklungsumgebung. Hierfür werden Stimuli in an die Softwaremodule gesendet und das Verhalten innerhalb der Entwicklungsumgebung überwacht. Applikation (Regler) und die Umwelt (Strecke) des Tests werden dabei durch

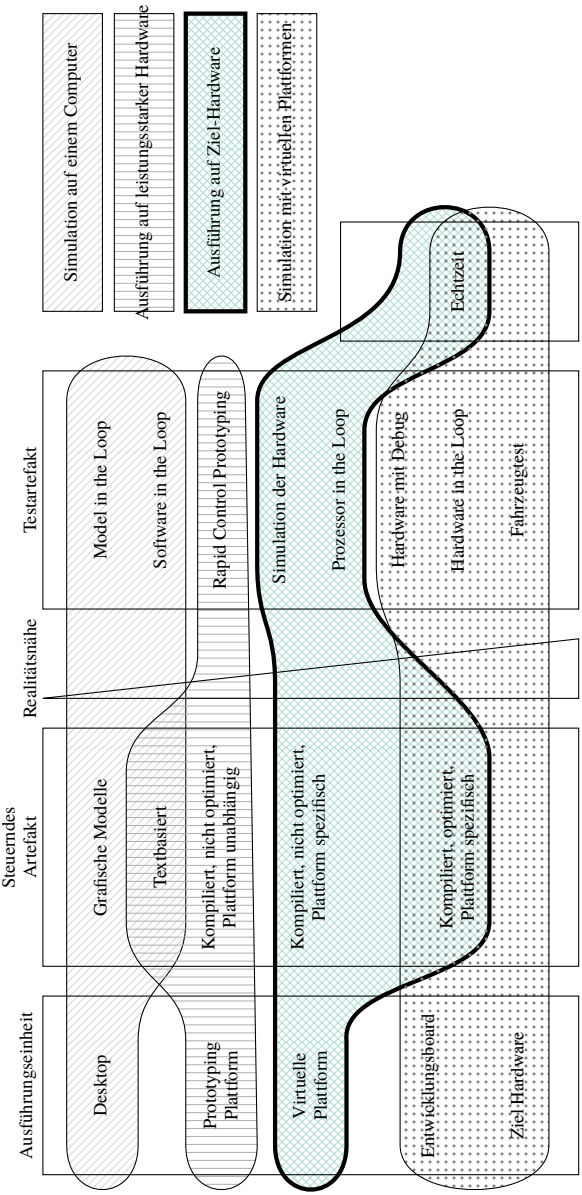



Abbildung 2.15: Übersicht über verschiedene Ausführungsmöglichkeiten zum Testen einer Software

den Computer simuliert. Für dieses Vorgehen muss das Gesamtsystem nicht vorhanden sein, sondern das Ausführen der einzelnen Softwaremodule ist ausreichend. Neben der Simulation der Anwendung können auch Hardwarekomponenten auf einem Computer simuliert werden. Dabei wird das Verhalten der Hardware, ohne Betrachtung der Software, in den Fokus gestellt.

Ziel der Simulation auf einem Computer ist es, die Robustheit und Anwendbarkeit der verwendeten Algorithmen und das daraus resultierende Verhalten am System zu untersuchen [1, 59]. Die Untersuchung der Robustheit bezieht sich dabei auf das Verhalten bei der Bewältigung von fehlerhafte Eingaben.

**Definition 32** (Robistheit)**:**
*Die Robustheit ist die Fähigkeit eines Computersystems, Fehler bei der Ausführung und durch fehlerhafte Eingaben zu bewältigen. [47]*

Die Untersuchung der Funktion des Algorithmus oder der Regelung steht dabei im Vordergrund. Durch die Simulation auf einem Computer ohne reale Hardware oder Umwelt, ist diese Testmethode nicht realitätsnah, wodurch weitere Schritte bis zum fertigen eingebetteten System unternommen werden müssen.

#### B) Ausführung auf leistungsstarker Hardware

Beim ersten Einsatz (Prototypen) werden die Algorithmen zuerst auf leistungsstarker Hardware getestet, bevor die Software in ein dediziertes Steuergerät integriert wird. Hierfür existieren in der Entwicklung beispielsweise spezielle Prototyp-Plattformen. Das Rapid Control Prototyping (RCP) ist dabei eine effiziente Methode, um neue Regelstrategien in einer realen Umgebung zu entwickeln, zu optimieren und zu testen. Um dies zu erreichen, werden mathematische Modelle auf einer Echtzeitplattform mit realen Schnittstellen zur Anbindung an reale Systeme genutzt.

Die für die Anwendung erzeugte Applikation ist in der Regel weder optimiert, noch für eine spezielle Plattform angepasst. Diese Art der Funktionsprüfung wird beispielsweise im RCP eingesetzt. Die programmierte Hochleistungshardware wird anschließend in HiL-Prüfständen oder im Realbetrieb getestet [1]. Obwohl beim RCP das realen Steuergerät zum Testen der Funktionalität nicht

vorhanden ist, wird dies durch Prototyp-Plattformen ersetzt, um bereits frühzeitig Steuerungsfunktionen anhand von realen Signalen zu verifizieren. Nach der Verifizierung muss die Applikation auf das entsprechende Steuergerät portiert und angepasst werden.

#### C) Ausführung auf Ziel-Hardware

Für die Ausführung auf der Ziel-Hardware muss das System bereits als Steuergerät oder zumindest als Entwicklungsplattform vorhanden und die Applikation als plattformspezifischer, optimierter und kompilierter Code auf dem Steuergerät installiert sein. Besonders beim Ausführen auf der Zielplattform gelten die selben Echtzeitbedingungen wie im späteren Einsatz. Eine große Realitätsnähe wird erreicht, da das Steuergerät anschließend direkt im Fahrzeug eingesetzt werden kann. Zum Testen werden Debug-Geäte an spezielle Schnittstellen angeschlossen (Hardware-Debug) oder das Steuergerät wie im Rahmen eines Hardware-in-the-Loop (HiL)-Prüfstands automatisiert getestet [91]. Bei diesen Tests werden weitere Teile des Fahrzeugs sowie die Umgebung simuliert um die Eingangsstimuli zu erzeugen. Im weiteren Verlauf der Tests kann das Prüfobjekt in ein Fahrzeug eingebaut werden, um das Gesamtsystem unter realen Bedingungen im Fahrversuch zu testen.

#### D) Simulation mit virtuellen Plattformen

Für die Simulation mit virtuellen Plattformen werden Modelle der Plattform (inklusive Prozessor und Applikation) auf einem Computer ausgeführt. Im Unterschied zur Simulation auf einem Computer wird jedoch auf der virtuellen Plattform der kompilierte Code ausgeführt, wird dies Processor-in-the-Loop (PiL) genannt. Dieser Code ist für die spezifische Prozessorplattform angepasst und kann sogar optimiert sein. Da keine Änderungen am Code durchgeführt werden müssen, um diesen auf die reale Hardware zu portieren, ist diese Art der Tests realitätsnäher als die Ausführung auf leistungsstarker Hardware oder die Simulation auf einem Computer. Für das Testen müssen jedoch hinreichend genaue Modelle in der Simulation vorhanden sein (siehe Kapitel 2.5).

Um die Tests durchführen und das Ergebnis bewerten zu können, müssen im Prüfobjekt verschiedene Punkte beobachtet werden. Diese Punkte werden als Beobachtungspunkte bezeichnet.

**Definition 33** (Beobachtung)**:**
*Bei der Beobachtung wird der Zustand eines Artefakts überprüft und protokolliert ohne dieses Artefakt zu steuern oder aktiv einzugreifen [24].*

Mithilfe der Beobachtungspunkte wird das Artefakt überwacht und der Ablauf der ausgeführten Einheit gesteuert. Das heißt während der Überwachung wird Einfluss auf die Ausführung des Programms genommen.

**Definition 34** (Überwachung)**:**
*Bei der Überwachung wird der Zustand eines Artefakts kontrolliert und für den richtigen Ablauf gesorgt. Hierfür wird das Artefakt gesteuert, um in den Ablauf aktiv einzugreifen [24].*

Innerhalb der Simulation auf einem Computer werden hierfür spezielle Debug-Artefakte erstellt, welche die Beobachtungspunkte kontrollieren und die Ausführung der Software zur Laufzeit unterbrechen können. Hierfür ist bei der Simulation auf einem Computer die RTE zuständig. Die Debug-Artefakte werden für die Simulation mit virtueller Hardware in die kompilierte Datei gespeichert und innerhalb der Simulation ähnlich zur RTE genutzt. Bei der Ausführung auf einer Ziel-Hardware oder im Fahrzeug können nur die Ein- und Ausgaben beobachtet werden. Ein steuernder Eingriff in Form einer Überwachung ist hier nicht möglich, da die Stimuli der Umwelt kontinuierlich eingespeist werden. Siehe Kapitel 6.3 für die Integration der Beobachtungspunkte in die verschiedenen Testgruppen.

Unabhängig davon, ob ein System beobachtet oder überwacht werden soll, müssen an den Eingangsschnittstellen des Prüfobjekts Stimuli angelegt werden. Hierfür werden entsprechende Testfälle erzeugt. Abhängig vom Ziel der Tests, können diese in zwei Klassen unterteilt werden (siehe Abbildung 2.16). Für funktionale Tests werden Testfälle aus den Anforderungen generiert und anschließend durch HiL oder SiL getestet. Im Fall der Datensicherheit werden Coderichtlinien für automatische Code-Analysen, zufällige Eingaben für Fuzz-

Tests (siehe Kapitel 3.5.4) oder bekannte Schwachstellen (siehe Kapitel 4.1) für Penetrationstest verwendet.

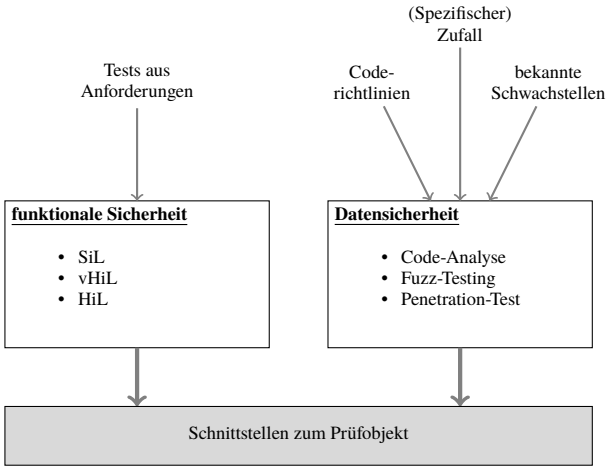



Abbildung 2.16: Testfallgenerierung für funktionale Sicherheit und Datensicherheit

# 3 Stand der Wissenschaft und Technik

Die Automatisierung und die Vernetzung von Fahrzeugen zählen zu den vier Mobilitätstrends im Bereich der Automobilindustrie (siehe Kapitel 1.1.1). Zugleich wird die Datensicherheit durch die steigende Anzahl der IT-Manipulationen, die bei der Vernetzung von Fahrzeugen ebenfalls nicht vernachlässigt werden darf, immer wichtiger (siehe Kapitel 1.1.3). Besonders durch Kombination mit dem fahrerlosen Fahren können Risiken durch IT-Manipulation entstehen, da kein Fahrer zum Eingreifen in das Fahrzeugverhalten vorhanden ist.

## 3.1 Vernetzte und automatisierte Mobilität

In den letzten zehn Jahren hat sich die Mobilität stark verändert. Ein Aspekt zukunftsweisender Mobilitätslösungen stellt dabei das vernetzte und automatisierte Fahren dar.

Heute schon unterstützten Fahrerassistenzsysteme den Fahrer. In Zukunft sollen hochautomatisierte Fahrfunktionen dem Fahrer die Fahraufgabe vollständig abnehmen, sodass der Fahrer sich anderen Aufgaben widmen kann [65]. Fahrerassistenzsysteme unterstützen dabei heute schon den Fahrer in Längs- und Querregelung. So regelt der Abstandsregeltempomat (ACC) die Beschleunigung und Verzögerung, um den Sicherheitsabstand zum vorausfahrenden Fahrzeug einzuhalten [124]. Ein Spurhalteassistent greift darüber hinaus in die Lenkung ein, um das Fahrzeug auf der Fahrspur zu halten [124]. Durch eine Kombination von Längs- und Querregelung werden Fahrerassistenzsysteme zur Folgefahrt im Stau bei niedriger Geschwindigkeit und das automatisierte Fahren auf der Autobahn die ersten Schritte zum sogenannten hoch automatisierten Fahren sein. Zum vollautomatisierten, fahrerlosen Betrieb eines Fahrzeugs müssen je-

doch auch die Verkehrssituationen in Städten vom Fahrzeug beherrscht werden. Eine Roadmap des Verband der Automobilindustrie (VDA) mit bereits erfolgter und geplanter Einführung der Fahrerassistenzsysteme ist in Abbildung 3.1 dargestellt.

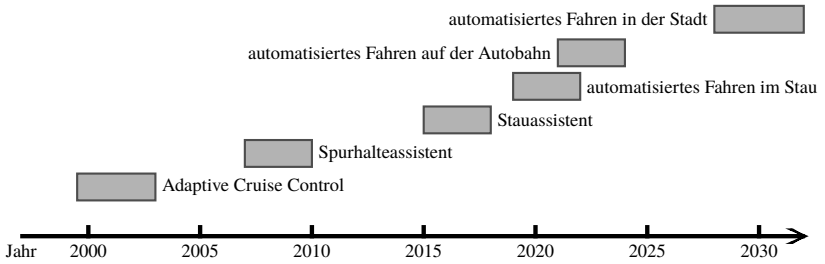



Abbildung 3.1: Einführungszeiträume von assistierten und automatisierten Fahrerassistenzsystemen nach [110]

Zusätzlich zu den Fahrerassistenzsystemen werden die Fahrzeuge immer weiter vernetzt, um Sicherheitsfunktionen und Komfort zu erhöhen. Anstatt dass ein Fahrzeug ein eigenständiges mechanisches Gerät ist, wird es zunehmend in eine mobile Plattform mit umfangreicher elektronischer Sensorik und Rechenleistung umgewandelt [124]. Heutzutage steht innerhalb eines Autos eine große Datenmenge zur Verfügung, die den Zustand des Fahrzeugs sowie das Verständnis seiner unmittelbaren Umgebung in Form von Sensordaten auf einem Bussystem darstellt.

Durch den Austausch von Daten und die Kombination mit automatisiertem Fahren werden neue Konzepte für effizientes Fahren, Verkehrsflussoptimierung und neue Komfortfunktionen möglich. Gleichzeitig ermöglicht der Zugriff über vernetzte Funktionen Möglichkeiten das automatisierte Fahrzeug manipulativ zu beeinflussen.

### 3.1.1 Automatisierte Mobilität

Die Machbarkeit des hochautomatisierten Fahrens wurde bereits im Rahmen diverser Demonstrationen für einzelne Verkehrssituationen gezeigt. Das autonome Fahren wird derzeit von klassischen Automobilherstellern und Zulieferern (z. B. Daimler [130], BMW [106], Audi [31], Bosch [99]) und zunehmend

von neuen spezialisierten Unternehmen (z. B.Tesla [106], Waymo [8]) oder Datenunternehmen wie Google [9] demonstriert.

Um die verschiedenen Entwicklungsstufen beim automatisierten Fahren zu unterscheiden, werden diese in fünf Stufen untergliedert [65, 88]. Ausgehend von einer Stufe 0, bei der der Fahrer sowohl für Längs- als auch Querführung des Fahrzeugs zuständig ist, kann beim assistierten Fahren (Stufe 1) eine der beiden Aufgaben von einem Assistenzsystem übernommen werden (siehe Abbildung 3.2). In Stufe 2, dem teilautomatisierten Fahren, ist der Mensch noch vollkommen verantwortlich und muss das System jederzeit überwachen. Dies entspricht dem heutigen Stand von Fahrerassistenzsystemen [65]. Beim vollautomatisierten Fahren (Stufe 4) ist das System bereits voll für bestimmte Fahraufgaben, zum Beispiel in Stausituation oder auf Autobahnen, verantwortlich. Ab Stufe 3 benötigt das Systeme eine Selbsteinschätzung und muss in der Lage sein den Fahrer zur Übernahme der Fahraufgabe aufzufordern. In der Entwicklungsstufe 5, dem fahrerlosen Fahren, kann die Fahraufgabe vom Fahrzeug voll umfänglich auf allen Straßentypen, in allen Geschwindigkeitsbereichen sowie unter allen Umfeld- und Umweltbedingungen durchgeführt werden [32].

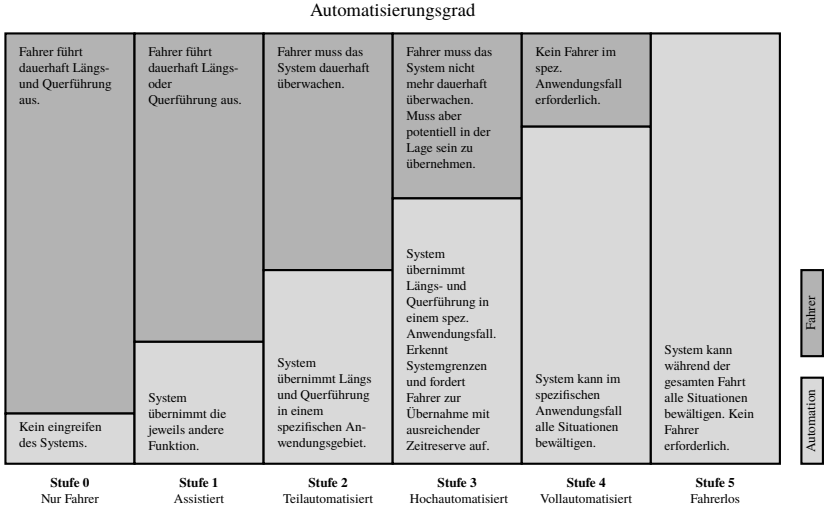



Abbildung 3.2: Stufen des automatisierten Fahrens [88]

Im Bereich der kommerziellen Logistikanwendungen werden autonome Fahrzeuge bereits in abgeschlossenen Bereichen eingesetzt, um Fahrpersonal einzusparen und Abläufe effizienter zu gestalten. Beispiel hierfür ist das Containerterminal Altenwerder in Hamburg, wo Transportfahrzeuge Container automatisch und ohne Fahrer zwischen Kai und Bestimmungsort befördern [49, 104]. Weitere Beispiele für den Einsatz von kommerziellen Anwendungen ist das automatisierte Fahren auf Depots und Betriebshöfen [LGSW16, LSW18b].

Auf öffentlichen Straßen ist dies jedoch aufgrund des Wiener Übereinkommens über den Straßenverkehr [112] bisher nicht zulässig gewesen. “Jedes Fahrzeug und miteinander verbundene Fahrzeuge (Anhänger) müssen, wenn sie in Bewegung sind, einen Führer haben“ [112]. Dies wird darin begründet, dass das Fahrzeug durch den Fahrzeugführer jederzeit beherrschbar sein muss. Im Jahr 2017 wurde dies durch die Bundesregierung und den Bundesrat gelockert, wodurch automatisierte Systeme erlaubt sind, solange “diese Systeme vom Führer übersteuert oder abgeschaltet werden können“ [15]. Zusätzlich wurde vereinbart, dass Fahrer und Computer als rechtlich gleichgestellt gelten [15]. Besonders für die zukünftigen fahrerlosen Systemen der Stufe 5 ist dies wichtig, da ein Eingreifen des Fahrers nicht nötig und evtl. sogar nicht möglich ist und diese Systeme sogar ohne Fahrer bewegt werden können. Dadurch muss die Steuerelektronik besser abgesichert sein, da manipulierte Systeme nicht durch den Fahrer überstimmt werden können. Um nachzuweisen ob Fahrer oder Computer das Fahrzeug gesteuert haben, wird eine Blackbox vorgeschrieben.

Durch die Möglichkeit Fahrzeuge ohne Fahrer fortzubewegen, ergeben sich nicht nur neue Möglichkeiten, sondern auch potenziell neue Gefahren (siehe Kapitel 3.4). Die Herausforderung besteht nun darin, diese im Rahmen der Fahrzeugentwicklung zu erkennen und abzuwenden. Damit gehen ebenfalls neue Anforderungen an Plattformen für die Entwicklung, das Testen und die Absicherung dieser Systeme einher [LS18].

### 3.1.2 Vernetzte eingebettete Systeme im Kraftfahrzeug

Um die verschiedenen Situationen im Straßenverkehr bewältigen zu können, erfassen eine Vielzahl von Sensoren die Umgebung des Fahrzeugs und fassen dies in einer Sensorfusion zu einem Gesamtbild zusammen. Für die Überwa-

chung werden Kameras, RADAR, LiDAR und Ultraschallsensoren eingesetzt, die jeweils unterschiedliche Aspekte der Umgebung wahr nehmen [124]. Bereits heute besteht ein Fahrzeug aus einem Verbund an Steuergeräten (siehe Kapitel 2.1.2), die durch die Vernetzung sowohl Fahrfunktionen als auch Komfort bereitstellen. Hierfür kommunizieren Steuergeräte teilweise über mehrere Bussysteme. Für einen Spurhalteassistent nimmt eine Kamera Bilder auf, um die Fahrbahnmarkierung zu detektieren. Diese werden an ein weiteres Steuergerät übermittelt, das die detektierten Spuren mit der Position des Fahrzeugs vergleicht, eine Trajektorie berechnet und Befehle an die Querregelung versendet. Kommen weitere Funktionen zum automatisierten Fahren hinzu, so nimmt auch die Vernetzung der Steuergeräte zu. Zusätzlich wird künftig zur Fahrzeug internen Vernetzung eine Kommunikation zu weiteren Verkehrsteilnehmern und Infrastruktur hinzukommen (siehe Kapitel 2.1.3).

Vernetztes Fahren geht noch einen Schritt weiter: Telematische Systeme vernetzen die Verkehrsteilnehmer sowie die Verkehrsteilnehmer mit der Infrastruktur (Car2X) [124]. Dadurch können Verkehrs- und Fahrzeuginformationen zwischen den Teilnehmern ausgetauscht werden, um einen höheren Grad an Autonomie zu erreichen. Intelligente Algorithmen auf den Steuergeräten der Fahrzeuge interpretieren die Umwelt durch die Sensordaten, erstellen ein Verständnis der Umgebung und planen ein gemeinsames Fahrmanöver für jedes der beteiligten autonomen Fahrzeuge. Auch eine Anbindung an Backend-Systeme und Cloud-Dienste wird in zukünftigen Fahrzeugen integriert. Hierzu werden schon heute drahtlose Schnittstellen wie Bluetooth (z. B. Anbindung der Freisprechanlage), WLAN (z. B. Verbindung zwischen Navigationssystem und Smartphone) und teilweise Car2X integriert. Darüber hinaus ist seit dem 1. April 2018 das sogenannte eCall-System europaweit für neue Fahrzeuge Pflicht [26], das über eine integrierte GSM Verbindung im Notfall mit der Rettungsleitstelle kommuniziert.

Durch die Vernetzung der Systeme entsteht jedoch auch die Möglichkeit von außen auf das Fahrzeug zuzugreifen. Sind hier fehlerhafte Implementierungen oder Sicherheitslücken vorhanden, können Funktionen des Fahrzeugs gesteuert werden. Besonders durch die Kombination der Vernetzung und dem fahrerlosen Fahren auf Stufe 5 entstehen Möglichkeiten ein Fahrzeug zu steuern, ohne das ein Fahrer eingreifen kann.

## 3.2 Evolutionäres Testen in der Automobilindustrie

Neben den in Kapitel 2.6.5 vorgestellten Testverfahren wird in der Automobilindustrie das evolutionäre Testen zur Testfallgenerierung eingesetzt, um beispielsweise Stimuli zur Validierung eines Parkassistenten zu erzeugen [12].

Das evolutionäre Testen basiert auf populationsbasierten Metaheuristiken, die genutzt werden, um ein Problem näherungsweise zu lösen. Die Tests geben dabei einen abstrakten, nicht problemspezifischen Prozess an, der das Finden der optimalen Lösung ermöglicht, jedoch nicht garantiert. Allgemein beschreiben evolutionäre Algorithmen eine Gruppe von verschiedenen Vorgehensweisen [120].

Evolutionäre Tests eignen sich für Problemstellungen, deren Suchräume nicht numerisch abgebildet werden können. Vergrößert sich der Suchraum auf n-dimensionale Lösungsräume oder besteht dieser aus Unstetigkeiten sowie Lücken, können klassische Optimierungsverfahren, wie beispielsweise das Gradientenverfahren, nicht angewandt werden [79].

Das evolutionäre Testen wurde erstmals 1992 für Tests zur Überprüfung der Codeabdeckung (engl. Code-Coverage) verwendetet [128]. Diese Codeabdeckung beinhaltet die Überdeckung von Anweisungen, Zweigen, Bedingungen und Pfaden.

- Anweisungsüberdeckung: Jede Anweisung im gesamten Code muss einmal ausgeführt werden.
- Zweigüberdeckung: Jede Verzweigung (z. B. if-else Block) muss einmal durchlaufen werden.
- Bedingungsüberdeckung: Alle Bedingungen zur Verzweigung müssen mit wahr und falsch ausgeführt werden.
- Pfadüberdeckung: Die Pfadüberdeckung fordert die Ausführung aller Pfade, die in der Applikation existieren.

Für die Optimierung der Stimuli wird der Ablauf in der Programmstruktur beobachtet.

Später wurde das evolutionäre Vorgehen für das automatisierte Durchlaufen von funktionalen Tests erweitert. Im Gegensatz zu den Tests aus Kapitel 2.6.5

werden die Stimuli anhand der Ausgabewerte der Schnittstellen optimiert. Das Ziel der Optimierung ist eine Verletzung der definierten Anforderung des Systems. Ziel des evolutionären Vorgehens ist, dass das untersuchte System die geforderte Spezifikation nicht mehr erfüllt und dadurch Fehler erkannt werden.

Das allgemeine Vorgehen bei evolutionären Algorithmen ist in Abbildung 3.3 dargestellt. Der erste Schritt besteht aus der Initialisierung einer Menge von Lösungsvorschlägen (Population) [84]. Diese können zufällig, basierend auf vorangegangenen Auswertungen oder Erfahrungswerten erstellt werden. Eine Population besteht dabei aus einzelnen Individuen (Chromosomen) und beschreibt eine Menge von Testdaten (Stimuli).

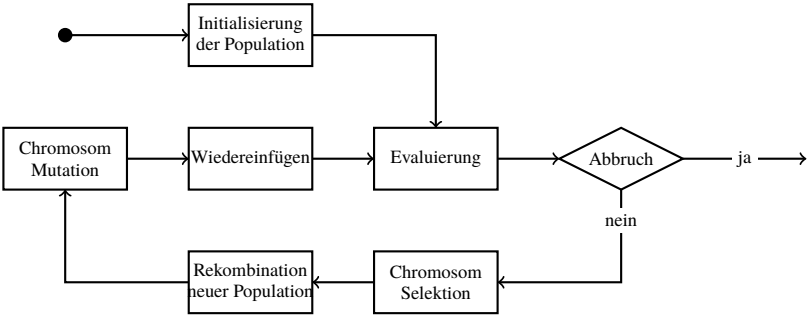



Abbildung 3.3: Ablauf eines evolutionären Optimierungskreislaufs nach [120]

Anschließend werden die beobachteten Ausgaben evaluiert und anhand eines Abbruchkriteriums (Zielfunktion) entschieden, ob ein weiterer evolutionärer Schritt durchlaufen wird. Abbruchkriterien sind neben funktionalen Kriterien (Stagnation des Bewertungskriteriums oder Erreichen eines Ziels), das Überschreiten einer Zeitspanne oder maximalen Anzahl an Durchläufen. Während dem evolutionären Vorgehen folgen die Schritte Selektion, Rekombination und Mutation, bevor die Stimuli wieder eingefügt und erneut bewertet werden [84]. Die Selektion ist der Prozessschritt, der bestimmte Individuen einer Population auswählt, die für die Produktion neuer Chromosomen der Nachfolgegeneration genutzt werden. Die Rekombination erstellt anschließend aus den Chromosomen eine neue Population, wobei aus der Kreuzung verschiedener Chromosomen eine neue Kombination an Stimuli entsteht [84]. Durch die Mutation werden zufällige Veränderungen der Individuen hervorgerufen, um die Diversität zu erhalten und einer vorzeitigen Konvergenz beziehungs-

weise der Einnahme lokaler Optima entgegenzuwirken [84]. Im Gegensatz zur Rekombination findet die Mutationen nur innerhalb eines Individuums statt. Im letzten Schritt werden die Individuen wieder in den Prozess eingefügt, wobei die Individuen anhand einer sogenannten Fitnessfunktion bewertet werden [84]. Die Fitness-Funktion bewertet dabei, wie gut die Stimuli für die Lösung des Problems geeignet sind [120].

Das evolutionäres Vorgehen wurde für den funktionalen Test eines Parkassistenten [13, 119] und eines Bremskraftverstärkers [12] bereits erfolgreich in der Automobilindustrie eingesetzt. Beim Parkassistent wurde untersucht, ob beim Einparkvorgang eine Kollision entstehen kann [13, 119]. Hierfür wurde als Population die Position (Parkplatzkoordinaten und Startwinkel des Fahrzeug) und die Größe (Länge und Breite) eines Parkplatzes variiert. Die Parkplatzkoordinaten, Startwinkel des Fahrzeug, Länge und Breite dienen dabei als Chromosomen der Population, die während dem evolutionären Vorgehen rekombiniert und mutiert wurden. Die veränderten Parameter wurden genutzt, um die Steuersoftware zu stimulieren und den Parkvorgang durchzuführen. Die Zielfunktion minimiert anschließend die Distanz zwischen dem Parkbereich und einer möglichen Kollision. Wird während der Evaluierung eine Kollision erkannt, ist eine funktionale Anforderung verletzt und der Test negativ beendet.

Ein ähnliches Vorgehen ist das iterativen Verfahrens zum Test des Bremskraftverstärkers. Mittels evolutionären Algorithmen wurden Fahrmanöver gesucht, um in unkritischer Situation den Bremsdruck zu verstärken oder in kritischen Situationen einen verminderten Bremsdruck anzulegen [12].

## 3.3 Datensicherheit von vernetzten eingebetteten Systemen in der Automobilindustrie

Um die Datensicherheit im Automobil bewerten zu können, muss zuerst definiert werden, welches Gut (engl. Asset) geschützt werden muss. Die Datensicherheit in Automobilen wird hierzu in vier Kategorien eingeordnet [67]:

1. Datenschutz: Der Datenschutz behandelt ungewollte oder unbefugte Erfassung von Daten, die Folgendes betreffen:

- Fahrzeug- oder Fahrertätigkeiten
- Fahrzeug- oder Fahreridentifikation
- Fahrzeug- oder Subsystem-Design und Implementierungen (z. B. Intellectual Property (IP) von Herstellern und Zulieferern)

2. Nicht autorisierter Zugriff: Diese Kategorie bezeichnet funktionale Zugriffe oder ungewollte Transaktionen bei Bezahlsystemen ohne Autorisierung des Anwenders.
3. Betriebsfähigkeit: Die Betriebsfähigkeit wird durch Manipulation von
   - nicht sicherheitsrelevanten Fahrzeugsystemen wie Infotainment, Heizung, Lüftung und Klimatisierung oder
   - Car2X-Kommunikation, die nicht sicherheitsrelevante Auswirkungen auf das Betriebsverhalten des Fahrzeugs oder des Intelligent Transportation System (ITS) hat

   beeinträchtigt.
4. Funktionale Sicherheit: Interferenzen mit dem Bordsystemen des Fahrzeugs oder Car2X-Kommunikation sorgen für eine Beeinflussung der funktionalen Sicherheit, die den sicheren Betrieb des Fahrzeugs oder der ITS beeinträchtigt.

Besonders der Schutz vor Eingriffen in die funktionale Sicherheit wird im folgenden berücksichtigt, da hierdurch eine Gefährdung von Personen möglich ist. Zusätzlich wird die Betriebsfähigkeit ohne direkte sicherheitsrelevante Auswirkungen betrachtet, da indirekte Bedrohungen durch Ablenkung des Fahrers (Lautstärke des Radios, Scheibenwischer, Beleuchtung, etc.) entstehen können (siehe Kapitel 1.1.3).

Um Manipulationen und Angriffe im Betrieb des Fahrzeugs zu erkennen, wird auf sogenannte Intrusion Detection System (IDS) gesetzt, ein solches System zur Erkennung von Anomalien auf Bussystemen beschreibt [117]. Das IDS erkennt fehlerhafte Kommunikation und verhindert das fehlerhafte Verhalten. Zudem meldet es dies an den Fahrer oder den Fehlerspeicher. Neben der Erkennung von Fehlern zur Laufzeit wird auf eine redundante Ausführung und Kommunikation gesetzt. Systeme wie [BBB+17] nutzen unabhängige Ressourcen und Übertragungskanäle, um das erhaltene Ergebnis zu plausibilisieren, wodurch Manipulationen in umgebenden Fahrzeugen erkannt werden. Im weiteren Verlauf wird der Fokus auf die Vermeidung von Angriffen auf die Betriebsfähigkeit (Punkt 3) und funktionale Sicherheit (Punkt 4) während dem Testprozess ge-

legt, nicht jedoch auf die Überwachung des bestehenden Systems während der Ausführung im Feld.

Während der Entwicklung von Steuergeräten wird durch sogenanntes "Security-by-Design" durch Analysen der Anforderungen und Gefährdungen ein entsprechendes Konzept für die Datensicherheit im Steuergerät erstellt [64] (siehe auch Kapitel 3.5.1 und Kapitel 3.5.2). Hierzu werden als sicher geltende Software-Pattern, Best Practices und Denkweisen eingesetzt [20].

Der anschließende Test der Datensicherheit ist in Standards wie SAE J3061 [89] zur Verifikation vorgeschrieben (siehe Kapitel 3.5). Diese Tests werden erst seit einiger Zeit entwickelt und konzentrieren sich auf funktionale Absicherung, Fuzz-Testing und Penetrationstests (siehe Kapitel 3.5.4).

Während der Entwicklung von Steuergeräten wird durch sogenanntes "Security-by-Design" durch Analysen der Anforderungen und Gefährdungen ein entsprechendes Konzept für die Datensicherheit im Steuergerät erstellt.

## 3.4 Angriffe auf Kraftfahrzeuge

Durch die Vernetzung der Systeme im Automobil wird es möglich Kommunikation oder Systeme zu manipulieren und damit das Verhalten des Fahrzeugs zu beeinflussen (siehe Kapitel 2.1.3). Besonders die Beeinflussung der Betriebsfähigkeit (Kapitel 3.3 Punkt 3) und der funktionalen Sicherheit (Kapitel 3.3 Punkt 4) des Fahrzeugs sind für Angriffe relevant. Datenschutz (Kapitel 3.3 Punkt 1) sowie nicht autorisierte oder ungewollte Transaktionen bei Bezahlsystemen (Kapitel 3.3 Punkt 2) sind zwar ebenfalls Schutzziele, die nicht vernachlässigt werden dürfen, eine Gefahr für die Passagiere besteht hierbei jedoch nicht. Daher wird im folgenden der Fokus auf Manipulationen der Betriebsfähigkeit und der funktionalen Sicherheit gelegt.

Erste Angriffe auf Fahrzeuge im akademischen Umfeld erfolgten bereits 2010 durch Verbindung eines manipulierten Diagnosegeräts (Angriff 1). Bei Fahrzeugen von Ford und Toyota wurde damit Zugriff auf Bremsen und Motor ermöglicht [54]. Der Angriff wurde später detailreich veröffentlicht, sodass dieser nachgebaut werden kann [109]. Da dieser Angriff von den Herstellern als irrelevant eingestuft wurde, da ein Angreifer physikalischen Zugriff auf das Fahrzeug haben muss, um einen Stecker in der OBD anzubringen, wurde im

Jahr 2011 der Angriff auf die Infotainment Einheit des Fahrzeugs erweitert (Angriff 2). Bei diesem Angriff wurde ein Fehlverhalten durch das Abspielen eines Musiktitels im Infotainment erzwungen, wodurch das Fahrzeug über eine Funkverbindung gesteuert werden konnte [19].

Untersuchungen über mehrere Fahrzeuge zeigten 2014, dass die oben genannten Probleme nicht nur bei Ford und Toyota auftreten können. Zu den darin untersuchten Fahrzeugen zählten neben Ford und Toyota unter anderen auch Audi, BMW, Dodge, Infiniti, Jeep und Range Rover [69]. Durch diese Analysen erfolgte im Jahr 2015 der erste Angriff auf ein Fahrzeug über die Telematik-Schnittstelle, die das Fahrzeug über Mobilfunk und Wi-Fi an die Umwelt anbindet (Angriff 3). Hierbei wurde ein Fehler im Uconnect Telematik-System von Jeep ausgenutzt, um die Firmware über das Mobilfunknetz zu verändern [70]. Durch diese Veränderung konnte das Fahrzeug über das Mobilfunknetz ferngesteuert werden. Dies beinhaltete die Steuerung von direkten Bedrohungen auf die Fahrsicherheit (Bremsen, Lenken, abschalten des Motors, etc.) und indirekte Bedrohungen durch Ablenkung des Fahrers (Lautstärke des Radios, Scheibenwischer, Beleuchtung, etc.).

Nach dem oben genannten Jeep-Angriff, wurden zunehmend Fahrzeuge untersucht und Schwachstellen identifiziert. Hierzu zählen bei BMW Fehler beim Abspielen von Musikliedern [143], die das Ausschalten des Infotainment bewirken und fehlerhafte Algorithmen bei der Identifikation über Mobilfunk [81], die das zurücksetzen des Passworts erlauben (Angriff 4, Angriff 5 und Angriff 6).

Für Angriff 5 wurde ein Fehler in der Telematikeinheit ausgenutzt, um Administrationsrechte auf dem Telematik-Steuergerät zu erhalten und damit die Sicherheitsmechanismen bei der Installation von Firmware zu umgehen. Dadurch konnte eine manipulierte Software auf dem Steuergerät installiert und auf weitere Steuergeräte verbreitet werden. Neben Tesla sind auch weitere Hersteller von Angriffen betroffen. Bei Infotainment-Systemen von VW und Audi können über Mobilfunk das Mikrofon, das Navigationssystemen und die Lautsprecher ferngesteuert werden (Angriff 7) [107].

Durch fehlerhafte Implementierungen wurde 2018 die verschlüsselte Funkfernbedienung eines Tesla kopiert (Angriff 8), wodurch der Diebstahl dieser Fahrzeuge ermöglicht wird [127].

Zusätzlich zu herstellerspezifischen Fehlern entstehen Probleme der Datensicherheit in den Systemen der Zulieferer (Angriff 9). Durch fehlerhafte Rechteverwaltung in einem System von Continental konnte auf dem Telematik-Steuergerät von Herstellern wie BMW, Ford, Infiniti und Nissan schadhafter Code eingespielt und ausgeführt werden, wodurch Zugriff auf den gesamten Speicher ermöglicht wurde [144, 145]. Laut Herstellerangaben kann durch diese Schwachstelle das gesamte System beeinflusst werden. Da die Schwachstelle derzeit jedoch noch nicht behoben ist, sind keine genauen Auswirkungen bekannt.

Im Jahr 2018 wurden bei BMW 14 weitere Schwachstellen entdeckt [48] und 2019 veröffentlicht, die remote Zugriff über Mobilfunk erlauben und teilweise Zugriff auf die Steuerung von Motor und Bremsen erlauben (Angriff 10).

Tabelle 3.1: Auswahl an Angriffen auf Fahrzeuge zwischen 2010 und 2019

| ID | Jahr | Hersteller | | Zugriff | Schnittstelle |
|---|---|---|---|---|---|
| 1 | 2010 | Ford, Toyota | [54, 109] | lokal | Onboard Diagnose |
| 2 | 2011 | Ford | [19] | remote | Infotainment |
| 3 | 2015 | Jeep | [70] | remote | Telematik |
| 4 | 2016 | BMW | [81] | remote | Telematik |
| 5 | 2016 | Tesla | [78] | remote | Information Display |
| 6 | 2017 | BMW | [143] | remote | Multimedia |
| 7 | 2017 | VW, Audi | [107] | remote | Infotainment |
| 8 | 2018 | Tesla | [127] | remote | Funkschlüssel |
| 9 | 2018 | Continental | [144, 145] | remote | Telematik |
| 10 | 2019 | BMW | [16, 48] | remote | Telematik |

Eine Auflistung über die oben aufgeführten Angriffe zwischen 2010 und 2019 ist in Tabelle 3.1 aufgelistet. Für eine vollständige Auflistung aller Angriffe wird auf die Literatur verwiesen. In der Auflistung wurden besonders die remote Angriffe herausgestellt, da diese ohne physikalischen Zugriff auf das Fahrzeug durchgeführt werden können.

Die aufgelisteten Angriffe werden in Kapitel 4.3 bewertet und die Ursachen untersucht. Anschließend wird hinterfragt, wie diese Ursachen bereits wäh-

rend der Entwicklung durch Testen entdeckt werden können. Detailtiere Auflistungen und Untersuchungen zur Reproduzierbarkeit aller Angriffe liegen derzeit noch nicht vor, um den Fahrzeugherstellern Zeit zum Nachbessern der Schwachstellen zu geben.

Neben direkten Angriffen auf die Fahrzeuge kann das Verhalten von automatisierten Fahrzeugen ebenfalls über die Kommunikation mit der Umwelt beeinflusst werden. So können Manipulation des globalen Navigationssatellitensystems (engl. Global Navigation Satellite System (GNSS) für verschiedenen Angriffen ausgenutzt werden. Darunter sind ein erzwungener Spurwechsel, das Verlassen der Straße und das Anhalten des Fahrzeugs [75]. Im weiteren Verlauf werden nur Manipulationen innerhalb der Steuergerätesoftware berücksichtigt, nicht jedoch die Verfälschung von Positions- oder Kommunikationsdaten. Da die letztgenannten Angriffe durch Signaturen oder Plausibilisieren in Form von Sensorfusion gelöst werden müssen und nicht durch Sicherheitslücken in der Software entstehen, sind diese nicht in Tabelle 3.1 aufgelistet und werden im folgenden nur am Rande adressiert.

## 3.5 Datensicherheit im V-Modell

In der PC-Branche sind Standards zum Vorgehen für eine Implementierung von sicheren Funktionen bereits weit verbreitet, insbesondere die Standards ISO/IEC 27001 [45] und ETSI TS 102 165 [28, 29]. In der Automobilbranche werden diese Standards für Steuergeräte häufig nicht eingesetzt. Hierfür sind besonders drei Gründe zu nennen. Zum einen steht die Performance (Leistung der Prozessoren und Speicher) vorgeschlagenen Implementierungen in der Automotive Domain meist nicht zur Verfügung, da kostensparende Lösungen bevorzugt werden. Zweitens müssen für die funktionale Sicherheit Echtzeitanforderungen eingehalten werden, die mit den vorgeschlagenen Security-Maßnahmen nicht vereinbar sind. Und drittens sind die Funktionen im Fahrzeug sehr stark vernetzt, wodurch viele Wechselwirkungen zwischen Funktionen entstehen.

Um diesen neuen Anforderungen gerecht zu werden, wurde für die Automobilbranche im Jahr 2016 das Security-Guidebook J3061 [89] der Society of Automotive Engineers (SAE) veröffentlicht. Hier sind Richtlinien für die Automobilindustrie enthalten um Entwickler in der Designphase zu unterstützen.

Um die Security in der Automobilindustrie zu entwickeln, führt der Standard eine abgewandelte Form des V-Modells ein (siehe Abbildung 3.4). Das Vorgehen im Cyber-Security V-Modell entspricht dem des klassischen V-Modells (siehe Kapitel 2.6.3) mit der Ergänzung um Security-Anforderungen und Security Tests. In Abbildung 3.4 ist dieser Unterschied dargestellt, indem die spezifischen Änderungen farblich herausgestellt sind. Zusätzlich zu den Schritten der Datensicherheit innerhalb des V-Modells, werden weiterhin die Schritte der funktionalen Sicherheit parallel ausgeführt. Ein ähnliches Modell wird vom europäischen Standard ISO 21434 [46] Ende 2019 erwartet, der zum aktuellen Zeitpunkt jedoch noch nicht verabschiedet oder veröffentlicht ist.

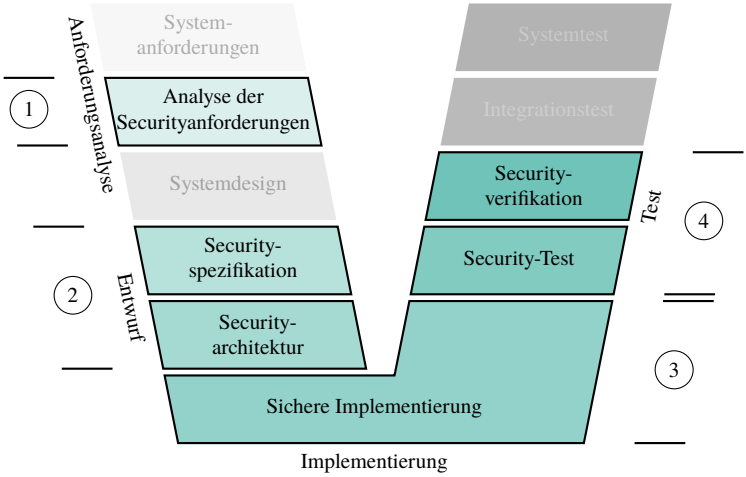



Abbildung 3.4: Einbindung der Datensicherheit in das V-Modell [89]

Das Vorgehen im Standard kann in vier Bereich unterteilt werden (siehe Abbildung 3.4), die ähnlich zum klassischen V-Modell (siehe Kapitel 2.6.3) sind:

1. Analysen der Securityanforderungen (siehe Kapitel 3.5.1)
2. Entwurf des Systems für Datensicherheit (siehe Kapitel 3.5.2)
3. Sichere Implementierung (siehe Kapitel 3.5.3)
4. Test der Datensicherheit (siehe Kapitel 3.5.4)

### 3.5.1 Analysen der Securityanforderungen

Analog zur funktionalen Sicherheit existieren im linken Ast des V-Modells Analysen der Security-Anforderungen. Diese Analysen dienen zum Finden von Designfehlern und der Erzeugung von Anforderungen für die Datensicherheit. Durch die sogenannte Angriffsbaumanalyse werden Systeme systematisch nach Schwachstellen aufgespalten und anschließend in einer Bedrohungsanalyse nach Risiken bewertet [89].

#### Angriffsbaumanalyse

Die Angriffsbaumanalyse (engl. Attack Tree Analysis (ATA)) ist eine Analysemethode, um die Bedrohung auf höchster Ebene zu identifizieren und potentiellen Angriffspfade zu bestimmen [89]. Diese Methode ist analog zur in der funktionalen Sicherheit bekannten Fehlerbaumanalyse (engl. Fault Tree Analysis (FTA)).

Obwohl die Methoden vergleichbar zueinander sind, werden sie unterschiedlich eingesetzt. In der Fehlerbaumanalyse identifiziert der Analytiker potenzielle Ursachen für die Gefährdung und sucht nach zufälligen Hardwarefehlern. Bei der Angriffsbaumanalyse geht es nicht um Hardwareausfälle mit einem oder mehreren Fehlern, sondern um die Ermittlung potenzieller Pfade, die ein Angreifer durch das System nehmen kann, um zur Bedrohung zu werden [89].

Basierend auf den Erkenntnissen aus dem Angriffsbaum werden in der Bedrohungsanalyse und Risikobewertung die Dringlichkeit für das Behandeln der erkannten Schwachstellen identifiziert. Während der Implementierung werden statischen Codeanalysen (siehe Kapitel 3.5.4) verwendet, um die erkannten potenziellen Schwachstellen im Code zu erkennen. Gültiger oder korrekter Code kann jedoch immer noch Schwachstellen aufweisen, wodurch weitere Testverfahren nötig sind [89].

#### Bedrohungsanalyse und Risikobewertung

Die Bedrohungsanalyse und Risikobewertung (engl. Threat and Risk Analysis (TARA)) wird verwendet, um die identifizierten potenziellen Bedrohungen

für das System zu bewerten und das mit jeder Bedrohung verbundene Risiko zu bestimmen [89]. Die Ergebnisse der TARA definieren die Datensicherheitsanforderungsstufe (engl. Automotive Cybersecurity Integrity Level (ACsIL)) und helfen die zukünftigen Analysetätigkeiten auf die größten Bedrohungen zu konzentrieren. Dadurch wird es ermöglicht, eine Vielzahl an Ressourcen für die größten potenziellen Bedrohungen einzusetzen.

Die Bedrohungsanalyse und Risikobewertung besteht dabei aus drei Komponenten [89]:

1. Bedrohungsanalyse (Threat Identification): Identifizierung der potenziellen Bedrohungen für ein System oder eine Organisation (Stakeholder), wobei die Angriffspfade durch die ATA vorgegeben sind.
2. Risikobewertung (Threat Classification): Bewertung und Klassifizierung der identifizierten Bedrohung zur Festlegung des verbundenen Risikos. Hierbei geht man vom schlimmsten anzunehmenden Szenario aus. Die Risikobewertung basiert dabei auf der Schwere des Vorfalls (funktionale Sicherheit, finanzieller Schaden und Datenschutz), dem Angriffspotential und einer geschätzten Angriffswahrscheinlichkeit.
3. Risikoanalyse: die Bedrohungen werden nach Risikograd eingestuft und festgestellt, ob das mit einer bestimmten Bedrohung verbundene Risiko auf einem akzeptablen Niveau liegt oder ob Maßnahmen zur Risikominderung erforderlich sind.

Die Risikobewertungskomponente einer TARA berücksichtigt die Schwere der möglichen Folgen eines möglichen Angriffs auf das System und die Wahrscheinlichkeit, dass ein möglicher Angriff erfolgreich durchgeführt werden kann. Die Wahrscheinlichkeit, dass ein möglicher Angriff erfolgreich durchgeführt werden kann, wird als “Angriffspotenzial“ bezeichnet. Das Angriffspotenzial berücksichtigt eine Reihe verschiedener Faktoren, einschließlich der verstrichenen Zeit (Zeit, um eine Schwachstelle zu identifizieren, einen Angriff zu entwickeln und einen Angriff erfolgreich durchzuführen), Fachkenntnisse, Kenntnisse des untersuchten Systems, Niveau der Cyber-Security-Maßnahmen, Zeitfenster (Zugang zum System) und erforderliche Ausrüstung [89]. Die Durchführung einer TARA wurde in Projekten wie EVITA [136] erarbeitet.

Ziel einer Bedrohungsanalyse und Risikobewertung ist es, potenzielle Bedrohungen für das System zu identifizieren, das potenziellen Risiko zu bewerten

und die Bedrohungen zu anhand vier Klassen einzugruppieren und dadurch das weitere Vorgehen zu bestimmen [89].

1. Risikominderung: Minderung des Risikoniveaus durch Einführung, Entfernung oder Änderung von Sicherheitskontrollen, damit das Restrisiko als akzeptabel bewertet werden kann.
2. Risikovermeidung: Vermeidung der Tätigkeit oder Bedingung, die zum Risiko führt.
3. Risikoakzeptanz: Entscheidung zur Aufrechterhaltung des Risikos ohne weitere Maßnahmen.
4. Risikotransfer: Risiko wird mit einer anderen Partei geteilt, die das jeweilige Risiko je nach Risikobewertung am effektivsten bearbeiten kann.

Basierend aus der Einordnung der Datensicherheitsanforderungsstufe in die vier Klassen, wird die Datensicherheitsspezifikation erstellt und das System entworfen.

### 3.5.2 Entwurf des Systems für Datensicherheit

Anhand der Datensicherheitsziele aus der TARA wird das System entworfen. Dies erfolgt analog zum Systementwurf der funktionalen Sicherheit. Jedoch wird bei der Datensicherheit besonders auf die Schnittstellen und Sicherheitsmodule geachtet. Bei Fahrzeugen umfasst dieser Schutz einen hardwarebasierten Schutz, softwarebasierte Abwehrsysteme in der jeweiligen ECU und eine globale Netzwerküberwachung. Die Grundsätze der Datensicherheit müssen unter Wahrung der funktionalen Sicherheit des Automobils gewahrt werden. Hierfür wird beim Entwurf bereits auf ein Security-by-Design geachtet, wodurch nur als sicher geltende Funktionen (Firewalls, Intrusion Detection System (IDS), kryptografische Funktionen, etc.) in die Gesamtarchitektur eingebracht werden. Als sicher geltende Funktionen sind über Software-Pattern und Best Practices definiert [20]. Zusätzlich werden in diesem Schritt Maßnahmen zur Manipulationssicherheit und zur Überprüfung der Integrität und Authentizität (siehe Kapitel 2.4) spezifiziert [64].

### 3.5.3 Sichere Implementierung

Der untere Bereich des V-Modells beschäftigt sich mit der Implementierung. Um eine sichere Implementierung zu gewährleisten, werden Coding-Richtlinien als Vorgaben zur Programmierung eingesetzt und die Umsetzung im Quellcode durch statische Code-Analysen überprüft. Für die Implementierung empfiehlt sich zunächst die Vorgabe von Coding-Richtlinien.

#### Coding-Richtlinien

Durch Einhaltung von Regeln bei der Implementierung werden Security-Schwachstellen bereits in der Implementierungsphase vermieden. Die Coding-Richtlinien beinhalten dabei Vorgaben für den Umgang mit verschiedenen Datentypen, Bedingungen und Schleifen. Durch diese Richtlinien werden zwar Hilfestellungen zum sicheren Implementieren gegeben, die Richtlinien können aber nicht sicherstellen, dass eine Software ohne Schwachstellen implementiert wird. Eine Großzahl der sicherheitskritischen Fehler entstehen trotz Coding-Richtlinien durch eine schlechte Implementierung des Programmierers (siehe Kapitel 3.4).

Für die funktionale Sicherheit sind bereits einige Coding-Richtlinien in der Automotive Domain vorhanden. Diese leiten sich hauptsächlich aus Standards für die funktionale Sicherheit wie IEC 61508 [42], IEC 61511 [43] und ISO 26262 [44] ab. Die Standards sind dabei nicht nur im Automotive-Bereich, sondern auch für den Einsatz in Regelungssystemen, Maschinen und Anlagen etabliert.

Als klassischer Vertreter von Coding-Richtlinien in der Automobilindustrie gelten regeln der Motor Industry Software Reliability Association (MISRA) [71]. MISRA ist eine Richtlinie speziell für die Entwicklung von Software in der Automobilindustrie. Diese ist hauptsächlich für die funktionale Sicherheit im Fahrzeug ausgelegt, kann aber auch für die Datensicherheit verwendet werden. Die Richtlinie beschreibt hierbei eine eigene Programmiersprache, in der sicherheitsgefährdende Aspekte (wie beispielsweise Zeiger) verboten sind, damit diese bei der Entwicklung von Software keine Schwachstellen darstellen. Durch die Einschränkung der erlaubten Konstrukte müssen Funktionen jedoch teilweise komplexer geschrieben werden, wodurch neue Fehlerquellen entstehen können. Da die Datensicherheit in den MISRA-Richtlinien jedoch nicht

betrachtet wird, müssen sie um Vorgaben des IT-Bereichs, wie die aus dem Computer Emergency Response Team (CERT) [95], ergänzt werden.

Das CERT ist Teil des Software Engineering Institute der Carnegie Mellon University. Durch diese Gemeinschaft wurden spezielle Richtlinien zur sicheren Programmierung von verschiedenen Programmiersprachen herausgegeben. In den veröffentlichten Richtlinien werden für verschiedene Kategorien Regeln und Vorschläge zur sichereren Implementierung vorgestellt [95]. In vielen Fällen sind Implementierungen des CERT jedoch in der Automobildomäne nicht einsetzbar. Besonders Performance und Speicherbegrenzungen spielen hierbei eine große Rolle, wodurch ein größeres Spektrum an Security- und Verschlüsselungsimplementierungen zum Einsatz kommt als dies in der IT-Branche der Fall ist [4].

### 3.5.4 Test der Datensicherheit

Nach der sicheren Implementierung muss die Datensicherheit der implementierten Algorithmen und Funktionen getestet werden, um die fehlerfreie Implementierung zu prüfen. Analog zum Testen der funktionalen Sicherheit (siehe Kapitel 2.6.3) beschreibt der rechte Ast des V-Modells für die Datensicherheit ebenfalls die Verifizierung des Produkts durch entsprechende Tests.

Beim Testen von Software wird bei den Testtechniken zwischen statischen und dynamischen Analysen unterschieden [59].

Statische Analysen werden zum Prüfen von Anforderungen oder Quelltext eingesetzt, ohne dass das Artefakt ausgeführt wird. Bei den zu testenden Artefakten handelt es sich um Dokumente oder Quellcode. Da der Quellcode bei den Analysen nicht ausgeführt werden muss, können statische Tests bereits in frühen Entwicklungsphasen eingesetzt werden. Der gesamte Quellcode wird dabei auf Programmierkonventionen (siehe Kapitel 3.5.3) überprüft [152]. Für eine Interaktion zwischen verschiedenen Systemen sind statische Testtechniken ungeeignet, da zum Testzeitpunkt nur Schnittstellen anderer Systeme bekannt sind. Zudem entdecken statische Methoden keine Fehlerzustände, die erst zur Laufzeit entstehen. Eine Zuweisung einer Null als Divisor kann durch eine Analyse des Quelltextes nicht erkannt werden [152].

Dynamische Analysen führen das Artefakt (z. B. Applikation) zur Auffindung von Fehlern auf einer Plattform aus. Durch die Ausführung des Testobjekts, kommt das Testverhalten dem Verhalten im fertigen Produkt näher. Zusätzlich kann die Interaktion verschiedener Systemteile getestet werden. Abweichungen können jedoch durch unterschiedliche Ausführungsplattformen entstehen [152]. Nachteil beim Analysieren mit dynamischen Testtechniken ist, dass ein ausführbares Programm und eine entsprechende Testumgebung benötigt wird. Die Erstellung der Testumgebung ist für verknüpfte Teilsysteme mit zusätzlichem Aufwand verbunden, da alle Teilsysteme hinreichend genau abgebildet sein müssen. Wird ein Fehler beim dynamischen Testen entdeckt, müssen die internen Fehlerzustände, die den Fehler ausgelöst haben, in einem separaten Arbeitsschritt (Debugging, siehe Definition 27) untersucht werden.

Zusätzlich zur Aufteilung der Testtechnik, wird das Testziel zwischen funktionalen und logischen Analysen unterschieden [11].

Die funktionalen Analysen haben dabei das Ziel, die funktionale Architektur des Systems zu definieren und das Funktionsverhalten zu charakterisieren. Das funktionale Verhalten, wie Funktionen und Operationen, werden dabei den Systemelementen der physikalischen Architektur zugeordnet. Dabei ist zu beachten, dass die funktionale Analyse als “Black-Box“ getestet wird, bei der das Systemverhalten nur durch die Ein- und Ausgabe-Funktionen definiert ist, jedoch ohne Bezugnahme auf anderen Artefakte.

Die logische Analyse verbindet die funktionale Analyse mit der Modellierung und Simulation, um das dynamische Verhalten des Systems vorherzusagen. Die logische Architektur des Systems wird in der Design Phase erstellt (siehe Kapitel 2.6.3). Anschließend können Modelle mit der logischen Struktur zum Testen aufgebaut werden. Die Logische Analyse ist daher eine “White Box“, da das System aus einzelnen Elementen besteht, die in ihren Teilen zugänglich sind [11].

Bezogen auf den Test der Datensicherheit eines Steuergeräts lassen sich die logischen Analysen durch statische Tests während den frühen Entwicklungsphasen abdecken, da zu späteren Zeitpunkten eine “White Box“ nicht mehr vorhanden ist und nicht benötigte Zugriffe für das Serienprodukt geschlossen werden. Die funktionalen Analysen werden durchgeführt sobald die Module oder Systeme vollständig sind. Um das genaue Verhalten bei unterschiedlichen Eingaben zu beobachten werden hierfür dynamische Test eingesetzt.

### Statische Security Tests

Bei den statischen Tests (oder auch Schwachstellenscanner) wird der Quellcode automatisiert anhand von formalen Kriterien untersucht, um Flüchtigkeitsfehler zu identifizieren und die Einhaltung von Konventionen und Schnittstellen zu prüfen [59].

Hierfür werden die zu verwendenden Coding-Richtlinien herangezogen und um typische Programmierfehler (use after free, double free, etc.) ergänzt. Um Probleme in der Programmstruktur zu erkennen, wird die Applikation über Graphen rekonstruiert und die Abhängigkeiten beurteilt. Durch die Abbildung auf solche Graphen ist zudem die Berechnung der Ausführungszeit mittels statischer Methoden möglich [23]. Bei diesen Analysen muss man zwischen High-Level Designanalysen und Detailanalysen unterscheiden. In Designanalysen werden hauptsächlich Protokolle, Schnittstellen und Spezifikationen analysiert, um systematische Angriffsvektoren wie schlechte Verschlüsselungsverfahren oder zu kurze Schlüssel zu finden. Die Erkenntnisse aus diesen Analysen werden zurück in die Anforderungsanalyse gegeben und mit einer TARA bewertet. Während bei Designanalysen nur eine theoretische Beschreibung des Systems vorliegen muss, benötigt man für die Detailanalysen explizites Wissen über die Umsetzung der Algorithmen. Ein weiterer Punkt der statischen Verfahren ist die Überprüfung der Zugriffe auf Variablen und Funktionen. Hierfür wird während der Untersuchung des Kontrollflusses in der Applikation protokolliert, welche Funktionen lesend oder schreibend auf Variablen zugreift [59]. Dadurch können Abhängigkeiten und Beeinflussungen zwischen Funktionen und Variablen erkannt werden.

Durch eine statische Code-Analyse können Implementierungsfehler zwar identifiziert werden, funktionale Fehler oder Fehler im Design werden von der statischen Analyse jedoch nicht gefunden und benötigen eine Überprüfung durch den Menschen [59]. Statische Analysen können keine Implementierungsfehler durch Fahrlässigkeit, falsche Interpretation der Spezifikation oder Fehler aus Drittanbieter-Software entdecken. Hierfür müssen explizite Tests der Funktionen auf das Gesamtsystems durchgeführt werden [61]. Zudem werden diese Analysen als unzuverlässig angesehen und der Einsatz von expliziten Code-Reviews bei sensitiven Funktionen empfohlen [51]. Für die statische Code-Analyse ist keine Ausführung der zu prüfenden Applikation nötig. Dafür ist

für die statische Code-Analyse der Quellcode in Form einer “White Box“ erforderlich.

#### Dynamische Security Tests

Neben den oben aufgeführten statischen Analysen existieren in der IT-Branche auch eine Reihe an dynamischen Security-Tests. Diese Tests können hierbei im wesentlichen zwischen funktionalen Tests, dem Überprüfen der Robustheit offener Schnittstellen (z. B. Fuzzing, siehe Kapitel 3.5.4) und Penetrationstests unterschieden werden.

Alle dynamischen Tests haben gemeinsam, dass ein Zugriff auf die fertige Soft- und Hardware, sowie teilweise die Spezifikation vorhanden sein muss. Für einen entsprechenden Zugriff, müssen zudem Debug-Schnittstellen eingefügt und die Soft-/Hardware entsprechend erweitert werden. Beides ist beim Testen einer “Block Box“ jedoch nicht möglich, weshalb mit den vorhandenen Schnittstellen ausgekommen werden muss. Zusätzlich ist es meist nötig, die Soft-/Hardware in der realen Umgebung zu betreiben oder die Schnittstellen entsprechend zu simulieren [4], um das gewünschte funktionale Verhalten der Umwelt zu erzielen.

Das dynamische Testen dient zur funktionalen Sicherstellung der korrekten Ausführung von Algorithmen. Für die Ausführung von funktionalen Test, wird an dieser Stelle auf die Literatur verwiesen [101]. Eine sorgfältige Ausführung dieser Tests kann Implementierungsfehler und daraus resultierenden Datensicherheitsschwachstellen bereits frühzeitig entdecken. Um Fehlerfreiheit der Tests sicherzustellen wird mit offiziellen Testparametern gearbeitet, die auch typische Grenzfälle der Algorithmen abdecken.

Automatische Sicherheitsprüfungen gewährleisten dabei die generelle Einhaltung von Spezifikationen und Standards der implementierten Sicherheitsfunktionalität, beispielsweise der Verschlüsselungsalgorithmen oder der Authentifizierungsprotokolle.

Bei funktionalen Tests werden die Algorithmen jedoch nicht nur auf das richtige Verhalten entsprechend der Spezifikation getestet, sondern auch die Robustheit (Definition 32) dieser geprüft. Darüber hinaus wird die Performance von (oft rechenintensiven) Sicherheitsalgorithmen getestet, um potenzielle Flaschen-

hälse zu identifizieren, welche die gesamte Sicherheitsleistung beeinträchtigen könnten [4].

Neben den funktionalen Tests der Sicherheitsfunktionalität (Verschlüsselungsalgorithmen, etc.) steht bei der Datensicherheit besonders die Anfälligkeit auf falsche Eingaben im Vordergrund. Hierfür werden im Stand der Technik zwei Verfahren (Fuzzing und Penetrationstests) eingesetzt. Beide Testverfahren greifen auf die Methoden (HiL, SiL, etc.) der funktionalen Tests zurück.

**Codeabdeckung**
In der Softwaretechnik ist die Codeabdeckung (engl. Code-Coverage) ein Maß dafür, wie viel Teile des Quellcode während den Tests ausgeführt wurden. Eine hohe Testabdeckung bedeutet dabei, dass der Quellcode während des Tests mehrfach durchlaufen wurde, wodurch die Wahrscheinlichkeit auf unentdeckte Softwarefehler verringert wird [7].

Für sicherheitskritische Anwendungen ist eine Codeabdeckung oft erforderlich. So schreibt die Norm ISO 26262 für Systeme der Stufe ASIL D im Automobil eine vollständige Codeabdeckung beim Testen vor. Das heißt alle Teile des Quellcodes müssen mindestens einmal durchlaufen werden.

Durch den Einsatz der Codeabdeckung, kann folgendes überprüft werden [76]:

- Testabdeckung: Wurde jede Funktion oder jedes Unterprogramm im Programm aufgerufen?
- Anweisungsabdeckung: Wurde jede Anweisung im Programm ausgeführt?
- Zweigabdeckung: Wurde jeder Zweig jeder Kontrollstruktur (z.B. in if- und case-Anweisungen) ausgeführt?
- Entscheidungsabdeckung (oder Prädikatabdeckung): Wurde jeder boolesche Ausdruck sowohl mit wahr als auch falsch bewertet?
- Pfadabdeckung: Wurde jede mögliche Route durch einen bestimmten Teil des Codes ausgeführt?
- Ein- und Ausgangsabdeckung: Wurde jeder mögliche Aufruf und Rücksprung der Funktion ausgeführt?
- Schleifenabdeckung: Wurde jede mögliche Schleife nullmal, einmal und mehrmals ausgeführt?

Einige der vorgenannten Kriterien sind miteinander verbunden, wodurch Dopplungen entstehen. Die Pfadabdeckung impliziert beispielsweise die Abdeckung

von Ein- und Ausgängen. Die Anweisungsabdeckung impliziert die Zweigabdeckung und Schleifenabdeckung, da jede Anweisung mindestens einmal ausgeführt wird.

In der Datensicherheit werde Tests der Codeabdeckung eingesetzt, um Funktionen zu erkennen, die nicht erreicht werden können, wie dies beim Angriff aus [5] der Fall war. Zusätzlich kann sichergestellt werden, dass jede Codezeile getestet wurde. Ein Fehlerfreies Verhalten der jeweiligen Codezeilen kann die Codeabdeckung jedoch nicht garantieren.

**Fuzzing**
Fuzzing ist eine Technik die verwendet wird, um Software und Netzwerke zu testen. Hierfür werden die Implementierungen einer unerwarteten, ungültigen oder zufälligen Eingabe unterzogen, mit der Hoffnung, dass das Ziel auf unerwartete Weise reagiert und dadurch neue Schwachstellen identifiziert werden [55]. Die Reaktionen auf solche Angriffe gehen von seltsamen Verhalten an Schnittstellen über unspezifiziertes Verhalten des Systems bis hin zu Systemabstürzen und Neustarts. Fuzzing als Testverfahren ist für automobile Systeme relativ neu [4], obwohl moderne Fahrzeuge viele Gemeinsamkeiten zu gängigen Computersystemen oder IoT aufweisen.

In der Regel kann das Fuzzing in drei Schritte unterteilt werden:

1. Erzeugung der Eingabedaten
2. Einspeisen der Eingabedaten in das System
3. Analyse des Verhaltens

Die im ersten Schritt erzeugten Eingabedaten werden entweder strukturiert anhand der Spezifikation oder aber völlig zufällige Permutationen eines Startwerts generiert. Das Testsystem muss hierfür auf das Prüfgerät abgestimmt und angepasst werden [55]. Anschließend werden die Daten in Schnittstellen des Systems eingespeist und der Ausgang überwacht. Als letzter Schritt muss das aufgezeichnete Verhalten von erfahrenen Programmierern analysiert werden, um mögliche Schwachstellen zu identifizieren. Im Gegensatz zu den ersten beiden Schritten kann die Identifizierung nicht automatisiert werden.

Da das Fuzz-Testing auf zufällig ausgewählten Mustern beruht, ist es statistisch unwahrscheinlich komplexe Zusammenhänge an den Schnittstellen zu entdecken. Dennoch hilft diese Methode fehlerhaftes Verhalten an den Schnittstellen zu identifizieren. Um den Nutzen des Fuzz-Testings zu erhöhen, muss die inter-

ne Struktur der Software bekannt sein und überwacht werden können (“White Box“) [55].

**Penetrationstests**
Während die statische Tests und Teile der dynamischen Tests (funktionale Tests sowie das Fuzzing) automatisiert ausgeführt werden können, bilden die Penetrationstests eine Testmethode, bei der weiterhin menschliche Experten als Tester eingesetzt werden. Bei diesen Tests wird versucht bekannte Schwachstellen auszunutzen und dadurch Zugriff auf das System zu erlangen. Das entsprechende Vorgehen basiert auf jahrelanger Erfahrung von Experten, die diese Tests durchführen. Beispiel für typische Penetrationstests ist das Ausnutzen undokumentierter Debug-Schnittstellen, um Zugriff auf Busse und interne Signale zu erlangen. Aber auch durch Öffnen der Chips und Zugriff auf das Silizium erhoffen sich die Tester Informationen über mögliche Angriffsvektoren [4]. Das bereitgestellte Wissen für die Tester reicht in der Regel von keinen Informationen (“Black Box“), über Zugriff auf die Spezifikation (“Grey Box“), bis hin zur Information über den Quellcode (“White Box“).

**Definition 35** (Interne Signale)**:**
*Interne Signale werden in einem Prozessorsystem übertragen und sind nach außen nicht direkt sichtbar. Internen Signale sind beispielsweise Zugriffe auf Speicherbereiche, Ausführen von Instruktionen oder Steuern der angeschlossenen Peripherie.*

Das Vorgehen für Penetrationstests kann durch vier Phasen beschrieben werden [51]:

**Reconnaissance:** Beschaffung der Informationen durch öffentliche Informationsquellen, Spezifikationen oder des zur Verfügung stehenden Quellcodes. Ziel hierbei ist das Sammeln von Informationen über mögliche Angriffsvektoren.

**Enumeration:** Erstes Ansprechen der Angriffsvektoren um weitere Informationen über das System zu erhalten und Erstellen eines Rankings der potentiellen Schwächen des Systems.

**Exploitation:** Zum Angriff auf das System werden die zuvor festgestellten Angriffsvektoren ausgenutzt und so versucht das System zu manipulieren.

**Dokumentation:** Abschließend werden die Ergebnisse aus den vorherigen Phasen dokumentiert.

Alle genannten dynamischen Tests garantieren keinen vollständige Abdeckung. Hier muss, wie auch beim funktionalen Testen, ein Kompromiss zwischen Test-Aufwand, Zeit und Vollständigkeit getroffen werden. Es ist zu beachten, dass es sich bei allen vorgestellten Tests um Negativ-Tests handelt, die weniger deterministisch sind als Positiv-Tests einer Funktion. Das bedeutet, dass keine vollständige Testabdeckung möglich ist, da nicht alle möglichen und denkbaren Fehlerfälle getestet werden. Darum muss eine sinnvolle und angemessene Auswahl an relevanten Testfällen getroffen werden [51]. Daher dienen dynamische Tests der Datensicherheit immer nur als Ergänzung zu den statischen Tests und den Analysen in der Design-Phase (siehe Kapitel 3.5.1).

Eine Übersicht und Einordnung der verschiedenen Methoden im V-Modell ist in Abbildung 3.5 dargestellt. Wobei die statischen Analysen auf dem Quellcode basieren und daher im unteren Bereich des V-Modells angesiedelt sind sowie die dynamischen Test im rechten Ast des V-Modells.

## 3.6 Kritik am Stand der Wissenschaft und Technik

Durch die Trends der Digitalisierung und des vernetzten und automatisierten Fahrens nimmt die Anzahl der Funktionen und deren Kommunikation untereinander zu (siehe Kapitel 1). Neben neuen Mobilitätslösungen entstehen dadurch aber auch Risiken (siehe Kapitel 2.1.3). Besonders der Aspekt der Datensicherheit muss daher bei der Vernetzung berücksichtigt werden.

Eine systematische Bewertung der Datensicherheit wird jedoch über statische und dynamische Tests derzeit in der Automobilindustrie kaum sichergestellt [4, 125]. Zudem erfolgt der systematische Einsatz von Security Analysen nur langsam [125]. Veranschaulicht wird dies auch dadurch, dass erste Standards wie SAE J3061 [89], die auf dem Vorgehen der Automobilindustrie aufsetzen, erst im Jahr 2016 eingeführt wurden. Der europäische Standard ISO 21434 ist sogar erst für Anfang 2020 geplant. Das darin enthaltene Vorgehen zur Analyse der Bedrohungen mit einer TARA kann zwar durchgeführt werden, allerdings schreiben die Standards keine Methoden für die Implementierung und

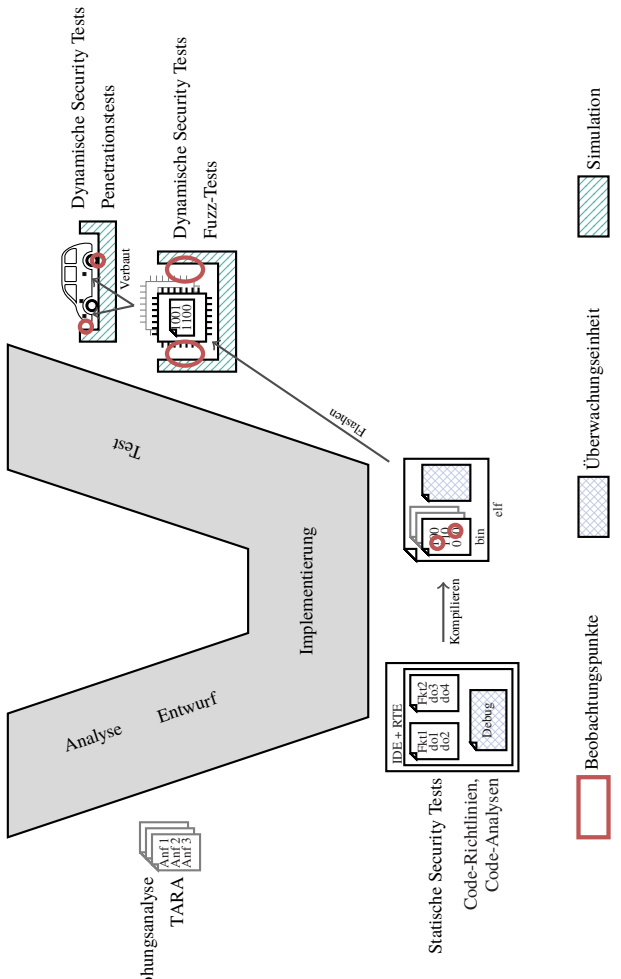



Abbildung 3.5: Einordnung der Datensicherheitstests im V-Modell

wichtiger, für die anschließenden Tests zur Gewährleistung der Datensicherheit vor.

Die vorgestellten Testmethoden des Stands der Wissenschaft und Technik für den Test der Datensicherheit fokussieren sich analog zur funktionalen Sicherheit auf die Schnittstellen der Systeme (“Black Box“). Da Fehler der Datensicherheit jedoch durch interne Probleme oder Datenmanipulation entstehen und diese nicht zwingend nach außen sichtbar werden, ist es nötig während den Tests die internen Zustände, Speicher und Rechenoperationen zu überwachen (“White Box“).

Unter bestimmten Bedingungen können Speicherverfälschungen nicht zu einem beobachtbaren und sofortigen Absturz des Systems führen [72]. Dies wird dann als stille Speicherkorruption bezeichnet. Bei einer stillen Speicherkorruption setzt das Programm seine Ausführung fort oder geht in einen unbeabsichtigten fehlerhaften Zustand. Die wichtige Konsequenz ist, dass der tatsächliche Fehler erst zu einem späteren Zeitpunkt bemerkbar wird [72]. Dies geschieht beispielsweise erst, wenn eine bestimmte Funktionalität angefordert wird oder eine bestimmte Sequenz an Ereignissen auftritt. Solange das System weiter läuft, wird das Problem daher nicht erkannt, stellt aber dennoch eine Bedrohung der Datensicherheit dar. Sobald das System in den fehlerhaften Fall übergeht, ist die Integrität gefährdet.

In der PC-Welt sind stille Speicherkorruptionen während der Laufzeit bereits bekannt und werden durch Schutzmechanismen wie Speicherisolation und Integritätsprüfungen von Speicherstrukturen erkannt, analysiert und dem Anwender gemeldet werden. Auf eingebetteten Systemen fehlen solche Mechanismen oft aufgrund ihrer begrenzten Schnittstellen, begrenzten Kosten und begrenzten Rechenleistung [72]. Zusätzlich ist eine Interaktion zur Fehlermeldung nur über die Umwelt möglich, wodurch der Test auf beobachtbare Effekte angewiesen ist, um dadurch den internen Zustand des Systems nachzuvollziehen. Interne Zustände in einem Hardwareaufbau sind nur zugänglich, indem diese durch Simulation einer virtuelle Plattform ebenfalls beobachtet werden.

Fehler und Sicherheitslücken, die durch Vernetzung der Steuergeräte und Funktionen entstehen, können nur durch Betrachtung des Steuergeräts oder des Gesamtsystems nach Definition 15 gefunden werden. Hierfür ist eine ganzheitliche Betrachtung des Systems nötig. Dies kann entweder erreicht werden, indem alle Teilsysteme physikalisch aufgebaut oder simuliert (z. B. Restbussimulati-

on) werden. Während beim physikalischen Aufbau das Steuergerät vorhanden sein muss, reicht bei der Simulation die Spezifikation der Schnittstellen.

Ein weiterer Kritikpunkt ist das zu testende Artefakt. Während statische Analysen nur grafische Beschreibungen oder den Quellcode berücksichtigen, untersuchen dynamische Verfahren den kompilierten Code auf dem Steuergerät. Bei der Analyse des kompilierten Maschinencodes als "Black Box" sind jedoch keine Rückschlüsse auf die Beschreibung und den Quellcode möglich. Beim Testen der grafischen Beschreibung oder des Quellcodes, ist die Anwendung noch nicht kompiliert, optimiert und für die Hardware angepasst. Durch das Kompilieren und Optimieren können neue Fehler entstehen. Durch das Testen der Software auf der realen Hardware ist entweder nur der Zugriff auf Schnittstellen möglich oder die Anwendung muss um entsprechende Testschnittstellen erweitert werden. Um eine nachträgliche Veränderung des Codes zu vermeiden, muss auf dem realen Maschinencode getestet werden, so wie der Code auf dem späteren System eingesetzt wird.

Fasst man die vorgestellten Testprozesse (siehe Kapitel 2) mit Kapitel 3 zusammen, ergibt sich die in Tabelle 3.2 dargestellte Bewertung. In Tabelle 3.2

Tabelle 3.2: Übersicht und Bewertung der Testprozesse

| | Test | | Codeart | | | | Hardware | | | Zugriffe | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Testprozess | statisch | dynamisch | Graphisch | Quellcode | Assembler | Maschinen | Echtzeit | Optimiert | Reale HW | Schnittstellen | Prozessor | Speicher | Peripherie |
| Model-in-the-Loop | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ |
| Software-in-the-Loop | ✓ | (✓) | (✓) | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ |
| Rapid Prototyping | ✗ | ✓ | (✓) | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | (✓) |
| Hardware-Simulation | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ |
| Prozessor-in-the-Loop | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✗ | ✓ | (✓) | ✗ | (✓) | ✓ | ✗ |
| Hardware-Debug | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | (✓) | (✓) | ✗ |
| Hardware-in-the-Loop | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ |
| Fahrzeug-Test | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ |

sind die jeweiligen Abdeckungen mit einem grünen Haken markiert, Punkte die durch den jeweiligen Testprozess hingegen nicht abgedeckt sind, wurden mit einem roten Kreuz markiert. Eingeklammerte Symbole können teilweise realisiert werden. So ist beim Testen mit Debug-Hardware oder beim PiL zwar

ein Zugriff auf Prozessor und Speicher möglich, dieser ist jedoch nicht so detailliert, wie dies bei der Simulation der Fall ist.

Kombiniert man die Hardware-Simulation mit einem PiL-Ansatz, kann durch Simulation sowohl die Software-Applikation als auch die Hardware mit Peripherie getestet werden (siehe Überdeckung in Tabelle 3.2). Eine Übertragung der Software auf die reale Hardware hängt jedoch vom gewählten Simulator ab.

## 3.7 Forschungsfragen

Um Fehler zu erkennen, die an den Schnittstellen nach außen nicht sichtbar sind, wird eine weitere Stufe zwischen die Simulation mit virtueller Hardware und die Ausführung auf der Ziel-Hardware gelegt (siehe Abbildung 3.6). Bei dieser Stufe wird das Steuergerät aus der Ausführung auf der Ziel-Hardware virtualisiert und innerhalb eines Computersystems ausgeführt (dynamisches Testen). Auf diesem virtuellen Steuergerät wird der selbe Maschinencode ausgeführt, wie er in der späteren Ziel-Hardware eingesetzt wird. Vorteil dieses Vorgehens ist, dass die Überwachungseinheit aus dem Quellcode ausgegliedert wird, sodass dieser unverändert auf der Ziel-Hardware eingesetzt werden kann und gleichzeitig zusätzliche Beobachtungspunkte im virtuellen Steuergerät eingefügt werden. Dadurch ist es möglich zur Laufzeit Überwachungen auszuführen, wie diese nur bei der Simulation auf einem Computer oder mit virtueller Hardware möglich sind. Für den Einsatz des Tests der Datensicherheit stellt sich folgende Frage:

**Forschungsfrage 1:**
*Welche Beobachtungspunkte und Überwachungsfunktionen müssen für die Tests der Datensicherheit gegeben sein?*

Für die Beobachtungspunkte sollen jedoch keine Schnittstellen oder zusätzliche Anpassungen generiert werden, die im späteren Seriensteuergerät nicht vorhanden sind. Basierend auf den Anforderungen zur Beobachtung und Überwachung wird anschließend eine Testmethodik vorgestellt, die Schwachstellen in der Datensicherheit aufzeigt. Hierfür stellt sich die weitere Forschungsfrage:

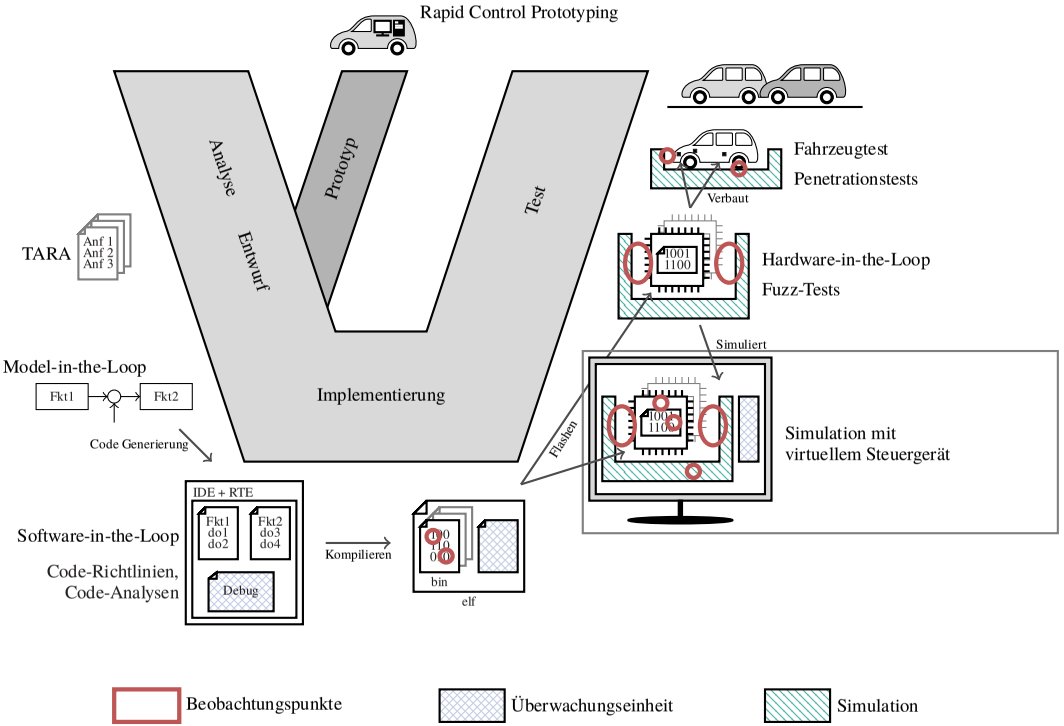



Abbildung 3.6: Analyse der Lücke zwischen Simulation auf Computer und Ausführung auf Ziel-Hardware aus Abbildung 2.13 und Abbildung 3.5

**Forschungsfrage 2:**
*Wie muss eine Testmethodik gestaltet sein, damit Schwachstellen der Datensicherheit durch eine statische oder dynamische Untersuchung erkannt werden?*

Abschließend soll das Vorgehen zum Test der Datensicherheit in der Entwicklung von Automobilsteuergeräten eingesetzt werden können. Hierzu ist folgende Frage klären:

**Forschungsfrage 3:**
*Wie kann das vorgestellte Vorgehen in den Entwicklungszyklus bei Herstellern und Zulieferern eingesetzt werden?*

Um die Testmethode in den Entwicklungszyklus einzubinden, muss in Forschungsfrage 3 zudem die Skalierbarkeit, der Aufwand und die Performanz der Testmethode untersucht werden.

# 4 Analyse und Kategorisierung der Datensicherheit in vernetzten Automotive Systemen

In diesem Kapitel werden mögliche Angriffe in der IT-Domäne aufgezeigt und kategorisiert. Anschließend werden wesentliche Unterschiede für den Einsatz von eingebetteten Systemen in der Automobilindustrie und anderen Industrien aufgezeigt. Aufbauend auf diesen Erkenntnissen werden potentielle Angriffe auf die Automobilindustrie übertragen und diese mit den Angriffen aus Kapitel 3.4 verglichen.

## 4.1 Analyse der Angriffe auf Systeme

In der PC-Welt werden Sicherheitslücken (engl. vulnerability) seit 1999 in einer Datenbank der MITRE Cooperation im Auftrag des U. S. Departments of Homeland Security gesammelt. Diese Common Vulnerabilities and Exposures Datenbank (CVE) [142] speichert alle bekannten Sicherheitslücken in vorhandenen Anwendungen ab. Bis zum Ende des Jahres 2018 wurden in dieser Datenbank über 110.000 Angriffe auf verschiedene Systeme protokolliert. Die Einträge der vergangenen zehn Jahre sind im Folgenden, basierend auf einer durch das National Institute of Standards and Technology (NIST) vorgeschlagenen Kategorisierung, aufgelistet und deren Verteilung in Abbildung 4.1 dargestellt. Kategorien mit ähnlichem Verhalten wurden dabei zusammengefasst.

Aufgrund der großen Anzahl an Kategorien durch das NIST [146] beschränkt sich die folgende Ausführung auf die größten Kategorien. Weiter können die aufgelisteten Kategorien in drei Klassen (Speicherfehler, numerische Fehler sowie systematische, funktionale und Anwendungsfehler) aufgeteilt werden (siehe Abbildung 4.1). Speicherfehler bezeichnen dabei Sicherheitsprobleme,

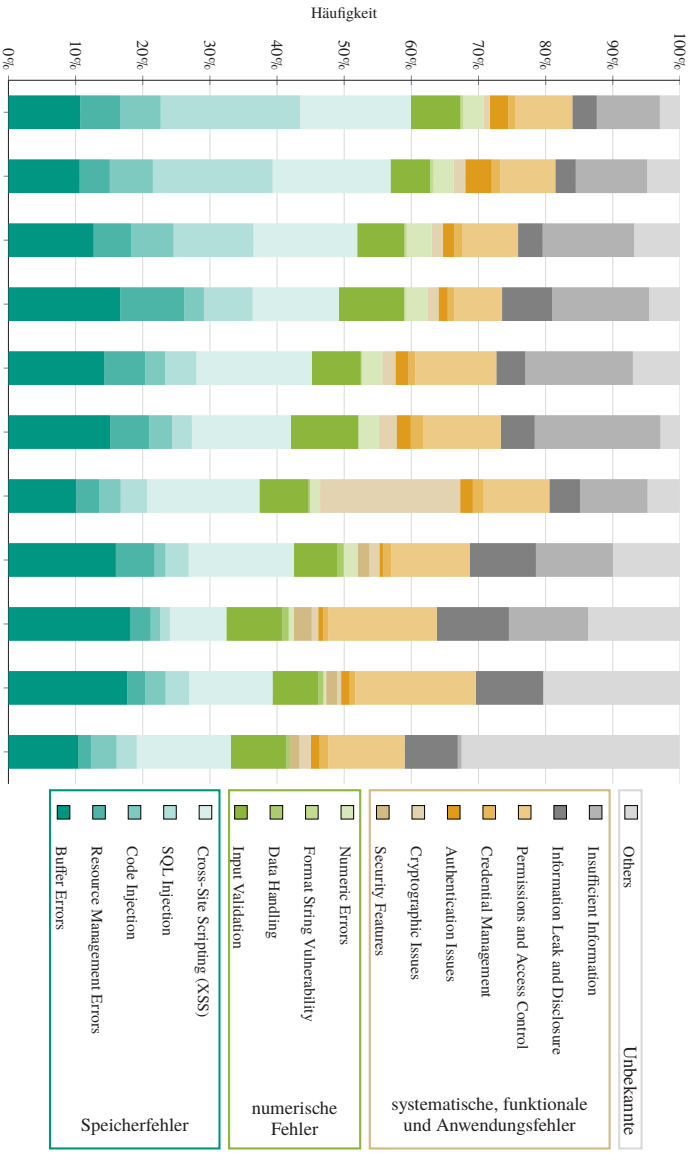



Abbildung 4.1: NIST Klassifikation der Schwachstellen anhand CVE [146]

die durch Schreiben oder Lesen an unerlaubten Speicherstellen auftreten. Bei numerischen Fehlern handelt es sich um Probleme durch falsches Abfragen von Eingaben oder Berechnungen. Beides kann während der Softwareentwicklung vermieden werden. Systematische, funktionale und Anwendungsfehler entstehen hingegen durch falsche Benutzung von Sicherheitsfunktionen, Schwachstellen in kryptografischen Verfahren oder der Weitergabe von Informationen. Alle vier Punkte sind in der Softwareentwicklung nicht zu beheben.

1. Speicherfehler:

   **Buffer Errors** (oder Buffer Overflows) überschreiben Speicherbereiche gezielt mit zu langen Datensätzen, um die im Speicher folgenden Datensätze zu manipulieren (siehe auch Kapitel 2.4.5).

   **Resource Management Errors** entstehen durch den unkontrollierten oder gleichzeitigen Zugriff von Anwendungen auf die selbe Ressource (z. B. Speicher, Peripherie, Cache). Hierzu zählen auch die Anfang 2018 bekannt gewordenen Sicherheitslücken Spectre [52] und Meltdown [60].

   **Code Injection** beruht auf dem Prinzip der Buffer Overflows. Wobei nicht nur Variablen überschrieben werden, sondern gezielt die Rücksprungadresse auf schadhaften, injizierten Code einer eigenen Funktion gesetzt wird (siehe auch Kapitel 2.4.6).

   **SQL Injection** schreiben Befehle bei der Abfrage oder beim Schreiben von Datenbanken in das Datenfeld und damit werden diese Befehle auf dem Datenbank-Server ausgeführt.

   **Cross-Site Scripting** ermöglicht es Angreifern Skripte in Webseiten einzufügen, die von anderen Benutzern angezeigt und ausgeführt werden.

2. Numerische Fehler:

   **Input Validation** sind Probleme, die sich dadurch ergeben, dass Eingabedaten vor der Verarbeitung nicht überprüft werden.

   **Data Handling and Numeric Errors** beinhaltet den Umgang mit Daten, das Überprüfen der Datentypen und Wertebereiche. Das Rechnen mit solchen Daten kann zu numerischen Rechenfehlern oder sogar Abstürzen führen (z. B. Nulldivision oder Vorzeichenfehler). Ursache hierfür können falsche Aufrufe oder Übergabeparameter bei Bibliotheksfunktionen sein.

**Format String Vulnerability** entstehen bei der Verarbeitung von Variablen, wenn der gültigen Wertebereichs und der Datentyp nicht auf Korrektheit geprüft wird.

3. Systematische, funktionale und Anwendungsfehler:

**Security Features** sind falsch verwendete Hilfen (z. B. Passwort-Safe), die es Nutzern erleichtern sollen IT-Sicherheit einzusetzen oder unzureichende Passwörter.

**Cryptographic Issues** sind Fehler in kryptografischen Verfahren, wie sie beispielsweise 2014 bei SSL durch Heartbleed [17] aufgetreten sind.

**Authentication Issues** lassen Zugriffe auf Ressourcen (Speicher, Variablen, etc.) zu ohne die Rechte der zugreifenden Instanz zuvor zu prüfen.

**Credential Management** ist ein standardisiertes Vorgehen, um Passwörter und Anmeldeinformationen zu speichern.

**Permission, Privileges and Access Control** sind Sicherheitslücken, die durch unzureichende Rechtevergabe oder Rechtekontrolle entstehen.

**Information Leak/Disclosure** beinhaltet Schwachstellen durch die Weitergabe von Informationen oder Passwörtern.

**Insufficient Information** beinhaltet die Sicherheitslücken, die durch Unwissenheit schlecht oder sogar überhaupt nicht abgesichert sind.

4. Unbekannte:

**Others** beziehen sich auf Schwachstellen die entweder in keine der obigen Kategorien eingegliedert werden können oder über die derzeit keine Informationen zur Verfügung stehen. Falls Sicherheitslücken noch nicht in allen Systemen geschlossen werden können, wird mit der Veröffentlichung der Details gewartet, um keine Systeme zu gefährden.

Grund für die obigen Schwachstellen ist meist eine schlecht implementierte Software, die durch Programmierfehler entsteht [82]. Besonders die konsequente Abfrage des Speicherbereichs bei dynamischen Variablen kann einen Buffer Error und Code Injection in den meisten Fällen verhindern [30]. Aufgrund von Zeit- und Speicherplatzanforderungen wird darauf jedoch häufig verzichtet. Ein weiterer Grund für Fehlverhalten ist die Division durch Null, die in Prozessoren und Mikrocontrollern nicht einheitlich spezifiziert ist und daher zu unterschiedlichem Verhalten oder gar zum Programmabbruch führt. Zusätzlich gibt es in der Softwareentwicklung häufig undefiniertes Verhalten beim Dereferenzieren von sogenannten Null-Pointern, die auf keinen Spei-

cher verweisen, beim Verwenden von Speicher oder Objekten nach Ausführen von “free“ (“use after free“), beim mehrfachen Freigeben von Speicherplatz (“double free“) oder beim Lesezugriff auf nicht allokierten Speicher. Meist können die vorgenannten Probleme durch konsequente Abfragen in der Programmierung zwar vermieden werden, dennoch werden die Abfragen selten aus Laufzeit- und Speichergründen implementiert.

Für die Betrachtung der Datensicherheit sind besonders die Speicherfehler (Buffer Error, Code Injection, SQL Injection und Cross-Site Scripting) relevant, da hierdurch schadhafter Code auf dem System eingebracht werden kann, um das Verhalten der Anwendung zu beeinflussen.

## 4.2 Abgrenzung der Automobilindustrie zu anderen eingebetteten Systemen

Der Automobilbereich unterscheidet sich von anderen Domänen wie dem Industriesektor, den Konsumprodukten, der Avionik oder der Medizintechnik. Vier typische Eigenschaften von eingebetteten Systemen sind die Sicherheit, Zuverlässigkeit, Verfügbarkeit und Echtzeitfähigkeit [59]. Zusätzlich werden hier noch die Ressourcen, Stückzahlen und Kosten der jeweiligen Domänen berücksichtigt, um die Unterschiede und Gemeinsamkeiten herauszustellen.

Die Unterschiede der verschiedenen Branchen sind in Tabelle 4.1 zusammengefasst und werden im folgenden näher erläutert.

Tabelle 4.1: Zusammenfassung der Unterschiede verschiedener Branchen

| | Automotive | Industrie 4.0 | Konsum | Avionik | Medizin |
|---|---|---|---|---|---|
| Sicherheitskritisch | ja | ja | nein | ja | teilweise |
| Lebensdauer | 20-25 Jahre | 6-7 Jahre | 2 Jahre | 30-40 Jahre | 5-15 Jahre |
| Updates | selten | selten | täglich | selten | selten |
| Wartung | Werkstätten | Fachpersonal | keine | Fachpersonal | Fachpersonal |
| Echtzeit | ja | ja | nein | ja | teilweise |
| Stückzahlen | Millionen | Hunderttausend | Milliarden | Tausend | unbekannt |
| Ressourcen | gering | gering | hoch | gering | gering |
| Verantwortung | Hersteller | Hersteller | Kunde | Hersteller | Hersteller |

### Sicherheitskritische Systeme

Die Sicherheit von Personen muss im Automobil, der Avionik und bei Industriemaschinen in allen Situationen gewahrt bleiben. Ein Fehlverhalten kann die Personensicherheit bei Fahrzeugen, Flugzeugen oder Industrierobotern und Industrieanlagen direkt beeinflussen. Fehlererkennungsmechanismen müssen daher so ausgelegt sein, dass sie von einem Angreifer nicht ausgenutzt werden können. Für die funktionale Sicherheit existieren in den Domänen verschiedene Standards, so müssen beispielsweise sicherheitskritische Systeme im Fahrzeug nach ISO 26262 [44] entwickelt werden, um ein entsprechendes ASIL Level zu erhalten (siehe Kapitel 2.3).

Bei Computern und Smartphones sind bei Angriffen Daten der Anwender oder Firmen betroffen, die keinen direkten Einfluss auf die Sicherheit von Personen haben.

Bei der Medizintechnik wird zwischen verschiedenen Risikoklassen unterschieden [25], welche über die Verletzbarkeit des menschlichen Körpers definiert sind. Medizinische Produkte mit einer Klasse höher als IIb (beispielsweise Beatmungsgeräte oder Implantate) müssen die Sicherheit des Patienten berücksichtigen und dürfen von einem Angreifer nicht beeinflusst werden.

### Lebensdauer und Updates

Im Bereich der IT-Branche (Computer und Smartphones) sind kurze Lebensdauern zu erwarten, hier wird von einer durchschnittlichen Lebensdauer von 2 Jahren ausgegangen [132]. Um eine Sicherheitslücke zu beheben, ist eine schnelle Reaktion und Update-Verteilung erforderlich. Im Konsumbereich können die Geräte jederzeit mit Updates gewartet werden.

Ein Industrieroboter wird hingegen schon 6 bis 7 Jahre eingesetzt [141] und erreicht dabei im Dauerbetrieb 60.000 Arbeitsstunden. Da Maschinen in Industrieanlagen durchgängig eingesetzt werden, sind Sicherheitsupdates nur durch eine Unterbrechung der Fertigung möglich und werden daher vermieden.

Medizinische Geräte haben in Krankenhäusern eine Lebensdauer von 10 Jahren [35]. Implantate wie beispielsweise Herzschrittmacher werden ebenfalls nach

5 bis 15 Jahren ausgetauscht [129]. Besonders beim Einsatz von Implantaten wird eine Aktualisierung der Gerätesoftware nicht durchgeführt.

Ein Fahrzeug hat eine Fahrleistung von 150.000 bis 300.000 km [36]. Dies entspricht einer erwarteten Fahrzeuglebensdauer von 20 bis 25 Jahren [87]. Während dieser Zeit muss das Fahrzeug funktional sicher sein, auch wenn Firmware aktualisiert wird oder Fahrzeugteile ausgetauscht werden. Updates werden heutzutage nur durch fachkundiges Personal in einer Werkstatt durchgeführt.

Moderne Flugzeuge, wie beispielsweise der Airbus A320, werden bis zu 180.000 Flugstunden eingesetzt. Dies entspricht einer erwarteten Flugzeuglebensdauer von bis zu 30 bis 40 Jahren [140]. In dieser Zeit wird das Infotainment regelmäßig dem Stand der Technik des Konsumbereichs angepasst, ohne die Steuerelektronik zu verändern. Im Vergleich zur Automobilindustrie hat die Avionik den Vorteil, dass eine strikte Trennung zwischen Steuerelektronik und Infotainment der Passagiere besteht. Das heißt Angreifer in der Kabine können die Steuerung nicht manipulieren, unabhängig von Sicherheitslücken im System.

#### Wartung und Reperatur

Autorisierte Werkstätten müssen in der Lage sein, das Fahrzeug zu diagnostizieren und im Fehlerfall Module auszutauschen. Hierbei ist es irrelevant, ob die Wartung oder Reparatur in einer Vertragswerkstatt oder in einer freien Werkstatt durchgeführt wird. Hinzu kommt, dass in freien Werkstätten nicht immer Ersatzteile der Hersteller verbaut werden.

Computer oder Smartphones werden bei defekten Komponenten in der Regel ersetzt, ohne das einzelne Komponenten ausgetauscht werden.

Industrieanlagen, Flugzeuge und medizinisches Equipment werden hingegen von Fachpersonal gewartet. Zusätzlich werden bei Reparaturen nur Ersatzteile der Hersteller verwendet.

### Echtzeitfähigkeit

Steuergeräte, die für die funktionale Sicherheit im Fahrzeug, Industrieanlagen und Avionik zuständig sind, müssen echtzeitfähig sein. Das heißt, die Software muss Aufgaben innerhalb einer begrenzten Zeitspanne bearbeiten, da sonst Personen gefährdet werden können. Diese Bedingungen sind in Konsumprodukten nicht anzutreffen. In der Medizintechnik hängt die Echtzeitfähigkeit vom Einsatz des Geräts. So müssen Herzschrittmacher beispielsweise Echtzeitanforderungen erfüllen, Ultraschall oder Blutdruckmessgeräte hingegen nicht.

### Stückzahlen und Kosten

Besonders die Automobilindustrie und die Konsumprodukte erreichen hohe Stückzahlen. So wurden im Jahr 2018 weltweit etwa 2,6 Milliarden Smartphones verkauft [156] und 82 Millionen PKWs neu zugelassenen [155].

Im Vergleich zur Automobilindustrie sind die Stückzahlen des Industriesektors, mit 381.000 Industrierobotern [153] und der Luftfahrt mit 2.300 Flugzeugen [154] sehr gering. Obwohl derzeit keine Statistiken der Stückzahlen aus der Medizintechnik vorhanden sind, ist davon auszugehen, dass diese im Vergleich zur Automobilindustrie und zum Konsumbereich ebenfalls sehr gering sind.

Durch die hohe Stückzahl ergibt sich, dass besonders die Automobilindustrie und der Konsumbereich durch Kosten getrieben sind.

### Ressourcen

Während in Konsumprodukten wie Computern und Smartphones leistungsstarke Hardware verbaut ist, wird in der Automobilindustrie, der Avionik und der Medizintechnik überwiegend günstige, leistungsschwache Hardware eingesetzt. Dadurch müssen spezielle Sicherheitsmechanismen für diese Bereiche entwickelt werden. Im Industriesektor werden leistungsstarke Rechner oder Server mit verteilten Sensoren eingesetzt. Da in der hierbei typischerweise nur die Server eine Anbindung an das Internet haben, können die selben Algorithmen ausgeführt werden wie bei Konsumprodukten.

#### Verantwortung

In der Automobilindustrie, im Industrie 4.0 Sektor, der Avionik und der Medizintechnik stellen Lieferanten Module mit Funktionalitäten zur Verfügung und müssen die Sicherheit ihrer Produkte aufrechterhalten. Hersteller integrieren Software- und Hardwaremodule von Drittanbietern und selbst entwickelten Modulen in die Produkte und müssen daher die Sicherheit aller Module einzeln und kombiniert gewährleisten. Im IT-Bereich wird diese Verantwortung geteilt. So sind Sicherheitslücken nicht in der Verantwortung des Computerherstellers, sondern der Betriebssystem- oder Anwendungsentwicklung. Für eine Installation der neuen Software ist der Kunde verantwortlich.

Im weiteren Verlauf wird aufgrund der hohen Stückzahlen, langen Lebensdauern und geringen Fähigkeit für Updates überwiegend auf die Fahrzeugindustrie fokussiert. Da hierdurch Sicherheitslücken besonders lange bekannt sind, aber nicht behoben werden können. Teile der Methodik können jedoch auch auf die anderen Domänen übertragen werden.

## 4.3 Ableitung der Angriffe auf eingebettete Systeme in der Automobilindustrie

Die in Kapitel 4.1 vorgestellten Schwachstellen lassen sich nicht ganzheitlich auf eingebettete Systeme aller Domänen übertragen.

So sind SQL-Injections aufgrund der fehlenden Webseiten mit Datenbankanbindung in Steuergeräten nicht wirksam. Ebenso können Angriffe über Cross-Site Scripting aufgrund der fehlenden Webseiten nicht auf eingebettete Systeme im Fahrzeug übertragen werden. In der Industrie 4.0 werden hingegen Datenbanken angebunden, wodurch Cross-Site Scripting und SQL Injektion ermöglicht werden. Durch den restriktiven Zugriff, wodurch eine Steuerung von außen in der Luftfahrt verhindert wird, entfällt das Rechtemanagement, dass den Zugriff regelt. Fehler, die durch Überschreiben von Speicherbereichen, Rechenoperationen oder durch systematische Fehler in der Kryptographie entstehen, können jedoch auf beliebige eingebettete Systeme übertragen werden, da diese Arten der Fehler unabhängig von Anwendung, Einsatz und Ressour-

cen des Systems sind. Eine Zusammenfassung aller Domänen ist in Tabelle 4.2 aufgelistet.

Tabelle 4.2: Übersicht der Schwachstellen in eingebetteten Domänen

| | Speicherfehler | | | | | Num. Fehler | | | systematische, funktionale, Anwendungsfehler | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Domäne | Buffer Errors | Resource Management Error | Code Injection | SQL Injection | Cross-Site Scripting | Input Validation | Data Handling and Numeric Errors | Format String Vulnerability | Security Features | Cryptographic Issues | Authentication Issues | Credential Management | Permission, Privileges, Access Control | Information Leak/Disclosure | Insufficient Information |
| Automotive | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ |
| Industrie 4.0 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ |
| Konsum | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Avionik | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ |
| Medizintechnik | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ |

Für eingebettete Systeme der Automobilindustrie ergibt sich damit eine reduzierte Auswahl an Schwachstellen, die während dem Testen betrachtet werden müssen. Diese Auswahl ist im Folgenden mit den drei Gruppen aus Abbildung 4.1 zusammenfassend aufgelistet:

1. Speicherfehler durch:
   - Buffer Errors
   - Code Injection
   - Resourcen Management Errors
2. Numerische Fehler durch:
   - Input Validation
   - Data Handling and Numeric Errors
   - Format String Vulnerability
3. Systematische, funktionale und Anwendungsfehler durch:
   - Security Features

- Cryptographic Issues
- Permission, Privileges and Access Control
- Information Leak / Disclosure
- Insufficient Information

Bei den aufgeführten drei Gruppen werden nur die ersten beiden im weiteren Verlauf detailliert betrachtet. Systematische und funktionale Fehler, die durch unzureichende Informationen (Insufficient Information) oder Weitergabe von Informationen (Information Leak / Disclosure) entstehen sowie Fehler durch schwache kryptografische Funktionen (Cryptographic Issues) können beim Test eines eingebetteten Systems nicht erkannt werden. Ebenfalls können Anwendungsfehler, wie unsichere Passwörter, durch Tests in der Entwicklung nicht erkannt werden.

## 4.4 Verknüpfung zu Angriffen im Fahrzeug

Bisherige Angriffe auf Fahrzeuge (siehe Kapitel 3.4) wurden über verschiedene Schwachstellen ausgeführt. Betrachtet man diese genauer, so können die Angriffe in die oben dargestellten Klassen eingeteilt werden. Angriff 1 wird an dieser Stelle jedoch nicht weiter betrachtet, da bei diesem Angriff eine Hardware im Fahrzeug angebracht wurde, die direkte Daten in die Bussysteme einspielt. Alle weiteren Angriffe sind in Tabelle 4.3 dargestellt und im folgenden ausgeführt.

Bei Angriff 2 wurde im Jahr 2011 das Fahrverhalten (inkl. Bremse) eines Fahrzeugs über das Infotainment manipuliert. Bei diesem Angriff wurde ein Fehlverhalten durch das Abspielen eines Musiktitels im Infotainment erzwungen, wodurch das Fahrzeug über eine Funkverbindung gesteuert werden konnte [19]. Für diesen Angriff wurde ein Fehler in der Implementierung der Musikwiedergabe ausgenutzt, wodurch Speicherbereiche überschrieben und schadhafter Code eingespielt werden konnte.

Angriff 3 ist derzeit der bekannteste Angriff. Hierbei wurde ein Jeep über die Telematik Schnittstelle so gesteuert, dass ein Eingriff in Motor, Bremse und Lenkung möglich war [70]. Dieser Angriff nutzte einen Fehler in der Generierung der WLAN-Passwörter aus, wobei das Password beim ersten Einschalten anhand der Zeit gesetzt wurde. Das Problem hierbei war jedoch, dass die Zeit

Tabelle 4.3: Verknüpfung der Angriffsarten mit Angriffen auf bestehende Systeme

| ID | Schnittstelle | Angriffstyp |
|---|---|---|
| 2 | Infotainment | Buffer Overflow, Code Injection |
| 3 | Telematik | Security Features |
| 4 | Infotainment | noch nicht veröffentlicht |
| 5 | Telematik | Buffer Overflow, Code Injection |
| 6 | Multimedia | Input Validation, Format String Vulnerability |
| 7 | Infotainment | noch nicht veröffentlicht |
| 8 | Funkschlüssel | Security Features, Cryptographic Issues |
| 9 | Telematik | Permission, Privileges and Access Control |
| 10 | Telematik | Buffer Overflow, Code Injection, Data Handling, Input Validation |

der Telematikeinheit erst nach dem Setzen des Passworts synchronisiert wurde, wodurch sich eine geringe Anzahl an möglicher Kombinationen ergab. Nach einem ersten Zugriff über WLAN, kann der Angriff über die GSM-Verbindung fortgesetzt werden. Über diese Verbindung konnte anschließend der Microprozessor des Gateways umprogrammiert werden, sodass dieser nicht mehr als Firewall funktionierte, sondern alle Kommandos weiterleitete.

Der Web Browser im Central Information Display (CID) des Fahrzeugs bei Angriff 5 stellte einen Angriffsvektor durch einen Buffer Overflow (siehe Kapitel 2.4.5) dar. Zusammen mit der Internetverbindung per WLAN oder Mobilfunknetz konnte ein Angriff auf das CID durchgeführt werden. Die Schwachstelle befand sich dabei in der Software des Browsers, die die Webseiten darstellt. Durch einen Buffer Overflow wurden Speicherbereiche überschritten und Administrationsrechte erhalten. Dadurch konnten die Sicherheitsmechanismen zur Prüfung neuer Software deaktiviert und anschließend manipulierte Firmware installiert werden. Durch eine weitere Code Injection konnten weitere Steuergeräte in der E/E-Architektur manipuliert werden, wodurch ein Eingriff in Bremse und Lenkung ermöglicht wurde.

Bei Angriff 6 wurde das Format eines Eingabewerts nicht überprüft, wodurch eine fehlerhafte Eingabe den Absturz des Bluetooth-Stack und damit der CD/Multimedia-Software im Infotainment-System ermöglichte.

Durch eine fehlerhafte Implementierungen wurde bei Angriff 8 die verschlüsselte Funkfernbedienung eines Tesla kopiert, wodurch der Diebstahl dieser Fahrzeuge ermöglicht wird [127]. Bei diesem Angriff wurde die Hardware-Verschlüsselung mit einem zu kurzen Schlüssel durchgeführt, wodurch eine Rekonstruktion des Schlüssels ermöglicht wurde.

Die weiteren Angriffe aus Kapitel 3.4 können derzeit noch nicht im Detail betrachtet werden, da hierzu noch nicht alle Informationen veröffentlicht wurden. Da die Schwachstellen derzeit noch nicht behoben wurden, sind nur abstrakte Beschreibungen der Fehler bekannt. Angriff 10 beschreibt dabei 14 Schwachstellen, die sowohl durch Überschreiben von Speicherbereichen als auch durch fehlerhafte Abfragen von Eingaben zustande kommen [48]. Unzulässige Einschränkungen innerhalb der Grenzen des Speichers wurde bei Angriff 9 entdeckt, wodurch der Speicher nicht wie definiert verwendet werden kann.

Für Angriff 4 und Angriff 7 wurden bisher keinerlei Details veröffentlicht, wodurch eine Bewertung zum aktuellen Zeitpunkt nicht möglich ist. Im weiteren Verlauf werden daher die zuvor genannten Angriffe als Proof of Concept verwendet.

## 4.5 Zusammenfassung

Angriffe auf Systeme in der IT-Industrie sind bereits zahlreich bekannt und werden über öffentliche Datenbanken protokolliert und kategorisiert. Von diesen protokollierten Angriffen lassen sich jedoch nicht alle auf eingebettete Systeme und die Automotive Domäne übertragen, daher wurden Angriffe bewertet und in drei relevante Klassen unterteilt. In der ersten Klasse ermöglichen Speicherfehler das Überschreiben von nicht autorisierten Bereichen bis hin zum Einfügen von schadhaftem Code. In der zweiten Klasse verursachen Rechenoperationen wie die Nulldivision und der falsche Umgang mit Daten (Data Handling) auf eingebetteten Systemen undefiniertes Verhalten oder Systemabstürze. Und drittens sorgen Systematische-, Funktionale- und Anwendungsfeh-

ler durch die falsche Nutzung von Sicherheitsfunktionen für Schwachstellen in Systemen.

Alle drei Klassen finden sich in den bekannten Angriffen auf eingebettete Systeme der Automobilindustrie. Die Kategorie der systematischen, funktionale und Anwendungsfehler kann jedoch durch das Testen von eingebetteten Systemen nicht behoben werden. Schlechte Passwörter oder die Weitergabe von Informationen sind unabhängig von der korrekten Implementierung ein Problem, das im weiteren Verlauf jedoch nicht weiter behandelt wird. Eine korrekte Auswahl der Sicherheitsfunktionen und -algorithmen muss bereits im Design berücksichtigt werden und wird daher ebenfalls nicht weiter betrachtet.

# 5 Bewertung bestehender Systeme zum Testen der Datensicherheit

Um die Datensicherheit von eingebetteten Systemen im automobilen Umfeld zu testen, wird zunächst untersucht, welche Methoden aus Kapitel 2.6.5 hierfür geeignet sind.

Eine Simulation auf einem Computer bietet die Möglichkeit bereits während der Algorithmenentwicklung Tests modellbasiert ohne Hardware durchzuführen (siehe Kapitel 5.3). Durch den Einsatz von leistungsstarker Hardware können die Tests frühzeitig in Prototypen unter realen Bedingungen durchgeführt werden, ohne dass die Software auf die Zielplattform angepasst ist (siehe Kapitel 5.4). Für eine Überwachung interner Signale werden auf Hardware Prototypen spezielle Debug-Schnittstellen (sogenannte Joint Test Action Group (JTAG) oder In-Circuit-Emulator) benötigt (siehe Kapitel 5.5). Virtuelle Plattformen bieten neben dem frühen Einsatz in der Entwicklung die Möglichkeit interne Signale, Speicher und Busse auf virtueller Hardware zu überwachen (siehe Kapitel 5.6).

## 5.1 Anforderung für den Test der Datensicherheit

Die Probleme der Datensicherheit aus Kapitel 4 entstehen häufig durch falsche Implementierungen, die nicht mutwillig programmiert wurden [51]. Diese Probleme können jedoch die Sicherheit eines gesamten Steuergeräts oder sogar des Fahrzeugs beeinflussen. Fehler, die durch Flüchtigkeit oder Unwissenheit entstehen, lassen sich jedoch durch Tests finden und anschließend beheben.

Um Datensicherheit zu überprüfen, muss zuerst festgelegt werden, gegen welche Angreifergruppen das Steuergerät abgesichert werden muss. Je nach technischen Möglichkeiten der unter Kapitel 2.4.4 vorgestellten Gruppen muss die Testtiefe angepasst werden. Zur Bewertung der Sicherheitsanforderung sind

alle verbundenen Elemente kritisch auf mögliche Schäden (siehe Definition 5) und Relevanz für die funktionale Sicherheit zu prüfen [51] (siehe auch Kapitel 3.5.1). Da mögliche Schäden im gesamten System auftreten können, muss immer das Gesamtsystem und nicht nur Einzelfunktionen untersucht werden. Das heißt die Testmethode muss die E/E-Architektur soweit abdecken, dass die Gesamtfunktionalität dargestellt werden kann. Hierfür muss auch die Echtzeitfähigkeit der Anwendung erhalten bleiben.

Zudem müssen die Tests alle bekannten Schwachstellen eines Systems adressieren. Das heißt, die Schwachstellen aus Kapitel 4.1 müssen durch Testfälle abgeprüft werden können. Zur Priorisierung der Schwachstellen ist es hilfreich Angriffe entsprechend zu gewichten und dadurch eine Testfolge zu erstellen (siehe Kapitel 3.5.1).

Die Tests der Datensicherheit müssen die drei Bereiche Speicherfehler, numerische Fehler sowie systematische, funktionale und Anwendungsfehler erkennen (siehe Kapitel 4.1). Besonders gezielte Speicherfehler (Buffer Overflows, siehe Kapitel 2.4.5 und Code Injection, siehe Kapitel 2.4.6) müssen vermieden werden, da hierdurch das Verhalten des Systems verändert werden kann (siehe Kapitel 2.4.6). Daher ist es wichtig diese Schwachstellen beim Testen zu identifizieren. Als weiterer Angriffsvektor speziell für Denial of Service (DoS) wurden numerische Fehler (z. B. Null-Divisionen) identifiziert, da hier das Verhalten des Prozessors undefiniert ist. Daher muss beim Testen überprüft werden, ob numerische Fehler vorkommen, diese in der Applikation verhindert werden oder wie sich das System beim Auftreten eines solchen Fehlers verhält. Systematische Fehler werden bereits durch funktionale Tests (siehe Kapitel 3.5.4) erkannt und daher im folgenden nicht berücksichtigt.

Um die Speicher- und numerischen Fehler zu identifizieren, muss der Speicher im Prozessor sowie die ausgeführten Rechenbefehle überwacht werden. Hierfür muss sowohl Zugriff auf die zu überwachenden Speicherbereiche, als auch auf die ausgeführten Instruktionen des Prozessors vorhanden sein.

Dynamische Security-Tests können nur auf vorhandenen Zielarchitekturen ausgeführt werden [4]. Tests die bereits in die Entwicklung der Software einfließen, lassen sich durch virtuelle Plattformen realisieren. Diese können eingesetzt werden bevor eine realer Ziel-Hardware existiert und diese simulieren. Weiter benötigt ein Tester möglichst viele Zugriffe (Beobachtungspunkte) auf interne Signale, Register und Speicherbereiche, um das System transparent

darzustellen, das Verhalten nachzuvollziehen und dadurch Schwachstellen zu identifizieren. Dadurch werden interne Fehler frühzeitig erkannt und nicht nachdem diese zu einer externen Schnittstelle propagiert sind.

Für den Test der Datensicherheit muss das Gesamtsystem (inklusive E/E-Architektur) betrachtet werden. Dennoch wird im Weiteren zuerst der Fokus auf ein einzelnes Steuergerät gelegt und die Methodik daran veranschaulicht. In Kapitel 8 wird anschließend die Methodik von einem Steuergerät auf einen Steuergeräteverbund skaliert, um Systemtests darzustellen.

## 5.2 Kriterien für die Eignung zur Beobachtung interner Signale

In einem Prozessorsystem werden zur Laufzeit der Applikation Signale intern übertragen, die nach außen nicht direkt sichtbar sind. Diese internen Signale sind beispielsweise Zugriffe auf Speicherbereiche, Ausführen von Instruktionen, Steuern der angeschlossenen Peripherie, Überwachung der Interrupts und Systemkontrolle (siehe Definition 35).

Zur Bewertung der Beobachtbarkeit interner Signale für den Test der Datensicherheit mit verschiedenen Arten der Funktionsprüfungen aus Kapitel 2.6.5 werden zunächst Kriterien hergeleitet:

1. Eine Möglichkeit zur Beobachtung (siehe Definition 33) der Anwendung muss bestehen.
2. Beim Test der Datensicherheit muss zusätzlich das Verhalten mit internen Zuständen und Signalen, nicht nur Ein- und Ausgängen, analysiert werden. Fehler der Datensicherheit entstehen durch Datenmanipulation im Speicher und werden nicht zwingend nach außen sichtbar (siehe Kapitel 3.6).
3. Die erstellte Anwendung muss innerhalb der Testmethode ausgeführt und auf die späteren Hardware (Steuergerät) portiert werden können.
4. Um Aussagen über die Güte eines Systems treffen zu können, muss der ausführbare Code unverändert auf der Hardware und dem Testsystem ausgeführt werden. Das heißt, es dürfen keine speziellen Instruktionen

im Code enthalten sein und der Code darf nach den Tests nicht verändert werden. Dies ist besonders für den Test der Datensicherheit wichtig, da zusätzliche oder fehlende Instruktionen das Verhalten des Codes verändern (siehe Kapitel 4).

5. Während den Tests muss nicht nur die Software im Prozessor ausgeführt werden, sondern ebenfalls die Peripherie, Speicherbereiche sowie interne Busse des Systems, um das Verhalten des Steuergeräts zu erhalten.

6. Zur Verifikation einer Gesamtfunktionalität, die auf mehreren Steuergeräten verteilt ist, muss neben dem Steuergerät ebenfalls das Gesamtsystem als Steuergeräteverbunds getestet werden können.

7. Um den Testvorgang bei steigender Komplexität der Funktionen effizient zu gestalten, muss der Testprozess entweder parallelisiert mit mehreren Tests oder beschleunigt ausgeführt werden können. Dies ermöglicht das Ausführen vieler Testfälle in einer begrenzten Zeit. Zusätzlich ist hierbei die Skalierung bei der Ausführung mehrerer Einheiten zu betrachten.

8. Die Testplattform muss bereits in der Entwicklung zur Verfügung stehen, um frühzeitig mit dem Testen beginnen zu können. Eine Änderungen der späteren Zielplattform oder Peripheriekomponenten muss am Testsystem möglich sein.

## 5.3 Beobachtung der Signale durch Simulation auf einem Computer

Eine Möglichkeit Anwendungen frühzeitig zu testen, besteht in der Simulation auf dem Computer (siehe Kapitel 2.6.5). Ziel der Simulation auf einem Computer ist die Untersuchung der verwendeten Algorithmen auf korrekte Ausführung (Kriterium 1). Dies kann für einzelne Softwaremodule oder für die Gesamtfunktionalität erfolgen (Kriterium 6). Durch die Simulation auf einem Computer steht die Software im Quellcode oder als Modell zur Verfügung, wodurch Einblicke in die internen Abläufe und Variablen möglich sind (Kriterium 2). Durch die Simulation ohne reale Hardware oder Umwelt müssen weitere Schritte bis zum fertigen eingebetteten System unternommen werden.

Läuft die Simulation auf einem Computer, ist eine Beobachtung der Anwendung und der internen Zustände ohne Anpassung möglich. Durch den Einsatz einer Integrated Development Environment (IDE) können Variablen und Funktionen während dem Test direkt beobachtet werden. Allerdings muss die Applikation zu einem späteren Zeitpunkt auf die Zielarchitektur angepasst werden, wodurch Änderungen in der Ausführung entstehen können. Dies steht jedoch in Widerspruch zu Kriterium 3 und Kriterium 4, da durch die Anpassung die finalen Instruktionen nicht bekannt sind. Die Peripherie und der Speicher können in dieser Testmethode ebenfalls nicht betrachtet werden (Kriterium 5), da diese Bausteine nicht vorhanden sind, sondern die Simulation auf einem Computer stattfindet (siehe Abbildung 5.1). Die Ein- und Ausgaben der Simulation erfolgen direkt auf Softwareebene, ohne Hardwarekomponenten des Computers, als ausführende Einheit, einzubeziehen. Zum Testen können sowohl Variablen als auch Funktionen der Applikation beobachtet werden (siehe Abbildung 5.1).

Da das Prüfobjekt nur aus Software besteht, können diese auf mehrere Computer verteilt oder beschleunigt ausgeführt werden (Kriterium 7), um eine große Anzahl an Testfällen in begrenzter Zeit auszuführen. Dies steigert die Effizienz der Tests. Die Modelle zur Ausführung auf dem Computer können bereits frühzeitig in der Entwicklung angefertigt und bei Änderungen angepasst werden (Kriterium 8).

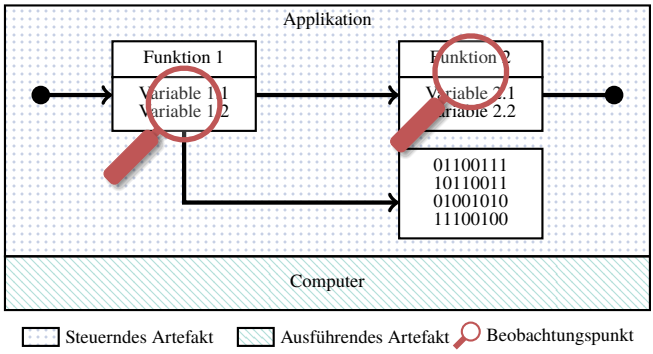



Abbildung 5.1: Beobachtungsmöglichkeiten einer simulierten Applikation

Durch den Widerspruch bei Kriterium 2, Kriterium 3, Kriterium 4 und Kriterium 5 zu den in Kapitel 5.2 vorgestellten Kriterien, ist die Beobachtung zum Test der Datensicherheit durch Simulation auf einem Computer nicht geeignet.

## 5.4 Beobachtung der Signale bei Ausführung auf leistungsstarker Hardware

Beim Testen der Anwendung auf leistungsstarker Hardware (siehe Kapitel 2.6.5) werden die Algorithmen getestet, bevor die Software in ein dediziertes Steuergerät integriert wird (Kriterium 1). Die für die Anwendung erzeugte Applikation ist weder für das spätere Steuergerät optimiert (Kriterium 3), noch für eine spezielle Plattform angepasst (siehe Kapitel 2.6.5). Durch den Einsatz einer leistungsstarken Hardware, muss die Software für die Ziel-Hardware angepasst und optimiert werden, wodurch eine direkte Übertragung auf die Ziel-Plattform nicht möglich ist (Kriterium 4). Zudem können nur Peripherie und Speicher der leistungsstarken Hardware, nicht jedoch von der Ziel-Hardware beobachtet werden (Kriterium 5). Die Algorithmen der Anwendung werden dabei unter realen Bedingungen, als Einzelmodul oder Verbund getestet (Kriterium 6). Dies kann beispielsweise über den Einsatz von HiLs oder RCP durchgeführt werden. Die Ein- und Ausgabe-Schnittstellen der leistungsstarken Hardware dienen zur Beobachtung des Verhaltens, ein direkter Rückschluss auf das interne Verhalten (Kriterium 2) ist bei diesem Verfahren nicht möglich (siehe Abbildung 5.2).

Abhängig von der verwendeten Hardware ist die Software-Applikation dabei entweder als kompilierte Version verfügbar, wodurch eine Beobachtung der Software nur an den Schnittstellen möglich ist oder als Modell der Applikation, wodurch Eingriffe wie in Kapitel 5.3 möglich sind.

Während den Tests mit leistungsstarker Hardware existieren dafür jedoch nur wenige Plattformen, wodurch das parallelisierte Testen nicht möglich ist (Kriterium 7). Durch die leistungsstarke Hardware kann bereits frühzeitig in der Entwicklung in der realen Umgebung getestet werden, ohne dass die Ziel-Plattform oder das Steuergerät zur Verfügung steht (Kriterium 8).

Besonders Kriterium 2, Kriterium 3 und Kriterium 4 stehen dabei im Widerspruch zu den in Kapitel 5.2 vorgestellten Kriterien, wodurch die leistungs-

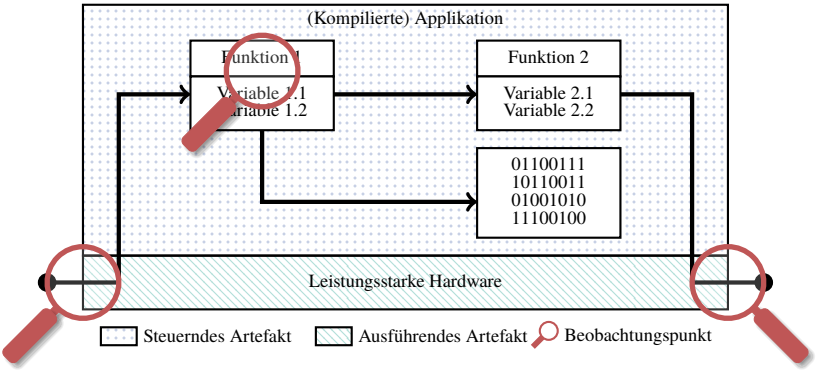



Abbildung 5.2: Beobachtungsmöglichkeiten bei Ausführung auf einer leistungsstarken Hardware

starke Hardware für die Einsatz zum Testen der Datensicherheit nicht geeignet ist.

## 5.5 Beobachtung der Signale bei Ausführung auf Ziel-Hardware

Um Software auf eingebetteten Systemen in der Ziel-Hardware (Definition 17) zu testen und zu debuggen, wird eine externe Hardware über definierte Schnittstellen angeschlossen, welche auf die internen Signale zugreift und die Ausführung der Applikation überwacht (Kriterium 1). Da jedoch nur die kompilierte Anwendung ausgeführt wird, wird ein sogenannter Disassembler[1] und der Quellcode benötigt, um Rückschlüsse auf verwendete Funktionen und Variablen zu ziehen. Abbildung 5.3 zeigt diesen Aufbau beispielhaft anhand eines ARM Cortex-M Prozessors.

Um eine Applikation über die Überwachungsschnittstelle zu debuggen oder um interne Signale zu überprüfen, ist ein externes Gerät nötig (Debug-Hardware am Computer), das an den Prozessor angeschlossen wird. Dieser Adapter wird an spezielle Pins des Prozessors angeschlossen, die an das “Test und Debug

[1] Ein Programm zum Umwandeln von Maschinencode in eine symbolische Sprache.

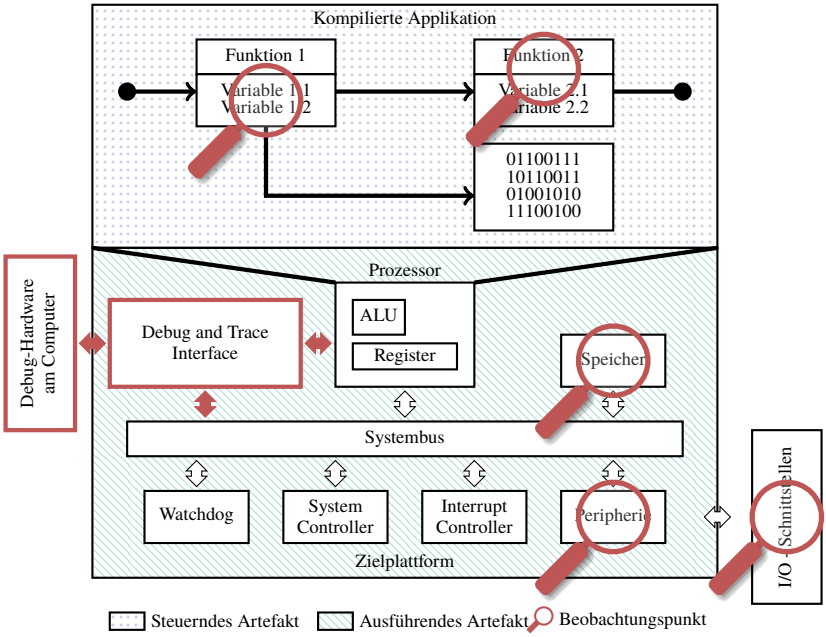



Abbildung 5.3: Beispielhafter Aufbau einer Zielplattform mit Speicher-, Bus- und Überwachungsschnittstelle über JTAG, FTDI [2]

Interface“ Modul verdrahtet sind. Dieses Modul gibt die Daten an dem Prozessor und lokalen Busse weiter, um so Zugriff auf die Peripherie, Prozessoren, Speicher und Bussysteme zu bekommen (Kriterium 5). Ein direkter Zugriff auf die Peripherie ist durch diese Debug-Module jedoch nicht möglich.

Ein sogenannter Watchdog bezeichnet in der Ziel-Hardware eine Funktion zur Ausfallerkennung des Systems. Wird dabei eine mögliche Fehlfunktion erkannt, so wird entweder dies gemäß Systemvereinbarung an andere Komponenten signalisiert (z. B. Umschalten auf ein redundantes System), eine geeignete Sprunganweisung bzw. ein Reset zur selbsttätigen Behebung des Ausfalls eingeleitet, oder ein sicheres Abschalten veranlasst. Der Watchdog ist dabei als Timer realisiert und wird in regelmäßigen Abständen auf 0 gesetzt. Läuft der Timer ab, wird eine Fehlfunktion erkannt (z.B. Absturz eines

Programms). Dadurch wird Fehlverhalten innerhalb des Systems erkannt, eine Beobachtung des Watchdogs von außen ist jedoch nicht möglich.

Vorteil der Beobachtung der Signale bei Ausführung auf der Ziel-Hardware ist, dass die Applikation bereits für die Ziel-Architektur angepasst und optimiert ist (Kriterium 3), da die Tests direkt auf der Ziel-Architektur ausgeführt werden. Eine nachträgliche Änderung oder Optimierung der Instruktionen oder Softwaremodule ist nicht nötig (Kriterium 4). Zusätzlich werden jedoch spezielle Softwaremodule benötigt, die in die Anwendung eingebunden sind. Dies ist nötig, um die Applikation zu beobachten und den Prozessor bei Bedarf über die externe Schnittstelle anzuhalten. Eine solche Anpassung der Software steht jedoch im Widerspruch zu Kriterium (Kriterium 4).

Speziell in der Serienentwicklung und Produktion von Steuergeräten ist die Debug-Schnittstelle jedoch nicht mehr vorhanden oder deaktiviert, da sie im späteren Steuergerät als Einfallstor genutzt werden kann. Dadurch ist das Beobachten der internen Signale in der Entwicklung und den damit verbundenen Integrationstests von Seriensteuergeräten erschwert (Kriterium 2).

Da die Tests, bzw. das Debugging direkt auf der Hardware ausgeführt werden, muss für das parallelisierte Ausführen von Tests, für jeden Testfall eine Hardware zur Verfügung stehen (Kriterium 7). Das heißt, dass für das parallelisierte Durchführen von verschiedenen Testfällen ebenso viele Hardwaregeräte wie parallele Tests vorhanden sein müssen.

Die fertige Ziel-Hardware steht in der Entwicklung erst später als die ersten Software-Module zur Verfügung, wodurch Tests der Software-Module auf der Ziel-Hardware erst in späten Entwicklungsschritten möglich sind (Kriterium 8).

## 5.6 Beobachtung der Signale durch Simulation mit virtuelle Plattformen

Virtuelle Plattformen bilden die reale Hardware ab (siehe Kapitel 2.5), verschieben jedoch die Beobachtungspunkte von der Debug-Schnittstelle in die Simulation. Abbildung 5.4 zeigt diesen Aufbau beispielhaft anhand eines ARM Cortex-M Prozessors. Das heißt, auch wenn keine speziellen Hardwareschnitt-

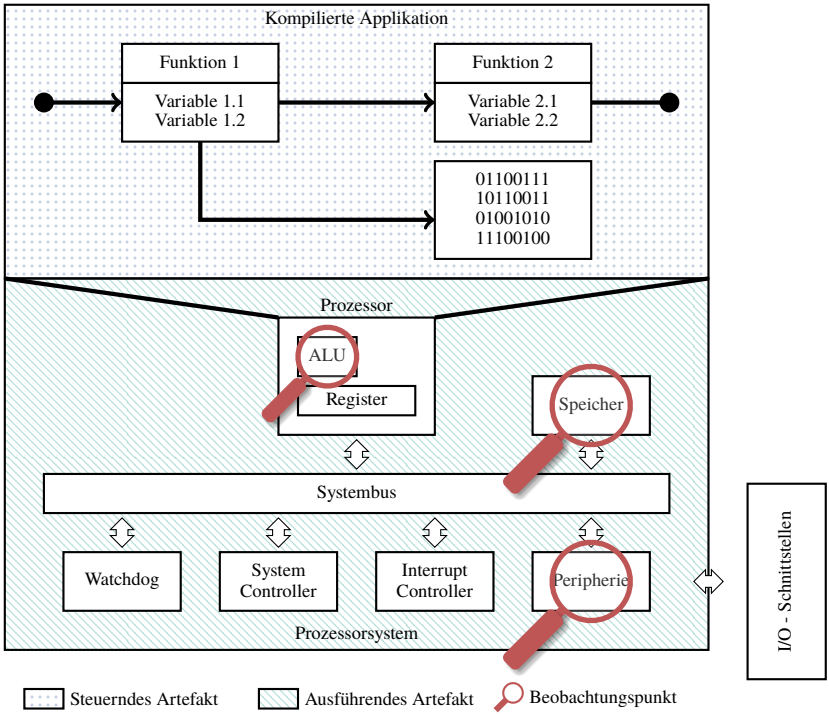



Abbildung 5.4: Aufbau einer virtuellen Plattform zur Beobachtung von Speicher-, Bus- und Instruktionen

stellen wie in Kapitel 5.5 vorhanden oder diese deaktiviert sind, wird die Beobachtung von der Simulationsumgebung übernommen.

Durch die Ausführung der Anwendung in der Simulation (Kriterium 1) werden alle Zugriffe auf die internen Speicher und Bussysteme in der Simulation ausgeführt (Kriterium 2). Dadurch können die Zugriffe direkt aus der Simulation beobachtet und analysiert werden, ohne zusätzliche Debugmodule in der Software oder eine externe Debug-Hardware im Steuergerät einzubringen (Kriterium 4). Jeglicher Eingriff in die Ausführung wird von der Simulationsumgebung durchgeführt. Dadurch muss der Prozessor nicht von der Debug-Schnittstelle gestoppt werden, da dies von der Simulationsumgebung übernommen wird. Da die kompilierte Anwendung jedoch nur im Maschi-

nencode (siehe Definition 31) vorliegt, sind keine direkten Rückschlüsse auf Funktionen und Variablen möglich. Diese müssen aus dem Maschinencode mit einem Disassembler gewonnen werden.

Durch die Abbildung des Gesamtsystems in einer virtuellen Simulation, inklusive Prozessor, Speicher, Bussysteme und Peripherie, ist es möglich das kompilierte Programm sowohl auf der realen Hardware als auch in der virtuellen Welt auszuführen (Kriterium 3) [WLK+16]. Hierfür muss der Prozessor jedoch mit dem entsprechenden Instruktionssatz in der virtuellen Simulation abgebildet sein (siehe Kapitel 2.5).

Zusätzlich zur Beobachtung des Prozessors und der Ausführung der Anwendung, können weitere Komponenten des Systems analysiert werden (Kriterium 5). Diese Beobachtungen schließen die einzelnen Peripheriebausteine, die Busse sowie die Speicher ein (siehe Abbildung 5.4). Dadurch können alle Vorgänge innerhalb des Prozessor-Systems analysiert werden (siehe Kapitel 6). Außerdem ist es möglich eine Analyse der ausgeführten Instruktionen innerhalb des Prozessors durchzuführen, wodurch die Instruktionen während der Laufzeit beobachtet und auf ihr Verhalten analysieren werden können (siehe Kapitel 6.5.4).

Durch die Virtualisierung des Systems kann die Simulation parallelisiert auf verschiedenen Computern ausgeführt werden (Kriterium 7). Da es sich um eine Simulation handelt, kann die Geschwindigkeit der Tests beliebig variiert werden, ohne dass Prozessor, Speicher und Busse asynchron ausgeführt werden (siehe Kapitel 6.6). Dadurch lässt sich die Ausführungszeit zusätzlich zur Parallelisierung beschleunigen. Plattformen mit über 1000 Prozessoren wurden effizient auf Desktop-PCs simuliert [138], wodurch der Test von verteilter Funktionalität ermöglicht wird (Kriterium 6). Die Ausführung eines typischen 32-Bit-Prozessors für Automotive [115] wird mit bis zu 500 MIPS[2] auf einem Desktop-PC ausgeführt und Spitzengeschwindigkeiten von mehreren Milliarden Anweisungen pro Sekunde wurden erreicht [138]. Im Antriebsstrang (engl. Powertrain) eines Automobils werden dabei Prozessoren mit einer Frequenz von bis zu 150 MIPS eingesetzt [147]. Bei Fahrassistenzsystemen (engl. Advanced Driver Assistance Systems (ADAS)) werden Steuergeräte mit mehreren Prozessoren und einer Frequenz bis zu 12.000 MIPS [158] eingesetzt (siehe

[2] Millionen Instruktionen pro Sekunde (MIPS)

Kapitel 2.1.1). Daher ist die Betrachtung der Skalierbarkeit und Performanz (Kriterium 7) anhand einer beispielhaften Applikation für die Ausführung von verteilter Funktionalität mittels virtuellen Steuergeräten gesondert zu untersuchen (siehe Kapitel 8).

Die virtuellen Modelle zur Simulation der ausführbaren Plattform (Ziel-Hardware) oder des gesamten Steuergeräts können bereits frühzeitig in der Entwicklung angefertigt und bei Änderungen angepasst werden (Kriterium 8). Um die Modelle zu generieren ist anfangs eine Spezifikation nötig, um ein grobes Modell zu erstellen. Während der Entwicklung werden die Modelle dann mit den verbauten Komponenten und jeweiligen Änderungen der Spezifikation angepasst.

## 5.7 Eignung der Testmethoden zur Beobachtung interner Signale

Im Folgenden wird eine Bewertung der vorgestellten Methoden zur Beobachtung der internen Signale beim Test der Datensicherheit durchgeführt. Diese Bewertung beruht auf den in Kapitel 5.2 hergeleiteten Kriterien. Diese Kriterien sind nachstehend noch einmal zusammenfassend aufgelistet:

1. Beobachtung der Anwendung
2. Beobachtung der internen Signale
3. Code muss auf Ziel-Hardware lauffähig sein
4. Keine speziellen Instruktionen im Code
5. Beobachtung der Peripherie und des Speichers
6. Testen eines Steuergeräteverbunds
7. Effizienz der Testausführung
8. Zeitpunkt und Anpassung der Plattform

Die Beobachtung und Analyse der Applikation ist mit allen vorgestellten Testmethoden möglich (1). Jedoch ist die Applikation nur bei der Ausführung auf der Zielplattform oder der Simulation mit virtuellen Plattformen direkt auf das Steuergerät übertragbar (3). Die Speicher und Peripherie können jedoch nur beobachtet werden, wenn diese in einer virtuellen Plattform simuliert oder an der Ziel-Plattform eine Debug-Schnittstelle angeschlossen ist. Eine Überwachung der Peripherie ist bei der Ziel-Hardware nur über die Kontrolle des

“Test and Debug Interface“ möglich, wohingegen eine virtuelle Plattform detaillierte Einblicke in jegliche Peripherie, Busse und Speicher bietet (5), ohne zusätzliche Software in die Applikation einzubringen. Bei einer Ausführung der Anwendung durch Simulation auf einem Computer oder durch leistungsstarke Hardware werden hingegen zusätzliche Schritte bis zur Ausführung auf der Ziel-Hardware benötigt (4). Beim Einsatz von virtuellen Plattformen (Prozessorsystems, inkl. Instruktionssatz) und einer instruktionsakkuraten Simulation kann die Anwendung direkt auf die Ziel-Hardware übertragen werden [WLBS16], ebenso beim Einsatz der Ziel-Hardware selbst.

Da die beiden Testmethoden, Simulation auf einem Computer und leistungsstarker Hardware, verschiedene Instruktionssätze wie das Zielsystem haben, müssen diese im weiteren Verlauf angepasst werden (4). Dadurch ist keine Aussage über Speicher- und numerische Fehler möglich. Virtuelle Plattformen besitzen Debug-Schnittstellen in der Simulation, wodurch interne Zustände überprüft werden können (2), ohne zusätzlichen Code in die Anwendung zu integrieren. Zur Ausführung der Tests auf der Ziel-Plattform wird hingegen zusammen mit der Debug-Hardware zusätzlicher Code benötigt, um das Stoppen und Überwachen des Prozessors zu kontrollieren.

Da virtuelle Plattformen und die Simulation auf einem Computer aus einer Simulation bestehen, können diese ohne Mehraufwand parallelisiert und sogar beschleunigt ausgeführt werden (7). Tests auf Hardware-Plattformen (Ziel-Hardware und leistungsstarke Hardware) sind hingegen immer an den vorhandenen Takt gekoppelt, was ein parallelisiertes oder beschleunigtes Ausführen nicht ermöglicht. Für eine parallelisierte Ausführung müssen mehrere Hardware-Plattformen zur Verfügung stehen oder für die Beschleunigung der Takterzeuger ausgetauscht werden.

Simulationen auf dem Computer können bereits früh in der Entwicklung beginnen, da hierfür lediglich die Softwaremodule bestehen müssen (8). Leistungsstarke Hardware und virtuelle Plattformen werden in den frühen Entwicklungsphasen erstellt und laufend an die reale Hardware angepasst. Beides bildet das reale System jedoch nur hinreichend genau ab. Die Zielplattform existiert erst nach der Hardwareentwicklung. Für Änderungen an der Hardware muss diese erneut gefertigt werden.

Tabelle 5.1 zeigt eine Zusammenfassung der diskutierten Auswertung anhand der definierten Kriterien.

Tabelle 5.1: Übersicht zur Auswertung der Kriterien aus Kapitel 5.2

| | Simulation auf Computer | Leistungsstarke Hardware | Ziel-Hardware | Virtuelle Plattform |
|---|---|---|---|---|
| 1. Beobachtung der Anwendung | + | + | + | + |
| 2. Beobachtung der internen Signale | - | - | 0 | + |
| 3. Übertragbarer Code | - | - | + | 0/+ |
| 4. Spezielle Instruktionen | - | - | -/0 | + |
| 5. Beobachtung der Peripherie | - | + | 0 | + |
| 6. Verbundstest | - | + | + | 0 |
| 7. Effizienz & Skalierung | + | + | - | + |
| 8. Zeitpunkt & Anpassung | + | 0 | - | 0 |

## 5.8 Zusammenfassung

Neben der Ausführung der Applikation ist für die Tests der Datensicherheit eine Beobachtung der internen Signale und Peripherien nötig. Falls die Anwendung nicht auf der Ziel-Plattform ausgeführt wird, ist zusätzlich auf die Übertragbarkeit ohne Anpassung zu achten.

Für das Testen von internen Zuständen der Software können dabei verschiedene Testmethoden eingesetzt werden. Die Anwendungen lassen sich jedoch nach den Tests nur bei virtuellen Plattformen oder den Tests auf der Ziel-Hardware, ohne Anpassung auf das spätere Steuergerät übertragen. Die Simulationen auf dem Computer und die Ausführung auf einer leistungsstarken Hardware arbeiten mit verschiedenen Instruktionssätzen, wodurch eine Anpassung der Anwendung nötig ist. Eine instruktionsgenaue Beobachtung ist für den Tests der Datensicherheit jedoch erforderlich (siehe Kapitel 4).

Die Auswertung anhand der Kriterien ergibt, dass neben der Ausführung auf der Ziel-Plattform ebenso die Simulation mit einer virtuelle Plattformen geeignet ist. Einzige Voraussetzung dafür ist jedoch, dass der getestete Code direkt von der virtuellen Plattform auf die Hardware übertragbar ist. Beim Einsatz der Ziel-Hardware ist darauf zu achten, dass eine spezielle Debug-Schnittstelle mit externer Hardware vorhanden sein muss, wodurch eine Anpassung des Codes zum Eingriff in den Ablauf benötigt wird.

Im Unterschied zur Nutzung eines Watchdogs zur Detektion von Fehlverhalten auf der Zielplattform, kann die Beobachtung durch die Simulation eines

virtuellen Steuergeräts passiv erfolgen. Der Watchdog hingegen muss zyklisch neugestartet werden. Zudem kann ein Watchdog nur überwachen, ob die Software läuft und regelmäßig den Timer neu startet, die Beobachtungspunkte des virtuellen Steuergeräts greifen zusätzlich auf Hardware Entitäten (Speicher, ALU, etc.) zu und können damit zusätzliche Funktionalität beobachten .

Zusätzlich bieten virtuelle Plattformen im Vergleich zu den weiteren Testmethoden die Möglichkeit Peripherie und Speicher direkt zu überwachen und den Test der Anwendung parallelisiert zu gestalten.

# 6 Testmethodik für Datensicherheit

Für den Test der Datensicherheit muss das ganzheitliche System untersucht werden. Daher wird im Folgenden eine Testmethodik vorgestellt, die virtuelle Steuergeräte nutzt, um die Kriterien aus Kapitel 5 zu erfüllen. Hierfür werden neben einer Systemsimulation (siehe Definition 23) und einer Co-Simulation (siehe Definition 21), ebenfalls Testmethoden benötigt, die über Beobachtungspunkte auf das Verhalten des virtuellen Steuergeräts zugreifen (siehe Abbildung 6.1).

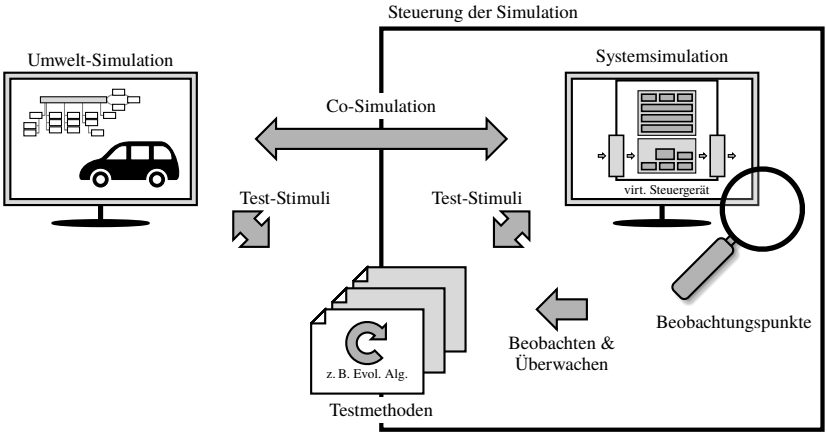



Abbildung 6.1: Einbindung der Methoden in die Gesamtsimulation durch Beobachtungspunkte und Co-Simulation

Die Methodik zum Test der Datensicherheit ist unterteilt in eine Systemsimulation des virtuellen Steuergeräts (siehe Kapitel 6.1), in eine Co-Simulation für die Simulation der Umwelt des Steuergeräts (siehe Kapitel 6.2), in Beobachtungspunkte (siehe Kapitel 6.3), um das Verhalten der Applikation und interne Zustände zu überwachen und in Testmethoden zur Erzeugung der Test-Stimuli

(siehe Kapitel 6.4). Eine Steuerung der Simulation ergänzt das Testsystem und kontrolliert die Abläufe (siehe Kapitel 6.5).

## 6.1 Simulation der Applikation auf einem virtuellen Steuergerät

Der Fokus von virtuellen Steuergeräten liegt darauf, die Entwicklung von eingebetteter Software zu beschleunigen. Durch eine Instruction Set Simulation (ISS) (siehe Kapitel 2.5) lässt sich das Verhalten einer Applikation (steuerndes Artefakt) instruktionsgenau untersuchen, ohne diese auf einer realen Hardware (ausführendes Artefakt) auszuführen. Hierfür werden entsprechende Modelle des ausführenden Artefakts (Prozessor, Speicher und Peripherie) benötigt, um das Verhalten hinreichend genau nachzubilden. Zusätzlich werden Modelle in der Umwelt-Simulation für das Verhalten der externen Komponenten (Artefakte außerhalb des Prozessorsystems, siehe Kapitel 2.1), mit denen die Software kommuniziert, benötigt und über eine Co-Simulation (siehe Kapitel 6.2) simuliert.

Die wesentliche Voraussetzung für den Test mit virtuellen Steuergeräten ist, dass es möglich ist, die kompilierte Applikation unverändert auf realem und virtuellem Steuergerät zu betreiben. Das bedeutet, dass die Simulation vollständig instruktionsakurat (siehe Definition 25) sein muss, dass die Prozessoren mit allen Interrupts und dem Befehlssatz modelliert wurden sowie für die Peripheriegeräte entsprechende Verhaltensmodelle der Funktionalität bereitgestellt sind. Da Steuergeräte aus Mehrprozessorsystemen bestehen können, muss die Simulation mehrere Prozessoren sowie deren Kommunikation untereinander ebenfalls abbilden können.

Beim Testprozess der Softwareentwicklung ist es nicht notwendig, die gesamte Funktionalität aller Bestandteile (gesamter Steuergeräteverbund des Fahrzeugs, inkl. Sensoren und Aktoren) als Modell bereitzustellen. Wichtig ist, dass die Funktionalität der Komponenten, die für den jeweiligen Test benötigt werden in der Simulation existieren. Für das Beispiel der Gesamtfunktion des ACC sind ein Sensor (z. B. Radar), ein Steuergerät zur Berechnung der Funktion (ACC), ein Aktor zur Ausgabe der Geschwindigkeit (z. B. Motorsteuergerät) und ein Display zur Fahrerinformation nötig (siehe Abbildung 6.2). Wird nur

die Funktionalität des ACC ohne Gesamtsystem getestet, ist es ausreichend das ACC-Steuergerät zu modellieren und die weiteren Komponenten über eine Co-Simulation abzubilden.

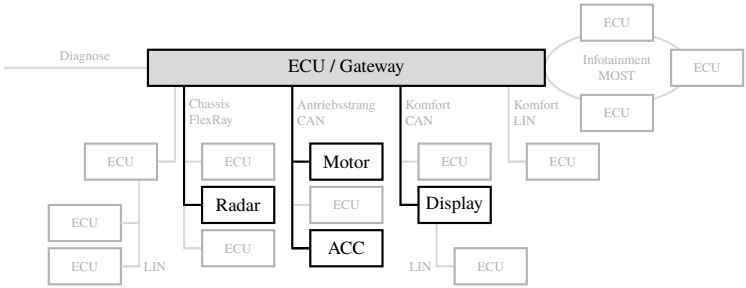



Abbildung 6.2: Exemplarischer Aufbau einer E/E-Architektur zur Simulation eines ACC

Um für den Tests der Datensicherheit das Gesamtsystem zu simulieren, muss neben der Prozessorplattform ebenfalls die Umgebung betrachtet werden. Hierfür wird das Verhalten entweder statisch vorgegeben oder die Umgebung über sogenannte Co-Simulationen nachgebildet. Diese Co-Simulationen werden parallel zur Simulation des Prozessorsystems ausgeführt und über Schnittstellen an die Peripherie angebunden (siehe Kapitel 6.2).

## 6.2 Co-Simulation der Umwelt und der Peripherie

Die ausgeführte Applikation wird in einem eingebetteten System (siehe Kapitel 2.6.1) ausgeführt. Dazu reagiert das System auf Eingaben und regelt oder steuert einen Ausgang. Bezogen auf ein ACC liest die Applikation Distanz und Geschwindigkeit anderer Verkehrsteilnehmer über einen Sensor ein (Eingaben) und regelt die eigene Fahrzeuggeschwindigkeit (Ausgabe), um einen definierten Mindestabstand einzuhalten. Für die Tests müssen Stimuli aus den Testfällen (siehe Kapitel 2.6.5) an das Steuergerät übergeben werden. Um ein System mit Ein- und Ausgaben überwachen zu können, müssen neben der Simulation des Prozessorsystems zur Ausführung der Applikation Testdaten (Stimuli) eingelesen werden. Diese Stimuli werden an die Peripherie übertragen, um sie anschließend in der Applikation zu verarbeiten. Hierfür werden Artefakte außerhalb des Steuergeräts durch statische Eingaben (Text-

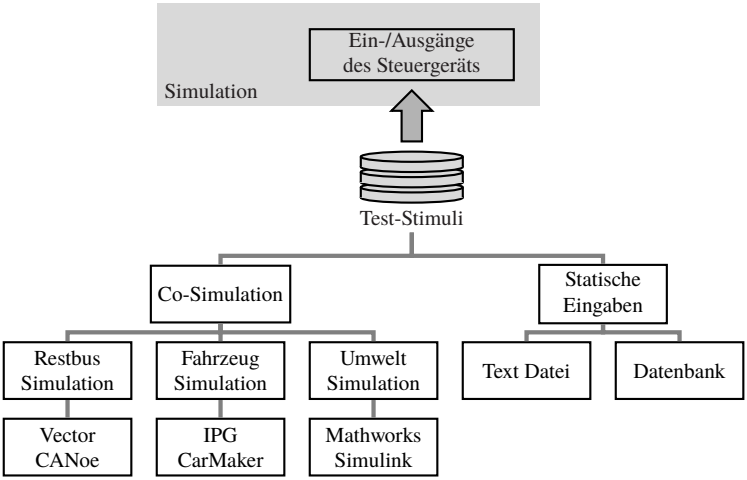



Abbildung 6.3: Kategorisierung der Test-Stimuli für die Simulation

dateien, Datenbanken) oder eine Co-Simulation (Simulation anderer Tools) in die virtuelle Plattform eingespeist (siehe Abbildung 6.3 für eine kategorisierte Darstellung).

Durch Test-Stimuli werden die Schnittstellen des Systems mit zuvor definierten Werten versehen, um den Verlauf der Anwendung zu analysieren. Abbildung 6.3 unterteilt die möglichen Test-Stimuli, die für die virtuelle Plattform berücksichtigt werden. Statische Eingaben sind Daten, die in Textdateien oder Datenbanken gespeichert sind, ausgelesen und sequenziell an das virtuelle Steuergerät übergeben werden. Eine Regelung der Stimuli findet in diesem Fall nicht statt. Im Gegensatz hierzu können Co-Simulatoren auf die Ausgaben der Steuergeräteschnittstellen reagieren und neue Stimuli anhand des Steuergerätverhaltens generieren. Durch diese Rückkopplung können Simulatoren für den Restbus, das Fahrzeug oder die Umwelt in einer geschlossenen Regelschleife betrieben werden.

Innerhalb der Simulation erfolgt eine Übergabe der Stimuli an die Peripherie, die diese verarbeitet und der Applikation zur Verfügung stellt. Durch die Simulation der Peripherie werden in der Applikation die selben Schnittstellen verwendet wie auf dem realen Steuergerät.

Durch die Anbindung einer Co-Simulation, kann die Simulation an beliebigen Stellen aufgetrennt und die Simulation an die jeweilige benötigte Genauigkeit durch Anbindung verschiedener Simulatoren erhöht werden. So kann die Umwelt beispielsweise mit Fahrzeug-Simulatoren oder Restbus-Simulatoren dargestellt und die erzeugten Stimuli in das virtuelle Steuergerät eingespielt werden. Durch eine Rückkopplung der Ausgangsperipherie wird ein geschlossener Regelkreis aufgebaut.

## 6.3 Beobachtungspunkte zur Überwachung des virtuellen Steuergeräts

Neben der Eingabe der Stimuli ist für die Tests eine Beobachtungsmöglichkeit der Ausgaben der Anwendung wichtig (siehe Kapitel 5.2 Kriterium 1). Zusätzlich muss eine Möglichkeit zur Beobachtung und Analyse der Software-Module, internen Signale, ausgeführten Instruktionen, Speicherzugriffe und Zugriffe auf die Peripherie des Steuergeräts gegeben sein (siehe Kapitel 5.2 Kriterium 2).

### 6.3.1 Beobachtungspunkte zur Überwachung der Datensicherheit

Für die Informationsgewinnung aus dem steuernden und ausführendem Artefakt werden Beobachtungspunkte eingeführt (siehe Definition 33). Generell wird beim steuernden Artefakt zwischen der System- und Anwendungssoftware (Applikation) unterschieden (siehe Abbildung 6.4). Beide Einheiten können dabei in mehrfacher Ausführung innerhalb des zu steuernden Artefakts vorhanden sein.

Die Systemsoftware stellt dabei die grundlegende Funktionalität bereit. Hierzu zählen der Bootloader, Betriebssystem oder Basissoftware und eine Laufzeitumgebung (Runtime Environment). Die Anwendungssoftware beschreibt das funktionale Verhalten des Steuergeräts gemäß den spezifizierten Anforderungen (siehe Kapitel 2.1.1).

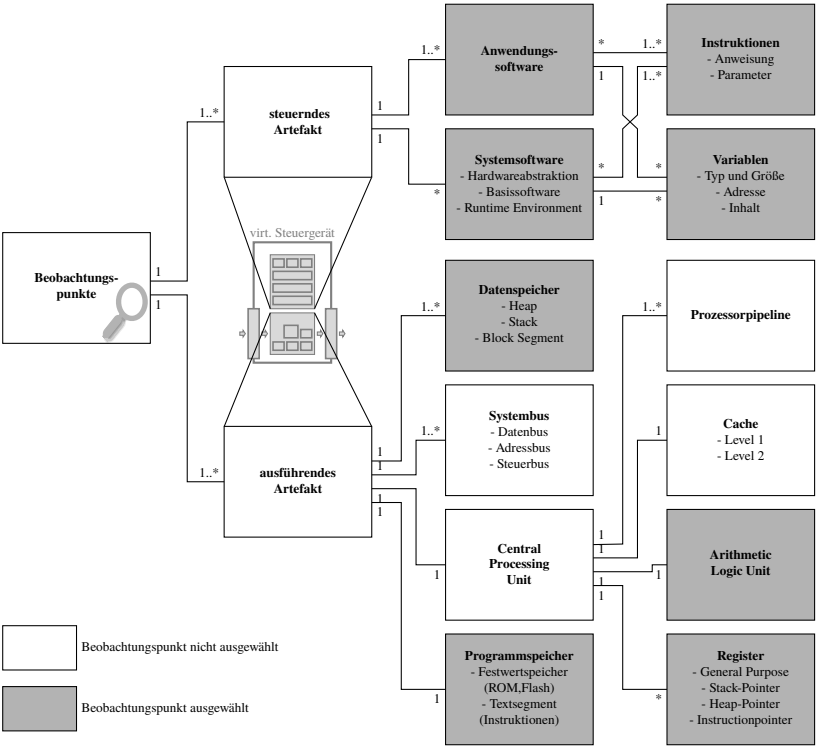



Abbildung 6.4: Kategorisierung von Beobachtungspunkten eines virtuellen Steuergeräts aus Abbildung 2.2

Beide Softwarearten bestehen dabei aus Instruktionen (Anweisung und Parameter), die ausgeführt werden und Variablen (Speicherbereiche). Instruktionen werden über einen Systembus aus dem Programmspeicher in die Central Processing Unit (CPU) geladen und ausgeführt. Hierbei muss beobachtet werden, welche Parameter übergeben wurden, ob bei Ausführung der Instruktion Fehler durch falsche Ausführung entstehen (z. B. Division durch Null) oder diese schreibend auf Speicherbereiche zugreifen (siehe Kapitel 5.1). Variablen speichern den Inhalt von Berechnungen ab. Hierfür besteht eine Variable aus Typ, Größe, Adresse und dem Inhalt der Variablen. Mit Typ, Größe und Adresse wird ein definierter Speicherbereich erzeugt, der nur für die entsprechende Va-

riable gültig ist, das heißt, von anderen Speicherzugriffen nicht überschrieben werden darf.

Die ausführende Hardware beschreibt die Elemente, die für das Ausführen der Software notwendig sind und beinhaltet Speicherbereiche für Daten und Programm sowie eine CPU. Die CPU führt die Instruktionen der Software in einer Arithmetic Logic Unit (ALU) aus. Mögliche Ausführungseinheiten sind dabei Integer, Floating Point und Fixed Point Unit. Zur ALU werden die Daten, Adressen und Steuersignale mittels prozessorinternen Bussystemen übertragen. Zudem speichert die CPU Daten in entsprechenden General Purpose Register (GPR). In einem Instruktionpointer wird die Adresse der nächsten auszuführenden Instruktion und in Heap-/Stack-Pointern die Adressen des letzten Heap[1] und Stack[2] Wertes gespeichert.

Im steuernden Artefakt wurden aus Abbildung 6.4 die Beobachtungspunkte für Anwendungs- und Systemsoftware zum einen anhand des Quellcode, zum anderen anhand des Maschinencodes (Unterscheidung siehe Kapitel 2.6.5) gewählt:

a) Variablen im Quellcode der Anwendungs- und Systemsoftware
b) Instruktionen im Quellcode der Anwendungs- und Systemsoftware
c) Variablen im Maschinencode der Anwendungs- und Systemsoftware
d) Instruktionen im Maschinencode der Anwendungs- und Systemsoftware

Im ausführende Artefakt werden folgende Beobachtungspunkte gewählt:

e) Register der CPU
f) Arithmetic Logic Unit der CPU
g) Daten- und Programmspeicher

Um die Beobachtungspunkte und damit die spätere Methodik gering zu halten, wird auf folgende Beobachtungspunkte verzichtet (siehe auch Abbildung 6.4), die bereits durch andere Punkte abgedeckt sind. Der Cache des ausführenden Artefakts wird im Folgenden nicht betrachtet, da ein temporäres Speichern im

[1] Auf dem Heap werden zur Laufzeit Speicherbereiche allokiert, und in beliebiger Reihenfolge wieder freigegeben. Der Heap dient als Speicherplatz für dynamische Variablen.

[2] Auf dem Stack zur Laufzeit neue Elemente gespeichert und in umgekehrter Reihenfolge wieder freigegeben (LIFO = Last In First Out). Der Stack dient als Speicherplatz für Rücksprungadressen, Übergabeparameter und lokale Variablen.

Cache bereits über die Beobachtung der Variablen abgedeckt ist. Ebenso wird die Summe der Einzelschritte der Prozessorpipeline über die Beobachtung der Instruktionen abgedeckt. Die Systembusse verbinden die Bestandteile der CPU mit weiteren Teilen des ausführenden Artefakts. Die kommunizierten Daten werden in der ALU verarbeitet oder im Speicher abgelegt. Da beides als Beobachtungspunkt ausgewählt ist, wird auf eine Beobachtung des Systembusses verzichtet. Würde man die vorgenannten Punkte ebenfalls ergänzen, so gibt sich keine Änderung an der Methodik. Daher können diese bei Bedarf ergänzt werden.

Alle in Abbildung 6.4 dargestellten Beobachtungspunkte können in einem virtuellen Steuergerät überwacht werden, nicht jedoch bei einer Simulation auf einem Computer oder in einem realen Steuergerät. Bei der Simulation auf einem Computer ist das ausführendes Artefakt nicht gleich der späteren Plattform im Steuergerät, wodurch sich Änderungen im ausführenden Artefakt und dem Befehlssatz (Instruktionen) ergeben. Bei einer Ausführung auf der realen Plattform können interne Strukturen wie Register oder ALU nicht beobachtet werden, da hierfür keine Testschnittstellen vorhanden sind (siehe Kapitel 5.5).

Zum Erkennen von Schwachstellen durch Überläufe im Speicher müssen einerseits die Variablen auf ihren Typ und Grenzen beobachtet werden (siehe Kapitel 5.1), andererseits muss das Verhältnis zwischen Heap und Stack überwacht werden, um ein Überlauf beider zu erkennen (siehe Kapitel 5.1). Für die Überwachung der Variablen ist der Zugriff auf den Speicher und die Lokalisierung der Variablen mit Typ und Länge aus dem Quellcode nötig. Um den Überlauf zwischen Heap und Stack zu erkennen, müssen prozessorinterne Register (Stack-, Heap- und Base-Pointer) und für numerische Rechenfehler der Zugriff auf Instruktionen zur Verfügung stehen. Besonders die Eigenschaften des steuernden Artefakts sind vom Quellcode abhängig.

### 6.3.2 Analyse des Quellcodes für das steuernde Artefakt

Für die Erzeugung der Überwachungsfunktionen (siehe Kapitel 6.4) werden Variablen und Funktionen der Applikation genutzt. Für das Finden der Funktionen und Variablen wird dabei zwischen drei Varianten unterschieden (siehe auch Kapitel 2.6.5):

1. Hochsprache

2. Maschinencode
3. Maschinencode mit Debug-Symbolen

Die drei Varianten unterscheiden sich nicht nur in der Darstellung der Variablen und Funktionen, sondern auch im Umfang der im Code enthaltenen Informationen. Während die Software in einer Hochsprache vom Menschen lesbar ist, sind keine Informationen über Speicherort der Variablen enthalten. Maschinencode ist hingegen nicht vom Menschen lesbar, dafür enthält er alle Instruktionen, die auf dem Prozessor ausgeführt werden und damit ebenfalls Informationen über Speicherort und Speicherplatz .

### 1. Hochsprache

(a) Funktionskopf und -aufruf

```
1  void doSomething (void) {
2  ...
3  }
4
5  ...
6  doSomething();
7  ...
```

(b) Variablendefinition

```
1  int   var1;
2  int   var2;
3  char  var3;
```

Abbildung 6.5: Funktionskopf, -aufruf und Variablendefinition in einer Hochsprache

In einer Hochsprache ist die Form einer Funktionen strukturiert (siehe Abbildung 6.5a). Diese beinhaltet im Funktionskopf den Namen, die Übergabeparameter und den Rückgabetyp. Der Funktionsaufruf erfolgt im weiteren Programm über den Namen der Funktion und die Übergabe der Parameter. Nachteilig an dieser Variante ist, dass keine Aussage über den Speicherort der Funktion getroffen werden kann, da dieser erst während dem Kompilieren vergeben wird. Mittels Ablaufgraphen kann dabei die Programmstruktur anhand des Quellcodes analysiert und Abhängigkeiten können gefunden werden.

Ebenso sind die Variablen über ihren Typen definiert (siehe Abbildung 6.5b). Eine Variablendefinition legt die Länge der Variablen fest, nicht jedoch den Speicherort. Um Variablen an einem spezifischen Ort abzuspeichern, sind Zeiger nötig, die jedoch nicht von allen Hochsprachen unterstützt werden.

### 2. Maschinencode

(a) Funktionskopf und -aufruf

```
1  0x0454   ...
2  0x0458   call 0x1234
3  0x045C   ...
4
5  0x1234   ...
6  ...
7  0x1258   ret
```

(b) Variablendefinition

```
1  0x0454   ...
2  0x0458   mov mem, 9
3  0x045C   ...
```

Abbildung 6.6: Funktionskopf, -aufruf und Variablendefinition im Maschinencode

Im Gegensatz zur Hochsprache sind im Maschinencode keine Namen und Variablentypen vorhanden. Um diesen wie in Abbildung 6.6 lesbar zu machen, muss der Maschinencode zuerst disassembliert werden (siehe Kapitel 2). Im Assemblercode werden die Funktionen dann über Sprungfunktionen (z. B. "`call`") aufgerufen. An der aufgerufenen Stelle wird die Rücksprungadresse abgespeichert und am Ende an diese Speicheradresse zurück gesprungen (siehe Abbildung 6.6a). Da Sprünge jedoch nicht nur bei Funktionen vorkommen, sondern ebenso bei Schleifen und dem Abfragen von Bedingungen, sind diese nicht eindeutig Funktionen zuzuordnen. Daher müssen weitere Kriterien, wie das Versetzen des Stack-Pointers beim Speichern der Rücksprungadresse oder das rekonstruieren des Kontrollflusses (Control Flow Graph Recovery (CFGR)) [21] herangezogen werden. Vorteil ist jedoch, dass der Maschinencode direkt auf der Zielplattform eingesetzt wird und für diese bereits kompiliert ist, wodurch die Speicheradressen der Funktionen bekannt sind.

Analog verhält es sich bei Variablen. Durch das Kompilieren für die Zielplattform sind die Speicherbereiche bekannt, jedoch gehen die Variablentypen verloren. Durch die Compiler-Optimierung wird zudem der Speicherplatz aller Variablen mit einem Befehl reserviert (siehe Abbildung 6.6b). Dadurch ist zwar der Speicherbereich, nicht aber die Anzahl und die Länge der einzelnen Variablen bekannt.

(a) Funktionskopf und -aufruf

```
0x0454   ...
0x0458   call doSomething
0x045C   ...

doSomething:
0x1234   ...
...
0x1258   ret
```

(b) Variablendefinition

```
(gdb) info variables
All defined variables:
...
0x0400610   var1
0x0400614   var2
0x0400618   char3
...
```

Abbildung 6.7: Funktionskopf, -aufruf und Variablendefinition im Maschinencode mit Debug-Symbolen

#### 3. Maschinencode mit Debug-Symbolen

Um die Vorteile beider Varianten zu verbinden, können während dem Kompilieren Zusatzinformationen (sog. Debug-Symbole) erzeugt werden. Diese Informationen beinhalten die Adressen der Funktionen und der einzelnen Variablen, ohne Zusatzcode in der Applikation zu erzeugen (siehe Abbildung 6.7). Diese Debug-Symbole werden stehen in der Regel zu Beginn oder Ende der kompilierten Anwendung, sodass diese Informationen nach den Tests gelöscht werden können, ohne die Applikation neu zu kompilieren.

### 6.3.3 Offenen Schnittstellen zur Beobachtung

Angriffe auf Steuergeräte erfolgen über die Schnittstellen zur Umgebung (siehe Kapitel 3.4). Diese können von einem Angreifer genutzt werden, um fehlerhaftes Verhalten des Steuergeräts hervorzurufen. Schnittstellen dienen jedoch auch dem Steuergerät zur Ein- und Ausgabe sowie der Kommunikation und können daher nicht pauschal vermieden werden. Im Beispiel des ACC muss der Abstand zum Vorderfahrzeug von einem Radarsensor gemessen, an eine Recheneinheit übertragen und eine Soll-Geschwindigkeit angesteuert werden. Eine Manipulation der dadurch entstehenden Schnittstellen zwischen den Einheiten kann ein Fehlverhalten zur Folge haben. Im Verlauf der Tests, muss analysiert werden, ob die Schnittstellen nach außen sichtbar sind (offene Schnittstellen). Unter der Annahme einer manipulierten Kommunikation auf dem Bussystem oder

der Manipulation eines anderen Steuergeräts gelten alle Schnittstellen die im Steuergerät genutzt werden als offen, unabhängig, ob diese an die Umwelt des Fahrzeugs angebunden sind (siehe Kapitel 2.1.3). Daher muss für jede Schnittstelle geprüft werden, ob Fehleingaben in der Software abgefangen sind und nicht zu Fehlverhalten führen. Daher ist es während der Tests wichtig, alle Schnittstellen zur Umgebung des Steuergeräts zu kennen und diese über Testfälle zu stimulieren.

Zur Erkennung der offenen Schnittstellen wird der Aufbau der virtuellen Plattform herangezogen. Innerhalb dieser ist das Peripherieverhalten für die Schnittstellen definiert. Durch diese Konfiguration werden vor Beginn der Simulation alle Schnittstellen aufgelistet. Damit sind innerhalb der virtuellen Umgebung alle modellierten Schnittstellen bekannt. Durch dieses Wissen wird mit Hilfe der kompilierten Software festgestellt, worauf die ausgeführte Applikation zugreift. Ist innerhalb der Applikation ein Zugriff auf eine Schnittstelle definiert, so wird diese zu den Beobachtungspunkten als offene Schnittstelle hinzugefügt. Anhand der offenen Schnittstellen wird während der Simulation überprüft, ob die Schnittstelle mit Testfällen stimuliert wurde, um eine Testabdeckung aller Schnittstellen zu gewährleisten.

Zur Erweiterung der Tests oder falls eine Schnittstelle nicht stimuliert wurde, können neue Testfälle durch Fuzz-Tests oder Datenbankeinträge generiert werden (siehe Kapitel 7.3).

### 6.3.4 Beobachtungspunkte im virtuellen Prozessorsystem

Um die Zustände der Beobachtungspunkte (siehe Kapitel 6.3.1) beim Testen überwachen zu können, müssen diese im Prozessor des virtuellen Steuergerät integriert werden.

Für das steuernde Artefakt wird die Applikation genutzt und als bare-metal oder mit Betriebssystem ausgeführt. Im steuernden Artefakt werden Daten (Variablen, siehe Kapitel 6.3.1 Punkt a)) und Programmcode (Instruktionen, siehe Kapitel 6.3.1 Punkt b)) beobachtet. Die Beobachtung erfolgt dabei für die Anwendungssoftware und für die Systemsoftware (Hardware-Abstraktion, Basissoftware, Runtime Environment). Variablen und Instruktionen können dabei sowohl im Quellcode als auch im Maschinencode vorliegen (siehe Kapitel 6.3.1 Punkt c) und Punkt d)).

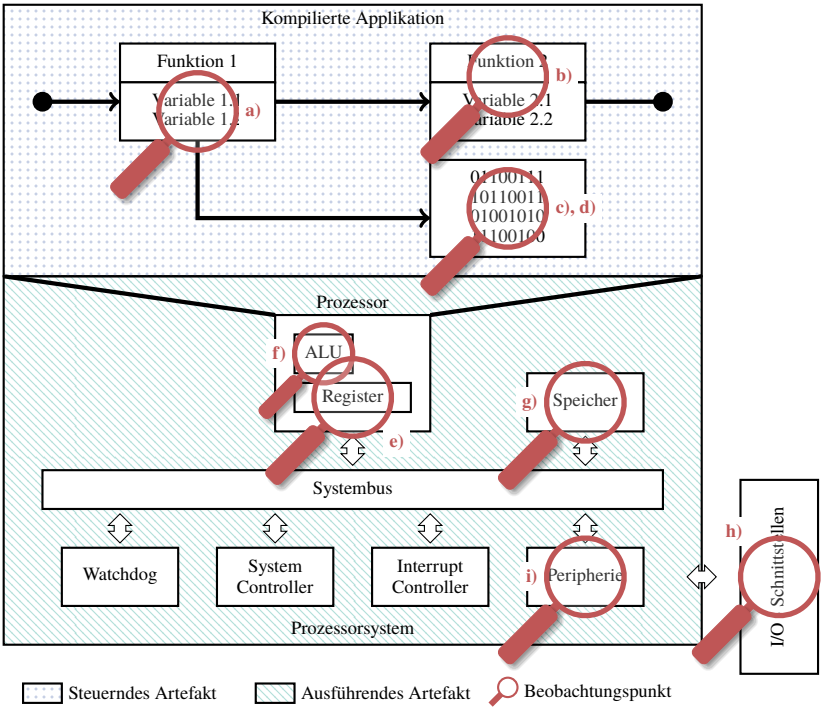



Abbildung 6.8: Übersicht der Beobachtungspunkte aus Kapitel 6.3.1 zur Testmethodik mit einer virtueller Plattform (siehe Kapitel 5.6) und Prozessorsimulation (sieh Kapitel 6.1)

Innerhalb des virtuellen Prozessormodells (virtualisierte CPU) werden die Register (siehe Kapitel 6.3.1 Punkt e)), die Arithmetic Logic Unit (ALU) (siehe Kapitel 6.3.1 Punkt f)) und der Speicher ((siehe Kapitel 6.3.1 Punkt g)) während der Laufzeit beobachtet. Die Register werden benötigt, um Beginn sowie Ende des Heap und Stack im Speicher zu erkennen (Heap- und Stack-Pointer, siehe Kapitel 2.4.5). In der ALU werden die Instruktionen als Rechenoperationen ausgeführt und die übergebenen Parameter verarbeitet[3].

[3] Für die Addition von 3 + 5 wird beispielsweise die Instruktion *addition* mit den Parametern 3 und 5 ausgeführt.

Zusätzlich zu den Beobachtungspunkten aus Kapitel 6.3.1 müssen die Schnittstellen beobachtet werden, da über sie die Ein- und Ausgaben erfolgen (siehe Kapitel 5.2 Kriterium 5). Innerhalb des Prozessorsystems sind die Schnittstellen an die Peripherie angeschlossen, weshalb diese ebenfalls zu den Beobachtungspunkten ergänzt wurden. Daher werden die Beobachtungspunkte erweitert mit:

h) I/O Schnittstellen
i) Prozessorperipherie

Das so entstehende Gesamtsystem inklusive aller Beobachtungspunkte ist in Abbildung 6.8 dargestellt.

Weitere Komponenten innerhalb des Prozessormodells sind Watchdog, Interrupt-Controller, Systembus, etc. und können in der Simulation ebenfalls zur Beobachtung eingebunden werden, sind jedoch für die vorgestellten Tests der Datensicherheit (siehe Kapitel 6.4) nicht notwendig. Um das Gesamtsystem minimal zu halten, werden im weiteren nur die in Abbildung 6.8 gezeigten Beobachtungspunkte betrachtet.

## 6.4 Testmethoden zur Überwachung der Datensicherheit

Aufbauend auf den Beobachtungspunkten (siehe Kapitel 6.3.1 und Kapitel 6.3.4) sowie der Simulation eines virtuellen Steuergeräts, können Testmethoden die Datensicherheit überwachen. Neben den Beobachtungsmöglichkeiten, die eine Simulation durch virtuelle Plattformen bietet, wird in der Testmethodik besonders auf eine Debug-Schnittstelle in der Hardware verzichtet. Stattdessen wird die Beobachtung der internen Parameter (Instruktionen, Peripherie, Speicher, etc.) durch die Simulation durchgeführt. Zusätzlich wird die Applikation in Form einer kompilierten Software auf dem virtuellen Prozessorsystem ausgeführt und das Verhalten mit allen internen Zuständen simuliert. Durch Kenntnisse des Quellcodes der Applikation können Rückschlüsse auf Variablen und Funktionen gezogen werden (siehe Kapitel 6.3.4). Aufbauend auf den Beobachtungspunkten ergeben sich neben den Methoden aus Kapitel 3.5.4 folgende statische Testmethoden zur Überwachung der Ausführung der Applikation auf dem virtuellen Steuergerät:

i Überwachung der Instruktionen
ii Überwachung der Register
iii Analyse der Eingaben
iv Markierung des genutzten Speichers
v Analysemöglichkeiten der Codeabdeckung

Die Testmethoden nutzten dabei verschiedene Beobachtungspunkte. Eine Verknüpfung zwischen Methoden und Beobachtungspunkten wird in Kapitel 6.4.1 vorgestellt.

Durch den zusätzlichen Einsatz von evolutionären Algorithmen werden Tests als Optimierungsproblem angesehen und durch Mutation und Rekombination die Stimuli solange variiert, bis das spezifizierte Verhalten nicht mehr gegeben ist (siehe Kapitel 3.2).

Bezieht man das evolutionäre Testen der Datensicherheit auf ein virtuelles Steuergerät und der darin enthaltenen Beobachtungspunkte (siehe Kapitel 6.3.1), so können dynamische Methoden, zusätzlich zur Analyse der Beobachtungspunkte, Eingaben generieren, um ein gezieltes Fehlverhalten herbeizuführen. Dynamische Methoden für ein evolutionäres Vorgehen sind:

vi Maximierung des Variablenwachstums
vii Maximierung des Speicherzugriffs
viii Minimierung des Abstands zwischen Heap und Stack
ix Minimierung des Variableninhalts
x Maximierung des Variableninhalts

#### i) Überwachung der Instruktionen

Bei der Überwachung der Instruktionen müssen die Anweisungen und Parameter im Prozessor beobachtet werden. Diese werden dann zur Laufzeit auf Schwachstellen (siehe Kapitel 4.1) untersucht. Für die Detektion der Rechenfehler (z. B. Integer Overflows und Nulldivisionen, siehe Kapitel 4.1) werden die Instruktionen analysiert. Hierzu wird die Anweisung geladen und die Parameter der Anweisung im jeweiligen Kontext auf Gültigkeit geprüft. Um Nulldivisionen zu erkennen, darf der Divisor einer Anweisung nicht Null sein. Integer Overflows werden erkannt, falls das Ergebnis einer Addition kleiner als die jeweiligen Summanden ist.

Aufgrund der Tatsache, dass Speicherfehler nur im Speicher detektierbar sind, kann die Überwachung der Instruktionen nicht für die Aufdeckung von Speicherfehlern genutzt werden. Ausnahme hierzu ist der Zugriff auf nicht initialisierte Speicherbereiche, solange der allokierte Speicher protokolliert und mit dem Parametern der Instruktion abgeglichen wird.

#### ii) Überwachung der Register

Die Überwachung der Register ist eine Erweiterung zur Überwachung der Instruktionen. Zusätzlich zu den Instruktionen werden die allgemein verfügbaren Register (General Purpose Register (GPR)) analysiert. Diese Überwachung ist nötig, da Parameter von Instruktionen nicht immer als absoluter Wert angegeben sind, sondern in GPR ausgelagert werden können.

Ähnlich wie bei der Überwachung der Instruktionen werden numerische Rechenfehler (siehe Kapitel 4.1) durch Überwachung der Register entdeckt. Anhand des Inhalts der Register können ungültige Divisionen und Überläufe erkannt werden.

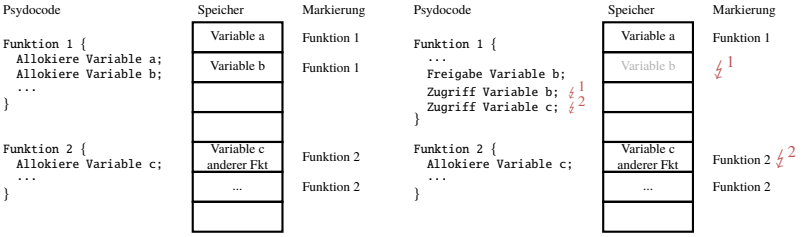
#### iii) Analyse der Eingaben

Die Analyse der Eingaben untersucht Bit-Sequenzen innerhalb der Stimuli. Dies erfolgt entweder direkt an der Schnittstelle zur simulierten Umwelt oder innerhalb der Peripherie des Steuergeräts. Dadurch können Injection Fehler (z. B. Code Injection, siehe Kapitel 2.4.6) und numerische Fehler (z. B. falsch formatierte Eingaben und falsche Datentypen, siehe Kapitel 4.1) aufgedeckt werden. Zum Erkennen müssen die ungültigen Sequenzen jedoch bekannt und entsprechend definiert sein.

#### iv) Markierung des genutzten Speichers

Die Speichermarkierung kennzeichnet verwendete Speicherbereiche und schützt diese vor Zugriff. Hierfür wird während der Applikationslaufzeit der Speicherzugriff analysiert und bei Allokation als genutzter Speicher markiert (siehe Abbildung 6.9a). Anschließend darf dieser Speicherbereich nicht erneut al-

lokiert werden und freie Speicherbereiche nicht beschrieben werden. Wird die Markierung mit der allokierenden Funktion verknüpft, so darf nur diese Funktion in den Speicherbereich schreiben (siehe Abbildung 6.9b).




(a) Markierung des Speichers bei Allokierung

(b) Zugriff auf markierten Speicher

Abbildung 6.9: Markierung des allokierten Speichers zur Überwachung von Zugriffen

Das Beobachten des Arbeitsspeichers erlaubt zudem eine Aussage über Memory Exhaustion (siehe Kapitel 4.1) als Maximum der markierten Speicherbereiche. Dadurch kann Memory Exhaustion erkannt werden. Eine Memory Exhaustion allokiert Speicher, ohne diesen wieder freizugeben. Dadurch werden Ressourcen belegt und Funktionen an der Ausführung gehindert.

Eine wichtige Erweiterung dieses Verfahrens ist der temporale Aspekt (siehe Abbildung 6.9b), da Speicherbereiche zur Laufzeit freigegeben werden können. Hiermit können beim Zugriff auf nichtmarkierte Bereiche des Stacks “Use After Scope“ und “Use After Return“ sowie auf dem Heap “Use After Free“ und “Double Free“ erkannt werden. Durch das Überwachen der lesenden und schreibenden Befehle können auch Race Conditions detektiert werden, indem Zugriffe auf das gleiche markierte Byte innerhalb kurzer Zeit erkannt werden.

#### v) Analysemöglichkeiten der Codeabdeckung

Während den Tests werden alle ausgeführten Befehle und Speicherzugriffe protokolliert. Dies ermöglicht es festzustellen, wie häufig Codezeilen ausgeführt oder Speicherinhalte geschrieben oder gelesen wurden. Mithilfe dieser Parameter lässt sich die Güte der Tests bewerten (siehe Kapitel 3.5.4).

Durch das Protokollieren der ausgeführten Codezeilen, kann folgendes überprüft werden [76]:

- Funktionsumfang: Wurde jede Funktion oder jedes Unterprogramm im Programm aufgerufen?
- Anweisungsabdeckung: Wurde jede Anweisung im Programm ausgeführt?
- Zweigabdeckung: Wurde jeder Zweig jeder Kontrollstruktur (z.B. in if- und case-Anweisungen) ausgeführt?
- Bedingungsabdeckung (oder Prädikatabdeckung): Wurde jeder boolesche Ausdruck sowohl mit wahr als auch mit falsch bewertet?
- Pfadabdeckung: Wurde jede mögliche Route durch einen bestimmten Teil des Codes ausgeführt?
- Entry/Exit Abdeckung: Wurde jeder mögliche Aufruf und Rücksprung der Funktion ausgeführt?
- Schleifenabdeckung: Wurde jede mögliche Schleife nullmal, einmal und mehrmals ausgeführt?

Zusätzlich wird die Anzahl der Aufrufe protokolliert, wodurch Rückschlüsse auf die Güte der Testfälle möglich sind. Beispielsweise lässt sich erkennen, ob Funktionen nur vereinzelt durchlaufen wurden und hierfür spezielle Testfälle generiert werden müssen.

#### vi) Maximierung des Variablenwachstums

Durch Variablen werden Speicherbereiche allokiert. Wächst dieser Bereich zu stark an, können Memory Exhaustion und Overflows (siehe Kapitel 2.4.5) die Folge sein. Durch ein evolutionäres Vorgehen werden Zielfunktionen aufgestellt, um die Stimuli durch Mutation und Rekombination anzupassen und damit ein Fehlverhalten herbeizuführen. Um eine Memory Exhaustion oder Overflow zu erzeugen, variiert der Optimierungsalgorithmus die Stimuli in der Art, dass der Speicherplatz aller Variablen maximiert wird (siehe Gleichung 6.1).

$$Zielfunktion = MAX\left(\sum_{i=1}^{n} Speicherplatz_{Variable,i}\right) \tag{6.1}$$

Um das Wachstum einzelner Variablen zu verfolgen, müssen Beobachtungspunkte auf die Variablengröße und den Speicherort vorhanden sein. Das Beobachten der Instruktionen reduziert den Aufwands des evolutionären Vorgehens, da Instruktionen einen Hinweis geben, wann eine Variable geschrieben wird und der Optimierungsalgorithmus nur dann die Zielfunktion auswertet.

#### vii) Maximierung des Speicherzugriffs

Analog zur Maximierung des Speicherwachstums können durch Maximierung des Schreibzugriffs Buffer-Overflows (siehe Kapitel 2.4.5) gefunden werden. Hierzu versucht der evolutionäre Algorithmus die Stimuli zu mutieren, um möglichst viele Daten am Stück in den Speicher zu schreiben (siehe Gleichung 6.2).

$$Zielfunktion = MAX\,(Adresse_{Ende} - Adresse_{Start}) \tag{6.2}$$

Der zusammenhängende Speicherzugriff sucht nach der längsten schreibenden Sequenz innerhalb des Speichers und versucht durch Maximierung einen Overflow (siehe Kapitel 2.4.5) zu erzeugen. Der Overflow wird in diesem Verfahren dadurch erzeugt, dass angenommen wird, dass alle Variablen eine begrenzte Länge besitzen und daher das Schreiben von langen Sequenzen den Überlauf in benachbarte Speicherbereiche ermöglicht. Instruktionen bieten die Möglichkeit den Aufwand zu reduzieren, indem nur schreibende Zugriffe im Speicher beobachtet werden.

#### viii) Minimierung des Abstands zwischen Heap und Stack

Der Vergleich von Stack und Heap zielt darauf ab, beide Segmente aufeinander wachsen zu lassen[4] (siehe Abbildung 6.10). Hierfür werden Heap und Stack beobachtet und sofern ein Bereich den anderen überschneidet, ist ein Stack oder Heap Overflow (siehe Kapitel 2.4.5) aufgetreten. Der Zugriff auf Stack

[4] Der Heap wächst durch Allokation von dynamischen Variablen, der Stack wird durch lokale Variablen und Rücksprungadressen vergrößert. Heap und Stack wachsen dabei im Speicher aufeinander zu, wodurch Überläufe entstehen können.

und Heap im Speicher sowie der Zugriff auf Stack- und Heap-Pointer müssen vorhanden sein.

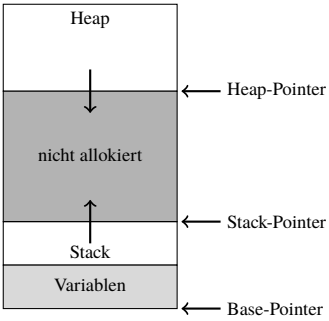



Abbildung 6.10: Overflow durch Minimierung des nicht allokierten Speichers zwischen Heap und Stack (siehe Abbildung 2.7)

Fasst man den Speicherabstand zwischen Stack und Heap als Optimierungsproblem auf, können durch evolutionäre Verfahren die Stimuli so mutiert und rekombiniert werden, dass der Abstand minimal wird (siehe Gleichung 6.3). Eine Überscheidung der Speicherbereiche zeigt anschließend ein Stack- oder Heap-Overflow als Schwachstelle an.

$$Zielfunktion = MIN\,(StackPointer - HeapPointer) \tag{6.3}$$

Ist der Heap-Pointer im verwendeten Prozessor nicht vorhanden, kann dieser aus Base-Pointer (siehe Kapitel 2.4.5) und geschriebenen Daten im Heap rekonstruiert werden.

#### ix) Minimierung des Variableninhalts

Um Rechenfehler herbeizuführen, wird der Inhalt der überwachten Variablen minimiert. Anhand der beobachteten Instruktionen wird anschließend erkannt, zu welchem Zeitschritt Divisionen durchgeführt werden. Durch ein evolutionäres Vorgehen werden die Variablen zu diesem Zeitschritt minimiert, bis sich die

Variable als Null ergibt (siehe Gleichung 6.4) und dadurch eine Nulldivision erzwungen wird.

$$Zielfunktion = Variable_i = 0 \quad \forall i \tag{6.4}$$

Wird die Variable evolutionär weiter reduziert, erfolgt ein Überlauf vom positiven Zahlenraum in den negativen Zahlenraum. Dadurch entstehende Rechenfehler werden anschließend erkannt und protokolliert. Als Zielfunktion ergibt sich Gleichung 6.5

$$Zielfunktion = MIN\,(Variable_i) \quad \forall i \tag{6.5}$$

#### x) Maximierung des Variableninhalts

Analog zum Vorgehen aus Methode x wird der Inhalt der Variablen verändert. Eine Erhöhung des Inhalts führt dazu, dass die Variable vom Maximalwert überläuft und wieder beim kleinstmöglichen Wert beginnt. Eine entsprechende Zielfunktion ist in Gleichung 6.6 dargestellt.

$$Zielfunktion = MAX\,(Variable_i) \quad \forall i \tag{6.6}$$

### 6.4.1 Zusammenhang zwischen Testmethoden und Beobachtungspunkten

Die verschiedenen Testmethoden benötigen unterschiedliche Beobachtungspunkte (siehe Tabelle 6.1).

Für die Überwachung der Instruktionen (Methode i) werden diese als Beobachtungspunkte benötigt. Eine Erweiterung der Instruktionsüberwachung mittels Registerinhalt (Methode ii) fordert zudem einen Beobachtungspunkt für die Register und die Arithmetic Logic Unit (ALU). Eine Analyse der Eingaben (Methode ii) wird an den Schnittstellen und in der Peripherie durchgeführt.

Wird der Speicher für eine Markierung (Methode iv), Maximierung des Variablenwachstums (Methode vi) oder des Speicherzugriffs (Methode vii) sowie

Tabelle 6.1: Zusammenfassung der Beobachtungspunkte aus Kapitel 6.3.4 und der Testmethoden aus Kapitel 6.4

| Testmethode \ Beobachtungspunkte | a) Variablen im Quellcode | b) Instruktionen im Quellcode | c) Variablen im Maschinencode | c) Instruktionen im Maschinencode | e) Register der CPU | f) Arithmetic Logic Unit der CPU | g) Daten- und Programmspeicher | h) I/O Schnittstellen | i) Prozessorperipherie |
|---|---|---|---|---|---|---|---|---|---|
| i) Überwachung der Instruktionen | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ |
| ii) Überwachung der Register | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ |
| iii) Analyse der Eingaben | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ |
| iv) Markierung des genutzten Speichers | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ |
| v) Analysemöglichkeiten der Codeabdeckung | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ |
| vi) Maximierung des Variablenwachstums | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ |
| vii) Maximierung des Speicherzugriffs | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ |
| viii) Minimierung des Abstands zwischen Heap und Stack | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ |
| ix) Minimierung des Variableninhalts | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ |
| x) Maximierung des Variableninhalts | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ |

Minimierung zwischen Heap und Stack (Methode viii) verwendet, so wird neben dem Zugriff auf Variablen ebenfalls der Speicher als Beobachtungspunkt genutzt. Zur Erkennung von Rechenfehlern, die auf dem Inhalt einer Variable beruhen, dienen Methode ix und Methode x. Beide Methoden benötigen Zugriff auf die Instruktionen und die Parameter der Instruktionen aus den Registern der CPU, da diese minimiert werden. Eine Verknüpfung der Testmethoden mit den benötigten Beobachtungspunkten ist in Tabelle 6.1 dargestellt.

## 6.5 Steuerung der Simulation und Überwachung

Die Steuerung des Testsystems (siehe Abbildung 6.1) basiert auf der Simulationsumgebung für virtuelle Plattformen und der Simulation der Peripherie und untergliedert sich in drei Bereiche:

1. Simulation der Applikation auf einem virtuellen Steuergerät (siehe Kapitel 6.1)
2. Simulation der Umwelt des Steuergeräts (siehe Kapitel 6.2)
3. Testmethoden und Beobachtungspunkte für die Simulation (siehe Kapitel 6.3 und Kapitel 6.4)

Die Simulation der Applikation (steuerndes Artefakt in Abbildung 6.8) nutzt die kompilierte Applikation im Maschinencode, die aus dem Quellcode erzeugt wurde, und führt diese auf einem virtuellen Prozessor aus, wodurch die Übertragbarkeit auf das reale Steuergerät gegeben ist. Während die kompilierte Software-Applikation in einem virtuellen Prozessor ausgeführt wird, muss zusätzlich zum Prozessor die gesamte virtuelle Plattform (Steuergerät inklusive Peripherie) und das Umfeld des virtuellen Steuergeräts simuliert werden (siehe Abbildung 6.1).

Für die Virtualisierung des Steuergeräts dient eine Beschreibung des Peripherieverhaltens und einer Beschreibung des virtuellen Steuergeräts (Plattformbeschreibung). Beides zusammen (Prozessor und Peripherie Simulation) dient als Grundlage der Simulation, um das Verhalten des Steuergeräts abzubilden. Über eine Testumgebung oder parallel ausgeführten Simulationen (Co-Simulation wie in [56]) werden die Schnittstellen zur Umwelt stimuliert. Zudem werden Zugriffe auf Beobachtungspunkte und Unterbrechungen der Simulation in einem Protokoll (Trace und Log) abgespeichert (siehe Abbildung 6.11).

Durch eine Erweiterung der Simulationssteuerung durch eine Überwachungsfunktion wird zum einen der Zustand aller Beobachtungspunkte überwacht und zum anderen die Simulation gesteuert, um den Ablauf während den Tests zu beeinflussen. Über Schnittstellen im Simulator wird die Ausführung überwacht und kann unterbrochen werden (siehe Kapitel 6.5.4). Zeitpunkte der Unterbrechung werden aus dem Quellcode anhand von Funktionsaufrufen und Speicherzugriffen (Variablen) entnommen und in einer separaten Datei (Beobachtungskonfiguration) abgespeichert. Hierfür wird der bereits kompilierte Programmcode (Maschinencode) analysiert und Rückschlüsse auf Speicherzugriffe und Funktionsaufrufe gezogen.

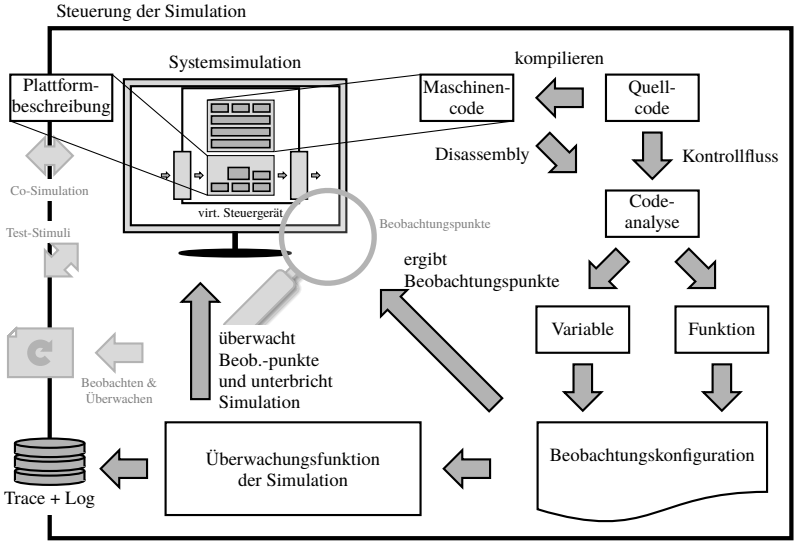



Abbildung 6.11: Aufbau der Testmethodik mit Eingriff zur Steuerung der Simulation als Erweiterung von Abbildung 6.1

### 6.5.1 Ablauf der Überwachungsfunktionen

Anhand der Applikation werden Überwachungsfunktionen generiert. Überwachungsfunktionen sind Funktionen, die innerhalb der Simulation genutzt werden, um die Applikation anzuhalten, Parameter zu beobachten, zu analysieren und die Simulation anschließend fortzusetzen.

Für die Generierung der Überwachungsfunktionen (engl. Interceptions) wird die Applikation zuerst für die Zielplattform kompiliert, um spätere Veränderungen durch die Optimierungen des Compilers zu vermeiden. Die entstandenen Dateien werden anschließend eingelesen und auf Speicherbereiche und Funktionsaufrufe untersucht (siehe Kapitel 6.3.2). Mithilfe dieser wird eine Tabelle zur Beobachtungskonfiguration (siehe Anhang B) aufgestellt, die alle Speicherbereiche für Variablen und Funktionsaufrufe enthält (siehe Kapitel 6.5.2). Um Nulldivisionen zu erkennen, wird zusätzlich nach plattformabhängigen Funktionen für die Division (mit und ohne Vorzeichen, “udiv“/“sdiv“) gesucht und diese als kritische Instruktionen gespeichert. Durch eine optionale Reduk-

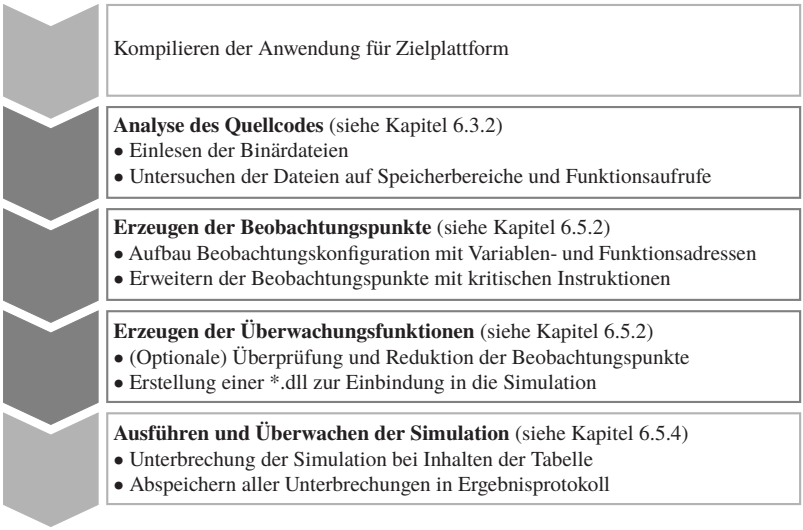



Abbildung 6.12: Ablaufdiagramm für die Generierung, Ausführung und Unterbrechungen der Simulation mittels Überwachungsfunktion für virtuelle Steuergeräte

tion der Tabelleninhalte kann der Umfang der Test verringert werden. Eine vollständige Auflistung aller kritischen Instruktionen ist in Anhang A gegeben.

Aufbauend auf dem Inhalt der Beobachtungskonfiguration wird abschließend eine Bibliothek erstellt (siehe Kapitel 6.5.2), die während der Simulation aufgerufen wird und die entsprechenden Unterbrechungen steuert (siehe Kapitel 6.5.4). Zum nachträglichen Bewerten der Tests werden die Unterbrechungen in einem Ergebnisprotokoll abgespeichert. Ein Ablaufdiagramm für diesen Prozess ist in Abbildung 6.12 dargestellt.

### 6.5.2 Erzeugen der Überwachungsfunktionen

Um die Simulation auszuführen, Zugriff auf die Beobachtungspunkte zu gewähren und durch die Testmethoden Stimuli zu erzeugen, werden Überwachungsfunktionen benötigt.

Für die Überwachungsfunktionen wird zuerst die Applikation untersucht. Hierbei wird der Maschinencode auf Funktionen, Funktionsaufrufe und Speicherbereiche für Variablen überprüft, da Speicher- und numerische Fehler zu Fehlern in der Datensicherheit führen können (siehe Kapitel 4.3).

Nur durch die beiden Alternativen (2. Maschinencode und 3. Maschinencode mit Debug-Symbolen, siehe Kapitel 6.3.2) lassen sich die Überwachungsfunktionen für diese Testmethode generieren, da bei der ersten Variante (1. Hochsprache) noch kein Programm vorliegt, das direkt auf einem Mikrocontroller eingesetzt wird und daher keine Zugriffe auf Speicherbereiche überwacht werden können. Weiter entstehen durch die Verwendung des Maschinencodes zwischen Simulation und Ausführung auf der realen Hardware keine Unterschiede in der Applikation.

Durch die Rekonstruktion der Variablen und Funktionen anhand des Maschinencodes (engl. Disassemble[5]) können ohne weitere Informationen über die Applikation zwischen 80 % und 99 % der Funktionen und Variablen erkannt werden [21]. Um dabei einen möglichst gute Abdeckung zu erhalten, wird nach dem Dissasemble der Kontrollfluss anhand von Verzweigungen ermittelt und somit die Struktur des Programms rekonstruiert. Dennoch können nicht immer alle Funktionen und Variablen erkannt werden. Grund hierfür ist, dass Funktionsaufrufe auch im Kontrollfluss nicht immer eindeutig identifizierbar sind. Daher werden diese später gesondert betrachtet (siehe Kapitel 6.5.3).

Um die Aussagekraft der Überwachungsfunktionen weiter zu erhöhen, wird im Folgenden davon ausgegangen, dass der Maschinencode und der Quellcode zur Verfügung stehen, um zusätzliche Debug-Symbole zu generieren. Dadurch werden alle Funktionen und Variablen der Applikation erkannt, solange diese im Quellcode definiert sind.

Durch die Analyse des Codes werden die Variablen, Funktionen und Stack-Operationen der Applikation in einer Beobachtungskonfiguration gespeichert (siehe Abbildung 6.13). Basierend auf den Funktionsaufrufen werden Rücksprungadressen zu den Stack-Operationen sowie lokale Variablen ergänzt. Da lokale Variablen erst während dem Ausführen der Applikation im Speicher

[5] Disassemble bedeutet, ein Programm aus seiner ausführbaren Form mit Maschienenbefehlen in eine Darstellung von Assemblerbefehlen zu konvertieren, so dass es für den Menschen lesbar ist.

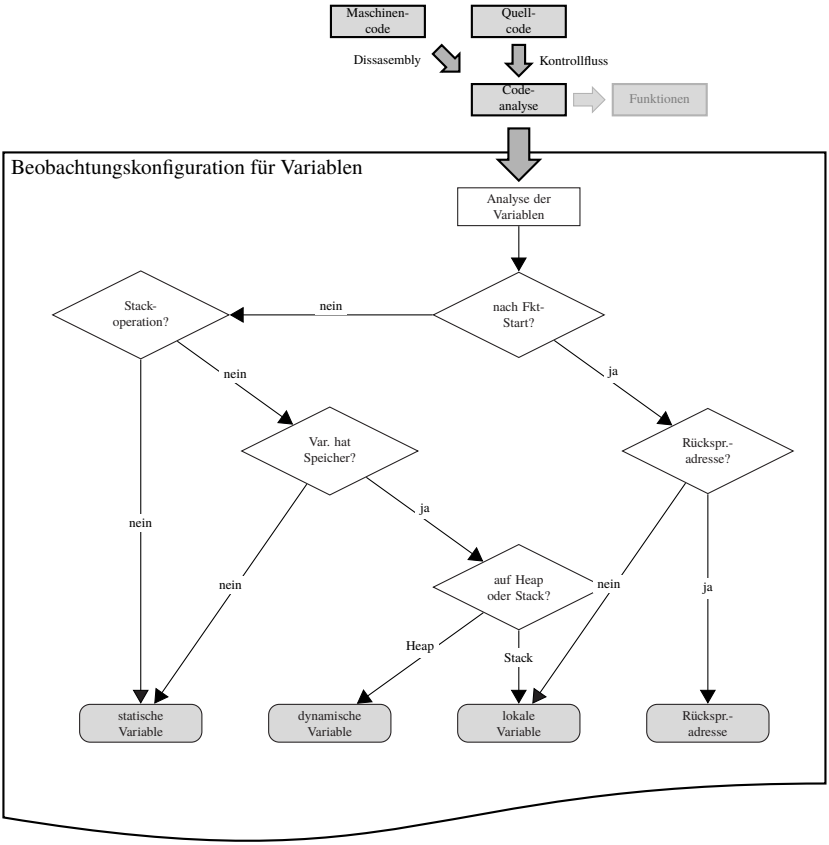



Abbildung 6.13: Erzeugung der Beobachtungskonfiguration aus Abbildung 6.11 für Variablen

angelegt werden, wird der Befehl zum Erstellen der lokalen Variablen in der Beobachtungskonfiguration vermerkt. Dadurch wird während der Simulation der Speicherort der lokalen Variable ausgelesen und in der Beobachtungskonfiguration zur Überwachung ergänzt. Anschließend werden durch den Kontrollfluss dynamische und statische Variablen bestimmt und die jeweiligen Adressen abgespeichert. Bei der Analyse des Maschinencodes ohne Debug-Symbole werden alle Variablen nur als Block abgespeichert, womit während der Simulation keine Rückschlüsse auf einzelne Bereiche in diesem Block mög-

lich sind. Durch die Debug-Symbole werden die einzelnen Variablen aufgelöst und individuell zur Überwachung abgespeichert.

Werden Speicherbereiche zur Laufzeit in der Applikation freigegeben (z. B. lokale Variablen nach dem Verlassen einer Funktion) wird dies ebenfalls in der Beobachtungskonfiguration vermerkt, damit dieser Speicher anschließend nicht weiter beobachtet wird.

Anschließend wird in der Code-Analyse die Software basierend auf dem Quellcode untersucht und alle Funktionen und Speicherzugriffe werden kategorisiert. Abbildung 6.14 zeigt dieses Vorgehen. Zuerst werden die Funktion untersucht und kategorisiert, ob diese beobachtet werden müssen. Funktionen die nicht beobachtet werden, haben keinen Speicherzugriff (d. h. keine Variablen), speichern keine Rücksprungadressen auf dem Stack ab (d. h. besitzen keine Subfunktionen) und besitzen keine kritischen Instruktionen (siehe Anhang A). Beispiel hierfür sind Routinen, die vom Compiler als “Inline“ übersetzt werden oder Berechnungen ausführen, ohne eigenen Speicher zu allokieren. Alle weiteren Funktionen werden während der Applikationsausführung beobachtet, können jedoch vom Entwickler zu den unbeobachteten Funktionen verschoben werden, um die Simulation zu beschleunigen. Beobachtete Funktionen allokieren Speicher und greifen auf diesen zu, rufen Unterfunktionen auf und speichern damit die Rücksprungadresse auf dem Stack oder enthalten kritische Funktionen (z. B. Divisionen oder Schreibzugriffe).

Die zu beobachtenden Variablen und Funktionen werden in einer Beobachtungskonfiguration abgespeichert (siehe Kapitel 6.5.2) und an die Überwachung und Steuerung der Simulation übergeben. Durch das Schreiben einer Variablen wird in der Überwachungsfunktion eine Speichermarkierung hinterlegt, die es nur der jeweiligen Funktion erlaubt auf den Speicherbereich zu schreiben. Durch einen Abgleich der Speichermarkierungen vor jedem Schreib- und Lesebefehl wird ein Speicherüberlauf erkannt und protokolliert.

Für die beobachteten Funktionen und Speicherbereiche werden in der Simulation Unterbrechungen erzeugt, die bei jedem Zugriff ausgeführt werden. Hierzu werden die Funktionsaufrufe und Instruktionen zur Laufzeit mit Überwachungsfunktionen überschrieben und eine Software zum Protokollieren des Verhaltens gestartet. Anschließend wird die ursprüngliche Funktion ausgeführt. Zusätzlich können Ausführungsbedingungen einer Überwachungsfunktion durch folgendes initiiert werden (siehe Abbildung 6.14, ①):

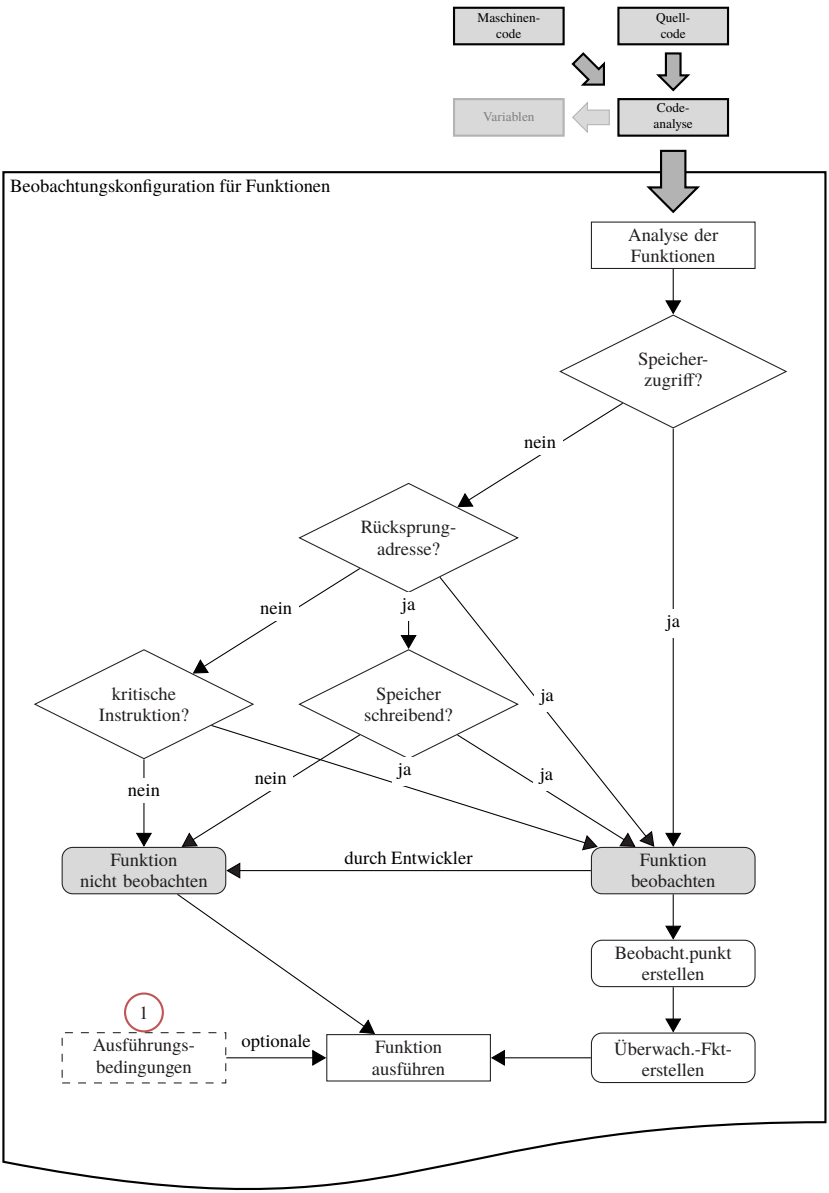



Abbildung 6.14: Code-Analyse zur Kategorisierung der Funktionen für Überwachung

- Simulationsbeginn und Simulationsende
- Operationsaufrufe
- bestimmte Instruktionsarten
- Erreichen einer vorher gewählten Instruktion
- Schreibender oder lesender Zugriff auf eine überwachte Speicherzelle oder Speicherbereich
- Erreichen einer bestimmten Programmzeile
- Nach dem Erreichen einer bestimmten Anzahl an Instruktionen oder Speicherzugriffen
- Bei Veränderung der Inhalte von Registern oder Speichern

### 6.5.3 Umgang mit "unbestimmten" Speicherbereichen

Beim Generieren der Überwachungsfunktionen anhand des Maschinencodes sind Funktionen nicht immer eindeutig durch Schlüsselwörter definiert, wie dies bei Hochsprachen beispielsweise durch den Zusatz "`function`" erfolgt. Daher kommt es zu sogenannten unbestimmten Speicherbereichen (siehe Kapitel 6.3.2). Diese Speicherbereiche ergeben sich zum einen dadurch, dass im Programmcode Variablenbereiche definiert werden, die jedoch zugeordnet sind. Zum anderen werden im Maschinencode die gleichen Ausdrücke (Instruktionen) für Funktionsaufrufe und bedingte Sprünge (z. B. `IF-THEN-ELSF`, `FOR-Schleifen`) verwendet[6].

Damit durch diese Speicherbereiche keine Sicherheitslücken entstehen, werden diese Speicherbereiche protokolliert und ebenfalls während der Simulation überwacht. Zusätzlich werden die Bereiche dem Entwickler oder Testingenieur gemeldet. Dadurch kann nach Durchlaufen der Tests entschieden werden, ob das Verhalten innerhalb dieser Speicherbereiche korrekt war.

[6] Beispiel: Im Maschinencode ist nicht erkenntlich, ob ein Sprungbefehl einen Funktionsaufruf darstellt bei dem im Anschluss eine Rücksprungadresse geschrieben wird oder ob der Sprungbefehl durch eine Schleife mit anschließendem Schreiben einer lokalen Variablen erfolgt (siehe auch Anhang A).

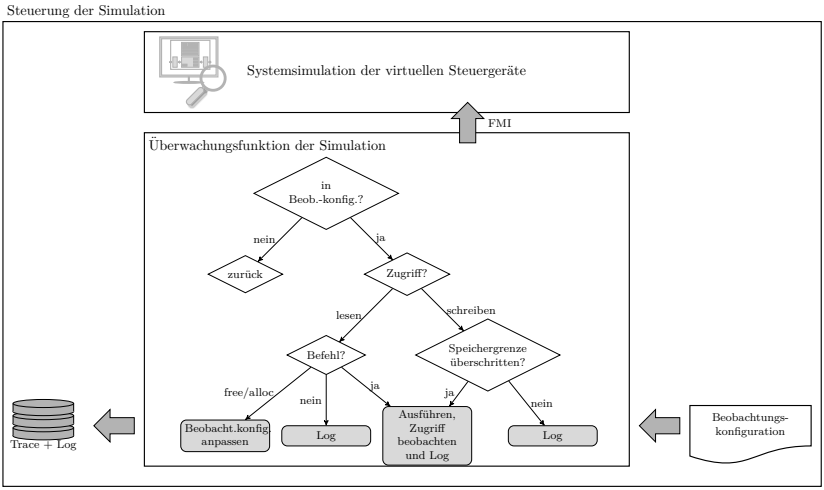



Abbildung 6.15: Steuerung der Simulation mit Entscheidung bei Unterbrechung

### 6.5.4 Steuerung der Simulation

Während der Laufzeit der Simulation wird diese durch eine Überwachungsfunktion unterbrochen (siehe Abbildung 6.15). Diese Überwachungsfunktion dient erstens zur permanenten Kontrolle der ausgeführten Befehle und Speicheroperationen, zweitens dem Vergleich der Inhalte der Beobachtungskonfiguration (siehe Kapitel 6.5.2) und drittens der Kontrolle der Simulation. Ein Überblick über den Ablauf der Steuerung ist in Abbildung 6.15 dargestellt.

Durch ein sogenanntes Functional Mock-up Interface (FMI) der Prozessorsimulation wird jeder auszuführende Befehl oder Speicherzugriff übermittelt und überprüft, ob dieser in der Beobachtungskonfiguration enthalten ist. Ist dies nicht der Fall, wird die Simulation mit dem nächsten Befehl fortgesetzt.

Ist der Befehl in der Beobachtungskonfiguration enthalten, wird im nächsten Schritt geprüft, ob dieser lesend oder schreibend auf den Speicher zugreift. Ist das lesende Artefakt kein Befehl, also ein Auslesen eines Speicherbereichs (Variable), wird dies in der Protokolldatei (Log) abgespeichert, jedoch während der Laufzeit nicht weiter betrachtet (siehe Abbildung 6.15). Handelt es sich um einen Befehl, wird die Ausführung beobachtet, protokolliert und bei falscher

Ausführung (d. h. bei Erkennen von Speicher- oder numerische Fehler, siehe Kapitel 4.1 und Kapitel 6.3.4) dem Entwickler mitgeteilt. An dieser Stelle werden nur Befehle überwacht, die in der Beobachtungskonfiguration enthalten sind, da die Befehle, die nicht in der Beobachtungskonfiguration vermerkt sind, bereits zu Beginn zur Weiterführung der Simulation geführt haben (siehe Abbildung 6.15).

Ist der Zugriff auf den Speicher schreibend, so wird überprüft, ob die Grenzen des Speicherbereichs überschritten werden. Ist dies der Fall, wird dies protokolliert und dem Entwickler im Trace + Log mitgeteilt, da es sich hierbei um einen Buffer-Overflow (siehe Kapitel 2.4.5) handelt. Werden Grenzen nicht überschritten, wird eine Variable beschrieben. Auch wenn das Schreiben einer Variablen kein Fehlverhalten darstellt, wird der Zugriff dennoch protokolliert, um bei der Auswertung der Protokolldatei überprüfen zu können, ob fehlerhaft Parameter geschrieben wurden. Anschließend wird mit dem nächsten Befehl der Applikation fortgesetzt.

Durch die Überwachung werden alle ausgeführten Befehle und Speicherbereiche analysiert. Wird über die Anzahl der Ausführungen je Befehl und Speicherbereich kumuliert, lässt sich zudem eine Aussage über die Vollständigkeit der Testabdeckung treffen. So ist es beispielsweise möglich Funktionen, Instruktionen oder Speicherbereiche zu finden, die während den Tests nicht oder nur mit einer geringen Anzahl an Testfällen durchlaufen wurden.

Durch die Einbindung der Analyse zur Laufzeit, entsteht ein Overhead. Dieser Overhead ist abhängig von der Applikation und daraus generierten Überwachungsfunktionen. Je mehr Funktionsaufrufe, Sprungbefehle und Speicherzugriffe vorhanden sind, umso häufiger wird die Simulation durch die Überwachung und Steuerung unterbrochen. Eine Analyse des Mehraufwands der Simulationszeit und der Skalierbarkeit wird in Kapitel 8 aufgezeigt.

## 6.6 Echtzeitfähigkeit der Testmethodik

Um echtzeitfähige Anwendungen zu testen, muss die Echtzeit ebenso in der Simulation gewährleistet sein (siehe Kapitel 5.1). Besonders bei der Ausführung von parallelen oder nebenläufigen Anwendungen auf Mehrprozessorsystemen muss die Simulation das Echtzeitverhalten garantieren.

Zur Simulation des Echtzeitverhaltens muss die Anwendung zeitakkurat simuliert werden, um Rückschlüsse auf das zeitliche Verhalten zu ziehen. Hierfür werden zyklenakkurate Simulatoren eingesetzt, die das Zeitverhalten hinreichend genau nachbilden (siehe Kapitel 2). Durch eine instuktionsakkurate Simulation kann unter bestimmten Bedingungen ebenfalls auf das zeitliche Verhalten zurück geschlossen werden.

Voraussetzung für das Zurückrechnen ist die Kenntnis des Instruktionssatzes und der entsprechenden Ausführungszeiten je Instruktion (Länge in Prozessortakten). Anschließend wird über Gleichung 6.7 mithilfe der Prozessorfrequenz die Laufzeit je Instruktion bestimmt. Durch eine Kumulation der Einzellaufzeiten wird die Gesamtlaufzeit berechnet (siehe Gleichung 6.8). Diese Art der Berechnung wird als Logical Excecution Time (LET) bezeichnen.

$$Laufzeit_{Instruktion} = \frac{Länge_{Instruktion}}{Prozessorfrequenz} \tag{6.7}$$

$$Laufzeit_{Kumuliert} = \sum_{i=1}^{n} Laufzeit_{Instruktion,i} \tag{6.8}$$

Durch das Berechnen der kumulierten Laufzeit wird eine globale Zeit erstellt und die einzelnen Prozessoren, Peripherien und Co-Simulatoren synchronisiert. Das heißt, Instruktionen der Anwendung oder der Co-Simulation werden nur ausgeführt, wenn diese zum aktuellen Zeitschritt passen. Wenn der Simulationscomputer die Simulation der Anwendung unterbricht, um beispielsweise Co-Simulationen auszuführen, so läuft die Zeit in der Anwendung nicht weiter, da keine Instruktionen im Prozessor ausgeführt werden (siehe Abbildung 6.16).

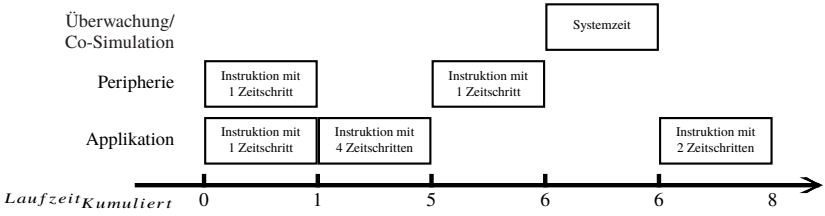



Abbildung 6.16: Kumulierte Laufzeit bei Simulation verschiedener Instruktionen

Durch dieses Vorgehen, wird auch das Echtzeitverhalten von Mehrprozessorsystemen simuliert. Um die Echtzeitfähigkeit in der Simulation, wie in der Realität, zu garantieren, gilt die Annahme, dass der Prozessor während einer laufenden Instruktion nicht mehr auf ändernde Eingaben reagiert. Änderungen an den Eingängen werden erst im nächsten Prozessortakt berücksichtigt.

Beim Testen eines realen Steuergeräts ist eine solche Unterbrechung für Überwachung oder Co-Simulation nicht möglich, da die Zeit im Steuergerät nicht unterbrochen werden kann. Das heißt, eine Überwachung oder Co-Simulation muss auf einem realen Steuergerät so abgebildet sein, dass zu jedem Zeitschritt des Steuergeräts die Überwachung und Co-Simulation abgeschlossen sind. Durch die Unterbrechung der Simulation und Berechnung der kumulierten Laufzeit ist der Test mittels virtuellem Steuergerät nicht an schnelle Simulatoren und Recheneinheiten gebunden, sondern die Ausführung der nächsten Instruktion wird entsprechend verzögert.

## 6.7 Zusammenfassung

Um die Datensicherheit in Automotivesystemen zu testen, wurde basierend auf einer Simulation mit virtuellen Steuergeräten eine Testmethodik aufgebaut. Diese Methodik wurde um eine Überwachung und Steuerung ergänzt, die Speicherzugriffe und Instruktionen in der Applikation erkennt und überwacht. Die Erkennung der Instruktionen und Speicherbereiche basiert auf der Untersuchung des Maschinencodes der Applikation. Zur Ergänzung werden Debug-Symbole des Compilers herangezogen.

Zur Laufzeit der Simulation werden die Artefakte der Beobachtungskonfiguration untersucht. Hierfür wird die auszuführende Funktion unterbrochen, eine Überwachungsfunktion gestartet und die Ergebnisse der ursprünglichen Funktion protokolliert. Es werden dabei verschiedene Überwachungsmethoden genutzt, um die Beobachtungspunkte auszunutzen. Diese Überwachungsmethoden reichen von statischer Überwachung der Instruktionen und Register über die Analyse der Eingaben bis hin zu einem evolutionären Vorgehen zur Maximierung des genutzten Speichers oder zur Minimierung des Abstands zwischen Heap und Stack.

Tabelle 6.2: Übersicht und Bewertung der Testmethodik im Vergleich zu Tabelle 3.2

| | Test | | Codeart | | | | Hardware | | | Zugriffe | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Testprozess | statisch | dynamisch | Graphisch | Quellcode | Assembler | Maschinen | Echtzeit | Optimiert | Reale HW | Schnittstellen | Prozessor | Speicher | Peripherie |
| Model-in-the-Loop | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ |
| Software-in-the-Loop | ✓ | (✓) | (✓) | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ |
| Hardware-Simulation | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ |
| Prozessor-in-the-Loop | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | (✓) | ✓ | ✗ |
| Überwachung virt. ECU | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Hardware-Debug | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | (✓) | (✓) | ✗ |
| Hardware-in-the-Loop | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ |
| Fahrzeug-Test | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ |
| Rapid Prototyping | ✗ | ✓ | (✓) | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | (✓) |

Durch die Kenntnis des Instruktionssatzes und der Ausführungszeiten der Instruktionen wird eine globale Zeit erstellt, die eine Simulation des Echtzeitverhaltens und eines geschlossenen Regelkreise durch Co-Simulatoren erlaubt.

Fasst man die Testmethodik mit Tabelle 3.2 zusammen, ergibt sich die Bewertung aus Tabelle 6.2. Die Hauptmerkmale der Methodik mit Überwachung eines virtuellen Steuergeräts sind, dass verschiedener Codearten (Quellcode bis Maschinencode) in einer dynamischen Umgebung getestet werden können und dass im Vergleich zu den Testverfahren aus Kapitel 3 zusätzliche Beobachtungspunkte bereitgestellt werden. Damit kann das Verfahren bereits in der Entwicklung eingesetzt werden, sobald ein Quellcode existiert. Das Testverfahren ist zudem für eine optimierte und hardwareangepasste Applikation einsatzfähig. Wie bei den Fahrzeugtests und Hardware-in-the-Loop (HiL) ist das Testverfahren für eine Applikation ohne Änderungen aus dem realen Steuergerät geeignet. Die benötigten Beobachtungspunkte ergeben keine Anpassung der Applikation, da diese nur in der Simulation des virtuellen Steuergeräts vorhanden sind. Ein weiterer Unterschied zu den Testmethoden aus Kapitel 3 besteht darin, dass neben den Schnittstellen der Prozessor, der Speicher und die Peripherie über Beobachtungspunkte analysiert werden können.

# 7 Einbindung in bestehende Testprozesse

Das Testen befindet sich im Entwicklungsprozess der Automobilindustrie generell auf der rechten Seite des V-Modells (siehe Kapitel 2.6.3). Hierbei werden während der Integration zuerst die Software-Module, dann das Steuergerät und abschließend der Steuergeräte-Verbund bis hin zum gesamten Fahrzeug getestet. Die vorgestellte Testmethodik für Datensicherheit wird im folgenden in das V-Modell eingeordnet und die Vorteile der Testmethodik erläutert. Die klassischen Testverfahren (MiL, SiL, HiL, Fahrzeugtest) sind zusammen mit der Einordnung der neu vorgestellten Testmethodik (Simulation mit virtuellem Steuergerät; farblich hinterlegt) in Abbildung 7.1 dargestellt.

## 7.1 Einbindung der Testmethodik im V-Modell

Beim klassischen Testen der Software werden zuerst einzelne Funktionen modellbasiert getestet (Model-in-the-Loop). Bei diesen Tests wird das funktionale Verhalten der Software untersucht, wobei die Datensicherheit nur am Rande berücksichtigt wird. Im weiteren Verlauf werden die Software-Komponenten erstellt und einzeln über Software-in-the-Loop ohne das Zusammenspiel der gesamten Software getestet. Solange hierbei keine Hardware-Schnittstellen benötigt werden, findet dies durch eine Simulation auf einem Computer, ohne Einbeziehung der Hardware-Plattform statt.

Anschließend werden in den klassischen Tests die Software-Komponenten auf ein reales Steuergerät integriert und das Verhalten auf der Hardware mittels Hardware-in-the-Loop sichergestellt. Die einzelnen Steuergeräte werden dann in ein Fahrzeug verbaut und im Fahrzeugtest die gesamte Funktionalität validiert. Eine Integration der Software-Komponenten erfolgt in den klassischen Tests auf einer realen Hardware, die anschließend am Prüfstand getestet wird.

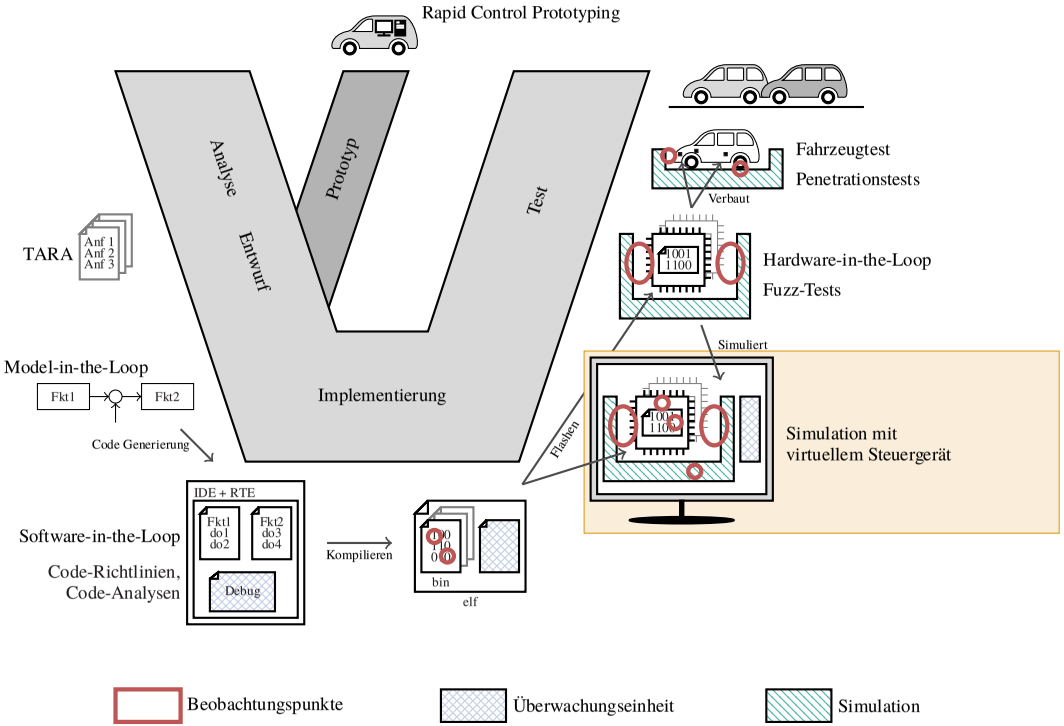



Abbildung 7.1: Analyse der Lücke zwischen Simulation mit virtueller Hardware und Ausführung auf Ziel-Hardware

Um die Gesamtsoftware bereits früher testen zu können, kann die vorgestellte Testmethodik durch Simulation eines virtuellen Steuergeräts eingesetzt werden. Hierbei werden der Prozessor, die Peripherie und die Umwelt auf einem Computer simuliert. Hierfür muss die virtuelle Plattform um die Schnittstellen zur Kommunikation erweitert werden. Anhand dieser Tests, wird die Applikation zusätzlich zu den Software-Tests auf Schwachstellen (siehe Kapitel 4.1) an den Schnittstellen und der Peripherie untersucht. Die Tests dienen der Untersuchung der Schwachstellen in der Software, daher ist es ausreichend, wenn das Verhalten der Schnittstellen gemäß ihrer Spezifikation modelliert ist und durch die Schnittstellen entsprechende Stimuli eingegeben werden können. Zudem bietet die Testmethodik mittels virtuellem Steuergerät, im Vergleich zur HiL die Möglichkeit, zusätzliche Beobachtungspunkte einzubringen (siehe Kapitel 6).

Weiter kann die Simulation durch eine Abbildung des gesamten Steuergeräte-Verbunds erweitert werden. Dies ermöglicht den Test der Applikation in einem Verbund von virtuellen Steuergeräten. Tests des Steuergeräte-Verbunds ermöglicht den Test komplexer und vernetzter Funktionen, die auf mehreren Steuergeräten verteilt sind. Für diese Tests ist es erforderlich, die Verbundsteuergeräte entweder über eine Co-Simulation mit Hilfe anderer Tools zu simulieren oder die weiteren Steuergeräte in der Simulationsumgebung aufzubauen. Für den Aufbau in der selben Simulationsumgebung, muss das Verhalten der Schnittstellen zur Kommunikation zwischen den Steuergeräten abgebildet werden. Hierfür müssen entsprechend hinreichend genaue Modelle der Schnittstellen und Kommunikation erstellt werden.

Die Steuergeräte-Verbund-Tests werden während der Integration von verschiedenen Komponenten durchgeführt. Da der Quellcode von den Softwareherstellern jedoch nicht zur Verfügung gestellt wird, können während den Tests nur Speicherblöcke überwacht werden, ohne deren Inhalt zu verifizieren (siehe Kapitel 6.3.2). Die in Kapitel 4.1 vorgestellten Speicher- und numerische Fehler werden dabei durch die Testmethodik in Steuergeräte-Verbund-Tests basierend auf den ausgeführten Instruktionen erkannt (siehe Kapitel 6.4).

## 7.2 Beobachtung und Überwachung verschiedener Artefakte

In klassischen Testprozessen (siehe Kapitel 3.5.4) werden entweder Softwareartefakte (Funktionen, Variablen) oder die Hardwareschnittstellen der Steuergeräte beobachtet. Diese Testmethoden sind in Abbildung 7.2 dargestellt, wobei hier eine Zweiteilung zwischen Software und Hardware entsteht. Im dargestellten Beispiel ist der oberer Teil nach einem AUTOSAR Software-Stack aufgebaut [86] und kann durch funktionale Tests und Codescanner validiert werden. Der untere Teil stellt ein abstraktes Steuergerät mit Prozessor, Speicher sowie Ein- und Ausgaben dar (siehe Kapitel 2.1.1), welches durch Fuzz- und Penetrationstests überprüft wird.

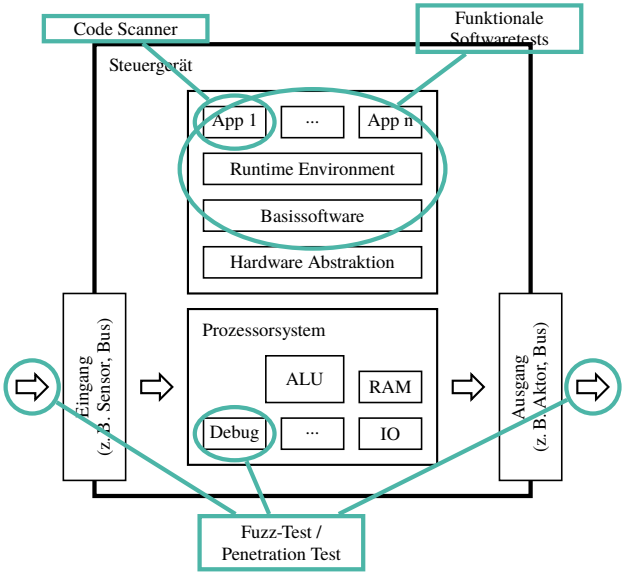



Abbildung 7.2: Verschiedene Sichtweisen der klassischen Tests für Datensicherheit

Zum Testen der Software werden die Funktionen unabhängig von der Hardware analysiert. Durch die Abstraktion von AUTOSAR ist die unterliegende Hardware irrelevant, wodurch der Test einzelner Software-Funktionen ermöglicht

wird. Da die Tests jedoch ohne Hardware ausgeführt werden, ist das genaue Verhalten des Steuergeräts während den Tests nicht prüfbar. Bei den Hardware-Tests werden die Schnittstellen des Steuergeräts beobachtet, wodurch keine Rückschlüsse auf das detaillierte Software-Verhalten möglich sind.

Durch eine Virtualisierung des Steuergeräts werden Schnittstellen und das Verhalten der Hardware sowie die Softwarefunktionen simuliert und können damit überwacht werden. Dadurch ist es möglich, bei Kenntnis der Software und Hardware, jegliches Artefakt im Steuergerät zu beobachten. Voraussetzung ist jedoch, dass für das zu überwachende Artefakt ein entsprechendes Modell existiert.

Unterscheidet man zwischen Zulieferer und Hersteller, stehen beiden unterschiedliche Ebenen des Testens zur Verfügung. Während dem Softwarezulieferer alle Informationen über die Software vorliegen, erhält der Hersteller nur das fertige Steuergerät oder den ausführbaren Maschinencode, um diesen am Bandende auf das Steuergerät zu programmieren. Im Gegensatz hierzu besitzt der Zulieferer nicht alle Steuergeräte, Sensoren und Aktoren, um die gesamte Funktion auszuführen und damit das Gesamtsystem zu testen.

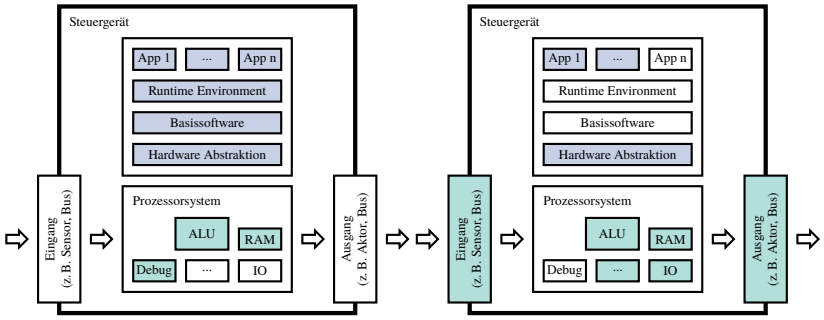



Abbildung 7.3: Möglichkeiten der Einblicke in das virtuelle Steuergerät aus Sicht des Zulieferers (links) und Herstellers (rechts)

Betrachtet man den Zulieferer, so kann dieser die in Abbildung 7.3 links dargestellten Bereiche testen, da für die Software bis zum Betriebssystem die Informationen vorliegen. Die Hardware-Abstraktion ist sowohl Hersteller als auch Zulieferer bekannt, da hierüber die Ansteuerung der Hardware aus der Software erfolgt. Das Prozessormodell kann der Zulieferer entweder frei wäh-

len, falls kein Prozessor vorgegeben ist, oder aus der Spezifikation erstellen. Durch eine Integration auf die Zielplattform kann der Zulieferer das Verhalten des gesamten Steuergeräts über die Debug-Schnittstelle überprüfen. Im Gegensatz zur realen Hardware können bei der vorgestellten Testmethodik durch ein virtuelles Steuergerät Einblicke in die Software, den virtualisierten Prozessor und die Steuergeräteperipherie generiert werden, ohne die Debug-Schnittstelle des realen Steuergeräts zu nutzen und dadurch auch ohne zusätzlichen Code in die Software einzubringen.

Der Hersteller besitzt hingegen den gesamten Steuergeräteverbund und kann diesen detailliert testen. Damit kann der Hersteller durch die vorgestellte Testmethodik die in Abbildung 7.3 rechts dargestellten Bereiche zum Testen nutzen. Da der Quellcode der Software vom Zulieferer nicht frei gegeben wird, muss der Hersteller Black-Box-Tests durchführen (siehe Kapitel 9.4). Durch eine Rekonstruktion der Applikation aus dem Maschinencode lässt sich zwar das Verhalten der Software und des Betriebssystems wiederherstellen, ein Einblick in alle Bereiche ist jedoch nicht möglich (siehe Kapitel 6.3.2).

Durch die Kombination des Wissens des Herstellers und des Zulieferers kann eine ganzheitliche Plattform erstellt werden, die alle Bereiche des Steuergeräts zum Test abdeckt. Hierfür müsste jedoch der Hersteller die entsprechende Hardwarekomponenten (inkl. E/E-Architektur) und der Zulieferer den Quellcode offen legen. Da dies zum Schutz der Intellectual Property (IP) weder vom Zulieferer noch vom Hersteller gewünscht ist, können Hersteller und Zulieferer weiterhin nur die in Abbildung 7.3 dargestellten Bereiche überwacht werden. Die Testmethode ermöglicht zwar zusätzliche Beobachtungspunkte, deckt jedoch nicht das Gesamtsystem ab. Folgt keine Kombination von Modellen der Hardwarekomponenten und dem Sourcecode der Software, eignet sich die Testmethode zum Ermittel von Schwachstellen besonders für den Zulieferer, da dieser die Software und Prozessormodelle für Einzelsteuergeräte besitzt und damit testen kann. Probleme die erst im Gesamtsystem entstehen, können jedoch nicht entdeckt werden.

## 7.3 Erweiterung der Testfallgenerierung

Die Datensicherheit wird durch die bestehenden Testmethoden der Automobilindustrie aktuell nur wenig betrachtet (siehe Kapitel 3.5.4). Daher werden

die bestehenden Testmethoden der Automobilindustrie für den Test der Datensicherheit erweitert, um neue Testfälle zu erzeugen. Die Test-Stimuli aus den klassischen Testprozessen (z. B. HiL, SiL) werden bereits im linken Pfad des V-Modells durch die Anforderungen generiert und anschließend gegen diese verifiziert (siehe Kapitel 2.6.3). Für den Test auf einer virtuellen Plattform werden diese Tests ebenfalls verwendet. Dadurch lässt sich ohne Erweiterung bereits die selbe Testabdeckung wie auf dem realen Steuergerät erreichen.

Für den Test der Datensicherheit müssen die bestehenden Test-Stimuli jedoch um spezielle Testfälle ergänzt werden. Für diese Erweiterung werden die Testmethoden aus Kapitel 3.5.4 herangezogen (siehe Abbildung 7.4). Für den Test der Datensicherheit wird bereits während der Entwicklung der Quellcode mittels Code-Analysen anhand vorgegebener Richtlinien untersucht (siehe Kapitel 3.5.4). Zusätzlich wird über Fuzz- und Penetrationstests im rechten Ast des V-Modells die reale Hardware auf Schwachstellen untersucht (siehe Kapitel 3.5.4). Für das Fuzz-Testing werden zufällige Eingabeparameter generiert oder diese zufälligen Eingabeparameter durch die vorhandene Spezifikation eingeschränkt (spezifischer Zufall), falls beispielsweise das Regelverhalten in der Analyse nicht betrachtet werden soll. Durch Fuzz-Tests werden ungültige Eingangs-Stimuli zufällig erzeugt, um neben der bereits bestehenden Testabdeckung, aus den klassischen Tests, Schwachstellen zu identifizieren. Hierzu werden zufällige Stimuli außerhalb der Spezifikation verwendet und das Verhalten der Schnittstellen und Peripherie zu untersucht. Durch den Einsatz der virtuellen Plattform werden zusätzlich zu den an Schnittstellen und Peripherien nach außen sichtbaren Effekten die internen Parameter des Prozessors analysiert. Durch die Überwachung der internen Zustände wird ein Fehlverhalten, das zu einer Schwachstelle führen kann, bereits in den internen Strukturen des virtuellen Prozessors erkannt, bevor es an den Schnittstellen durch die klassischen Testprozesse sichtbar ist. Da das Fuzz-Testing auf zufällig ausgewählten Mustern beruht, ist es statistisch unwahrscheinlich komplexe Zusammenhänge zu entdecken und muss daher mit weiteren Testmethoden und einem strukturieren Vorgehen ergänzt werden.

Für die Penetrationstests werden bekannte Schwachstellen analysiert oder mittels Fuzz-Tests neue Schwachstellen gefunden. Durch den Einblick in das Verhalten des Prozessors und des Speichers können, unter Hilfe des Einsatzes der Testmethode aus Kapitel 6, während diesen Tests Rückschlüsse auf das Fehlverhalten innerhalb des virtuellen Steuergeräts gezogen werden. Mit der

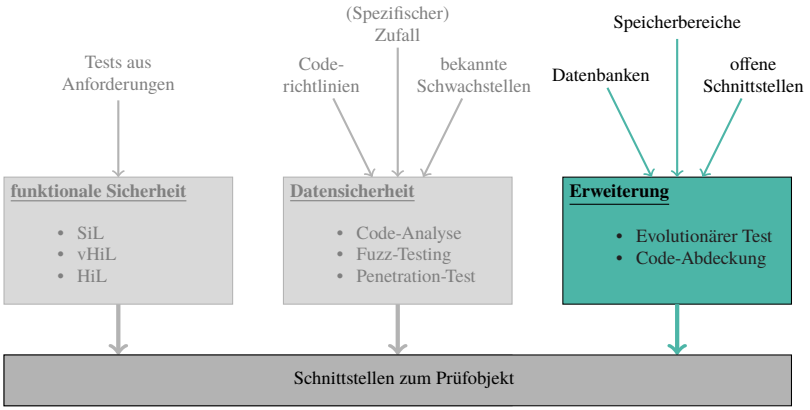



Abbildung 7.4: Erweiterung bestehender Testfallgenerierung aus Abbildung 2.16

Einbindung von Penetrationstests in den Testprozess werden weitere Testfälle generiert, die in der virtuellen Plattform analysiert werden. In den Phasen Enumeration und Exploitation (siehe Punkt 3.5.4) wird der Tester durch die Beobachtungspunkte im virtuellen Prozessor unterstützt. In diesen beiden Phasen erhält der Tester neben dem Quellcode auch Zugriff auf das interne Verhalten beim Ausführen der Angriffsvektoren und kann dadurch Rückschlüsse auf die Schwachstelle des Systems ziehen.

Zusätzlich werden die in Abbildung 2.16 dargestellten Methoden ergänzt, um weitere Testfälle zu generieren (siehe Abbildung 7.4). Hierbei wird zunächst das System auf offenen Schnittstellen[1] untersucht und zu den Testfällen ergänzt. Schnittstellen die nicht modelliert, aber im Code vorhanden sind, können während der Simulation beim Zugriff auf diese erkannt werden, durch die fehlende Modellierung wird das Verhalten jedoch nicht dargestellt. Für Schnittstellen, die in der Simulation nicht vorhanden sind, werden keine Test-Stimuli erzeugt und diese bei der Ausführung der Simulation nicht berücksichtigt. Daher ist es nötig, für den Test der offenen Schnittstellen diese zu modellieren und dadurch in den Testprozess einzuschließen. Anschließend werden Fuzz-Tests automatisiert ausgeführt (durch Zufallswerte stimuliert) und das interne Verhalten des

[1] Offene Schnittstellen, sind Schnittstellen, die in der Applikation vorhanden und verwendet werden (siehe Kapitel 6.3.3).

virtuellen Steuergeräts untersucht und Auffälligkeiten dieser Analyse protokolliert.

Zur Generierung der Testfälle für die offenen Schnittstellen dienen die Schwachstellen aus Kapitel 4.1 und den Datenbanken der Common Vulnerabilities and Exposures Datenbank (CVE) und Common Weakness Enumeration Datenbank (CWE). Beide listen die in Kapitel 4.1 verwendeten Schwachstellen und Angriffe auf verschiedenen Abstraktionsebenen auf und beinhalten die Durchführung der Angriffe. Basierend auf bekannten Angriffen können neue Testfälle generiert werden. Hierzu werden ähnliche oder identische Applikationen und Funktionen in der Datenbank gesucht, die Angriffe auf die bekannten Schnittstellen herangezogen und anschließend zu den Testfällen ergänzt. Das automatisierte Generieren der Testfälle aus solchen Datenbanken ist für ein automatisiertes Testen der Datensicherheit essentiell und wird in der Literatur [10, 97] bereits für einzelne Anwendungen betrachtet. Hierfür werden aus den gespeicherten Angriffsmustern neue Stimuli für das System erzeugt oder entsprechende Signaturen in der Anwendung gesucht, die auf die bekannte Schwachstelle hindeuten. Obwohl die Datenbanken für den Einsatz in der PC-Welt vorgesehen sind, existieren darin bereits Einträge über Angriffe auf eingebettete Geräte und Automotive Steuergeräte. Um eine effiziente Generierung der Testfälle zu ermöglichen müssen die Angriffe aus der Automotive-Domain weiterhin protokolliert und in einer Datenbank zusammengefasst werden.

Als weitere Ergänzung zu den klassischen Tests der funktionalen Sicherheit und der Datensicherheit ermöglicht die vorgestellte Testmethodik eine Erweiterung mit evolutionären Testverfahren (siehe Kapitel 3.2). Hierzu werden die Beobachtungspunkte des virtuellen Steuergeräts (siehe Kapitel 6) genutzt und die Überwachung als Optimierungsproblem angesehen. Mittels iterativem Vorgehen versucht die Überwachung und Steuerung den zusammenhängenden Schreibzugriff auf den Speicher zu maximieren oder den Speicher von Heap und Stack ansteigen zu lassen, um dadurch einen Overflow (siehe Kapitel 2.4.5) herbeizuführen. Durch die Beobachtung der internen Strukturen des Steuergeräts (Speicher, Register, Peripherie) kann der evolutionäre Testalgorithmus zusammen mit der kompilierten Applikation eingesetzt werden. Die Zielfunktion des evolutionären Test kann dabei den Speicher auswerten und so Probleme entdecken, bevor diese zu den Schnittstellen propagiert sind. Damit wird der Testaufwand reduziert, da der Testalgorithmus schneller erkennt, ob

die Variation der Eingabeparameter positiven oder negativen Einfluss auf die Zielfunktion besitzt.

Zusätzlich zur den oben beschriebenen Tests werden die ausgeführten Befehle und Speicherbereiche protokolliert. Dies ermöglicht eine Aussage über die Güte der ausgeführten Tests zu treffen. Durch die Überwachung der Code-Abdeckung wird festgestellt wie häufig Funktionen durchlaufen wurden und ob die Tests angepasst werden müssen, um Funktionen während des Testdurchlaufs häufiger abzuprüfen.

Da diese Tests auf dem kompilierten Quellcode ausgeführt werden, gelten die Ergebnisse nicht nur für das Programm im Quellcode, sondern auch für die ausführbare Applikation, ohne das weitere Änderungen durch den Compiler und Linker entstehen.

## 7.4 Zusammenfassung

Die vorgestellte Testmethodik zur Analyse der Datensicherheit befindet sich im rechten Ast des V-Modells (siehe Abbildung 3.6) und deckt die Bereiche der Softwaretests auf einem virtuellen Steuergerät und des Steuergeräte-Verbund ab. Diese verschiedenen Abdeckungen werden durch Simulation

- der Software auf einem Prozessor,
- des Steuergeräts inklusive Speicher und Peripherie oder
- der verteilten Steuergeräte mit Simulation der Kommunikation oder Co-Simulation

erreicht.

Durch den Einsatz von virtuellen Steuergeräten werden während der Tests unterschiedliche Artefakte überwacht. Der Zulieferer eines Steuergeräts kann somit beispielsweise einen zusätzlichen Einblick in Speicher, Register und Peripherie bekommen, was über die Debug-Schnittstelle nur mit zusätzlichem Code in der Applikation funktioniert. Fahrzeughersteller können die Simulation zu einem Steuergeräteverbund erweitern und damit die Gesamtfunktion testen, auch wenn diese über mehrere Steuergeräte verteilt ist. Zusätzlich bekommt der Hersteller beim Testen Zugriff auf interne Strukturen (Applikation,

Speicher, Peripherie, etc.) des virtualisierten Steuergeräts, ohne zusätzliche Debug-Schnittstellen einbauen zu müssen.

Für den Test auf einem virtuellen Steuergerät werden die Testfälle aus den klassischen funktionalen Tests (Hardware-in-the-Loop, Software-in-the-Loop, etc.) und den Tests der Datensicherheit (Fuzz-Test, Penetrationstest) ebenfalls verwendet, wodurch sich ohne Erweiterung bereits die selbe Testabdeckung wie auf dem realen Steuergerät erreichen lässt. Um die Testabdeckung zu erweitern, werden zusätzlich die Speicherbereiche und offenen Schnittstellen analysiert. Durch den Einsatz von evolutionären Testalgorithmen können Zielfunktionen aufgestellt werden, die Overflows bereits im Speicher erkennen, bevor diese zu den Schnittstellen propagiert und nach außen sichtbar sind.

# 8 Skalierbarkeit der Testmethodik

Die Anzahl der Steuergeräte in einem System ist nicht fest. Je nach System wird die Gesamtfunktion von mehreren verteilten Steuergeräten und intelligenten Sensoren und Aktoren realisiert. Um die maximale Anzahl an simulierbaren Steuergeräten beim Test von Steuergeräte-Verbünden zu bestimmen, wird die Skalierbarkeit der Simulationsumgebung und der Überwachungsfunktionen untersucht. Für die Untersuchung der Skalierbarkeit werden verschiedene Simulationsumgebungen untersucht und eine Simulationsumgebungen zur Ausführung ausgewählt.

## 8.1 Bewertung und Auswahl einer virtuellen Umgebung

Für virtuelle Plattformen existieren verschiedene Anwendungsgebiete. Diese reichen von Rapid Prototyping über die Analyse der Laufzeiten und Instruktionen bis hin zur Analyse der programmierbaren Hardware (FPGAs und MPSoC) [WLK+16].

Um die fehlerfreie Implementierung während des Entwicklungsprozesses zu gewährleisten, wird eine instruktionsakkurate Simulation eingesetzt (siehe Kapitel 2.5.3). Hierfür stehen verschiedene Simulationsumgebungen zur Auswahl, die in Tabelle 8.1 aufgelistet sind.

In der Simulation müssen Prozessoren und Betriebssysteme unterstützt werden, die in den späteren Steuergeräten eingesetzt sind, um keine Anpassung an der Software der Steuergeräte vornehmen zu müssen.

Durch eine Paravirtualisierung (siehe Definition 22) wird das Verhalten der Software simuliert, jedoch werden Schnittstellen bereitgestellt, die der Hardware ähnlich sind, deren Verhalten aber nicht genau nachbilden. Eine Emula-

tion (siehe Definition 19) ist die Nachahmung des nach außen hin sichtbaren Verhaltens, wobei der innere Zustand nicht dem inneren Zustand der realen Einheit entspricht. Die Gesamtsystemsimulation stellt eine Umgebung bereit, die das Verhalten der späteren Plattform realitätsnah abbildet (siehe Kapitel 2.5.2) und ist daher zu bevorzugen.

Für eine Überwachung (siehe Definition 34) der internen Zustände ist ein Eingriff in die Systemarchitektur nötig, um die Simulation zu steuern und zur Laufzeit Parameter auszulesen. Um die Peripherie des Steuergeräts zu überwachen, ist es zudem nötig nicht nur Prozessoren, sondern auch Schnittstellen nach außen zu emulieren oder zu simulieren und weitere Simulatoren für die Umwelt anbinden zu können.

Die Simulatoren GXemul, Imperas OVPsim, Synopsys Virtual Platform, QUEMU und Simics verfügen über eine Vielzahl an verschiedenen Prozessoren, die in der Automobilindustrie für eingebettete Systeme eingesetzt werden [134] (siehe Tabelle 8.1). Zusätzlich können verschiedene Betriebssysteme oder bare-metal Applikationen innerhalb der Simulatoren ausgeführt werden. Obwohl keiner der Simulatoren speziell für AUTOSAR ausgelegt ist, kann dies dennoch auf Imperas OVPsim und Synopsys Virtual Platform portiert werden.

Ein Vergleich der Simulatoren zeigt, dass nicht alle eine Gesamtsystem-Simulation erlauben. Durch den Einsatz einer Paravirtualisierung müssen Schnittstellen nach dem Test angepasst werden, was zu einer Änderung der Applikation führt (siehe Kapitel 2.5.1).

Ebenso verhält es sich beim Eingriff in die Systemarchitektur. Dieser Eingriff ist jedoch nötig, um interne Zustände und Speicherbereiche zu überwachen und die vorgestellte Steuerung (siehe Kapitel 6) in der Simulation einzubinden.

Eine Gesamtsystemsimulation und Eingriff in die Systemarchitektur ermöglichen die Werkzeuge Imperas OVPsim und Synopsys Virtual Platform. Beide unterstützen Prozessoren aus der Automobilindustrie mit verschiedenen Betriebssystemen. Im folgenden wird Imperas OVPsim als Simulatiosumgebung betrachtet und darauf aufbauend die Skalierbarkeit untersucht.

Tabelle 8.1: Übersicht und Bewertung bestehende Lösungen für virtuelle Plattformen von eingebetteten Systemen

| | Unterstütze Prozessoren | Unterstützte Betriebssysteme | Art der Virtualisierung | Eingriff in Systemarchitektur |
|---|---|---|---|---|
| GXemul [137] | ARM, MIPS, Motorola 88000, PowerPC, SuperH | NetBSD, OpenBSD, Linux, Ultrix, Sprite | Emulation | teilweise, da OpenSource |
| OKL4 Microvisor [37] | ARM | Linux, Android, QNX | Paravirtualisierung | nein |
| Imperas OVPsim [138] | OpenRisc (OR1K), MIPS32, ARC600/700, ARM, RISC, CISC, DSP, VLIW, MicroBlaze, eigene Prozessor Modelle | beliebige Betriebssysteme (inkl. eigener) | Gesamtsystemsimulation mit optionaler Simulation von Peripherie und Prozessoren | ja |
| Synopsys Virtual Platform [157] | ARM, MIPS, PowerPC, Renesas SH, Texas Instruments, Tensilica, ZSP | Abhängig von CPU (Linux, WindowsCE, Symbian, inkl. einener) | Gesamtsystemsimulation mit Simulation der Hardware und Schnittstellen | ja |
| QEMU [150] | x86, x64, ARM, CRIS, LM32, MicroBlaze MIPS, OpenRisc32, PowerPC, SPARC, Unicore | Linux, FreeBSD, OpenBSD | Gesamtsystem-Emulation mit Hardware und Schnittstellen | teilweise, da OpenSource |
| Simics [161] | ARM, MIPS, Cavium, Broadcom MIPS, Freescale Power Architecture, IBM Power Architecture, SPARC v8/v9, Renesas H8/SH | unmodified Software, QNX, Linux, Solaris, Windows, FreeBSD, TinyOS, VMware ESX | Simulation von Prozessoren, MMU, Speicher, Netzwerk, aber nur auf selbem Prozessor | nein |

## 8.2 Einfluss des Prozessortyps

Innerhalb der Steuergeräte werden verschiedene Prozessortypen eingesetzt (siehe Kapitel 2.1.1), die in der Simulation abgebildet werden müssen. Die Laufzeit der Funktion hängt dabei zum einen von der Anzahl der Instruktionen in der Anwendung ab, zum anderen hat der Instruktionssatz des verwendeten Prozessors einen maßgeblichen Einfluss. Je komplexer ein Instruktionssatz (CISC[1]) gestaltet ist, desto schneller werden spezifische Funktionen abgearbeitet, da hierfür spezielle Instruktionen existieren. Im Gegensatz hierzu müssen in einem reduzierten Instruktionssatz (RISC[2]) mehrere Instruktionen aufgerufen werden, um das selbe Ergebnis zu erzielen. Ein kleinerer Instruktionssatz reduziert dafür den benötigten Speicher des Programms, da die kodierte Länge der Instruktionen von der Gesamtzahl der zu kodierenden Instruktionen abhängt. Daher wird anhand der Anforderungen entschieden, welcher Instruktionssatz verwendet wird.

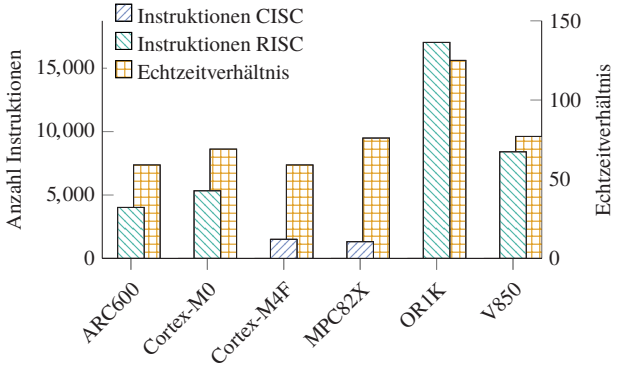



Abbildung 8.1: Vergleich der ausgeführten Instruktionen auf verschiedenen Prozessoren

In Abbildung 8.1 sind verschiedene Automotive Prozessoren mit einer identischen Applikation gegenübergestellt. Zum Vergleich der Anzahl an Instruktionen der verschiedenen Prozessortypen dient ein ACC, welches in der Hochsprache C/C++ mit rund 500 Codezeilen geschrieben und für den jeweiligen

[1] komplexen Befehlssatz (engl. Complex Instruction Set Computer) (CISC)
[2] reduzierter Befehlssatz (engl. Reduced Instruction Set Computer) (RISC)

Prozessor kompiliert und optimiert wurde. Durch die Abbildung der Applikation auf verschiedene Befehlssätze ergeben sich die Unterschiede der Anzahl an Instruktionen. In einem OR1K wird ein RISC mit einem sehr kleinen Befehlssatz verwendet, wodurch Funktionen in mehr Instruktionen (rund 17.000 Instruktionen) abgebildet werden müssen. Der Gegensatz hierzu ist bei einem Cortex-M4F zu sehen, der als CISC mit sehr vielen Spezialbefehlen arbeitet und daher mit weniger Instruktionen (rund 1.500 Instruktionen) für die gleiche Applikation auskommt.

Die Anzahl der Befehle fließt während der Simulation proportional in die Simulationszeit ein. Das heißt, eine Verdopplung der Befehle in der Applikation führt zu einer doppelt so langen Simulationszeit. Für das Verhalten der Applikation in der Simulation ist es jedoch unerheblich, ob ein RISC oder CISC Behelfssatz verwendet wird. Beide Befehlssätze müssen von der Simulation des virtuellen Prozessors gleichermaßen interpretiert werden können. Die Effizienz der Interpretation des Befehlssatzes hängt zudem ebenfalls von dem Befehlssatz ab. Während ein RISC durch den Simulationscomputer direkt abgebildet werden kann, sind Spezialbefehle eines CISC nicht zwingend im Befehlssatz de Simulationscomputers vorhanden und müssen daher in mehreren Befehlen simuliert werden.

Befehle eines RISC werden durch einen realen Prozessor in weniger Taktzyklen ausgeführt als bei einem CISC Prozessor. Da beide Befehlstypen in der Simulation jedoch auf die gleichen Befehlssätze abgebildet werden (den Befehlssatz des Simulationscomputers), entsteht für die Simulationslaufzeit hierdurch kein Unterschied, ob ein RISC oder CISC Prozessor simuliert wird. Da durch diese Abbildung Unterschiede zwischen der Simulationszeit und der tatsächlichen Applikationszeit entstehen, werden diese im Folgenden unterschieden. Die Simulationszeit ist dabei die Zeit, die benötigt wird, um auf einem Computer die Applikation zu simulieren. Die Applikationszeit ist die Zeit, die benötigt wird, um die Applikation auf einem realen Steuergerät auszuführen.

Bildet man den Quotienten aus Applikationszeit und Simulationszeit, so erhält man das Echtzeitverhältnis (siehe Gleichung 8.1).

$$Echtzeitverhältnis = \frac{Applikationszeit}{Simulationszeit} \tag{8.1}$$

Durch das Echtzeitverhältnis wird angegeben, wie viel die Simulation schneller als die Ausführung auf der realen Hardware ist. Das Echtzeitverhältnis der verschiedenen Befehlssätze für die Simulation mit einem virtuellen Prozessor ist in Abbildung 8.1 dargestellt. Dabei macht es für das Echtzeitverhältnis kaum einen Unterschied, ob ein RISC oder CISC simuliert wird. Bei der Simulation eines einzelnen Prozessors beträgt die Simulationsbeschleunigung in der Regel zwischen 59 und 77. Einzige Ausnahme bildet der OR1K, dessen Befehlssatz im gewählten Simulator besonders effizient dargestellt wird und dadurch in der Simulation 125-fach schneller ist als die Applikationszeit.

## 8.3 Grenzen des Echtzeitverhältnis von Multiprozessor-Systemen

Im Fahrzeug bestehen Funktionen jedoch nicht aus einem einzigen Prozessor, sondern aus der Verknüpfung mehrerer Einzelsysteme. So werden beispielsweise Daten von einem Steuergerät eingelesen, von einem weiteren System verarbeitet und von einem Aktor ausgeführt. Für den Test der Sicherheit im Fahrzeug und der damit verbundenen Betrachtung des Gesamtsystems muss die Testmethodik daher für vernetzte Steuergeräte skalieren.

Abbildung 8.2 zeigt einen exemplarischen Aufbau mit mehreren Steuergeräten, die über ein gemeinsames Bussystem kommunizieren. Um die Grenzen der Simulation mit virtuellen Plattformen zu testen, wurde die Anzahl der Steuergeräte und damit der Prozessoren schrittweise inkrementiert. Jedes Steuergerät besteht dabei aus einen Prozessor, einem Speicher, einer Peripherie für Test-Stimuli und der Kommunikation zu anderen Teilnehmern. Als verteilte Applikation dient dabei ein vereinfachter ACC. In Steuergerät 1 (Sensor) werden statische Eingaben eingelesen und zyklisch alle 1 ms über das Bussystem an Steuergerät 2 versendet. Steuergerät 2 wertet die Daten aus, berechnet einen Stellwert und übermittelt diesen zyklisch alle 1 ms über das Bussystem an Steuergerät 3 (Aktor). Für die Messungen wurden die Prozessoren in allen Steuergeräten anhand eines ARM Cortex-M4F (CISC Prozessor mit 180 MHz und bis zu 360 MIPS) umgesetzt und für 750 Sekunden zu 100 % ausgelastet, ohne die Prozessoren in Energiespar- oder Ruhemodus zu versetzen. Das heißt, in der Simulation wird für jeden Prozessor in jedem Zeitschritt eine Instruktion

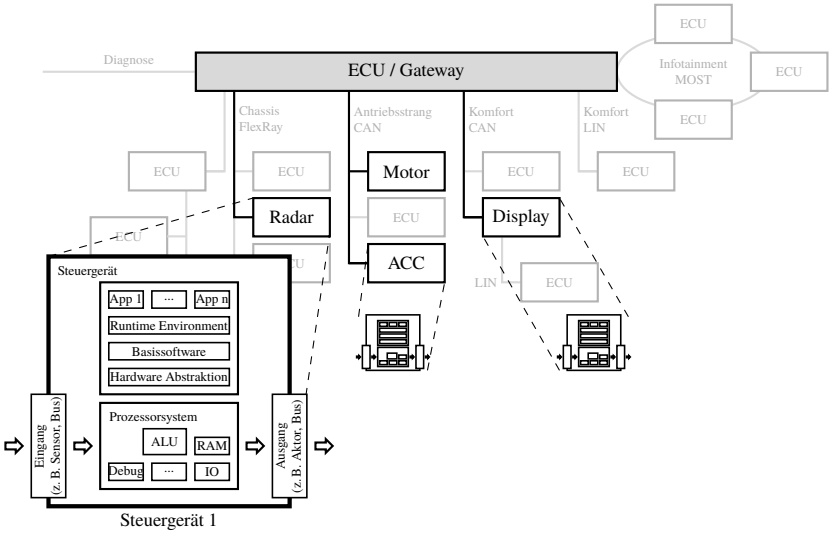



Abbildung 8.2: Aufbau der Simulation mit mehreren kommunizierenden Steuergeräten

ausgeführt. Die Peripherie wurde über statische Eingaben stimuliert, um den Einfluss einer Co-Simulation auszuschließen.

Um die Auslastung der Simulation festzustellen, wurde das System aus Abbildung 8.2 auf einem Linux PC[3] ausgeführt. Die Resultate sind in Abbildung 8.3 dargestellt.

Die in Kapitel 8.1 ausgewählte Simulationsumgebung (Imperas OVPsim) skaliert dabei linear für die Simulationszeit in Abhängigkeit von der Prozessoranzahl. Das heißt, eine Verdopplung der Prozessoren oder Steuergeräte (mit einem Prozessor) führt zu einer Verdopplung der Simulationszeit. Dies ist auch an der konstanten Zahl von etwa 600 MIPS zu erkennen. Die MIPS umfassen dabei nur die Instruktionen der Applikationen, nicht aber des Simulationscomputers. Durch die Ausführung der Linux-Betriebssystemprogramme schwankt die tatsächlich ausgeführte Zahl der MIPS in Abbildung 8.3. Besonders für die Simulation von wenigen Prozessoren, wird zwischen den simulierten In-

[3] Simuliert wurde auf einem Desktop-Computer mit 1,2 GHz und 8 GB RAM

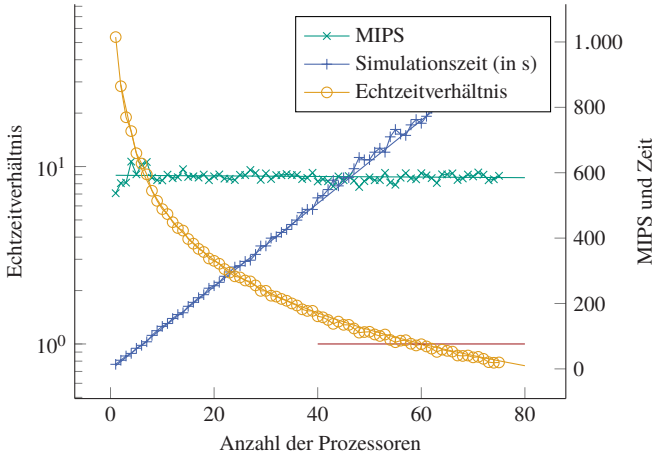



Abbildung 8.3: Vergleich der Simulationszeit und der Beschleunigung in Abhängigkeit der Prozessorzahl anhand eines ACC

struktionen des virtuellen Prozessors häufig eine Aufgabe des Betriebssystems ausgeführt. Während sich bei einer höheren Anzahl an simulierten Prozessoren die Auslastung des Simulationscomputers erhöht und weniger Betriebssystemaufgaben ausgeführt werden.

Das Echtzeitverhältnis aus Abbildung 8.3 ergibt sich als Beschleunigung der ausgeführten Applikation in der Simulation im Vergleich zur Realzeit und wird nach Gleichung 8.1 berechnet. Da die Applikationszeit mit 750 Sekunden konstant ist, ergibt sich die gezeigte Hyperbel. Zur besseren Übersicht, ist diese in Abbildung 8.3 logarithmisch aufgetragen. Die rote Markierung zeigt dabei das Echtzeitverhältnis von eins, bei dem Applikationszeit und Simulationszeit identisch sind.

Bei Ausführung der Simulation auf dem oben angegebenen Linux PC ergibt sich, dass mindestens 62 Steuergeräte parallel simuliert werden können, bevor die Simulationszeit langsamer als die Applikationszeit wird und dadurch eine Ausführung auf der realen Hardware schneller ist. Da die Simulationszeit linear ansteigt, kann die maximale Anzahl der Steuergeräte überschlagsmäßig mit dem Echtzeitverhältnis bei der Ausführung eines einzelnen Steuergeräts gleichgesetzt werden (siehe Gleichung 8.2).

$$max_{Steuergeräte} = Echtzeitverhältnis_{1\ Steuergerät} \tag{8.2}$$

Durch den Einsatz von Energiespar- oder Ruhemodi in den Prozessoren können simulierte Instruktionen eingespart und damit die tatsächliche Simulation sogar weiter beschleunigt werden.

Die Simulation kann durch den Einsatz eines schnelleren Computers oder parallelisiertes Rechnen auf mehreren Prozessorkernen eines Computers ebenfalls beschleunigt werden. Eine Skalierung der Ergebnisse erfolgt dann linear zur Erhöhung des Prozessortakts.

Die Untersuchungen erfolgten anhand eines ARM Cortex-M4F mit CISC Prozessor. Bei anderen Prozessoren verhält sich die Skalierung analog, durch die Änderung des Befehlssatzes und der damit verbundenen höheren Anzahl an Instruktionen ergibt sich jedoch eine entsprechend längere Simulationszeit und eine Änderung des Echtzeitverhältnisses (siehe auch Kapitel 8.2). Das Echtzeitverhältnis steht dabei im Verhältnis zur Änderung der Instruktionslänge und der damit Verbundenen Laufzeit je Instruktion. Nutzt man hierfür Gleichung 6.8 ergibt sich das Echtzeitverhältnis aus Gleichung 8.3.

$$Echtzeitverhältnis = \frac{\sum_{i=1}^{n} Länge_{Instruktion,i}}{Prozessorfrequenz \cdot Simulationszeit} \tag{8.3}$$

## 8.4 Verlängerte Simulationszeit durch Überwachungsfunktionen

Durch das Einbinden der Überwachungsfunktionen wird die Simulation verlangsamt. Beim Erkennen einer Instruktion oder eines Speicherzugriffs aus der Beobachtungskonfiguration wird die Simulation gestoppt und die Überwachungsfunktion gestartet (siehe Kapitel 6.4). Die Funktionalität der Überwachungsfunktion reicht dabei vom Protokollieren des Zugriffs bis hin zum Eingriff in die Ausführung der Instruktion. Die Verlängerung der Simulationszeit durch die Überwachung ist dabei abhängig von der implementierten Funktionalität. Beim Eingriff in die Ausführung der Instruktion ist die Länge der Überwachungsfunktion von der Implementierung des Testingenieurs abhängig. Je mehr Parameter in der Überwachungsfunktion analysiert werden, umso länger wird die Simulation unterbrochen.

Für jeden Aufruf der Überwachungsfunktion wird zudem eine konstante Laufzeit auf die Simulationsdauer addiert, um die Simulation zu stoppen, die Überwachungsfunktion aufzurufen und anschließend die Simulation fortzusetzen. Auf dem oben verwendeten Linux-PC dauert dies 0,1 µs je Unterbrechung.

Die entstandene kumulierte Gesamtlaufzeit der Überwachungsfunktionen hängt somit von zwei Faktoren, der implementierte Funktionalität und Anzahl der Aufrufe der Überwachungsfunktionen ab (siehe Gleichung 8.4). Wobei die Anzahl der Überwachungsfunktionen von den zu überwachenden Variablen, Speicherbereichen und Instruktionen abhängig ist.

$$Überwachungszeit = Anzahl_{Überwachungen} \cdot \left(0,1\mu s + Zeit_{Überwachungsfunktion}\right) \quad (8.4)$$

Addiert man die Überwachungszeit zur Simulationszeit aus Gleichung 8.1 ergibt sich das gesamte Echtzeitverhältnis (inkl. Überwachungsfunktion) in Abhängigkeit der Applikationszeit als Gleichung 8.5.

$$Echtzeitverhältnis = \frac{Applikationszeit}{Simulationszeit + Überwachungszeit} \quad (8.5)$$

Da die Überwachungszeit einen reziproken Einfluss auf das Echtzeitverhältnis hat, wird damit ebenfalls die maximale Anzahl an Steuergeräten beeinflusst, bevor die Applikation auf einer realen Hardware schneller als in der Simulation ausgeführt wird. Passt man Gleichung 8.2 entsprechend an, ergibt sich:

$$max_{Steuergeräte} = \frac{Applikationszeit}{Simulationszeit_{1\ Steuergerät} + Überwachungszeit} \quad (8.6)$$

Führt man nun die Applikation aus Kapitel 8.3 mit Überwachungsfunktionen aus, so steigt die Simulationszeit proportional zur Anzahl der Bobachtungspunkte (siehe Abbildung 8.4).

Durch einen Anstieg der Simulationszeit fällt das Echtzeitverhältnis in gleichem Maße reziprok ab, wodurch die maximale Anzahl an Steuergeräten die simuliert werden können abnimmt, bevor die Simulationszeit langsamer als die Applikationszeit wird. Beide Verhältnisse ergeben sich durch den Aufruf der

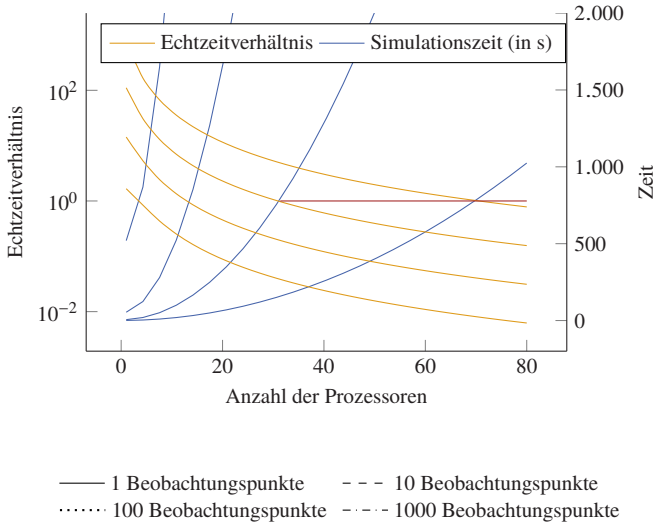



Abbildung 8.4: Vergleich der Simulationszeit und der Beschleunigung in Abhängigkeit der Überwachungsfunktionen anhand eines ACC

Überwachungsfunktionen[4] bei Zugriff auf den Inhalt der Beobachtungspunkte (siehe Gleichung 8.4 und Gleichung 8.5). Die Beobachtungspunkte rufen in der ausgeführten Applikation die jeweilige Überwachungsfunktion im Schnitt 100 mal durch Lesen/Schreiben eines Speicherbereichs oder Ausführen einer beobachteten Funktion auf. Zusätzlich erzeugt eine Co-Simulation basierend auf den Ausgabewerten neue Stimuli.

Bei Ausführung der Simulation mit 100 Beobachtungspunkten reduziert sich die maximale Anzahl der Steuergeräte bereits auf 17. Erhört man die Beob-

[4] Neben der Zeit zum Aufrufen der Überwachungsfunktion (0,1 µs) benötigt die Überwachungsfunktion je Aufruf beispielhaft 1 ms zur Analyse und Verarbeitung der Beobachtungspunkte, sowie zur Co-Simulation. Benötigt die Co-Simulation mehr Ausführungszeit, reduziert sich das Echtzeitverhältnis weiter.

achtungspunkte auf 1000, könnten 3 Steuergeräte[5] parallel simuliert werden, bevor das Echtzeitverhältnis kleiner 1 wird.

## 8.5 Zusammenfassung

Für die Simulation einer verteilten Funktion müssen mehrere Steuergeräte simuliert werden. Daher muss die Testmethodik skalieren. Hierfür wurden zunächst Simulatoren bewertet und ein Gesamtsystemsimulator ausgewählt. Aufgrund späterer Anpassungen in der Applikation sind Paravirtualisierungen und Emulatoren für die Testmethodik ungeeignet.

Zudem ist die Simulation von virtuellen Plattformen abhängig von der gewählten Prozessorarchitektur im Steuergerät, da durch verschiedene Prozessortypen der Befehlssatz auf die Plattform unterschiedlich ist. Abhängig von der Komplexität des Instruktionssatzes wird eine unterschiedliche Anzahl an Instruktionen in der Applikation verwendet, wodurch die Laufzeit der Applikation variiert. Für die Simulationszeit in Abhängigkeit der Anzahl an Instruktionen gilt, dass eine Verdopplung der Instruktionen zu einer Verdopplung der Simulationszeit führt. Durch die Abbildung der Applikation in der Simulation kann die Applikation beschleunigt ausgeführt werden, wodurch sich das Echtzeitverhältnis nach Gleichung 8.1 ergibt.

Variiert man die Anzahl der simulierten Steuergeräte, ergibt sich, dass eine steigende Anzahl an Steuergeräten und damit der Prozessoren exponentiell mit der Simulationszeit skaliert und das Echtzeitverhältnis reziprok als Hyperbel abfällt. Bei der Analyse der Skalierbarkeit sind zudem die simulierten Millionen Instruktionen pro Sekunde (MIPS) konstant, unabhängig der Anzahl an Steuergeräten und Prozessoren, das dem linearen Wachstum der Simulationszeit bei steigender Prozessorenzahl entspricht.

Durch den Aufruf der Überwachungsfunktionen entsteht eine Verlängerung der Simulationszeit, die eine konstante Anzahl an Funktionen zum Aufruf der Überwachungsfunktion benötigt und daher eine konstante Laufzeit von je 0,1 µs

[5] Eine Simulation mit 1000 Beobachtungspunkte war mit dem simulieren ACC nicht möglich, da dieses aus nur 1700 Codezeilen besteht. Daher ergibt sich der Wert aus theoretischen Berechnungen der vorgestellten Formeln.

pro Unterbrechung addiert. Die genaue Ausführungszeit der Überwachung ist abhängig von der Länge der Überwachungsfunktion und den darin enthaltenen Operationen. Da die Simulationszeit in das Echtzeitverhältnis einfließt, reduzieren die Überwachungsfunktionen das Echtzeitverhältnis und damit die Anzahl an parallel simulierten Steuergeräten bevor die Simulationszeit langsamer als die Applikationszeit wird.

Obwohl die vorgestellte Testmethodik beim Einsatz von Überwachungsfunktionen für 100 Beobachtungspunkten und Co-Simulation 35 fach schneller als die Ausführung auf einem realen Steuergerät ist, sind Tests auf dem realen Steuergerät dennoch nötig. Die Genauigkeit der Tests hängt von der Genauigkeit der Modelle ab. Um eine realitätsnahe Gesamtsystemsimulation darzustellen, müssen die Modelle ausreichend genau dargestellt sein sowie die Co-Simulation die Umwelt in allen Details simulieren. Eine Erhöhung der Genauigkeit erhöht jedoch auch die Kosten beim Erstellen der Modelle und reduziert das Echtzeitverhältnis während der Simulation, da sich für die Co-Simulation der Faktor $Zeit_{Überwachungsfunktion}$ aus Gleichung 8.4 und für die Steuergerätesimulation der Faktor $Applikationszeit$ aus Gleichung 8.5 erhöht.

# 9 Bewertung der Security-Tests

Abschließend wird die vorgestellte Testmethodik zum Testen der Datensicherheit durch virtuelle Steuergeräte bewertet. Da nicht immer alle Beobachtungspunkte des Steuergeräts und der Software zur Verfügung stehen, wird dies anschließend auf Black-Box-Tests übertragen und die Auswirkungen aufgezeigt. Aufbauend auf der Testmethodik, wird das Vorgehen abschließend anhand eines Fallbeispiels zusammengefasst.

## 9.1 Einsatz der Testmethodik auf Angriffe in der Automobilindustrie

In Kapitel 3.4 wurden Angriffe auf Fahrzeuge dargestellt und in Kapitel 4.4 klassifiziert. Um die Wirksamkeit der Testmethodik zu bewerten, werden die Schwachstellen der aufgezeigten Angriffe im Folgenden untersucht und bewertet, ob diese mit der vorgestellten Testmethodik erkannt werden können. Angriff 4 und Angriff 7 aus Kapitel 3.4 sind dabei nicht weiter berücksichtigt, da derzeit keine detaillierten Informationen zum Nachbau des Angriffs vorhanden sind. Angriff 1 wurde bereits in Kapitel 4.4 von einer näheren Betrachtung ausgeschlossen, da hier ein direkter Zugang zum Fahrzeug nötig ist, um ein zusätzliches Steuergerät am Bussystem anzubringen.

Eine Übersicht, welche Angriffe durch die Testmethodik erkannt werden können, ist in Tabelle 9.1 dargestellt.

In Angriff 2 wurde durch eine Funktion zum Abspielen von Musiktiteln ein Buffer Overflow (siehe Kapitel 2.4.5) erzielt, wodurch schadhafte Software auf das Infotainment aufgebracht werden konnte [19]. Die Auswirkungen des Buffer Overflows können innerhalb des Speichers festgestellt werden, obwohl das Verhalten an den Schnittstellen beim Abspielen der Musik der Spezifikation entspricht. Der Overflow (siehe Kapitel 2.4.5) kann beim Testen mit der

Tabelle 9.1: Übertragung der Testmethodik auf die Angriffe aus Kapitel 3.4

| ID | Schnittstelle | Angriffstyp | Erkennung |
|---|---|---|---|
| 2 | Infotainment | Buffer Overflow, Code Injection | ✓ |
| 3 | Telematik | Security Features | ✗ |
| 5 | Information Display | Buffer Overflow, Code Injection | ✓ |
| 6 | Multimedia | Input Validation, Format String Vulnerability | (✓) |
| 8 | Funkschlüssel | Security Features, Cryptographic Issues | ✗ |
| 9 | Telematik | Permission, Privileges and Access Control | ✗ |
| 10 | Telematik | Buffer Overflow, Code Injection, Data Handling, Input Validation | ✓ |

vorgestellten Testmethodik durch eine Speichermarkierung (siehe Kapitel 6) erkannt werden. Dabei wird festgestellt, dass Speicher außerhalb der gültigen Variablengrenzen überschrieben wird. Bedingung hierfür ist jedoch der Einblick in den Speicher, da das Fehlverhalten bei den getesteten Testfällen nicht an den Schnittstellen ersichtlich ist.

Angriff 3 nutzte einen Fehler im Telematik-System aus, um die Firmware über das Mobilfunknetz zu verändern [70]. Das Problem ließ sich auf unzureichende Sicherheitsmechanismen zurückführen, wodurch es möglich war, das Wi-Fi Passwort des Fahrzeugs zu erraten. Grund hierfür ist, dass das Passwort beim ersten Einschalten des Systems mithilfe der Uhrzeit gesetzt wurde, die Uhrzeit jedoch erst gestellt wurde, nachdem das Passworts erfolgreich gesetzt war. Ein solcher systematischer Fehler wird durch die Testmethodik nicht erkannt, da hierbei das Verhalten von Prozessor, Speicher und Peripherie korrekt funktioniert haben.

Angriff 5 wurden Speicherbereiche durch einen Buffer Overflow überschrieben, um Administrationsrechte zu erhalten. Diese Speicherüberschreitungen können mit der vorgestellten Methodik erkannt werden. Damit wäre ein deaktivieren der Sicherheitsmechanismen und überschreiben der Software nicht möglich gewesen. Das Deaktivieren der Sicherheitsmechanismen und das Überschreibend er Software kann mit der Methodik nicht erkannt werden, da dies Vorgänge sind, die ein Administrator auf dem Steuergerät durchführen darf.

Durch Sonderzeichen im Namen eines Musiktitels wurde in Angriff 6 das Ausschalten des Multimediasystems bewirkt [143]. Grund für das Ausschalten war, dass Sonderzeichen beim Einlesen nicht berücksichtigt wurden, sondern falsche Funktionsaufrufe starteten. Falsch formatierte Eingabewerte werden im virtuellen Steuergerät durch die Überwachung der Peripherie erkannt. Jedoch wäre das Problem beim Testen mit SiL ebenfalls aufgefallen, sofern Sonderzeichen in den Testfällen enthalten sind. Durch ein zusätzliches Ausführen von Fuzz-Tests kann dieser Fehler auch ohne die vorgestellte Testmethode erkannt werden.

Im Funkschlüssel aus Angriff 8 wurde die Verschlüsselung zwar aktiviert, jedoch nicht mit der spezifizierten Länge des Schlüssels [127]. Durch die Verkürzung des Schlüssel konnte dieser erraten und damit der Funkschlüssel kopiert werden. Dieser Angriff wird durch die Testmethodik nicht erkannt, da es sich hierbei um Fehler in der Umsetzung der Spezifikation handelt. Gleiches gilt für Angriff 9, die Rechteverwaltung kann nicht im Speicher erkannt werden, da die jeweiligen Zugriffe aus gültige Bereiche des Speichers lesen.

Im Angriff auf das Telematik-System in Angriff 10 wurden Speicherbereiche unerlaubt überschrieben, um schadhafte Software auf dem System auszuführen. Ein solcher Buffer Overflow (siehe Kapitel 2.4.5) kann durch die Testmethodik erkannt werden. Jedoch sind zum jetzigen Zeitpunkt noch nicht genügend Informationen des Angriffs veröffentlicht, um diesen nachzubauen und zu testen.

## 9.2 Fallbeispiel zur Validierung der Testmethodik anhand unabhängiger Software

Die Erkennung der Angriffe auf die Automobilindustrie (siehe Kapitel 9.1) zeigt, dass Rechen- und Speicherfehler auf virtuellen Steuergeräten erkannt werden können. Da die Probleme der Angriffe jedoch zuvor studiert und daher bekannt waren, sind diese Angriffe für eine Validierung der Testmethodik nicht repräsentativ. Aus diesem Grund wird die Testmethodik im Folgenden anhand einer unabhängigen Software validiert, deren Verhalten und Schnittstellen, nicht jedoch die konkrete Implementierung, bekannt ist.

Als unabhängige Software dienen Sensormodule, Motorsteuerung und Regler eines für die Lehre eingesetzten selbst-balancierenden einachsigen Beförderungsmittels (Segway, siehe Abbildung 9.1) sowie einer Vernetzungseinheit, um das Segway und ein Computer über Bluetooth zu verbinden (siehe Abbildung 9.2).

Um den Segway zu regeln, kontrolliert die Motorsteuerung die Geschwindigkeit der beiden Räder und deren Drehrichtung. Über Sensoren misst die Regelung den Winkel der Lenkstange (Lenkwinkel), um die Fahrtrichtung über unterschiedliche Geschwindigkeiten der beiden Räder zu steuern. Zusätzlich wird über Drehrate (Neigung des Segways), Beschleunigung und Geschwindigkeit beider Räder des Segways die Soll-Geschwindigkeit berechnet. Durch eine optionale Vernetzungseinheit können Parameter des Segways (Geschwindigkeit, Beschleunigung, etc.) über Bluetooth ausgelesen oder Steuerbefehle an den Segway (Notstopp, Geschwindigkeitsbegrenzung) übermittelt werden.

Abbildung 9.1: In der Lehre eingesetztes Segway

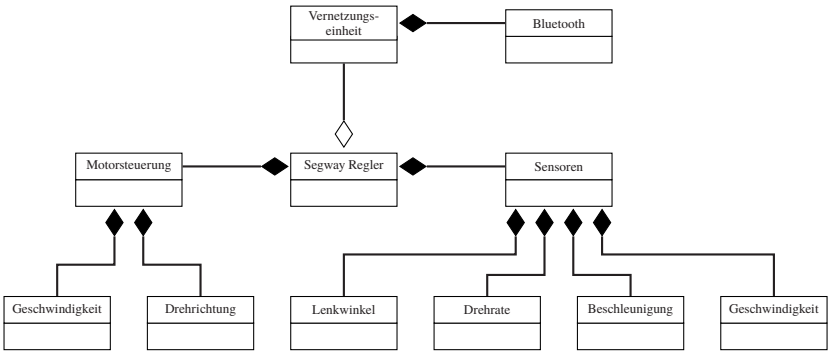



Abbildung 9.2: Klassendiagramm zur Übersicht der Module des in der Lehre eingesetzten Segways

### 9.2.1 Identifizierte Fehler in Software des Fallbeispiels

Die Module der Motorsteuerung und Sensoren wurden im Rahmen einer Lehrveranstaltung von verschiedenen Studierenden implementiert. Dadurch können die Testmethoden (siehe Kapitel 6) auf Softwaremodule angewandt werden, deren Implementierungen unbekannt sind. Für die Validierung der Software sind jedoch das spezifizierte Verhalten und die Schnittstellen bekannt sowie das Steuergerät bereits als Simulationsmodell für das Simulationstool OVP (siehe Kapitel 8) vorhanden. Für die Simulation wurde das Modell eines Xilinx MicroBlaze mit Peripherie (Analog-Digital-Wandler (ADC), GPIO, Pulsweitenmodulation (PWM) und Timer) erstellt [WLBS16].

Die Module des Segways wurden anhand von Testfahrten mit 105.35 Sekunden Laufzeit und den vorgestellten Testmethoden validiert. Hierfür wurden zu jedem Zeitschritt die Sensorwerte (Analogwert des Lenkwinkels, der Drehrate zur Bestimmung der Neigung des Segways und der Beschleunigung) sowie Parameter der Vernetzungseinheit eingelesen. Nach 100 simulierten Testfahrten mit mutierten Stimuli wurden die Testmethoden abgebrochen. Auf einem virtualisierten Steuergerät wurden zuerst funktionale Tests der Lehrveranstaltung und anschließend die Testmethodik (siehe Kapitel 6) mit evolutionärem Vorgehen angewandt. Dabei wurden durch den evolutionären Algorithmus die Stimuli einer Testfahrt von 105,35 Sekunden (Applikationszeit) variiert und durch die Gesamtsystemsimulation innerhalb von 2,89 Sekunden simuliert (Simulationszeit).

In den Modulen des Lenkwinkelsensors, Motorsteuerung und der Vernetzungseinheit wurden dabei Fehler identifiziert.

#### Lenkwinkelsensor

Vor dem Start des Segways wird der maximal nutzbare Bewegungsbereich der Lenkstange ermittelt, da sich der gemessene Spannungsbereich aufgrund verschieden positionierter Potentiometer, von Segway zu Segway unterscheidet. Hierfür muss die Lenkstange zuerst bis zum mechanischen Anschlag nach rechts (minimaler Lenkwinkel) und anschließend bis zum mechanischen Anschlag nach links (maximaler Lenkwinkel) bewegt werden. Diese beiden Werte dienen anschließend zur Normierung des gemessenen Lenkwinkels auf das In-

tervall [-1,1] und werden genutzt, um die relative Bewegung der Lenkstange zu ermitteln. Abschließend wird der Wert an den Segway Regler übergeben und zyklisch mit einer Frequenz von 10 Hz eine neue Messung gestartet (siehe Abbildung 9.3).

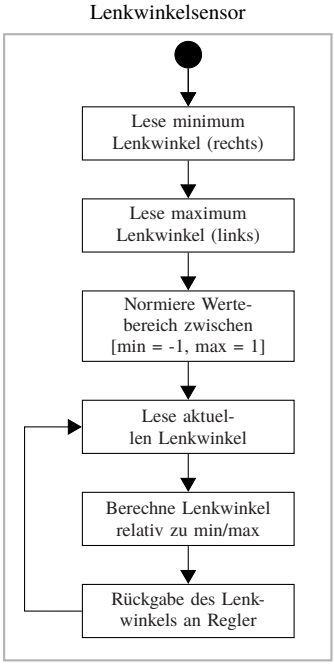



Abbildung 9.3: Ablauf der Initialisierung und Berechnung des Lenkwinkels eines Segways

Durch die evolutionären Testmethoden (siehe Kapitel 6) wurde der Wert der Lenkwinkelposition bei der Initialisierung (linker und rechter mechanischer Anschlag) und die zur Laufzeit gemessenen Lenkwinkel mutiert. Die mutierten Werte wurden als Stimuli der Simulation verwendet. Neben den Abbruchkriterien der verschiedenen Methoden wird die Simulation nach 100 Durchläufen abgebrochen.

Während der Simulation wurden durch Überwachung der Instruktionen (siehe Kapitel 6.4, Methode i), Minimierung des Variableninhalts (siehe Kapitel 6.4, Methode ix) und Maximierung des Variableninhalts (siehe Kapitel 6.4, Methode x) Fehler im Ablauf erkannt. Wurde zu Beginn der Initialisierung nicht der mechanische Anschlag (min/max) angefahren, so konnte zur Laufzeit ein Wert außerhalb des Intervalls zwischen min und max eingelesen werden. Die Normierung auf den maximal zulässigen Wertebereich führte im folgenden Schritt dazu, dass ein Wert größer als 1 oder kleiner als $-1$ berechnet wurde und zu einem Überlauf des Variablen-Wertebereichs führte. Folge dieses Überlaufs ist, dass der Maximaleinschlag der Lenkstange nach rechts zu einer Lenkbewegung nach links und der Maximaleinschlag der Lenkstange nach links zu einer Lenkbewegung nach rechts führen kann. Ein Begrenzen des Wertebereichs in der Lenkwinkelberechnung, behebt diesen Fehler.

Des weiteren wurde Nulldivision während der Simulation durch die Überwachung der Instruktionen (siehe Kapitel 6.4, Methode i) erkannt. Dabei wurde maximaler und minimaler Lenkwinkel durch Mutation identisch gewählt und

die Differenz beider zu Null. Um den aktuellen Lenkwinkel zu berechnen wird durch diese Differenz dividiert. Durch das Rauschen der Sensoren sind beide Werte zur Laufzeit zwar nie identisch, ein physikalischer Fehler im Potentiometer, eine schlechte Steckverbindung oder Manipulation der Eingabewerte durch andere Module können die Nulldivision jedoch ebenfalls auslösen.

### Motorsteuerung

Für die Motorsteuerung werden mit einer Frequenz von 10 Hz die Soll- und Ist-Drehzahl der Motoren eingelesen und verglichen. In Abhängigkeit der Differenz wurde die elektrische Leistung am Motor durch eine PWM und die Drehrichtung über eine H-Brücke gesteuert (siehe Abbildung 9.4).

Die eingesetzten evolutionären Tests variieren die übergebene Soll-Drehzahl und überwachen die internen Speicherbereiche und ausgeführten Instruktionen. Als zusätzliches Abbruchkriterium dienten 100 Durchläufe der Simulation.

Ein Variieren der Soll-Drehzahl der Motoren außerhalb der Spezifikation führt zu einem Überlauf in der Berechnung der Differenz. Durch den Überlauf kann die Drehrichtung der Motoren während der Fahrt umgekehrt werden.

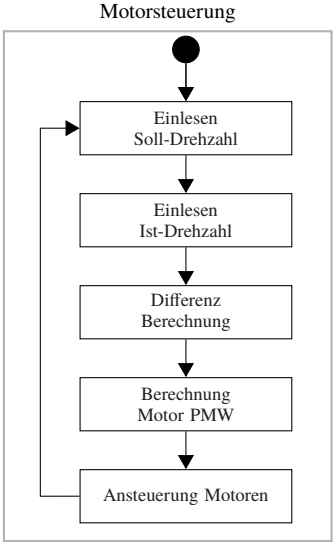



Abbildung 9.4: Ablauf der Motorsteuerung eines Segways

### Vernetzungseinheit

Für die Vernetzungseinheit wurde ebenfalls auf Softwaremodule des Segways zurückgegriffen. Dabei wird über diese Vernetzungseinheit eine Geschwindigkeitsbegrenzung und ein Notstopp realisiert. Hierfür werden zum einen die Daten der Geschwindigkeitsregelung ausgelesen und zur Diagnose über eine Bluetooth-Schnittstelle versendet, zum anderen werden Steuerinformationen

empfangen, um die Geschwindigkeitsregelung einzustellen oder die Motoren des Segways abzuschalten (siehe Abbildung 9.5).

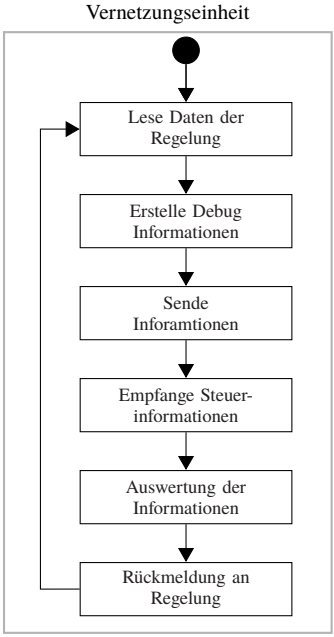



Abbildung 9.5: Ablauf der Vernetzungseinheit zur Kommunikation eines in der Lehre eingesetzten Segways

Für die Tests wurde die Kommunikation (Parameter der Steuerinformationen und Parameter der Aufrechterhaltung[1]) variiert und die internen Speicherbereiche überwacht. Im Laufe der Tests wurden dabei zwei Fehler erkannt.

Zum einen wurde durch Markierung des genutzten Speichers (siehe Kapitel 6.4, Methode iv) und Maximierung des Speicherzugriffs (siehe Kapitel 6.4, Methode vii) erkannt, dass zur Verschlüsslung der Kommunikation ein veraltetes SSL-Modul eingesetzt wurde. Dieses Modul beinhaltete die als "Heartbleed" bekannte Schwachstelle [17]. Bei SSL muss der Empfänger mit einer, zuvor zugesendeten, definierten Antwort bestätigen, dass die Verbindung noch besteht. Hierfür wird dem Empfänger ein Codewort und dessen Länge mitgeteilt (Payload und Padding). Bei der "Heartbleed" Schwachstelle wurde die Länge des Codeworts nicht verifiziert. So konnten Daten aus dem Speicher gelesen werden. Ist die übermittelte Länge größer als das Codewort, so wird ebenfalls der benachbarte Speicherinhalt als Antwort zurückgesendet.

Zum andern wurde durch die Analyse der Eingaben (siehe Kapitel 6.4, Methode iii) und Maximierung des Variableninhalts (siehe Kapitel 6.4, Methode x) eine fehlerhafte Datenübergabe erkannt. Die Verwendung von nicht spezifizierten Steuerinformationen führte zum Überlauf des Inhalts einer Variablen,

[1] Die Aufrechterhaltung ist ein Mechanismus der Datenübertragung mit dem Ziel, eine Kommunikation aufrechtzuerhalten und sich von der Konnektivität und Funktion eines Kommunikationspartners zu überzeugen.

die zum Auslösen des Notstopps benötigt wird. Nach Überlauf der Variablen konnte kein Notstopp ausgelöst werden.

### Segway Regler

Zur Ansteuerung des Segways dient nach der Initialisierung ein zyklischer Ablauf der Regelung (siehe Abbildung 9.6). In der Initialisierung wird dabei die Sensorik und Aktorik aktiviert. Anschließend wird zyklisch mit einer Frequenz von 10 Hz auf interne Fehler geprüft. Treten Fehler (geringe Batteriespannung,

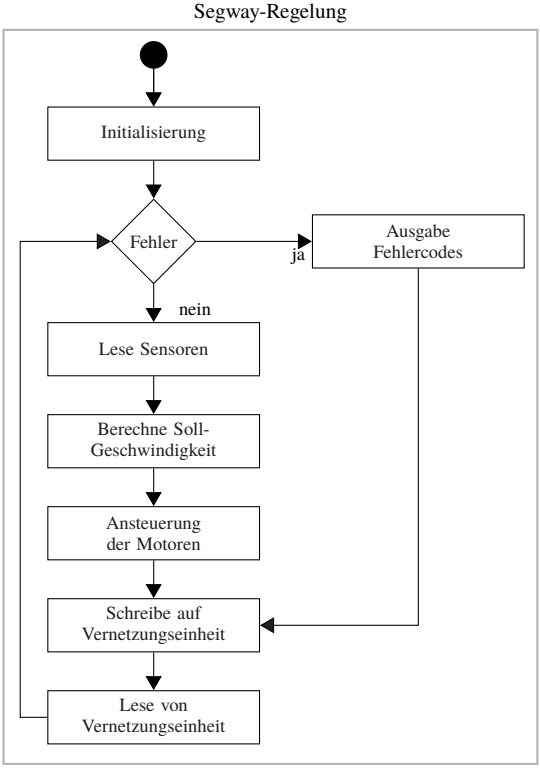



Abbildung 9.6: Ablauf der Regeleinheit des eingesetzten Segways

keine Sensorwerte, etc.) auf, so wird der Segway angehalten und ein Fehlercode ausgegeben. Ist der Segway fahrbereit, werden die Sensoren ausgelesen, eine Soll-Geschwindigkeit berechnet und diese an die Motorsteuerung übermittelt. Über die Vernetzungseinheit werden Parameter des Reglers an gekoppelte Bluetooth Geräte versendet und Steuerbefehle empfangen.

Beim Ausführen der vordefinierten Tests aus der Lehrveranstaltung und der obigen evolutionären Methoden wurde durch die Analysemöglichkeiten der Codeabdeckung (siehe Kapitel 6.4, Methode v) festgestellt, dass nicht alle Pfade gleichermaßen durchlaufen wurden. In keinem der verwendeten Testfälle traten Fehler wie geringe Batteriespannung oder fehlende Sensorwerte auf. Daher wurde die Ausgabe der Fehlercodes nicht überprüft. Um diesen Pfad ebenfalls zu durchlaufen, mussten die Stimuli der Eingabeschnittstellen manuell angepasst werden. Hierfür wurde die Batteriespannung variiert und Sensorwerte während den Tests abgeschaltet. Durch beides konnten zwar keine weiteren Fehler injiziert werden, die Analysemöglichkeiten der Codeabdeckung stellte allerdings eine Testabdeckung aller Pfade in der Software sicher.

### 9.2.2 Bewertung der identifizierten Fehler

Die Software wurde auf einem virtuellen Steuergerät simuliert und durch Testmethoden anhand Beobachtungspunkten überwacht. Die Variablen des Quellcodes wurden anhand der Variablendeklarationen ermittelt. Variablen wurden im Maschinencode durch einen Disassembler erkannt und beziehen sich auf Speicherbereiche die nach Aufruf einer Funktion allokiert wurden. Als Instruktionen des Maschinencodes wurden zur Laufzeit die kritischen Funktionen (siehe Anhang A) beobachtet. Diese Instruktionen werden ebenfalls durch die Arithmetic Logic Unit der CPU beobachtet, wodurch sich eine doppelte Aufführung in Tabelle 9.2 ergibt. Als Beobachtungspunkte der Daten- und Programmspeicher ergeben sich neben den Variablen im Quellcode und Variablen im Maschinencode weitere manuell hinzugefügte Speicherbereiche. Diese wurden für die Tests nicht genutzt, da keine Kenntnis über die Speicherstruktur der Software existierte. Die Beobachtete Prozessorperipherien sind für den Lenkwinkelsensor der ADC und für die Motorsteuerung eine PWM je Motor.

Um die Methoden auszuführen wurden Beobachtungspunkte des virtuellen Steuergeräts eingesetzt (siehe Kapitel 6.3.4) und greifen dabei auf eine unterschiedliche Anzahl der Beobachtungspunkte zu (siehe Tabelle 9.2).

Tabelle 9.2: Anzahl der Beobachtungspunkte zur Ausführung der Tests aus Kapitel 9.2.1

| Software \ Beobachtungspunkte | a) Variablen im Quellcode | b) Instruktionen im Quellcode | c) Variablen im Maschinencode | c) Instruktionen im Maschinencode | e) Register der CPU | f) Arithmetic Logic Unit der CPU | g) Daten- und Programmspeicher | h) I/O Schnittstellen | i) Prozessorperipherie | **Summe** |
|---|---|---|---|---|---|---|---|---|---|---|
| Lenkwinkelsensor | 12 | 0 | 2 | 5 | 0 | 5 | 0 | 2 | 1 | **27** |
| Motorsteuerung | 12 | 0 | 1 | 5 | 0 | 5 | 0 | 4 | 2 | **29** |
| Vernetzungseinheit | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | **4** |
| Segway Regler | 16 | 0 | 9 | 10 | 0 | 10 | 0 | 6 | 0 | **51** |

Um die Software zu analysieren, wurden zum einen funktionale Tests der Lehrveranstaltung, zum anderen statischen und evolutionären Testmethoden (siehe Kapitel 6.4) eingesetzt. Für die funktionalen Tests wurden drei verschiedene Probefahrten aufgezeichnet und diese mit Hilfe des virtuellen Steuergeräts simuliert. Die Testfahrten dauerten 100,35 Sekunden und wurden innerhalb 2,89 Sekunden simuliert, wodurch sich ein Echtzeitverhältnis von 35, 77 ergibt [WLBS16]. Die oben dargestellten Schwachstellen wurden durch verschiedene evolutionären Algorithmen entdeckt (siehe Kapitel 9.2.1 und Tabelle 9.3).

Die funktionalen Tests wurden durch das Vorgehen der statischen Methoden (siehe Kapitel 6.4, Methoden i bis v) und der evolutionären Methoden (siehe Kapitel 6.4, Methoden vi bis x) ergänzt. Hierfür wurden durch eine zufällige Mutation der seriellen Eingaben die Stimuli[2] variiert. Eine zufällige Variation der Mutationen verhindert zum einen das Finden eines lokalen Minimums

[2] In der vorliegenden Applikation werden zu jedem Zeitschritt die Sensorwerte (Analogwert des Lenkwinkels, der Drehrate zur Bestimmung der Neigung des Segways und der Beschleunigung) sowie Parameter Vernetzungseinheit eingelesen und die neue Drehzahl der Motoren berechnet.

(siehe Kapitel 3.2), zum anderen führt dies jedoch zu einem nicht deterministischen Verhalten, wodurch die Schwachstellen nach einer unterschiedlicher Anzahl an Mutationen und Applikationsdurchläufen durch die Methoden erkannt werden. Um das nicht deterministische Verhalten auszugleichen, wurden für jede Applikation 20 Messungen durchgeführt und die minimale und maximale Anzahl an Applikationsdurchläufen ermittelt. Durch Ausführung der Gesamtsystemsimulation auf einem Desktop-Computer mit 1,2 GHz und 8 GB RAM wurden die in Tabelle 9.3 dargestellten Simulationslaufzeiten ermittelt.

Tabelle 9.3: Zusammenfassung der Erkannten Schwachstellen mit Methoden, Anzahl der Durchläufe und Laufzeiten

| Getestete Software | Probleme | Methode | Mutationen | Laufzeit |
|---|---|---|---|---|
| Lenkwinkelsensor | Integer-Overflow | i, ix, x | 2-20 | 0,3-3 Sek. |
| Lenkwinkelsensor | Nulldivision | i | 12-20 | 2-3 Sek. |
| Motorsteuerung | Integer-Overflow | i, x | 50-100 | 5-10 Sek. |
| Vernetzungseinheit | SSL-Heartbleed | iv, vii | 2-10 | 0,5-2 Sek.[3] |
| Vernetzungseinheit | Input Validation | iii, x | 20-50 | 1-3 Min.[4] |
| Segway Regler | Codeabdeckung | v | 100 | 75 Min.[5] |

Eine Aussage mittels Analysemöglichkeiten der Codeabdeckung (siehe Kapitel 6.4, Methode v) ist erst nach Abschluss aller Tests möglich, da erst anschließend die Häufigkeit des Durchlaufens der Codezeilen (Pfade in der Software) fest steht.

Im dargestellten Fallbeispiel wurde die Testmethodik genutzt, um diese auf eine unbekannte Softwareimplementierung zu übertragen. Hierbei zeigt sich, dass die verschiedenen Testmethoden Schwachstellen in der Software identifizieren können. Jedoch kann durch die zufällige Mutation nicht garantiert werden, dass alle Fehler (vollständige Testabdeckung) gefunden werden. Für eine solche Garantie ist eine vollständige Testabdeckung mit allen vorkom-

[3] Payload und Padding, zur Überprüfung, ob die Verbindung noch besteht, wurden im verwendeten Code mit einer Frequenz von 5 Hz gesendet.
[4] Im Testzyklus wurde der Notstopp nur einmal pro Durchlauf getätigt
[5] Nach 100 Mutationen wurden die evolutionären Tests abgebrochen.

menden Parametermutationen nötig, um den gesamten Suchraum der Tests darzustellen.

## 9.3 Einsatzmöglichkeiten der Testmethodik

Neben der bereits gezeigten Einsatzmöglichkeit der Simulation mit virtuellem Steuergerät zwischen SiL und HiL (siehe Kapitel 7.1), kann die Testmethodik auch nach Bekanntwerden einer Schwachstelle genutzt werden.

Wird beispielsweise ein Angriff oder Fehlverhalten erkannt und protokolliert, so kann der Hersteller den Fehler reproduzieren. Durch die zusätzlichen Beobachtungspunkte des virtuellen Steuergeräts wird das Fehlverhalten bereits bei Auftreten innerhalb des Prozessorsystems erkannt, wodurch Rückschlüsse auf die Ursachen gezogen werden können. Damit kann die Testmethodik ebenfalls zur Fehlersuche bei bekannten Schwachstellen eingesetzt werden.

Als weiteres Einsatzgebiet kann die Testmethodik als durchgängiger Test über den gesamten Produktlebenszyklus genutzt werden. Durch Ausführen der Simulation können neu entdeckte Schwachstellen und deren Auswirkung auf das System getestet werden, ohne den Steuergeräteverbund in einem HiL-Prüfstand aufbauen zu müssen.

Obwohl die Testmethodik für den Test der Datensicherheit entwickelt wurde, können durch Anpassungen der Überwachungsfunktionen und Teststimuli auch funktionale Tests durchgeführt werden. Dies ermöglicht beim funktionalen Testen ebenfalls die Möglichkeit die internen Beobachtungspunkte zu analysieren. Die Überwachungsfunktionen fokussieren sich auf Speicherüberläufe und Instruktionen, die zu Schwachstellen in der Applikation führen können. Für das funktionale Testen sind diese so zu erweitern, dass die Überwachungsfunktion die erwarteten Ergebnisse mit den simulierten Ergebnissen vergleicht.

## 9.4 Grenzen der Testmethodik

Für die Simulation der Peripherie, Speicher, Ein- und Ausgaben ist das spezifizierte Verhalten hinreichend, solange nur Stimuli eingelesen und die Bausteine

nicht detailliert betrachtet werden. Daher ist die Beschreibung der Schnittstellen in Form einer Spezifikation ausreichend, um diese zu modellieren. Soll mehr als das Verhalten der Schnittstellen simuliert werden, so müssen die Verhaltensmodelle der weiteren Komponenten vorhanden sein oder erstellt werden. Hierdurch werden während der Entwicklung zusätzliche Kosten entstehen. Die Genauigkeit der Komponentenmodelle hat zudem einen Einfluss darauf, was überwacht werden kann und wie realitätsnah die Gesamtsystemsimulation ist. Um die Genauigkeit der Analysen zu erhöhen, müssen die Modelle der verschiedenen Bausteine (Peripherie, Speicher, Ein- und Ausgaben) hinreichend genau abgebildet werden. Eine Erhöhung der Genauigkeit erhöht jedoch die Kosten beim Erstellen der Modelle, wodurch die Wirtschaftlichkeit sinken kann.

Stehen die Modelle zum Test nicht zur Verfügung, kann die Testmethodik nicht eingesetzt werden. Ebenso können nur aufwendig modellierte Modelle die gesamte Realität in der Simulation abbilden, was durch die zusätzlichen Kosten zum Erstellen der Modelle einer Gesamtsystemsimulation unwirtschaftlich wird. Für die vorgestellte Testmethodik sind hinreichend genaue Modelle, die die Beobachtungspunkte unter Kapitel 6.3.4 abbilden können, ausreichend. Daher sind die Tests mit der vorgestellten Methode, auf einem virtuellen Steuergerät, nur als Ergänzung, nicht aber als Ersatz, zu den bestehenden Tests aus Kapitel 2.6.5 zu sehen.

Beim Testen steht der Quellcode der Software nicht immer zur Verfügung (siehe Kapitel 7.2). Betrachtet man nun beim Testen ausschließlich die Schnittstellen und die Spezifikation, so bezeichnet man das System als Black-Box. Für die Testmethodik werden jedoch die internen Zustände aus Prozessor, Speicher und Peripherie benötigt. Daher wird im folgenden der Einfluss einer Black-Box auf die Testmethodik betrachtet.

Für die Ansteuerung der Software werden bei Black-Box-Tests durch die Rückgewinnung des Codes aus dem Maschinencode keine zusätzlichen Informationen benötigt (siehe Kapitel 6.3.2). Im Fall der Black-Box-Tests ist jedoch nur der Maschinencode vorhanden. Da die Debug-Informationen fehlen, ist es nicht möglich auf einzelne Variablen zurück zu schließen, wodurch nur Speicherbereiche beobachtet werden. Buffer Overflows sind daher nur für den gesamten Speicher möglich, nicht jedoch für einzelne Variablen.

Die Kategorie der numerischen Fehler (siehe Kapitel 4.1) werden bei Black-Box-Tests erkannt, solange der Befehlssatz des Prozessors bekannt ist. Für die Ausführung der Applikation auf einem virtuellen Prozessor, muss der Befehlssatz jedoch sowieso bekannt sein, da sonst eine instruktionsakkurate Simulation nicht möglich ist. Daher können numerische Fehler (wie Nulldivisionen oder Formatierungsfehler) auch bei Black-Box-Tests erkannt werden.

Um eine Analyse der offenen Schnittstellen durchführen zu können, muss ebenfalls der Quellcode vorhanden sein. Basierend auf einer Black-Box können zwar alle Schnittstellen des Steuergeräts erkannt und zur Überwachung ergänzt werden, eine Einschränkung auf tatsächlich verwendete Schnittstellen ist jedoch nicht möglich.

Bei der Ausführung der Applikation werden alle ausgeführten Speicherbereiche protokolliert, um eine Aussage über die Code-Abdeckung zu treffen. Da bei einer Black-Box keine Rückschlüsse auf die Funktionen der Applikation möglich sind, ist die Analyse der Code-Abdeckung nicht in der Lage die Anzahl der Ausführungen auf einzelne Funktionen zu abzubilden. Ein Rückschluss über die Häufigkeit des Zugriffs auf Speicherzeilen und Instruktionen ist dennoch möglich, da dies während der Simulation des virtuellen Prozessors protokolliert wird.

Weitere Grenzen der Testmethodik entstehen, da die Software auf einem virtuellen Prozessor ausgeführt wird. Analog zum Test auf einem realen Steuergerät können Softwaremodule nicht einzeln getestet werden. Für die Ausführung der Applikation auf dem virtuellen und realen Steuergerät sind zusätzliche Module und Funktionen nötig, die beim Test der Gesamtsoftware vorhanden sind. Hierzu zählt die Software zum Starten und Steuern des Prozessors, wie beispielsweise die Startprozedur mit Initialisierung der Prozessorperipherie, Frequenzgeneratoren, Watchdog und Interrupts.

## 9.5 Übertragbarkeit der Methodik auf andere Domänen

In Kapitel 4.2 wurden die Unterschiede verschiedener Domänen betrachtet und dabei auf die Automobildomäne fokussiert. Die vorgestellte Testmethodik lässt sich jedoch auch auf weitere Domänen übertragen. Da Sicherheitslücken

in den weiteren vorgestellten Domänen ebenfalls auftreten können, soll hier betrachtet werden, wodurch sich die Testmethodik abgrenzt.

Im Konsumbereich ist, durch die Möglichkeit der tägliche Updates, die Softwarelebensdauer sehr kurz (siehe Kapitel 4.2). Sicherheitslücken werden durch ein Software-Update behoben, das jederzeit auf die Geräte verteilt werden kann. Obwohl die Testmethodik grundsätzlich auf den Konsumbereich übertragbar ist, wurde die Methodik für virtuelles Testen mit eingebetteten Systemen entworfen und muss daher für eine Übertragbarkeit angepasst werden. Zudem wird die Software im Konsumbereich nicht auf spezielle Prozessoren und Steuergeräte angepasst, sondern flexibel für den Einsatz mit einem Betriebssystem geschrieben. Daher wird diese Testmethodik im Konsumbereich vermutlich keine Anwendung finden.

In den eingebetteten Systemen aus Industrie 4.0, Avionik und Medizintechnik finden die täglichen Updates nicht statt. Durch eine zusätzlich längere Lebensdauer als im Konsumbereich müssen Sicherheitslücken während dem Test verstärkt betrachtet werden. Daher sind diese Bereiche der Automobildomäne gleichzusetzen. Anpassungen müssen jedoch bei der Auswahl der Schwachstellen gemacht werden (siehe Kapitel 4.3). So müssen bei der Anbindung einer Datenbank (z. B. Industrie 4.0) zusätzlich die Angriffe über diese Schnittstelle (Cross-Site Scripting, SQL Incection, Credential Management, etc) berücksichtigt werden.

Obwohl die vorgestellte Methode virtuelle Steuergeräte des Automotive verwendet, können diese für andere Domänen angepasst werden. Voraussetzung für eine Simulation der gesamten Plattform ist, dass die Co-Simulation für Stimuli und Bewertung der Ausgaben auf die jeweilige Domäne angepasst wird. Zudem ist die Methodik auf dem Testprozess des V-Modells aufgebaut, wodurch Änderungen bei der Anpassung an die Testprozesse der verschiedenen Domänen nötig sein können.

## 9.6 Zusammenfassung

Die Bewertung der Testmethodik mittels Angriffen der Automobilindustrie zeigt, dass nicht alle bekannten Angriffe verhindert werden könnten. Während sich die Testmethodik zur Erkennung von Buffer Overflows (siehe Kapi-

tel 2.4.5) und daraus resultierender Code Injection (siehe Kapitel 2.4.6) eignet, können Schwachstellen, die durch systematische Fehler (siehe Kapitel 4.1) entstehen, nicht erkannt werden. Hierfür sind weitere Testmethoden nötig. Angriffe über falsche Datenformatierung oder fehlende Überprüfung der Eingangsdaten (siehe Angriff 6) können durch die Testmethode entdeckt werden, solange entsprechende Testfälle existieren. Diese Art der Fehler kann durch entsprechende Testfälle jedoch ebenfalls mit Software-in-the-Loop (SiL) und Hardware-in-the-Loop (HiL) gefunden werden.

Steht beim Testen der Quellcode nicht zur Verfügung, so kann durch Rekonstruktion der Software aus dem Maschinencode, die Testmethodik zwar eingesetzt werden, jedoch entstehen hierbei Grenzen. Zum einen können nur Speicherbereiche und keine einzelnen Variablen überwacht werden, zum anderen sind keine Rückschlüsse auf einzelne Funktionen möglich. Eine Analyse der Speicherzugriffe, Rechenoperationen und der Code-Abdeckung ist jedoch durch Betrachtung der ausgeführten Instruktionen im virtuellen Prozessor weiterhin möglich.

Eine Betrachtung der Übertragbarkeit zeigt, dass die Testmethodik neben der Automobilindustrie zudem in den eingebetteten Systemen der Industrie 4.0, Avionik und Medizintechnik eingesetzt werden kann. Eine Anpassung an die entsprechenden Testprozesse muss jedoch vorgenommen werden.

# 10 Zusammenfassung und Ausblick

Der aktuelle Trend des automatisierten Fahrens sorgt für eine zunehmende Zahl an Steuergeräten und Funktionen im Fahrzeug. Zusätzlich steigt die Vernetzung der Fahrzeugfunktionen und der Fahrzeuge untereinander, wodurch Fahrzeuge nicht mehr als Insellösungen betrachtet werden können, sondern offen für Angriffe über die Fahrzeugschnittstellen sind. Beachtet man zudem die steigende Zahl der IT-Manipulationen, die in den meisten Fällen durch bekannte Schwachstellen geschehen, muss die Datensicherheit bereits während der Entwicklung durch entsprechende Tests sichergestellt werden.

Eine Klassifizierung der bekannten Angriffe auf IT-Systeme und eingebettete Systeme ergibt, dass Angriffe in die drei Klassen Speicherfehler, numerische Fehler sowie systematische, funktionale und Anwendungsfehler unterteilt werden können. Speicherfehler beinhalten dabei das unerlaubte Überschreiben von Speicherbereichen, wodurch Variablen verändert werden können oder sogar die Ausführung von schadhaftem Code ermöglicht wird. Numerische Fehler sind Probleme, die während dem Ausführen der Instruktionen auftreten und durch falsche oder falsch formatierte Eingaben hervorgerufen werden. Hierzu zählt das Überprüfen der Datentypen und Wertebereiche. Das Rechnen mit solchen Daten kann zu numerischen Fehlern oder sogar Abstürzen führen (z. B. Nulldivision oder Vorzeichenfehler). Im Bereich der systematischen, funktionalen und Anwendungsfehler finden sich Probleme durch kryptografische Verfahren, Rechteverwaltung und Sicherheitslücken durch Weitergabe von Informationen.

Im Stand der Wissenschaft und Technik wird die Datensicherheit für Speicherfehler und numerische Fehler durch statische Analysen des Quellcodes und dynamische Tests (Fuzz-Test und Penetrationstest) sichergestellt. Besonders die dynamischen Verfahren können dabei Fehler erst feststellen, wenn diese an den Schnittstellen des Steuergeräts sichtbar sind.

Um Schwachstellen während den Tests zu identifizieren, bevor diese an den Schnittstellen sichtbar sind, wurde eine Testmethodik vorgestellt, die mit Hilfe

eines virtuellen Steuergeräts zusätzliche Beobachtungspunkte im Speicher, in der Peripherie und im Prozessor ergänzt. Fehlerhaftes Verhalten kann dadurch bereits bei Auftreten im entsprechenden Artefakt identifiziert werden. Eine zusätzliche Überwachungsfunktion greift auf die Beobachtungspunkte zu und steuert die Simulation, um während dem Test Analysen des Speichers, der Instruktionen oder der Peripherie durchzuführen.

Die Testmethodik beruht dabei auf den drei Bereichen Simulation des Prozessors und der Applikation, Simulation der Peripherie und des Umfelds und einer Überwachung und Steuerung. Für die Simulation der Applikation auf einem virtuellen Prozessor wurde eine Gesamtsystemsimulation gewählt, damit die Applikation auf das reale Steuergerät ohne Anpassung übertragen werden kann. Durch eine Anbindung von Co-Simulatoren können Peripherie und Umfeld des Tests auf verschiedenen Abstraktionsebenen simuliert werden. Das heißt, eine einfache Simulation der Peripherie kann anhand des Verhaltens in der Spezifikation erstellt oder für komplexere Simulationen spezialisierte Simulationswerkzeuge über Co-Simulationsschnittstellen angebunden werden.

In der Testmethodik werden zum einen die Eingabewerte der verwendeten Testfälle aus Software-in-the-Loop (SiL) und Hardware-in-the-Loop (HiL) verwendet, zum anderen werden durch evolutionäre Algorithmen neue Eingabewerte generiert und somit neue Testfälle erzeugt. Dadurch werden zur Laufzeit Instruktionen, Register und Eingaben oder der genutzte Speicher analysiert, um die Eingabe so zu variieren, dass eine Maximierung des Variablenwachstums, des Speicherzugriffs oder eine Minimierung des Abstands zwischen Heap und Stack entsteht.

Die Überwachung und Steuerung erzeugt für die Speicherbereiche der Variablen, Rücksprungadressen, Heap und Stack Beobachtungspunkte und analysiert das Verhalten während der Laufzeit. Ergeben sich während der Laufzeit Überläufe des Speichers oder ein schreibender Zugriff auf unerlaubte Speicherbereiche, so wird dies protokolliert und dem Testingenieur mitgeteilt. Anhand einer beispielhaften ACC Applikation wurde die Skalierung der Testmethodik ermittelt. Dabei lässt sich das verteilte System mit bis zu 62 Steuergeräten simulieren, bevor die Applikationszeit schneller als die Simulationszeit wird (siehe Kapitel 8.3). Durch Hinzufügen der Überwachungsfunktion erhöht sich die Simulationszeit jedoch in Abhängigkeit der Anzahl der Überwachungsfunktionen.

Anhand des Befehlssatzes wird die Laufzeit jeder Instruktion berechnet und durch kumulieren eine Systemzeit bestimmt. Anhand dieser Systemzeit kann das Echtzeitverhalten eines Steuergeräts in der Simulation überprüft und Co-Simulatoren synchronisiert werden. Durch die Simulation des virtuellen Steuergeräts auf einem Computer kann die Ausführung zudem beschleunigt werden, ohne das Verhalten zu beeinflussen.

Zur Bewertung der Methodik dienen die Angriffe auf Fahrzeuge der vergangenen Jahre. Durch den Einsatz des virtuellen Steuergeräts werden Speicherfehler durch die Überwachung interner Zustände während den Test identifiziert, bevor diese an die Schnittstellen propagiert sind und nach außen sichtbar werden. Die Klasse der numerischen Fehler wird im virtuellen Steuergerät zwar erkannt, dies kann aber ebenfalls bereits über den Software-in-the-Loop und Hardware-in-the-Loop Tests erkannt werden. Systematische Fehler (z. B. falsche Verschlüsselung oder Zugriffsrechte) werden durch die Testmethodik nicht erkannt, da diese funktional getestet werden müssen. Dennoch kann die Testmethode durch zusätzliche Beobachtungspunkte im Steuergerät und dem Prozessor helfen diese Fehler zu identifizieren.

Die vorgestellte Testmethodik unterstützt den Entwickler und Testingenieur bereits in der Identifikation von Schwachstellen. Dennoch kann der Testprozess durch Ergänzungen effizienter gestaltet werden.

Durch erste Einträge zu Angriffen auf Fahrzeuge und eingebettete Systeme in den Datenbanken CWE und CVE können diese genutzt werden, um Testfälle zu generieren. Durch eine Anbindung der Datenbanken in den Testprozess können Schwachstellen automatisiert in den Datenbanken gesucht und entsprechende automatisierte Testfälle generiert werden.

Weiter ist zu untersuchen, wie neue Testfälle generiert werden können, um unbekannte Schwachstellen zu finden. Der Einsatz von evolutionäre Algorithmen oder auch künstlicher Intelligenz sind zwei Möglichkeiten, um Testfälle automatisiert generieren zu lassen. Beide Verfahren profitieren durch die Testmethodik, da im Gegensatz zum Test mittels realer Hardware zusätzliche Beobachtungspunkte entstanden sind.

# A Einteilung kritischer Funktionen

Instruktionen dienen der Ausführung eines Programms und werden im Maschinencode oder Assemblercode dargestellt. Für die Überwachung der Datensicherheit werden diese in kritische und unkritische Instruktionen unterteilt (siehe Tabelle A.1).

Die Unterteilung der Instruktionen basiert darauf, ob

1. Daten in den Speicher geschrieben werden (schreibende Instruktionen wurden als kritisch klassifiziert),
2. das Ergebnis der betrachteten Recheninstruktion Overflows verursachen kann (Instruktionen die einen Integeroverflow[1] verursachen können wurden als kritisch klassifiziert),
3. eine Division durchgeführt wird (bei einer Division muss der Divisor größer als Null sein) und
4. Sprungbefehle auf einen beliebigen Speicher zeigen können (Sprünge in beliebigen Speicher können den Inhalt einer Variable als Instruktion erkennen und diese ausführen, daher wurden diese Sprünge als als kritische Instruktionen klassifiziert).

[1] Ein Integeroverflow tritt auf, wenn eine Instruktion versucht, einen Zahlenwert zu erzeugen, der außerhalb des Bereichs der Variable liegt, entweder größer als das Maximum oder kleiner als der minimale darstellbare Wert.

Tabelle A.1: Auswahl und Bewertung von Maschinenbefehlen

| Instruktion | Beschreibung | unkritisch |
|---|---|---|
| AAx[2] | ASCII-Anpassung für Berechnung | ✓ |
| ADC | Addieren mit Übertrag | ✓ |
| ADD | Addieren (ohne Carry-Flag) | ✗ |
| AND | Und-Verknüpfung | ✓ |
| BL | Sprung zur angegebenen Adresse | ✗ |
| CALL | Aufruf der im Operanden angegebenen Unterroutine | ✗ |
| CBW | Byte vorzeichengerecht in Word umwandeln | ✓ |
| CLx | Status Flag löschen | ✓ |
| CMPx | Vergleichen von Operanden | ✓ |
| CWD | Wort vorzeichengerecht in Doppelwort umwandeln | ✓ |
| DAx | Dezimale Anpassung für Berechnung | ✓ |
| DEC | Dekrementieren | ✗ |
| DIV | Vorzeichenlose Division | ✗ |
| ESC | Umschalten zu ext. Controller/HW-Beschleuniger | ✓ |
| HLT | Prozessor anhalten | ✓ |
| IDIV | Division mit Vorzeichen | ✗ |
| IMUL | Multiplikation mit Vorzeichen | ✗ |
| IN | IO-Port Einlesen in Register | ✓ |
| INC | Inkrementieren | ✗ |
| INT | Software-Interrupt (Systemaufruf) | ✓ |
| INT3 | Breakpoint Interrupt | ✓ |
| INTO | Unterbrechung bei Überlauf | ✗ |
| IRET | Rückkehr von Interrupt-Routine | ✗ |

Weiter auf der nächste Seite

[2] Ein "x"fasst ähnliche Instruktionen zur Übersichtlichkeit zusammen

Tabelle A.1: Auswahl und Bewertung von Maschinenbefehlen (Fortsetzung)

| Instruktion | Beschreibung | unkritisch |
|---|---|---|
| JMP | Unbedingter Sprung | ✗ |
| Jx | Sprungbefehl | ✗ |
| LAHF | Lade Flags in das AH-Register | ✓ |
| LDS | Lade Register und DS-Register aus dem Speicher | ✓ |
| LEA | Effektive (Offset-) Adresse laden | ✓ |
| LES | Lade Register und ES-Register aus dem Speicher | ✓ |
| LOCK | Bus sperren | ✓ |
| LODSx | Bytes/Words in Register laden | ✓ |
| LOOP | Zähler erniedrigen und Sprung, bis Zähler gleich 0 | ✗ |
| MOVx | Byte/Word von Speicher zu Speicher übertragen | ✗ |
| MUL | Multiplikation (ohne Vorzeichen) | ✗ |
| NEG | Negieren | ✓ |
| NOP | Keine Operation | ✓ |
| NOT | Nicht-Verknüpfung | ✓ |
| OR | Oder-Verknüpfung | ✓ |
| OUT | Ausgabe eines Registers | ✓ |
| POPx | Wort vom Stack holen | ✓ |
| PUSHx | Wort auf dem Stack ablegen | ✗ |
| RCL | Mit Carry-Flag links rotieren | ✓ |
| RCR | Mit Carry-Flag rechts rotieren | ✓ |
| REPx | Wiederholung des folgenden Befehls | ✓ |
| RETx | Rückkehr von Unterprogramm | ✗ |
| ROL | Links rotieren | ✓ |
| ROR | Rechts rotieren | ✓ |
| SAL | Arithmetisch nach Links schieben | ✓ |

Weiter auf der nächste Seite

Tabelle A.1: Auswahl und Bewertung von Maschinenbefehlen (Fortsetzung)

| Instruktion | Beschreibung | unkritisch |
|---|---|---|
| SAR | Arithmetisch nach Rechts schieben | ✓ |
| SBB | Subtrahieren mit Übertrag | ✓ |
| SCASx | Durchsuchen eines Strings nach Registerinhalt | ✓ |
| STx | Setzen des ALU-Flags | ✓ |
| STOSBx | Register in Speicher schreiben | ✗ |
| SUB | Subtrahieren (ohne Carry-Flag) | ✗ |
| TEST | Vergleich durch logisches UND | ✓ |
| WAIT | Warten auf Signal am Test-Pin | ✓ |
| XCHG | Vertauschen | ✓ |
| XOR | Exclusive ODER-Verknüpfung (XOR) | ✓ |

# B Darstellung der Überwachungskonfigurationstabelle

Für die Überwachung des virtuellen Steuergeräts (siehe Kapitel 6.5.2) wird eine Überwachungskonfiguration benötigt, die die Parameter der Beobachtungspunkte speichert.

Durch eine Codeanalyse (siehe Kapitel 6.5.2) werden die Parameter der Beobachtungspunkte aus der Applikation extrahiert und anschließend zur Unterbrechung der Simulation (siehe Kapitel 6.5.1) verwendet. Für eine exemplarischen Funktionsablauf aus Abbildung B.1 werden Variablen und Instruktionen extrahiert und in Tabelle B.1 festgehalten.

Diese Überwachungskonfiguration ist in Form einer Tabelle dargestellt (siehe Tabelle B.1) und beinhaltet:

**Typ** enthält eine Information über die Art des Beobachtungspunktes. Hierbei wird zwischen Variablen, Rücksprungadressen und Instruktionen unterschieden.

**Speicheradresse** ist die Adresse innerhalb des Speichers der Variablen oder Instruktion.

**Variablenlänge** beschreibt die Länge des belegten Speichers einer Variablen. Rücksprungadressen belegen bei einer 32 bit-Architektur 32 bit oder 4 Byte. Eine Änderung der Architektur verändert die Länge der Rücksprungadressen entsprechend.

**Heap/Stack** unterscheidet, ob der Speicherbereich innerhalb des Heaps oder Stacks liegt, eine lokale Variable oder Konstante ist.

**Befehl** gibt die Art der Instruktion an. Hierdurch lassen sich die Parameter ableiten, welche beobachtet werden müssen.

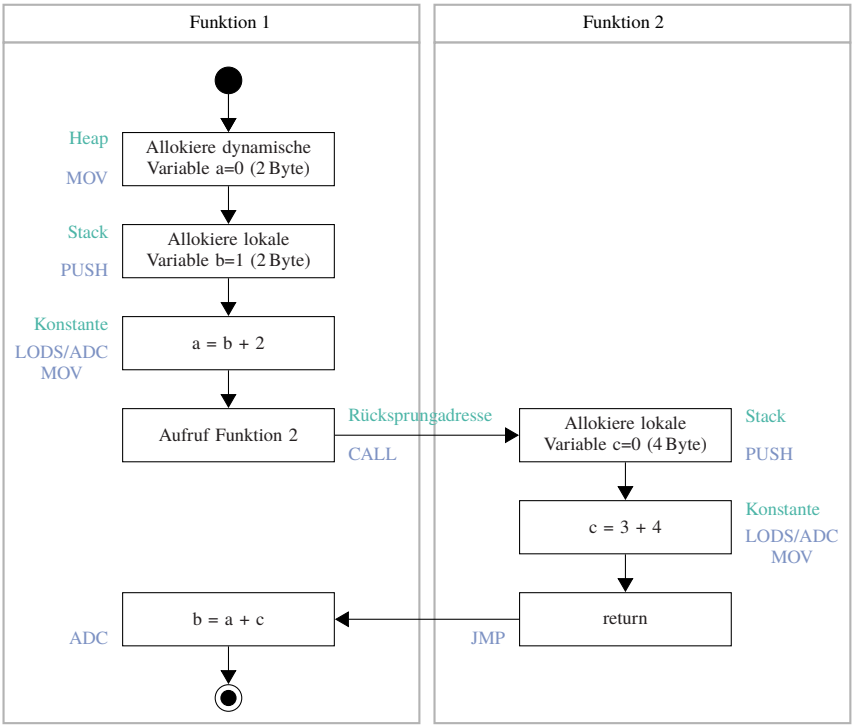



Abbildung B.1: Exemplarischer Programmablauf zur Erstellung der Beobachtungskonfiguration in Tabelle B.1

**Schreibend** unterscheidet, ob ein Befehl schreibend oder lesend auf den Speicher zugreift.

**Zugehörigkeit** beinhaltet die Funktion der Instruktion oder Variablen. Dadurch lassen sich Zugriffe auf einen Speicherbereich durch unberechtigte Funktionen erkennen (siehe Kapitel 6.4 iv).

**Beobachtet** gibt an, ob der Beobachtungspunkt innerhalb der Simulation ausgewertet wird.

Tabelle B.1: Exemplarische Beobachtungskonfiguration für Abbildung B.1

| Typ | Speicheradresse | Variablenlänge | Heap/Stack | Befehl | Schreibend | Zugehörigkeit | Beobachtet |
|---|---|---|---|---|---|---|---|
| Instruktion | `0x0000A1C0` | - | - | MOV | ja | Funktion 1 | ✓ |
| Variable | `0xFE14A5C4` | 2 Byte | Heap | - | - | Funktion 1 | ✓ |
| Instruktion | `0x0000A1C4` | - | - | PUSH | ja | Funktion 1 | ✓ |
| Variable | `0xFE00A500` | 2 Byte | Stack | - | - | Funktion 1 | ✓ |
| Instruktion | `0x0000A1C8` | - | - | LODS | nein | Funktion 1 | ✗ |
| Variable | - | 1 Byte | Konst. | - | - | Funktion 1 | ✗ |
| Instruktion | `0x0000A1CC` | - | - | ADC | nein | Funktion 1 | ✗ |
| Instruktion | `0x0000A1D0` | - | - | MOV | ja | Funktion 1 | ✓ |
| Instruktion | `0x0000A1D4` | - | - | CALL | ja | Funktion 1 | ✓ |
| Rücksprung | `0xFE00A502` | 4 Byte | Stack | - | - | Funktion 2 | ✓ |
| Instruktion | `0x0000A1D8` | - | - | PUSH | ja | Funktion 2 | ✓ |
| Instruktion | `0x0000A1DC` | - | - | LODS | nein | Funktion 2 | ✗ |
| Variable | - | 1 Byte | Konst | - | - | Funktion 2 | ✗ |
| Instruktion | `0x0000A1F0` | - | - | ADC | nein | Funktion 2 | ✗ |
| Instruktion | `0x0000A1F4` | - | - | MOV | ja | Funktion 2 | ✓ |
| Instruktion | `0x0000A1F8` | - | - | JMP | nein | Funktion 2 | ✓ |
| Instruktion | `0x0000A1FC` | - | - | LODS | nein | Funktion 1 | ✗ |
| Instruktion | `0x0000A200` | - | - | ADC | nein | Funktion 1 | ✗ |
| Instruktion | `0x0000A204` | - | - | MOV | ja | Funktion 1 | ✓ |

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Abkürzungsverzeichnis

| | |
|---|---|
| **ACC** | Abstandsregeltempomat |
| **ACsIL** | Automotive Cybersecurity Integrity Level |
| **ADAS** | engl. Advanced Driver Assistance Systems |
| **ADC** | Analog-Digital-Wandler |
| **ALU** | Arithmetic Logic Unit |
| **ASIL** | Automotive Safety Integrity Level |
| **ATA** | Attack Tree Analysis |
| **AUTOSAR** | Automotive Open Systems Architecture |
| **BL** | Branch with Link |
| **Car2X** | Fahrzeug zu Fahrzeug und Fahrzeug zu Infrastruktur Kommunikation |
| **CERT** | Computer Emergency Response Team |
| **CFGR** | Control Flow Graph Recovery |
| **CID** | Central Information Display |
| **CISC** | komplexen Befehlssatz (engl. Complex Instruction Set Computer) |
| **CPU** | Central Processing Unit |
| **CVE** | Common Vulnerabilities and Exposures Datenbank |
| **CWE** | Common Weakness Enumeration Datenbank |
| **DLL** | Dynamic-Link Library |

| | |
|---|---|
| **DoS** | Denial of Service |
| **DuT** | Device under Test |
| **ECU** | Electronic Control Unit |
| **E/E** | Elektrik-Elektronik |
| **FTA** | Fault Tree Analysis |
| **FMI** | Functional Mock-up Interface |
| **GUI** | Graphical User Interface |
| **GNSS** | globalen Navigationssatellitensystems (engl. Global Navigation Satellite System |
| **GPR** | General Purpose Register |
| **GSM** | Global System for Mobile Communications |
| **HAL** | Hardware Abstraction Layer |
| **HiL** | Hardware-in-the-Loop |
| **IDE** | Integrated Development Environment |
| **IDS** | Intrusion Detection System |
| **IoT** | Internet of Things |
| **IP** | Intellectual Property |
| **ISS** | Instruction Set Simulation |
| **ITS** | Intelligent Transportation System |
| **JTAG** | Joint Test Action Group |
| **LET** | Logical Excecution Time |
| **MaaS** | Mobility as a Service |
| **MiL** | Model-in-the-Loop |
| **MIPS** | Millionen Instruktionen pro Sekunde |

| | |
|---|---|
| **MISRA** | Motor Industry Software Reliability Association |
| **NIST** | National Institute of Standards and Technology |
| **OBD** | Onboard Diagnose |
| **OEM** | Original Equipment Manufacturer |
| **OVP** | Open Virtual Platform |
| **PiL** | Processor-in-the-Loop |
| **PWM** | Pulsweitenmodulation |
| **RCP** | Rapid Control Prototyping |
| **RISC** | reduzierter Befehlssatz (engl. Reduced Instruction Set Computer) |
| **RTE** | Runtime Environment |
| **SAE** | Society of Automotive Engineers |
| **SiL** | Software-in-the-Loop |
| **SOTA** | Software-Over-The-Air |
| **SSL** | Secure Sockets Layer |
| **StVO** | Straßenverkehrsordnung |
| **TARA** | Threat and Risk Analysis |
| **VDA** | Verband der Automobilindustrie |

# Literaturverzeichnis


[1] Abel, Dirk ; Bollig, Alexander: *Rapid Control Prototyping: Methoden und Anwendungen ; mit 16 Tabellen*. Berlin, Heidelberg : Springer-Verlag Berlin Heidelberg, 2006. http://dx.doi.org/10.1007/3-540-29525-9. http://dx.doi.org/10.1007/3-540-29525-9. – ISBN 978–3–540–29524–2

[2] ARM ; Arm Limited (Hrsg.): *ARM® Cortex®-M4 Processor*. 2015 (Technical Reference Manual)

[3] Balzert, Helmut: *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb*. 3. Aufl. Heidelberg : Spektrum Akademischer Verlag, 2011. – ISBN 978–3–827–42246–0

[4] Bayer, Stephanie ; Enderle, Thomas ; Oka, Dennis-Kengo ; Wolf, Marko: Automotive Security Testing—The Digital Crash Test. Version: 2016. http://dx.doi.org/10.1007/978-3-319-19818-7{_}2. In: Langheim, Jochen (Hrsg.): *Energy Consumption and Autonomous Driving*. Cham : Springer International Publishing, 2016 (Lecture Notes in Mobility). – DOI 10.1007/978–3–319–19818–7_2. – ISBN 978–3–319–19817–0, S. 13–22

[5] Bland, Mike: Finding More Than One Worm in the Apple. In: *Queue* 12 (2014), Nr. 5, S. 10–21. http://dx.doi.org/10.1145/2620660.2620662. – DOI 10.1145/2620660.2620662. – ISSN 15427730

[6] Bormann, René ; Fink, Philipp ; Holzapfel, Helmut ; Rammler, Stephan ; Sauter-Servaes, Thomas ; Tiemann, Heinrich ; Waschke, Thomas ; Weirauch, Boris: Die Zukunft der deutschen Automobilindustrie: Tranformation by disaster oder by design? In: *WISO Diskurs* 2018 (2018), Nr. 03/2018. https://library.fes.de/pdf-files/wiso/14086-20180205.pdf

[7] Brader, Larry ; Hilliker, Howie ; Wills, Alan C.: *Testing for continuous delivery with Visual Studio 2012*. S.l. : Microsoft, 2012 (Patterns & practices). – ISBN 1–621–14018–0

[8] Bresson, Guillaume ; Alsayed, Zayed ; Yu, Li ; Glaser, Sebastien: Simultaneous Localization and Mapping: A Survey of Current Trends in Autonomous Driving. In: *IEEE Transactions on Intelligent Vehicles* 2 (2017), Nr. 3, S. 194–220. http://dx.doi.org/10.1109/TIV.2017.2749181. – DOI 10.1109/TIV.2017.2749181. – ISSN 2379–8904

[9] Brown, Alan S.: Hiding in Plain Sight: Google's autonomous car applies lessons learned from driverless races. In: *Mechanical Engineering* 133 (2011), Nr. 02, S. 31. http://dx.doi.org/10.1115/1.2011-FEB-3. – DOI 10.1115/1.2011–FEB–3. – ISSN 0025–6501

[10] Brumley, David ; Poosankam, Pongsin ; Song, Dawn ; Zheng, Jiang: Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In: *IEEE Symposium on Security and Privacy, 2008*. Piscataway, NJ : IEEE, 2008. – ISBN 978–0–7695–3168–7, S. 143–157

[11] Brusa, Eugenio ; Calà, Ambra ; Ferretto, Davide: *Studies in Systems, Decision and Control*. Bd. 134: *Systems engineering and its application to industrial product development*. Cham : Springer International Publishing, 2018. http://dx.doi.org/10.1007/978-3-319-71837-8. http://dx.doi.org/10.1007/978-3-319-71837-8. – ISBN 978–3–319–71837–8

[12] Bühler, Oliver ; Wegener, Joachim: Automatic Testing of an Autonomous Parking System Using Evolutionary Computation. In: *SAE Technical Paper Series*, SAE International400 Commonwealth Drive, Warrendale, PA, United States, 2004 (SAE Technical Paper Series). – ISBN 0148–7191

[13] Bühler, Oliver ; Wegener, Joachim: Evolutionary functional testing of a vehicle brake assistant system. In: *6th Metaheuristics International Conference, Vienna, Austria*, 2005

[14] Bundesakademie für Sicherheitspolitik ; Bundesakademie für Sicherheitspolitik (Hrsg.): *Cyber-Security - Eine Frage der Begriffe*. https://www.baks.bund.de/sites/baks010/files/baks_arbeitspapier_2_2014.pdf. Version: 2, 2014 (Arbeitspapiere Sicherheitspolitik)

[15] Bundesregierung: *Entwurf eines Gesetzes zur Änderung der Artikel 8 und 39 des Übereinkommens vom 8. November 1968 über den Straßenverker*. 2016

[16] Cai, Zhiqiang ; Wan, Aohui ; Zhang Wenkai ; Gruffke, Michael ; Schweppe, Hendrik: 0-days & Mitigations: Roadways to Exploit and Secure Connected BMW Cars. In: *Black Hat Biefings*. Mandalay Bay, 2019

[17] Carvalho, Marco ; DeMott, Jared ; Ford, Richard ; Wheeler, David A.: Heartbleed 101. In: *IEEE Security & Privacy* 12 (2014), Nr. 4, S. 63–67. http://dx.doi.org/10.1109/MSP.2014.66. – DOI 10.1109/MSP.2014.66. – ISSN 1540–7993

[18] Charette, Robert N.: This Car Runs on Code. In: *IEEE Spectrum* (2009). https://spectrum.ieee.org/transportation/systems/this-car-runs-on-code

[19] Checkoway, Stephen ; McCoy, Damon ; Kantor, Brian ; Anderson, Danny ; Shacham, Hovav ; Savage, Stefan ; Koscher, Karl ; Czeskis, Alexei ; Roesner, Franziska ; Kohno, Tadayoshi: Comprehensive Experimental Analyses of Automotive Attack Surfaces. In: *USINEX Security Symposium*, 2011

[20] Deogun, Daniel ; Johnsson, Dan B. ; Sawano, Daniel: *Secure by Design*. Manning Publications Company, 2018. – ISBN 978–1–617–29435–8

[21] Di Federico, Alessandro ; Agosta, Giovanni: A jump-target identification method for multi-architecture static binary translation. In: *2016 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. Piscataway, NJ : IEEE, 2016. – ISBN 978–1–450–34482–1, S. 1–10

[22] Dori, Dov: *Model-Based Systems Engineering with OPM and SysML*. First edition. New York, NY : Springer New York, 2016. – ISBN 978–1–493–93295–5

[23] Dreyer, Boris ; Hochberger, Christian ; Lange, Alexander ; Wegener, Simon ; Weiss, Alexander: Continuous Non-Intrusive Hybrid WCET Estimation Using Waypoint Graphs. Version: 2016. http://dx.doi.org/10.4230/OASIcs.WCET.2016.4. In: Schoeberl,

Martin (Hrsg.): *16th International Workshop on Worst-Case Execution Time Analysis*. Saarbrücken/Wadern, Germany : Schloss Dagstuhl - Leibniz-Zentrum für Informatik GmbH Dagstuhl Publishing, 2016 (OASIcs). – DOI 10.4230/OASIcs.WCET.2016.4. – ISBN 978–3–959–77025–5, S. 4:1–4:11

[24] Dudenredaktion: *Duden: Die deutsche Rechtschreibung*. 2017. – ISBN 978–3–411–04017–9

[25] Europäisches Parlament: *Angleichung der Rechtsvorschriften der Mitgliedstaaten über aktive implantierbare medizinische Geräte und 93/42/EWG des Rates über Medizinprodukte sowie der Richtlinie 98/8/EG über das Inverkehrbringen von Biozid-Produkten*. http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=OJ:L:2007:247:0021:0055:de:PDF. Version: 2007

[26] Europäisches Parlament: *Anforderungen für die Typgenehmigung zur Einführung des auf dem 112-Notruf basierenden bordeigenen eCall-Systems in Fahrzeugen und zur Änderung der Richtlinie 2007/ 46/ EG*. http://eur-lex.europa.eu/legal-content/DE/TXT/PDF/?uri=CELEX:32015R0758&from=DE. Version: 2015

[27] Europäisches Parlament: *Verordnung zum Schutz natürlicher Personen bei der Verarbeitung personenbezogener Daten, zum freien Datenverkehr und zur Aufhebung der Richtlinie 95/46/EG (Datenschutz-Grundverordnung)*. https://eur-lex.europa.eu/legal-content/DE/TXT/PDF/?uri=CELEX:32016R0679&from=EN. Version: 2018

[28] European Telecommunications Standards Institute: *Telecommunications and Internet converged Services and Protocols for Advanced Networking (TISPAN); Methods and protocols; Part 1: Method and proforma for Threat, Risk, Vulnerability Analysis*. V4.2.1. 2007

[29] European Telecommunications Standards Institute: *Telecommunications and Internet converged Services and Protocols for Advanced Networking (TISPAN); Methods and protocols; Part 2: Protocol Framework Definition; Security Counter Measures*. V4.2.1. 2007

[30] Foster, James C. ; Osipov, Vitaly ; Bhalla, Nish: *Buffer Overflow Attacks: Detect, Exploit, Prevent*. 1. Aufl. Rockland : Syngress Publishing, Inc., 2005 https://repo.zenk-security.com/Magazine%20E-book/[Syngress]%20Buffer.Overflow.Attacks.-.Detect.Exploit.Prevent.pdf. – ISBN 1–932–26667–4

[31] Funke, Joseph ; Theodosis, Paul ; Hindiyeh, Rami ; Stanek, Ganymed ; Kritatakirana, Krisada ; Gerdes, Chris ; Langer, Dirk ; Hernandez, Marcial ; Muller-Bessler, Bernhard ; Huhnke, Burkhard: Up to the limits: Autonomous Audi TTS. In: *IEEE Intelligent Vehicles Symposium (IV), 2012*. Piscataway, NJ : IEEE, 2012. – ISBN 978–1–467–32118–1, S. 541–547

[32] Gasser, Tom M. (Hrsg.): *Berichte der Bundesanstalt für Strassenwesen. F, Fahrzeugtechnik*. Bd. Heft F 83: *Rechtsfolgen zunehmender Fahrzeugautomatisierung: Gemeinsamer Schlussbericht der Projektgruppe*. Bremerhaven : Wirtschaftsverlag NW, 2012. – ISBN 978–3–869–18189–9

[33] Ghosh, Anup K. ; Michael, Christoph ; Schatz, Michael: A Real-Time Intrusion Detection System Based on Learning Program Behavior. In: Debar, Hervé (Hrsg.): *Recent advances in intrusion detection*. Berlin : Springer, 2000 (Lecture notes in computer science). – ISBN 3–540–41085–6

[34] Grabowski, Tobias: Vernetzte Fahrzeuge: Neue Ermittlungsansätze im Strafverfahren? In: *Kriminalistik* 4/2018 (2018), S. 209

[35] Harer, Johann: *Anforderungen an Medizinprodukte: Praxisleitfaden für Hersteller und Zulieferer*. 2., überarbeitete Auflage. München : Hanser, 2014. – ISBN 978–3–446–44041–8

[36] Hawkins, Troy R. ; Gausen, Ola M. ; Strømman, Anders H.: Environmental impacts of hybrid and electric vehicles—a review. In: *The International Journal of Life Cycle Assessment* 17 (2012), Nr. 8, 997–1014. http://dx.doi.org/10.1007/s11367-012-0440-9. – DOI 10.1007/s11367–012–0440–9. – ISSN 1614–7502

[37] Heiser, Gernot ; Leslie, Ben: The OKL4 microvisor. In: Thekkath, Chandramohan A. (Hrsg.): *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*. New York, NY : ACM, 2010. – ISBN 978–1–450–30195–4, S. 19

[38] Hensher, David A.: Future bus transport contracts under a mobility as a service (MaaS) regime in the digital age: Are they likely to change? In: *Transportation Research Part A: Policy and Practice* 98 (2017), S. 86–96. http://dx.doi.org/10.1016/j.tra.2017.02.006. – DOI 10.1016/j.tra.2017.02.006. – ISSN 09658564

[39] Howard, Bill: Hack the diagnostics connector, steal yourself a BMW in 3 minutes. In: *ExtremeTech* (2012), 1–4. http://www.extremetech.com/extreme/132526-hack-the-diagnostics-connector-steal-yourself-a-bmw-in-3-minutes

[40] Howard, Michael ; Le Blanc, David: *Writing secure code: Practical strategies and proven techniques for building secure applications in a networked world*. 2. ed. Redmond, Wash. : Microsoft Press, 2003. – ISBN 978–0–735–61722–3

[41] Hutter, Alexander: *Informationstechnik im Maschinenwesen*. Bd. 25: *Eine Systematik zur Erstellung virtueller Steuergeräte für Hardware-in-the-Loop-Integrationstests*. München : Utz, 2006. – ISBN 978–3–831–60637–5

[42] International Electronical Commision: *Functional safety of electrical/electronic/programmable electronic safety-related systems*. 2010

[43] International Electronical Commision: *Functional safety - Safety instrumented systems for the process industry sector*. 2018

[44] International Organization for Standardization: *ISO 26262: Road Vehicles : Functinal Safety*. 2011. 2011

[45] International Organization for Standardization: *ISO/IEC 27001: Information security management*. 2013

[46] International Organization for Standardization: *Road Vehicles – Cybersecurity engineering*. 2019

[47] Kanoun, Karama (Hrsg.) ; Spainhower, Lisa (Hrsg.): *Dependability benchmarking for computer systems*. Hoboken, N.J and Piscataway, NJ : Wiley IEEE Computer Society and IEEE Computer Society, 2008. http://dx.doi.org/10.1002/9780470370506. http://dx.doi.org/10.1002/9780470370506. – ISBN 978–0–470–37083–4

[48] Keen Security Lab ; Tencent Technology (Shanghai) Co., Ltd (Hrsg.): *Experimental Security Assessment of BMW Cars: A Summary Report*. https://keenlab.tencent.com/en/Experimental_Security_Assessment_of_BMW_Cars_by_KeenLab.pdf

[49] Kemme, Nils: Effects of storage block layout and automated yard crane systems on the performance of seaport container terminals. In: *OR Spectrum* 34 (2012), Nr. 3, S. 563–591. http://dx.doi.org/10.

1007/s00291-011-0242-7. – DOI 10.1007/s00291–011–0242–7. – ISSN 0171–6468

[50] Khastgir, Siddartha ; Dhadyalla, Gunwant ; Birrell, Stewart ; Redmond, Sean ; Addinall, Ross ; Jennings, Paul: Test Scenario Generation for Driving Simulators Using Constrained Randomization Technique. In: *SAE World Congress Experience*, SAE International400 Commonwealth Drive, Warrendale, PA, United States, 2017 (SAE Technical Paper Series), S. 1–7

[51] Knechtel, Harry: Methoden zur Umsetzung von Datensicherheit und Datenschutz im vernetzten Steuergerät. In: *ATZelektronik* 10 (2015), Nr. 1, S. 26–31. http://dx.doi.org/10.1007/s35658-015-0500-6. – DOI 10.1007/s35658–015–0500–6. – ISSN 1862–1791

[52] Kocher, Paul ; Genkin, Daniel ; Gruss, Daniel ; Haas, Werner ; Hamburg, Mike ; Lipp, Moritz ; Mangard, Stefan ; Prescher, Thomas ; Schwarz, Michael ; Yarom, Yuval: Spectre Attacks: Exploiting Speculative Execution. In: *Computer Science*. ArXiv e-prints, 2018, S. 1–16

[53] Königs, Hans-Peter: *IT-Risiko-Management mit System: Von den Grundlagen bis zur Realisierung - Ein praxisorientierter Leitfaden*. 3., überarbeitete und erw. Aufl. Wiesbaden : Vieweg+Teubner Verlag / Springer Fachmedien Wiesbaden, Wiesbaden, 2009 (Edition <kes>). http://gbv.eblib.com/patron/FullRecord.aspx?p=749789. – ISBN 978–3–834–80359–7

[54] Koscher, Karl ; Czeskis, Alexei ; Roesner, Franziska ; Patel, Shwetak ; Kohno, Tadayoshi ; Checkoway, Stephen ; McCoy, Damon ; Kantor, Brian ; Anderson, Danny ; Shacham, Hovav ; Savage, Stefan: Experimental Security Analysis of a Modern Automobile. In: *IEEE Symposium on Security and Privacy*, 2010, S. 447–462

[55] Kreuzinger, Tobias ; Kengo Oka, Dennis ; Bayer, Stephanie ; Wolf, Marko: Effektive Security-Tests am HiL-System. In: *Hanser automotive* (2016), Nr. 11-12, 48–51. https://www.hanser-automotive.de/_storage/asset/2533709/storage/master/file/28791433/FA_Escrypt_stemp_02.pdf

[56] Lange, Walter ; Bogdan, Martin: *Entwurf und Synthese von Eingebetteten Systemen: Ein Lehrbuch*. München : Oldenbourg, 2013.

http://dx.doi.org/10.1524/9783486721096. http://dx.doi.org/10.1524/9783486721096. – ISBN 978–3–486–71840–9

[57] Lenhard, Thomas H.: *Datensicherheit: Technische und organisatorische Schutzmaßnahmen gegen Datenverlust und Computerkriminalität*. Wiesbaden : Springer Vieweg, 2017. http://dx.doi.org/10.1007/978-3-658-17983-0. http://dx.doi.org/10.1007/978-3-658-17983-0. – ISBN 978–3–658–17982–3

[58] Leupers, Rainer ; Temam, Olivier: *Processor and System-on-Chip Simulation*. Boston, MA : Springer Science+business Media LLC, 2010. http://dx.doi.org/10.1007/978-1-4419-6175-4. http://dx.doi.org/10.1007/978-1-4419-6175-4. – ISBN 978–1–441–96174–7

[59] Liggesmeyer, Peter: *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. 2. Aufl. s.l. : Spektrum Akademischer Verlag, 2009 http://gbv.eblib.com/patron/FullRecord.aspx?p=450796. – ISBN 3–827–42056–3

[60] Lipp, Moritz ; Schwarz, Michael ; Gruss, Daniel ; Prescher, Thomas ; Haas, Werner ; Mangard, Stefan ; Kocher, Paul ; Genkin, Daniel ; Yarom, Yuval ; Hamburg, Mike: Meltdown. Version: 2018. https://meltdownattack.com/meltdown.pdf. In: *Computer Science*. ArXiv e-prints, 2018, 1–16

[61] Lohmann, Katharina: *Hella Security testing framework: Integration of Security test methods on different test levels throughout the validation / verification process*. Stuttgart, 2016 (Vector Cyber Security Symposium)

[62] Long, Johnny: *Penetration tester's open source toolkit*. Rockland, MA : Syngress Pub, 2006. – ISBN 1–59749–021–0

[63] Magnusson, P. S. ; Christensson, M. ; Eskilson, J. ; Forsgren, D. ; Hallberg, G. ; Hogberg, J. ; Larsson, F. ; Moestedt, A. ; Werner, B.: Simics: A full system simulation platform. Version: 2002. http://dx.doi.org/10.1109/2.982916. In: *Computer* Bd. 35. IEEE Computer Society, 2002. – DOI 10.1109/2.982916, 50–58

[64] Masys, Anthony J. (Hrsg.): *Security by design: Innovative perspectives on complex problems*. Cham : Springer-Verlag, 2018 (Advanced sciences and technologies for security applications). – ISBN 978–3–319–78020–7

[65] Maurer, Markus (Hrsg.) ; Gerdes, J. C. (Hrsg.) ; Lenz, Barbara (Hrsg.) ; Winner, Hermann (Hrsg.): *Autonomes Fahren: Technische, rechtliche und gesellschaftliche Aspekte*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2015. http://dx.doi.org/10.1007/978-3-662-45854-9. http://dx.doi.org/10.1007/978-3-662-45854-9. – ISBN 978–3–662–45854–9

[66] McCammon, Marques: From Mars to Earth, Applying Three Decades of Experience to Thwart Vehicle Hacking. In: *Connected Vehicle Journal*. 2016, S. 13–15

[67] McCarthy, Charlie ; Harnet, Kevin ; Carter, Art: Characterization of Potential Securtiy Threads in Modern Automobiles: A Composite Modeling Approach. In: Gibbs, Craig (Hrsg.): *Automotive Cybersecurity*. New York : Novinka, 2016 (Transportation issues, policies and R&D). – ISBN 978–1–634–85987–5, S. 53–94

[68] McConnell, Steve: *Code Complete: [a practical handbook of software construction]*. 2nd ed. Sebastopol : O'Reilly Media Inc, 2009. – ISBN 978–0–735–61967–8

[69] Miller, Charlie ; Valasek, Chris: *A Survey of Remote Automotive Attack Surfaces*. 2014

[70] Miller, Charlie ; Valasek, Chris: Remote Exploitation of an Unaltered Passenger Vehicle. In: *White Paper* (2015)

[71] Motor Industry Software Reliability Association: *MISRA C:2012: Guidelines for the use of the C language in critical systems*. Nuneaton : Misra, 2013. – ISBN 978–1–906–40011–8

[72] Muench, Marius ; Stijohann, Jan ; Kargl, Frank ; Francillon, Aurelien ; Balzarotti, Davide: What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In: *Network and Distributed Systems Security (NDSS) Symposium 2018*, 2018. – ISBN 1–1891–56249–5

[73] Müller, Markus ; Hörmann, Klaus ; Dittmann, Lars ; Zimmer, Jörg: *Automotive SPICE in der Praxis: Interpretationshilfe für Anwender und Assessoren*. 2., aktualisierte und erweiterte Auflage. Heidelberg : dpunkt.verlag, 2016. – ISBN 978–3–960–88000–4

[74] Muller-Glaser, K. D. ; Frick, G. ; Sax, E. ; Kuhl, M.: Multiparadigm Modeling in Embedded Systems Design. In: *IEEE Transactions on Control Systems Technology* 12 (2004), Nr. 2, S. 279–292. http://dx.doi.org/10.1109/TCST.2004.824340. – DOI 10.1109/TCST.2004.824340. – ISSN 1063–6536

[75] Murray, Victor: Legal GNSS Spoofing and its Effects on Autonomous Vehicles. In: *Black Hat Biefings*. Mandalay Bay, 2019

[76] Myers, Glenford J. ; Badgett, Tom ; Thomas, Todd M. ; Sandler, Corey: *Business Data Processing*. Bd. v.28: *The art of software testing*. 2nd ed. Hoboken, N.J : John Wiley & Sons, 2004 http://site.ebrary.com/lib/academiccompletetitles/home.action. – ISBN 0–471–46912–2

[77] Navet, Nicolas: *Industrial information technology series*. Bd. 5: *Automotive embedded systems handbook*. Boca Raton : CRC, 2009. – ISBN 978–0–849–38027–3

[78] Nie, Sen ; Liu, Ling ; Du Yuefeng: Free-Fall: Hacking Tesla From Wireless to CAN Bus. In: *Black Hat Biefings*, 2017, 1–16

[79] Nocedal, Jorge ; Wright, Stephen J.: *Numerical Optimization*. Second Edition. New York, NY : Springer Science+Business Media LLC, 2006 (Springer Series in Operations Research and Financial Engineering). http://dx.doi.org/10.1007/978-0-387-40065-5. http://dx.doi.org/10.1007/978-0-387-40065-5. – ISBN 978–0–387–30303–1

[80] Organisation for Economic Co-operation and Development: Urban Mobility System Upgrade: How shared self-driving cars could change city traffic. In: *International Transportation Forum* 2015 (2015), Nr. OECD/ITF 2015. https://www.itf-oecd.org/sites/default/files/docs/15cpb_self-drivingcars.pdf

[81] Osborne, Charlie: Zero-day flaw lets hackers tamper with your car through BMW portal. (2016). https://www.zdnet.com/article/hackers-can-tamper-with-car-registration-through-bmw-connected-car-portal/

[82] Papp, Dorottya ; Ma, Zhendong ; Buttyan, Levente: Embedded systems security: Threats, vulnerabilities, and attack taxonomy. In: Ghorbani, Ali (Hrsg.): *2015 13th Annual Conference on Privacy, Security and Trust*

*(PST)*. Piscataway, NJ : IEEE, 2015. – ISBN 978–1–467–37828–4, S. 145–152

[83] Plassmann, Wilfried (Hrsg.): *Handbuch Elektrotechnik: Grundlagen und Anwendungen für Elektrotechniker ; mit 300 Tabellen*. 5., korrigierte Aufl. Wiesbaden : Springer Fachmedien, 2009 (Praxis). – ISBN 978–3–834–80470–9

[84] Pohlheim, Hartmut: *Evolutionäre Algorithmen: Verfahren, Operatoren und Hinweise für die Praxis*. Berlin : Springer, 2000 (VDI-Buch). – ISBN 978–3–540–66413–0

[85] Radzkewycz, Tim: Automotive Networks Can Benefit from Security. In: *Connected Vehicle Journal*. 2016, S. 7–11

[86] Roebuck, Kevin: *AUTOSAR - AUTomotive Open System ARchitecture: High-impact Strategies - What You Need to Know: Definitions, Adoptions, Impact, Benefits, Maturity, Vendors*. Dayboro : Emereo Pub, 2012. – ISBN 978–1–283–76532–9

[87] Rosenstatter, Thomas ; Olovsson, Tomas: Open Problems when Mapping Automotive Security Levels to System Requirements. In: *Proceedings of the 4th International Conference on Vehicle Technology and Intelligent Transport Systems*, SCITEPRESS - Science and Technology Publications, 2018. – ISBN 978–9–897–58293–6, S. 251–260

[88] SAE International: *Taxonomy and Definitions for Terms Related to On-Road Motor Vehicle Automated Driving Systems*. Jan2014. 2014

[89] SAE International: *Cybersecurity Guidebook for Cyber-Physical Vehicle Systems*. Jan2016. 2016

[90] Sander, Oliver: *Steinbuch series on advances in information technology*. Bd. 1: *Skalierbare adaptive System-on-Chip-Architekturen für Inter-Car und Intra-Car Kommunikationsgateways: Zugl.: Karlsruhe, KIT, Diss., 2010*. Print on demand. Hannover and Karlsruhe : Technische Informationsbibliothek u. Universitätsbibliothek and KIT Scientific Publishing, 2011 https://edocs.tib.eu/files/e01fn12/661529010.pdf. – ISBN 978–3–866–44601–4

[91] Sax, Eric (Hrsg.): *Automatisiertes Testen eingebetteter Systeme in der Automobilindustrie*. München : Hanser, 2008. http://dx.doi.org/10.3139/9783446419018. http://dx.doi.org/10.3139/9783446419018. – ISBN 978–3–446–41635–2

[92] Schäuffele, Jörg ; Zurawka, Thomas: *Automotive Software Engineering: Grundlagen, Prozesse, Methoden und Werkzeuge effizient einsetzen*. 6., überarb. u. akt. Aufl. 2016. 2016 (ATZ/MTZ-Fachbuch). http://dx.doi.org/10.1007/978-3-658-11815-0. – ISBN 978–3–658–11815–0

[93] Schmitt, Stephen: *Integrierte Simulation und Emulation eingebetteter Hardware/Software-Systeme*. 1st ed. Göttingen : Cuvillier Verlag, 2005 https://books.google.de/books?id=45rAPTgV7GQC. – ISBN 978–3–865–37511–7

[94] Scholz, Peter: *Softwareentwicklung eingebetteter Systeme: Grundlagen, Modellierung, Qualitätssicherung*. Berlin : Springer, 2005 (Xpert.press). http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10182931. – ISBN 978–3–540–23405–0

[95] Seacord, Robert C.: *The CERT C coding standard: 98 rules for developing safe, reliable, and secure systems*. 2nd ed. Upper Saddle River, NJ : Addison-Wesley, 2014 (SEI series in software engineering). – ISBN 978–0–321–98404–3

[96] Seiffert, Ulrich ; Rainer, Gotthard: *Virtuelle Produktentstehung für Fahrzeug und Antrieb im Kfz: Prozesse, Komponenten, Beispiele aus der Praxis*. Wiesbaden : Vieweg + Teubner Verlag / GWV Fachverlage GmbH Wiesbaden, 2008. http://dx.doi.org/10.1007/978-3-834-89479-3. http://dx.doi.org/10.1007/978-3-834-89479-3. – ISBN 978–3–834–80345–0

[97] Shahriar, Hossain ; Zulkernine, Mohammad: Automatic Testing of Program Security Vulnerabilities. In: Ahamed, Iqbal (Hrsg.) ; Bertino, Elisa (Hrsg.): *33rd Annual IEEE International Computer Software and Applications Conference, 2009*. Piscataway, NJ : IEEE, 2009. – ISBN 978–0–7695–3726–9, S. 550–555

[98] Siebenpfeiffer, Wolfgang: *Vernetztes Automobil: Sicherheit - Car-IT - Konzepte*. Wiesbaden : Springer Fachmedien Wiesbaden, 2014 (ATZ/MTZ-Fachbuch). http://dx.doi.org/10.1007/978-3-658-04019-2. http://dx.doi.org/10.1007/978-3-658-04019-2. – ISBN 978–3–658–04019–2

[99] Siegle, G. ; Geisler, J. ; Laubenstein, F. ; Nagel, H.-H. ; Struck, G.: Autonomous driving on a road network. In: *IEEE Intelligent Vehicles, 1992*. Piscataway : IEEE, 1992. – ISBN 0–780–30747–X, S. 403–408

[100] Singer, P. W. ; Friedman, Allan: *Cybersecurity and cyberwar: What everyone needs to know*. Oxford : Oxford Univ. Press, 2014 (What everyone needs to know). – ISBN 978–0–199–91811–9

[101] Spillner, Andreas ; Linz, Tilo: *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester - Foundation Level nach ISTQB-Standard*. 4. Aufl. s.l. : dpunkt.verlag, 2010. – ISBN 978–3–898–64642–0

[102] Steinbrink, Cornelius ; Lehnhoff, Sebastian ; Rohjans, Sebastian ; Strasser, Thomas I. ; Widl, Edmund ; Moyo, Cyndi ; Lauss, Georg ; Lehfuss, Felix ; Faschang, Mario ; Palensky, Peter ; van der Meer, Arjen A. ; Heussen, Kai ; Gehrke, Oliver ; Guillo-Sansano, Efren ; Syed, Mazheruddin H. ; Emhemed, Abdullah ; Brandl, Ron ; van Nguyen, Hoa ; Khavari, Ata ; Tran, Quoc T. ; Kotsampopoulos, Panos ; Hatziargyriou, Nikos ; Akroud, Akroud ; Rikos, Evangelos ; Degefa, Merkebu Z.: Simulation-based Validation of Smart Grids - Status Quo and Future Research Trends. In: *0302-9743* 10444 (2017), Nr. 8, 171–185. http://dx.doi.org/10.1007/978-3-319-64635-0{_}13. – DOI 10.1007/978–3–319–64635–0_13. – ISSN 0302–9743

[103] Symantec Corporation: Internet Security Threat Report. In: *2013 Trends* 19 (2014)

[104] TA Triumph-Adler GmbH: Der automatisierte Welthandel. In: *Talking Future*, 2015

[105] Thiemel, Arnulf V. ; Janke, Marcus ; Steurich, Björn: Speedometer Manipulation – Putting a Stop to Fraud. In: *ATZ elektronik worldwide Edition* 2013-02 (2013), 1–5. http://www.atzonline.com/Artikel/3/16068/Speedometer-Manipulation-%E2%80%93-Putting-a-Stop-to-Fraud.html

[106] Tian, Yuchi ; Pei, Kexin ; Jana, Suman ; Ray, Baishakhi: DeepTest: automated testing of deep-neural-network-driven autonomous cars. In: *2018 ACM/IEEE 40th International Conference on Software Engineering*. Piscataway, NJ : IEEE, 2018. – ISBN 978–1–450–35638–1

[107] Tung, Liam: VW-Audi security: Multiple infotainment flaws could give attackers remote access: Some VW and Audi models are vulnerable to remote hacking over Wi-Fi and cellular networks. (2018). https://www.zdnet.com/article/vw-audi-security-

multiple-infotainment-flaws-could-give-attackers-remote-access/

[108] United States Government Accountabiltity Office: Vehicle Cybersecurity: DOT and Industry Have Efforts under Way, but DOT Needs to Define Its Role in Responding to a Real-World Attack. In: Gibbs, Craig (Hrsg.): *Automotive Cybersecurity*. New York : Novinka, 2016 (Transportation issues, policies and R&D). – ISBN 978–1–634–85987–5, S. 1–52

[109] Valasek, Chris ; Miller, Charlie: *Adventures in Automotive Networks and Control Units*. http://dx.doi.org/10.1787/9789264208780-3-en

[110] Verband der Automobilindustrie e.V.: Automatisiertes Fahren: Der technologische Fortschritt zeigt sich bereits heute in modernen Fahrzeugen, die vermehrt Fahrerassistenzsysteme (FAS) besitzen, auf dem Weg zur Automatisierung. In: *Verband der Automobilindustrie e.V.* (2018). https://www.vda.de/de/themen/innovation-und-technik/automatisiertes-fahren/automatisiertes-fahren.html

[111] Verein zur Weiterentwicklung ders V-Modell XT e.V.: *V-Modell XT: Das deutsche Referenzmodell für Systementwicklungsprojekte*. 2015

[112] Vereinte Nationen: *Wiener Übereinkommen über den Strassenverkehr*. 1986

[113] Wagner, Stephan M.: *Europäische Hochschulschriften Reihe 5, Volks- und Betriebswirtschaft*. Bd. 2734: *Strategisches Lieferantenmanagement in Industrieunternehmen: Eine empirische Untersuchung von Gestaltungskonzepten: Zugl.: St. Gallen, Univ., Diss, 2000*. Frankfurt am Main : Lang, 2001. – ISBN 978–3–631–36225–9

[114] Wagner, Ulrike: *Schwachstellenanalyse der Informationssicherheit*. 1. Auflage, digitale Originalausgabe. München : GRIN Verlag, 2001. – ISBN 978–3–638–10127–1

[115] Wallentowitz, Henning (Hrsg.) ; Reif, Konrad (Hrsg.): *Handbuch Kraftfahrzeugelektronik: Grundlagen - Komponenten - Systeme - Anwendungen*. 2., verb. und aktualisierte Aufl. Wiesbaden : Vieweg + Teubner, 2011 (ATZ/MTZ-Fachbuch). – ISBN 978–3–8348–0700–7

[116] Weber, Marc: *Untersuchungen zur Anomalieerkennung in automotive Steuergeräten durch verteilte Observer mit Fokus auf die Plausibilisierung von Kommunikationssignalen*. KIT, Karlsruhe, 2019.

http://dx.doi.org/10.5445/IR/1000092815. http://dx.doi.org/10.5445/IR/1000092815

[117] Weber, Marc ; Klug, Simon ; Sax, Eric ; Zimmer, Bastian: Embedded Hybrid Anomaly Detection for Automotive CAN Communication. Version: 2018. https://hal.archives-ouvertes.fr/hal-01716805/document. In: *9th European Congress on Embedded Real Time Software and Systems*. Toulouse, France : Archives-Ouvertes, 2018, 1–10

[118] Wedeniwski, Sebastian: *The mobility revolution in the automotive industry: How not to miss the digital turnpike*. Heidelberg : Springer, 2015. http://dx.doi.org/10.1007/978-3-662-47788-5. http://dx.doi.org/10.1007/978-3-662-47788-5. – ISBN 978–3–662–47787–8

[119] Wegener, Joachim ; Buhr, Kerstin ; Pohlheim, Hartmut: Automatic Test Data Generation for Structural Testing of Embedded Software Systems by Evolutionary Testing. In: Langdon, W. B. (Hrsg.): *Proceedings of the Genetic and Evolutionary Computation Conference*. San Francisco, Calif. : Morgan Kaufmann, 2002 (GECCO'02). – ISBN 1–55860–878–8, 1233–1240

[120] Weicker, Karsten: *Evolutionäre Algorithmen*. 3., überarb. und erw. Aufl. Wiesbaden : Springer Vieweg, 2015. http://dx.doi.org/10.1007/978-3-658-09958-9. http://dx.doi.org/10.1007/978-3-658-09958-9. – ISBN 978–3–658–09957–2

[121] Weimerskirch, Andre: An Overview of Automotive Cybersecurity. In: Ray, Indrajit (Hrsg.) ; Wang, Xiofeng (Hrsg.) ; Ren, Kui (Hrsg.): *Proceedings of the 5th International Workshop on Trustworthy Embedded Devices*. New York, NY : ACM, 2015. – ISBN 978–1–450–33828–8, S. 53

[122] Whitaker, Andrew ; Shaw, Marianne, Gribble, Steven D.: Denali: Lightweight Virtual Machines for Distributed and Networked Applications. In: *Proceedings of the USENIX Annual Technical Conference* (2002). https://dada.cs.washington.edu/research/tr//2002/02/UW-CSE-02-02-01.pdf

[123] Williams, Jeff: *The Watcher of the Apple Watch*. Rancho Palos Verdes, CA, USA, 2015 (Re/code's Code Conference)

[124] Winner, Hermann (Hrsg.) ; Hakuli, Stephan (Hrsg.) ; Lotz, Felix (Hrsg.) ; Singer, Christina (Hrsg.): *Handbook of Driver Assistance Systems: Basic Information, Components and Systems for Active Safety and Comfort*. Cham and s.l. : Springer International Publishing, 2016. http://dx.doi.org/10.1007/978-3-319-12352-3. http://dx.doi.org/10.1007/978-3-319-12352-3. – ISBN 978–3–319–12352–3

[125] Wolf, Marko ; Scheibel, Michael: A Systematic Approach to a Quantified Security Risk Analysis for Vehicular IT Systems. In: *Automotive – Safety & Security*, 2012, 195–210

[126] Woody, Carol ; Ellison, Robert ; Nichols, William ; Software Engineering Institute (Hrsg.): *Predicting Software Assurance Using Quality and Reliability Measures*. 19.11.2018

[127] Wouters, Lennert ; Marin, Eduard ; Ashur, Tomer ; Gierlichs, Benedikt ; Preneel, Bart: Fast, Furious and Insecure: Passive Keyless Entry and Start Systems in Modern Supercars. In: *16th escar Europe - Embedded Security in Cars*, 2018

[128] Xanthakis, Spiros ; Ellis, C. ; Skourlas, Christos ; Le Gall, A. ; Katsikas, S. ; Karapoulios, K.: Application of genetic algorithms to software testing. In: *Proceedings of the 5th International Conference on Software Engineering and Applications*, 1992, S. 625–636

[129] Xiao, Chunyan ; Cheng, Dingning ; Wei, Kangzheng: An LCC-C Compensated Wireless Charging System for Implantable Cardiac Pacemakers: Theory, Experiment, and Safety Evaluation. In: *IEEE Transactions on Power Electronics* 33 (2018), Nr. 6, S. 4894–4905. http://dx.doi.org/10.1109/TPEL.2017.2735441. – DOI 10.1109/TPEL.2017.2735441. – ISSN 0885–8993

[130] Ziegler, Julius ; Bender, Philipp ; Schreiber, Markus ; Lategahn, Henning ; Strauss, Tobias ; Stiller, Christoph ; Dang, Thao ; Franke, Uwe ; Appenrodt, Nils ; Keller, Christoph G. ; Kaus, Eberhard ; Herrtwich, Ralf G. ; Rabe, Clemens ; Pfeiffer, David ; Lindner, Frank ; Stein, Fridtjof ; Erbs, Friedrich ; Enzweiler, Markus ; Knoppel, Carsten ; Hipp, Jochen ; Haueis, Martin ; Trepte, Maximilian ; Brenk, Carsten ; Tamke, Andreas ; Ghanaat, Mohammad ; Braun, Markus ; Joos, Armin ; Fritz, Hans ; Mock, Horst ; Hein, Martin ; Zeeb, Eberhard: Making Bertha

Drive—An Autonomous Journey on a Historic Route. In: *IEEE Intelligent Transportation Systems Magazine* 6 (2014), Nr. 2, S. 8–20. http://dx.doi.org/10.1109/MITS.2014.2306552. – DOI 10.1109/MITS.2014.2306552. – ISSN 1939–1390

[131] Zimmermann, Werner ; Schmidgall, Ralf: *Bussysteme in der Fahrzeugtechnik: Protokolle, Standards und Softwarearchitektur*. 5., aktual. und erw. Aufl. Wiesbaden : Springer Vieweg, 2014 (ATZ / MTZ-Fachbuch). http://dx.doi.org/10.1007/978-3-658-02419-2. http://dx.doi.org/10.1007/978-3-658-02419-2. – ISBN 978–3–658–02418–5

[132] Zink, Trevor ; Maker, Frank ; Geyer, Roland ; Amirtharajah, Rajeevan ; Akella, Venkatesh: Comparative life cycle assessment of smartphone reuse: Repurposing vs. refurbishment. In: *The International Journal of Life Cycle Assessment* 19 (2014), Nr. 5, 1099–1109. http://dx.doi.org/10.1007/s11367-014-0720-7. – DOI 10.1007/s11367–014–0720–7. – ISSN 1614–7502

# Webseitenverzeichnis


[133] Auto Motor Sport: *BMW 5er: Generationen ab 1972*. https://www.auto-motor-und-sport.de/bmw/5er/#generationen. Version: 2018

[134] AUTOSAR: *Specification of Platform Types*. https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_PlatformTypes.pdf

[135] Dobbins, Roland ; Bjarnson, Steinthor: *Mirai IoT Botnet Description and DDoS Attack Mitigation*. https://www.arbornetworks.com/blog/asert/mirai-iot-botnet-description-ddos-attack-mitigation/. Version: 2016

[136] Fraunhofer SIT: *EVITA: E-safety vehicle intrusion protected applications*. https://www.evita-project.org/. Version: 2011

[137] GXemul: *GXemul*. http://gxemul.sourceforge.net/. Version: 2018

[138] Imperas Software Limited: *Open Virtual Platforms: The source of Fast Processor Models & Platforms Open Virtual Platforms*. http://www.ovpworld.org/. Version: 2019

[139] Infineon Technologies AG: AURIX 32-bit microcontrollers for automotive and industrial applications: Highly integrated and performance optimized. (2019)

[140] Kingsley-Jones, Max: *Airbus begins tests to extend service life of A320 family*. https://www.flightglobal.com/news/articles/airbus-begins-tests-to-extend-service-life-of-a320-family-220962/. Version: 2008

[141] MH-Elektro- Steuerungstechnik: *Robotertechnik und Programmierung*. http://www.mh-hermann.de/Robotertechnik%20und%20Programmierung.html. Version: 2018

[142] MITRE Corporation: *Common Vulnerabilities and Exposures (CVE)*. https://cve.mitre.org/

[143] MITRE Corporation: *CVE-2017-9212*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9212. Version: 2018

[144] MITRE Corporation: *CVE-2017-9633*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9633. Version: 2018

[145] MITRE Corporation: *CVE-2017-9647*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9647. Version: 2018

[146] National Institute of Standards and Technology ; National Institute of Standards and Technology (Hrsg.): *National Vulnerability Database: CWE Over Time*. https://nvd.nist.gov/vuln/visualizations/cwe-over-time. Version: 2018

[147] NXP Semiconductors: *Ultra-Reliable MPC564xA MCU for Automotive & Industrial Engine Management*. https://www.nxp.com/products/processors-and-microcontrollers/power-architecture-processors/mpc5xxx-55xx-32-bit-mcus/ultra-reliable-mpc56xx-32-bit-automotive-and-industrial-microcontrollers-mcus/ultra-reliable-mpc564xa-mcu-for-automotive-industrial-engine-management:MPC564xA. Version: 2019

[148] NXP Semiconductors ; NXP Semiconductors (Hrsg.): *Ultra-Reliable MPC57xx 32-bit Automotive and Industrial Microcontrollers (MCUs)*. https://www.nxp.com/products/processors-and-microcontrollers/power-architecture-processors/mpc5xxx-55xx-32-bit-mcus/ultra-reliable-mpc57xx-32-bit-automotive-and-industrial-microcontrollers-mcus:MPC57XX. Version: 2019

[149] Online, Spiegel: *Mercedes S-Klasse: Alle Baureihen im Überblick*. http://www.spiegel.de/fotostrecke/mercedes-s-klasse-alle-baureihen-im-ueberblick-fotostrecke-96643.html. Version: 2013

[150] QEMU: *QEMU: the FAST! processor emulator*. https://www.qemu.org/. Version: 2018

[151] Reynolds, Matthew: *TalkTalk and Post Office customers hit by Mirai worm attack: The Mirai worm also hit the broadband,*

*internet and TV networks of 900,000 Deutsche Telekom customers*. http://www.wired.co.uk/article/deutsche-telekom-cyber-attack-mirai. Version: 2016

[152] Software Testing Academy: *Software Testing Methoden*. https://www.software-testing.academy/software-testing-methoden.html. Version: 2019

[153] Statista GmbH ; Statista - Das Statistik-Portal (Hrsg.): *Absatz von Industrierobotern weltweit in den Jahren 2004 bis 2016*. https://de.statista.com/statistik/daten/studie/29386/umfrage/weltweiter-absatz-fuer-industrie-roboter-seit-2004/ (Statista - Das Statistik-Portal)

[154] Statista GmbH ; Statista - Das Statistik-Portal (Hrsg.): *Anzahl der bei Airbus und Boeing bestellten Flugzeuge in den Jahren 2003 bis 2017*. https://de.statista.com/statistik/daten/studie/36629/umfrage/auftragsentwicklung-bei-airbus-und-boeing/ (Statista - Das Statistik-Portal)

[155] Statista GmbH ; Statista - Das Statistik-Portal (Hrsg.): *Anzahl der weltweiten Neuzulassungen von Pkw in den Jahren 2013 bis 2018*. https://de.statista.com/statistik/daten/studie/247129/umfrage/weltweite-neuzulassungen-von-pkw/ (Statista - Das Statistik-Portal)

[156] Statista GmbH ; Statista - Das Statistik-Portal (Hrsg.): *Prognose zur Anzahl der Smartphone-Nutzer weltweit von 2012 bis 2020*. https://de.statista.com/statistik/daten/studie/309656/umfrage/prognose-zur-anzahl-der-smartphone-nutzer-weltweit/ (Statista - Das Statistik-Portal)

[157] Synopsys Inc.: *Virtual Prototyping*. https://www.synopsys.com/verification/virtual-prototyping.html. Version: 2018

[158] Texas Instruments Incorporated: *Automotive Processors: TDAx ADAS SoCs – Driver monitoring*. http://www.ti.com/processors/automotive-processors/tdax-adas-socs/applications/driver-monitoring-system.html. Version: 2019

[159] VAU-MAX: *40 Jahre VW Golf*. https://www.vau-max.de/magazin/klassik/40-jahre-vw-golf-die-vau-max-

de-golf-uebersicht-die-vw-golf-geschichte-in-bildern-modelljahr-fuer-modelljahr.293. Version: 2016

[160] Wheatley, Mike: *Known vulnerabilities cause 44 percent of all data breaches*. http://siliconangle.com/blog/2016/01/12/known-vulnerabilities-cause-44-percent-of-all-data-breaches/. Version: 2016

[161] Wind River Systems Inc.: Wind-River-Simics: Product-Overview. (2015). https://www.windriver.com/products/product-overviews/Wind-River-Simics_Product-Overview.pdf

# Eigene Veröffentlichungen


[BBB+17] Bapp, Falco ; Becker, Jürgen ; Beyerer, Jürgen ; Doll, Jens ; Filsinger, Max ; Frese, Christian ; Hubschneider, Christian ; Lauber, Andreas ; Müller-Quade, Jörn ; Pauli, Mario ; Roschani, Masoud ; Salscheider, Ole ; Rosenhahn, Bodo ; Ruf, Miriam ; Stiller, Christoph ; Willersinn, Dieter ; Ziehn, Jens R.: A Non-Invasive Cyberrisk in Cooperative Driving. In: *TÜV-Tagung Fahrerassistenz*, 2017

[BLES19] Brenner, Nathalie ; Lauber, Andreas ; Eckert, Carsten ; Sax, Eric: Autonomous Driving of Commercial Vehicles within Cordoned Off Terminals. In: Gusikhin, Oleg (Hrsg.) ; Helfert, Markus (Hrsg.): *5th International Conference on Vehicle Technology and Intelligent Transport Systems (VEHITS)*, SCITEPRESS – Science and Technology Publications, Lda., 2019. – ISBN 978–9–897–58374–2, S. 521–527

[BLS+20] Böhme, Martin ; Lauber, Andreas ; Stang, Marco ; Pan, Luyi ; Sax, Eric: Using Machine Learning to Optimize Energy Consumption of HVAC Systems in Vehicles. Version: 2020. `http://dx.doi.org/10.1007/978-3-030-25629-6{_}110`. In: Ahram, Tareq (Hrsg.) ; Taiar, Redha (Hrsg.) ; Colson, Serge (Hrsg.) ; Choplin, Arnaud (Hrsg.): *Human Interaction and Emerging Technologies* Bd. 1018. Cham : Springer International Publishing, 2020. – DOI 10.1007/978–3–030–25629–6_110. – ISBN 978–3–030–25628–9, S. 706–712

[GLMS19] Guissouma, Houssem (Hrsg.) ; Lauber, Andreas (Hrsg.) ; Mkadem, Amir (Hrsg.) ; Sax, Eric (Hrsg.): *Virtual Test Environment for Efficient Verification of Software Updates for Variant-Rich Automotive Systems: The 13th Annual IEEE International Systems Conference (SysCon), Orlando, FL, April 8 –11, 2019*. IEEE, Piscataway, NJ, 2019

[HPL+15] Hartmann, Frank ; Pistorius, Felix ; Lauber, Andreas ; Hildenbrand, Kai ; Becker, Jürgen ; Stork, Wilhelm: Design of an embedded UWB hardware platform for navigation in GPS denied environments. In: *IEEE Symposium on Communications and Vehicular Technology in the Benelux (SCVT)*, 2015, S. 1–6

[IELS17] Ilahi, Seyed S. ; Ellwanger, Simon ; Lauber, Andreas ; Sax, Eric: Video Transmission for Autonomous Truck Platoons using WLAN Broadcast. In: *12th ITS European Congress*, 2017

[Lau19] Lauber, Andreas: Mobilität der Zukunft auf dem Testfeld Autonomes Fahren Baden-Württemberg: Roboterautos nehmen Fahrt auf! In: *E-Mobilität in der Metropolregion Rhein-Neckar* 2019 (2019), Nr. 1, S. 34–35

[LBFS20] Lauber, Andreas ; Böhme, Martin ; Fuchs, Kevin ; Sax, Eric: Evolutionary Algorithms to Generate Test Cases for Safety and IT-Security in Automotive Systems. In: *14th Annual IEEE International Systems Conference (SysCon)*, IEEE, 2020

[LBS19] Lauber, Andreas ; Brenner, Nathalie ; Sax, Eric: Automated Vehicle Depots as an Initial Step for an Automated Public Transportation. In: *UITP Global Public Transportaion Summit*, 2019

[LGS18] Lauber, Andreas ; Guissouma, Housemeddine ; Sax, Eric: Virtual Test Method for Complex and Variant-Rich Automotive Systems. In: *2018 IEEE International Conference on Vehicular Electronics and Safety (ICVES)*. Piscataway, NJ : IEEE, 2018. – ISBN 978–1–5386–3543–8

[LGSW16] Lauber, Andreas ; Glock, Thomas ; Sax, Eric ; Wiedemann, Markus: Analyzation and Evaluation of Vehicle and Infrastructure for Autonomous Driving on Public Transportation Depots. In: K. Berns, K. Dressler, P. Fleischmann, R. Ilsen, B. Jörg, R. Kalmar, T. Nagel, C. Schindler, and N. K. Stephan (Hrsg.): *Commercial Vehicle Technology*. Achen : Shaker Verlag, 2016. – ISBN 978–3–8440–4229–0, S. 3–12

[LS17] Lauber, Andreas ; Sax, Eric: Testing Security of Embedded Software through Virtual Processor Instrumentation. In: *Proceedings of 2017 14th International Conference on Remote Engineering and Virtual Instrumentation (REV)*, Springer-Verlag Berlin Heidelberg, 2017

[LS18] Lauber, Andreas ; Sax, Eric: Testing Security of Embedded Software through Virtual Processor Instrumentation // Online Engineering & Internet of Things. Version: 2018. http://dx.doi.org/10.1007/978-3-319-64352-6{_}9. In: Auer, Michael E. (Hrsg.) ; Zutin, Danilo G. (Hrsg.): *Online engineering & internet of things // Online engineering & Internet of Things* Bd. 22. Cham, Switzerland : Springer and Springer International Publishing, 2018. – DOI 10.1007/978–3–319–64352–6_9. – ISBN 978–3–319–64352–6, S. 85–94

[LSW18a] Lauber, Andreas ; Sax, Eric ; Wiedemann, Markus: Autonomes Fahren auf dem Busbetriebshof. In: *ATZ - Automobiltechnische Zeitschrift* 120 (2018), Nr. 6, 74–77. http://dx.doi.org/10.1007/s35148-018-0047-y. – DOI 10.1007/s35148–018–0047–y. – ISSN 0001–2785

[LSW18b] Lauber, Andreas ; Sax, Eric ; Wiedemann, Markus: Autonomous driving in public transportation depots. In: *ATZ worldwide* 120 (2018), Nr. 6, S. 68–71. http://dx.doi.org/10.1007/s38311-018-0051-6. – DOI 10.1007/s38311–018–0051–6. – ISSN 2192–9076

[LZK11] Lauber, Andreas ; Zimmermann, Werner ; Kappen, Nikolaus: Optimierung kryptographischer Methoden für eingebettete Systeme. In: *IT-Innovationen*. 2011, S. 22

[PLP+16] Pistorius, Felix ; Lauber, Andreas ; Pfau, Johannes ; Klimm, Alexander ; Becker, Jürgen: Development of a latency optimized communication device for WAVE and SAE based V2X-Applications. In: *SAE 2016 World Congress and Exhibition*, 2016

[TLW+15] Tradowsky, Carsten ; Lauber, Andreas ; Werner, Stephan ; Beuth, Thorsten ; Müller-Glaser, Klaus D. ; Sax, Eric: Porter for the ITIV-Labs - Objective-Related Engineering Education in an Undergraduate Laboratory. In: *International Journal of Teaching and Education (iJOE)* 04 (2015), 45–58. http://www.universitypublications.net/jte/0401/pdf/DE4C216.pdf

[WLBS16] Werner, Stephan ; Lauber, Andreas ; Becker, Jürgen ; Sax, Eric: Cloud-based Remote Virtual Prototyping Platform for Embedded Control Applications: cloud-based infrastructure for large-scale embedded hardware-related programming laboratories. In:

*Proceedings of 2016 13th International Conference on Remote Engineering and Virtual Instrumentation (REV)*, IEEE, 2016. – ISBN 9781467382465

[WLK+16] Werner, Stephan ; Lauber, Andreas ; Koedam, Martijn ; Becker, Jürgen ; Sax, Eric ; Goossens, Kees: Cloud-based Design and Virtual Prototyping Environment for Embedded Systems. In: *International Journal of Online Engineering (iJOE)* 12 (2016), Nr. 09, S. 52–60. http://dx.doi.org/10.3991/ijoe.v12i09.6142. – DOI 10.3991/ijoe.v12i09.6142

# Eigene Patente

[Nothalt2018] Ziehn, Jens; Ruf, Miriam; Weißenbach, Stefanie; Doll, Jens; Flad, Michael; Frey, Michael; Knoch, Eva-Maria Judith; Kohlhaas, Ralf; Lauber, Andreas; Pistorius, Felix; Rothfuß, Simon. Konzept zur Durchführung eines Nothaltmanövers. *Internationale Nummer: WO 2020/089270 A1*, Internationale Offenlegung am 7. Mai 2020

# Betreute studentische Arbeiten

[Blu16] Blum, Stefano: *Echtzeit- u. Latenzevaluation einer Plattform zur Konzeptionierung einer optimierten Car2X Kommunikationseinheit*. Karlsruhe, Karlsruhe Institute for Technologie, Bachelorarbeit, 01.05.2016

[Die15] Diewald, Axel: *Konzeptionierung und Entwicklung eines Infrarotbasierten "Follow-Me"-Demonstrator-Fahrzeuges*. Karlsruhe, Karlsruhe Institute for Technologie, Bachelorarbeit, 20.04.2015

[Ebb15] Ebbert, Sebastian: *Konzeptionierung und Aufbau einer Smart-ECU zur Trajektorien gestützten Ansteuerung von Mecanum Wheels*. Karlsruhe, Karlsruhe Institute for Technologie, Bachelorarbeit, 05.05.2015

[Fuc16] Fuchs, Kevin: *Aufbau einer Datenbank zur Klassifizierung und Kategorisierung von Security-Schwachstellen im Automobil*. Karlsruhe, Karlsruhe Institute for Technologie, Bachelorarbeit, 17.10.2016

[Fuc19] Fuchs, Kevin: *Testfallgenerierung durch evolutionäre Algorithmen zur Gewährleistung der Cyber-Security in Automotive Systemen*. Karlsruhe, Karlsruhe Institute for Technologie, Masterarbeit, 31.03.2019

[Had14] Haderer, Lukas: *ECDSA Implementation and Latency Studies for Car-to-X Systems*. Karlsruhe, Karlsruhe Institute for Technologie, Bachelorarbeit, 05.05.2014

[Hil15] Hildenbrand, Kai F.: *Konzept und Implementierung einer Embedded UWB-Lokalisierung zur Indoor Navigation*. Karlsruhe, Karlsruhe Institute for Technologie, Bachelorarbeit, 04.05.2015

[Hol17] Holzmann, Emanuel: *Aufbau einer Virtualisierungsumgebung zur Verifikation von Security Anwendungen*. Karlsruhe, Karlsruhe Institute for Technologie, Masterarbeit, 24.01.2017

[Kam14] Kamm, Simon: *Entwurf und Implementierung einer ZYNQ-Hardware-Schnittstelle zur Kommunikation mit dem Modellierungstool Ptolemy II*. Karlsruhe, Karlsruhe Institute for Technologie, Bachelorarbeit, 19.11.2014

[Kar15] Kara, Kaan: *Design of a SoC Framework for Dynamic Distributed Scheduling in Car2x Applications*. Karlsruhe, Karlsruhe Institute for Technologie, Masterarbeit, 03.09.2015

[Ngu15] Nguyen, Xuan V.: *Konzeptionierung und Simulation eines „Smart Hard Braking Assist" im Car2X-Kontext*. Karlsruhe, Karlsruhe Institute for Technologie, Bachelorarbeit, 04.05.2015

[Pfa15] Pfau, Johannes: *Evaluation und Implementierung eines latenzoptimierten Kommunikation-Managers im Car2X Kontext*. Karlsruhe, Karlsruhe Institute for Technologie, Bachelorarbeit, 26.10.2015

[Pis14] Pistorius, Felix: *Entwicklung einer latenzoptimierten Kommunikationseinheit für Car-to-X Anwendungen im WAVE-Kontext*. Karlsruhe, Karlsruhe Institute for Technologie, Masterarbeit, 01.07.2014

[Sch15] Schwär, Daniel A.: *Concept and Implementation of a Kalman-Filter for Optimization of an Inertial-Navigationsystem*. Karlsruhe, Karlsruhe Institute for Technologie, Bachelorarbeit, 08.12.2015

[Sch16] Schweizer, Dennis: *Aufbau einer Virtualisierungsumgebung zur Verifikation von Security-Anwendungen*. Karlsruhe, Karlsruhe Institute for Technologie, Bachelorarbeit, 05.12.2016

[See16] Seegert, Thomas: *Konzeptionierung und Entwicklung eines HiL-Teststandes für Car2X-Kommunikationsgeräte*. Karlsruhe, Karlsruhe Institute for Technologie, Masterarbeit, 17.02.2016

[Ste16] Stelzer, Florian: *Evaluierung der Herausforderungen und Potentiale einer elektronischen Deichsel und Platooning im ÖPNV*. Karlsruhe, Karlsruhe Institute for Technologie, Bachelorarbeit, 18.10.2016

[Vet17] Vetter, Andreas: *Automatisierte Strukturanalyse für Software in Eingebetteten Systemen*. Karlsruhe, Karlsruhe Institute for Technologie, Masterarbeit, 07.11.2017

[Weg15] Wegmann, Laura S.: *Konzeptionierung und Implementierung einer Fahrzeugverfolgung basierend auf einer low-cost Netzwerkkamera*. Karlsruhe, Karlsruhe Institute for Technologie, Bachelorarbeit, 04.05.2015

[Wob15] Wobker, Thilo: *Konzeptionierung und Implementierung von Car2X-basierten Fahrmanövern durch verteiltes kooperatives Rechnen*. Karlsruhe, Karlsruhe Institute for Technologie, Bachelorarbeit, 19.04.2015