Kryptografische Insel zur Realisierung von Angreifertoleranz in redundanten Systemen

DISSERTATION

zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

durch die Fakultät Wirtschaftswissenschaften
Universität Duisburg-Essen
Campus Essen

vorgelegt von
Johannes Formann
geboren am 07.08.1983 in Münster

Essen, 2016

Datum der Einreichung:	13.01.2016
Datum der mündlichen Prüfung:	22.04.2016
Erstgutachter:	Prof. Dr. Klaus Echtle
Zweitgutachter:	Prof. Dr.-Ing. Felix Freiling

Johanness Formann: *Kryptografische Insel zur Realisierung von Angreifertoleranz in redundanten Systemen,* Eine effektive Lösung zur Angreifertoleranz in selbstsichernden und fehlertolerierenden Systemen durch Isolation der beteiligten Komponenten mittels divers implementierten Maskeradeschutz, © 2016

ABSTRACT

In this thesis I will show why the traditional disjoint treatment of security and fault tolerance has weaknesses if the attacker gains access to the fault tolerant system and how an integrated approach that utilizes existing fault tolerance techniques can provide security effectively.

An efficient integrated safety and security approach is presented for fault tolerant systems (fail safe and fail operational), which achieves protection against attacks over the network by forming a logically isolated (sub-) network which is resilient against a bug in a software component. Isolation is obtained by diverse design of a general reusable component that prevents any unauthorized message transfer towards the secured application program. Messages from other compromised nodes are tolerated utilizing existing mechanims. The reuse of the existing fault tolerance techniques and the reuse of the additional parts needed for attacker tolerance can significantly reduce the required effort compared to existing systems.

ZUSAMMENFASSUNG

In dieser Arbeit wird ein neuer Ansatz zur Angreifertoleranz entwickelt, der durch eine hohe Wiederverwendbarkeit für das individuelle System nur geringen zusätzlichen Entwicklungsaufwand bedeutet. Dazu wird zunächst ein Überblick über die Schwachstellen von verteilten Systemen gegenüber Angreifern dargestellt und bestehende Techniken zur Eindämmung und Tolerierung beschrieben.

Der Hauptteil befasst sich mit einem neuen Ansatz, bei dem durch geschickte Kombination bestehender Sicherungstechniken für fehlererkennde Systeme und fehlertolerante Systeme, welche Mehrheitsentscheidungen nutzen, eine wiederverwendbare Sicherungsschicht entwickelt wird, die das System auch gegenüber Angreifern tolerant macht. Aufgrund der Wiederverwendbarkeit und der Mitnutzung der benötigten Fehlererkennungs- und Fehlertoleranztechniken ist ein signifikant geringerer Aufwand pro System, verglichen mit herkömmlichen Angreifertoleranztechniken, erzielbar.

VERÖFFENTLICHUNGEN

Einzelne Ideen und Abbildungen wurden bereits bei den folgenden Gelegenheiten veröffentlicht:

- 22.11.2012 Vortrag „The Byzantine Generals got new Ammo“ im Rahmen des Diskussionskreis Fehlertoleranz 2012 der GI/ITG-Fachgruppe "Fehlertolerierende Rechensysteme (FERS)" in Nürnberg
- 17.09.2013 Vortrag „An AUTOSAR compliant approach to secure applications against malicious attacker“ im Rahmen der 5. EUROFORUM-Jahrestagung "ISO 26262" in Stuttgart
- 18.03.2014 Vortrag „Security for safety relevant systems“ im Rahmen der IQPC "Automotive Embedded Systems" Konferenz in Düsseldorf
- 21.03.2014 Vortrag „An efficient approach to tolerate attackers in fault-tolerant systems“ auf der Sicherheit 2014 in Wien; veröffentlicht im GI Edition Proceedings Band 228 - Sicherheit 2014 - Sicherheit, Schutz und Zuverlässigkeit : Beiträge der 7. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI), ISBN: 978-3-88579-622-0
- 04.12.2014 Vortrag „Cryptographic island – An extension for fault tolerant systems to enable intrusion tolerance“ im Rahmen des Diskussionskreis Fehlertoleranz 2014 der GI/ITG-Fachgruppe "Fehlertolerierende Rechensysteme (FERS)" in Dresden
- 26.05.2015 Vortrag „Cryptographic island – An extension for fault tolerant systems to enable intrusion tolerance“ im Rahmen des Doktorandenseminars am Lehrstuhl Informatik 1 der Universität Erlangen
- 06.09.2015 Vortrag „Sicherheit meint nicht Sicherheit und erst recht nicht Verfügbarkeit“ im Rahmen der MRMCD 2015 in Darmstadt

DANKSAGUNG

Ich bedanke mich bei allen, die mich während der Erstellung der Arbeit unterstützt haben.

Insbesondere bedanke ich mich bei Prof. Echtle für die Unterstützung, für den eingeräumten Freiraum, der mir die Erforschung dieses Themas ermöglichte, und für den inhaltlichen Austausch sowie die hilfreichen Kommentare bei der Erstellung der Ausarbeitung.

Des Weiteren danke ich meiner Familie und meiner Partnerin, die mir Rückhalt boten und viel Verständnis für meine Arbeitsbelastung und die damit verbundenen Einschränkungen aufbrachten.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

ACRONYMS

AUTOSAR	AUTomotive Open System ARchitecture
API	Application Programming Interface
CRC	Cyclic Redundancy Check
HMAC	Keyed-Hash Message Authentication Code
IP	Internetprotokoll
MITM	man in the middle
PKI	Public-Key-Infrastruktur
SCADA	Supervisory Control and Data Acquisition
COTS	Commercial off-the-shelf
POSIX	Portable Operating System Interface
TCP	Transmission Control Protocol

NOMENCLATURE

Bug	Fehler in einem Programm, der zu einer Abweichung von dem spezifizierten und/oder vom Nutzer erwarteten Verhaltens führt.
Exploit	Programmcode oder Handlungsanweisungen, die geeignet sind, eine Sicherheitslücke in einem System zum Ausführen eigener Anweisungen zu missbrauchen.
Funktionale Sicherheit	Maßnahmen um schädliche ungewollte Interaktionen mit der Umwelt auf eine gesellschaftlich akzeptierte Wahrscheinlichkeit zu reduzieren. (Im Englischen safety)
Sicherheit	Maßnahmen um das System vor unbeabsichtigter Beeinflussung durch die Umgebung zu schützen. Dies betrift sowohl die Veränderung des Verhaltens als auch Gewinnung von Informationen. Im Englischen wird dieser Sicherheitsbegriff meist als security bezeichnet. Als Synonyme werden teilweise die Begriffe IT-Sicherheit und Informationssicherheit verwendet, um die Abgrenzung zur funktionalen Sicherheit auszudrücken.

Teil I

EINLEITUNG

1

EINLEITUNG UND MOTIVATION

1.1 MOTIVATION

Cyber-physikalische Systeme [153] sind aus dem Leben nicht mehr wegzudenken. Sie kontrollieren und beeinflussen eine Vielzahl von Prozessen. Ein fehlerhaftes cyber-physikalisches System kann dabei sehr unterschiedliche Auswirkungen haben. Diese reichen von geringen Komforteinbußen oder geringen finanziellen Verlusten (beispielsweise bei einen System zur automatischen Optimierung des Heizungsbetriebs) bis hin zu erheblichen Sachschäden und dem Risiko von schweren bis letalen Verletzungen (beispielsweise bei der Prozessteuerung in der chemischen Industrie, Steuerungen von Fahrzeugen).

Im Bereich der Forschung zur funktionalen Sicherheit (im englischen „safety“) wurden im Laufe der letzten Jahrzehnte eine Vielzahl von Techniken entwickelt, die durch Fehlererkennung oder Fehlertoleranz mit hoher Wahrscheinlichkeit verhindern, dass aus Fehlern innerhalb eines cyber-physikalisches Systems ein gefährliches Fehlverhalten für die Umwelt entsteht. Die grundlegenden generischen Techniken sind nahezu vollständig erforscht [48], so dass aktuelle Forschungen sich vorwiegend auf die Effizienz von Verfahren für bestimmte Anwendungszwecke konzentrieren [92, 133, 108].

Diese Kenntnisse werden auch in der Industrie angewendet. Die Norm IEC 61508 [20] definiert für Maschinensteuerungen die Kriterien, nach denen evaluiert werden kann, wie hoch die Gefahr, die von dem cyber-physikalischen System ausgeht, bewertet wird, und stellt daraufhin Forderungen an die funktionale Sicherheit (der sogenannte „safety integrity level“ oder kurz SIL).

Cyber-physikalische Systeme mit potentiell gefährlichen Fehlfunktionen waren traditionell in sich geschlossene Systeme, so dass die Probleme der (IT-)Sicherheit (im Englischen „security“) mit externen Angreifern mangels vorhandener Schnittstellen zu öffentlichen Systemen nicht von Relevanz waren. In den verschiedenen Anwendungsdomänen werden allerdings zunehmend verschiedene Systeme vernetzt. In der Industrie bezeichnet man beispielsweise diese Aspekte mit den Schlagworten „horizontale Integration“, „vertikale Integration“ oder auch „Industrie 4.0“. Dabei entstehen oftmals (ungeplante) schlecht gesicherte Verbindungen zu öffentlichen Netzen [30, 19]. Durch die Verbindung mit öffentlichen Netzen benötigen funktional sichere Systeme auch Sicherheit vor Angreifern, da sich diese in der Regel außerhalb des Fehlermodells bewegen (siehe Kapitel 2.2 und

Kapitel 2.3.6).
Beispiele hierfür finden sich in den verschiedensten Anwendungsbereichen, die traditionell funktionale Sicherheit analysieren und damit oftmals selbstsichernde oder fehlertolerierende Systeme einsetzen (Automatisierungstechnik/Anlagenbau [29, 109, 54], Automotive [94, 59, 110, 71] und Avionik [162, 145]).

Da die Begriffe (funktionale) Sicherheit (safety) und (IT-) Sicherheit (security) im Deutschen oftmals gleich mit Sicherheit bezeichnet werden, wird in dieser Arbeit Sicherheit als (IT-) Sicherheit verwendet und bei funktionaler Sicherheit diese entsprechend explizit benannt.

1.2 ZIELE

Ein Bug gehört aus Sicht der funktionalen Sicherheit zu der Klasse der systematischen Fehler, da er auf allen Instanzen der betroffenen Komponente reproduzierbar ist. Als generische Gegenmaßnahme gegen systematische Fehler dient eine diversitäre Entwicklung (siehe Kapitel 2.5.5), bei der dann die Instanzen einer Komponente durch von unterschiedlichen Teams geschriebenen Varianten ersetzt werden. Da diese jedoch einen hohen Entwicklungsaufwand mit sich zieht, findet dieser Ansatz in der Praxis nur sehr selten Anwendung (Kernelemente der Fly-by-Wire Steuerung von Flugzeugen, Steuerung von Atomkraftwerken, Bahn-Stellwerke) – und bei diesen motiviert durch die zu erreichende funktionale Sicherheit.

Das Ziel der Arbeit ist es daher, ein Framework zu entwickeln, welches mit geringem Aufwand ermöglicht, Bugs, die ein Angreifer ausnutzen kann, um Einfluss auf das Systemverhalten zu nehmen, zu tolerieren. Die im System vorhandenen Fehlertoleranztechniken sollen mitbenutzt werden, um den zusätzlichen Aufwand zu minimieren. Das Framework soll sich sowohl für Neuentwicklungen als auch Nachrüstungen von bestehenden Systemen eignen und dabei zur Aufwandsreduktion wiederverwendbare Implementierungen ermöglichen. Um den Aufwand für die Entwicklung des Frameworks auf das Nötigste zu beschränken, werden die Teile identifiziert, bei denen Diversität notwendig ist. Dabei werden die unterschiedlich umfangreichen möglichen Realisierungen diskutiert und Vorteile und Nachteile einander gegenübergestellt.

Dazu werden im Teil ii zunächst die Grundlagen der betrachteten Systeme sowie die später genutzten Basistechniken wie Fehlererkennung, Fehlertoleranz und Kryptographie vorgestellt und ein kurzer Überblick über andere Ansätze zur Angreifertoleranz gegeben.
Im Teil iii wird die kryptographische Insel in den Kapiteln 5 bis 7 entwickelt und in Kapitel 8 und Kapitel 9 die Eignung für verschiedene Anwendungen bewertet und der Aufwand verglichen.
In Teil iv findet sich zum Schluss die Zusammenfassung mit dem Ausblick.

Teil II

GRUNDLAGEN & PROBLEMANALYSE

In diesem Teil werden die Grundlagen der betrachteten Systeme sowie die später genutzten Basistechniken wie Fehlererkennung, Fehlertoleranz und Kryptographie vorgestellt und ein kurzer Überblick über andere Ansätze zur Angreifertoleranz gegeben. Dies unterteilt sich in die Grundlagen von fehlertoleranten Systemen, die genutzten Komponenten (Kommunikationssysteme, Software, Kryptographie), die Beschreibung der Fähigkeiten eines Angreifers und die Erläuterung der verschiedenen Maßnahmen gegen Störungen. Dabei werden sowohl einzelne Techniken, die bestimmte Angriffe erschweren/verhindern, als auch übergreifende Ende-zu-Ende Ansätze sowie bestehende Angreifertoleranzverfahren vorgestellt.

2

BETRACHTETE SYSTEME UND ANGRIFFSMÖGLICHKEITEN

Im Rahmen der Arbeit werden verteilte Systeme betrachtet. In Kapitel 2.1 wird die angenommene Struktur der Systeme definiert und in Kapitel 2.3 die unterschiedlichen Ausprägungen von Fehlertoleranz diskutiert.

2.1 SYSTEMDEFINITION

Die Definitionen eines verteilten Systems in der Literatur sind sehr ähnlich, unterscheiden sich aber durch den Abstraktionsgrad:

> „A distributed system is a collection of independent computers that appears to its users as a single coherent system.“ [144] (Ein verteiltes System ist eine Menge von unabhängigen Computern, die gegenüber den Nutzern als ein kohärentes System erscheinen)
>
> "Eine verteilte Anwendung besteht aus einer Menge kommunizierender Betriebssystemprozesse, die nicht über einen gemeinsamen Speicher verfügen und auf unterschiedliche Rechner verteilt sein können. Ein zugrunde liegendes verteiltes System, das sich aus diesen Rechnern zusammensetzt, ermöglicht die Kommunikation der Betriebssystemprozesse auf der Basis einer physikalischen Rechnerkopplung. Diese Kopplung kann direkt über ein lokales Netz oder auch indirekt über zwischengeschaltete Gateways zur Verbindung unterschiedlicher lokaler Netze erfolgen." [114]

Allen Definitionen ist dabei gemein, dass der Dienst des Systems für den Benutzer durch mehrere interagierende voneinander unabhängige Teilsystemen erbracht wird. Um den Dienst zu erbringen, interagiert ein System über Schnittstellen (z. B. Sensoren und Aktuatoren) mit der Umgebung. Ein verteiltes System in Abgrenzung zur Umwelt lässt sich wie in Abbildung 1 dargestellt visualisieren.

In der Literatur [144, 37] werden die Teilsysteme eines verteilten Systems als Knoten bezeichnet. Zur Interaktion der Knoten wird in der Regel eine mehrschichtige Hard- und Softwarearchitektur definiert, wie sie Abbildung 2 zeigt.

> "Die untersten Hardware- und Softwareschichten werden häufig als Plattformen für verteilte Systeme und Applikationen bezeichnet. Diese Low-Level-Schichten stellen Diens-

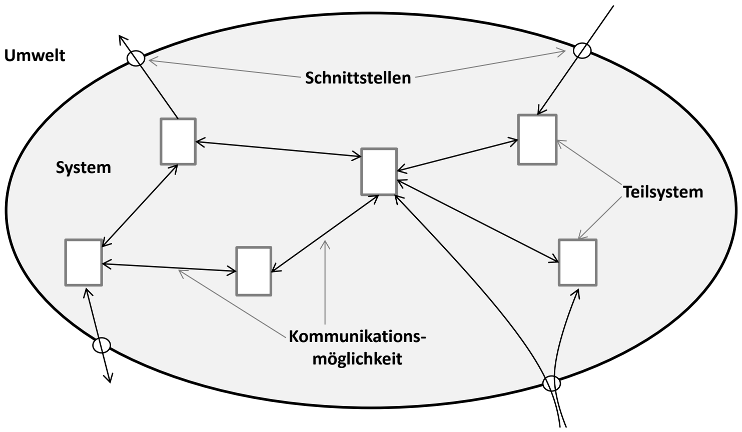


Abbildung 1: Aufbau eines verteilten Systems

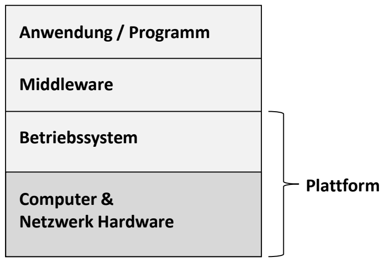


Abbildung 2: Aufbau eines Computersystems nach [37]

> te für die darüber liegenden Schichten bereit, die in jedem Computer unabhängig implementiert werden, wodurch die Programmierschnittstellen des Systems auf eine Ebene gebracht wird, die die Kommunikation zwischen den Prozessen vereinfacht. Wichtige Beispiele sind Intel x86/Windows, Sun SPARC/SunOS, Intel x86/Solaris, PowerPC/MacOS oder Intel x86/Linux.
> Middleware wurde [...] als Softwareschicht definiert, deren Aufgabe es ist, Heterogenität zu verbergen und den Applikationsprogrammierern ein praktisches Programmiermodell bereitzustellen. Middleware wird durch Prozesse oder Objekte in mehreren Computern dargestellt, die zusammenarbeiten, um die Kommunikation und die gemeinsame Nutzung von Ressourcen zu unterstützen." [37]

Die Anzahl der Abstraktionsebenen ist in einigen Anwendungsszenarien nicht immer realistisch (zum Beispiel bei preiswerten Sensoren, die an ein Kommunikationssystem angeschlossen sind), so dass im Folgenden mit einem etwas vereinfachten Modell wie in Abbildung 3 gearbeitet wird. Die optionale Hardwareabstraktionsebene kann dabei nur aus einem Betriebssystem oder auch zusätzlich wie bei [37] mit einer Middleware-Schicht. Damit ergeben sich die folgenden Komponenten, aus denen ein verteiltes System aufgebaut wird:

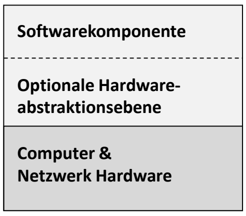


Abbildung 3: Aufbau eines Knotens

ANWENDUNG: Ein Dienst, den das System für die Umwelt erbringt. Sie kann durch einen einzelnen Knoten realisiert werden oder durch die Zusammenarbeit mehrerer Knoten.

AKTUATOR: Ein physikalisches Element an einer Schnittstelle zwischen Umwelt und System, welches auf die Umwelt einwirkt. Beispiele sind optische oder akustische Alarme, Stellmotoren und Ventile. Ein Aktuator ist an einen Knoten angeschlossen und wird durch ein oder mehrere Programme angesteuert.

BETRIEBSSYSTEM: Ein Betriebssystem ist ein Programm, welches eine Hardwareabstrahierungsschicht für die Softwarekomponenten bietet. Optional werden auch Ressourcen, die von verschiedenen Softwarekomponenten genutzt werden, durch das Betriebssystem verwaltet.

KNOTEN: Eine physikalische Komponente des Systems, welche einen Prozessor und entsprechende Peripherie – inklusive Anbindung an das Kommunikationssystem – besitzt, um Software auszuführen. An einen Knoten können Sensoren angeschlossen sein, deren Werte von einer Softwarekomponente ausgelesen werden, oder Aktuatoren, die von einer Softwarekomponente angesteuert werden. Ein Beispiel für einen Knoten mit entsprechender Software ist in Abbildung 3 dargestellt.

KOMMUNIKATIONSSYSTEM: Ein verteiltes System benötigt zur Koordinierung und Synchronisierung der verteilten Knoten eine Möglichkeit, Informationen zwischen Knoten auszutauschen. Da Kommunikationssysteme in verteilten Systemen eine so wesentliche Rolle innehaben, werden in Kapitel 2.4 deren Eigenschaften genauer dargelegt.

MIDDLEWARE: Eine Abstrahierungsschicht, die auf ein Betriebssystem aufbaut und für die Softwarekomponenten weitere Heterogenität verbirgt. Beispiele für Heterogenität, die durch die Middleware verborgen können, sind unterschiedliche Prozessorarchitekturen sowie die Verteilung der Softwarekomponenten auf die Knoten.

PROGRAMM: Eine Sammlung von Anweisungen für einen Prozessor, um eine definierte Funktion zu erfüllen.

SENSOR: Ein physikalisches Element an einer Schnittstelle zwischen Umwelt und System, welches auf definierte Umwelteinflüsse reagiert. Beispiele sind Thermometer, Schalter und Lichtschranken. Ein Sensor ist an einen Knoten angeschlossen und wird durch ein oder mehrere Programme ausgelesen.

SOFTWARE: Die Gesamtheit der auf einem Knoten ausgeführten Programme.

SOFTWAREKOMPONENTE: Ein Programm, welches einen Dienst implementiert, den das System zur Bereitstellung einer Anwendung benötigt.

2.2 ANGREIFER

Als Angreifer wird eine Person oder eine Gruppe von Personen bezeichnet, die unbefugt versuchen, mit einem System zu interagieren. Die geplante Interaktion läuft dabei typischerweise den Interessen des Eigentümers und Betreibers entgegen (z. B. Spionage, Sabotage). [135, 109, 51, 47]

Ein Angreifer kann dabei auf jede ihm mögliche Art und Weise mit dem System interagieren. Daher sind auch mechanische Eingriffe möglich, falls ein physikalischer Zugang gegeben ist. Die angenommenen möglichen Interaktionen mit dem System bilden das sogenannte Angreiferverhaltensmodell. Das Angreiferverhaltensmodell dokumentiert dabei alle Annahmen, sowohl die für den Angreifer nutzbaren Schnittstellen mit der Umwelt als auch zusätzliche Annahmen wie etwa seine Kompetenzen oder Limitierungen (z. B. Angriffsversuche pro Zeiteinheit).

Möglichkeiten eines Angreifers

Unabhängig von dem konkreten Angreiferverhaltensmodell werden im Folgenden die wichtigsten Möglichkeiten eines Angreifers geschildert, Einfluss auf ein System zu gewinnen.

INFORMATIONSBESCHAFFUNG UND REVERSE ENGINEERING Ein Angreifer kann alle für ihn verfügbaren Quellen dazu nutzen, um ein möglichst detailliertes Bild von dem Aufbau und Zustand des anzugreifenden Systems zu bekommen. Zu möglichen Quellen gehören öffentlich verfügbare Dokumentationen[1] und die Quellen, die sich im Rahmen des Angriffs erschließen. Im Rahmen des Angriffs kann ein Angreifer unter Umständen auch an interne Dokumentation des Systems kommen, Datenverkehr mitschneiden oder Zugriff auf einzelne Software- oder Hardwarekomponenten erlangen.

Falls die dabei gewonnenen Informationen das eigentliche Ziel des Angreifers waren, ist der Angriff – sofern keine ungewollten Seiteneffekte bei der Informationsbeschaffung auftreten – vom Standpunkt der funktionalen Sicherheit nicht relevant, da keine kritischen Betriebszustände resultieren. Der Angreifer kann aber auch die gewonnenen Informationen auswerten, um ein Angriffsvorgehen zu finden, bei dem die Sicherungsmechanismen der funktionalen Sicherheit fehlschlagen.

STÖREN/ZERSTÖREN VON KOMPONENTEN Ein Angreifer kann Komponenten eines Systems temporär stören oder sogar permanent zerstören. Durch gezieltes Stören von Komponenten kann ein Angrei-

---

[1] Beispielsweise in Form von Pressetexten, „Case Studies“ von Zulieferer oder Ausschreibungstexte für Ergänzungen/Reparaturen.

fer verhindern, dass das System auf bestimmte Ereignisse wie spezifiziert reagiert. Bei einem System, dass vom Standpunkt der funktionalen Sicherheit fehlertolerierend sein muss, kann ein Ausbleiben einer Reaktion zu einer Verletzung der Sicherheitsziele führen.

Mögliche Angriffsvorgehen dazu sind physikalische Krafteinwirkungen bei vorhandenen Zugangsmöglichkeit, Dienstblockadeangriffe (siehe auch Kapitel 2.4.2) oder ein Angriff auf eine Softwarekomponente mit einem Exploit (siehe Kapitel 2.5), der diese Komponente blockiert/abstürzen lässt.

ÄNDERN DES VERHALTENS VON KOMPONENTEN Ein Angreifer kann das Verhalten einer Komponente des Systems temporär oder permanent modifizieren. Falls die Komponente sicherheitsrelevante Aktuatoren ansteuert, können durch den Angriff damit Sicherheitsziele verletzt werden.

Mögliche Angriffsvorgehen dazu sind physikalische Veränderungen[2] bei vorhandenen Zugangsmöglichkeit, Modifikation einer Softwarekomponente oder das Fälschen relevanter Eingangssignale. Für eine Modifikation einer Softwarekomponente kann sowohl physikalischer Zugang genutzt werden[3] als auch ein Exploit (siehe Kapitel 2.5) der die Software zur Laufzeit manipuliert. Bei der Fälschung von Eingabesignale (meist mittels Maskerade, siehe Kapitel 2.4.2.2) kann der Komponente ein von der Realität abweichender Systemzustand präsentiert werden. Die Reaktionen erfolgen dann auf Basis des vorgetäuschten Systemzustands und können damit für den aktuellen realen Betriebszustand außerhalb des spezifizierten und funktional sicheren Rahmens liegen. Da bei der Fälschung von Eingabesignalen das Verhalten nur indirekt beeinflusst wird, ist zur gezielten Provokation von Reaktionen ein hohes Domänenwisssen und Reverse Engineering notwendig [68].

PROVOZIEREN VON „UNWAHRSCHEINLICHEN FEHLERBILDERN" Ein Angreifer kann versuchen Fehlerbilder zu erzeugen, die durch zufällige stochastisch unabhängige Fehler eine so geringe Auftrittsrate aufweisen, dass für die funktionale Sicherheit keine weiteren Gegenmaßnahmen getroffen wurden [65]. Da in diesem Fall das System außerhalb der Spezifikation operiert, kann dadurch eine Verletzung der Sicherheitsziele erfolgen.

Die möglichen Angriffsvorgehen sind sehr systemspezifisch. Ein einfaches Beispiele ist jedoch Maskerade bei Verwendung von Ende zu Ende Sicherungsprotokollen wie AUTomotive Open System ARchitecture (AUTOSAR) Profil 2. Bei diesen wird angenommen, dass ein zufälliger Fehler nicht mehrere aufeinanderfolgende gültige Nach-

---

[2] zum Beispiel wird der Aktuator durch Vertauschen der Pole invertiert angesteuert

[3] Austausch der Festplatte/ROM, Einspielen von Schadsoftware über Serviceschnittstelle

richten erzeugen kann. Ein Angreifer ist jedoch durch Mitschneiden des Datenverkehrs in der Lage, die verwendeten Sicherheitsmechanismen so zu rekonstruieren, so dass er beliebig viele gültige Nachrichten erzeugen kann.

BEEINFLUSSUNG VON MENSCHEN Ein Angreifer kann versuchen durch gezielte Manipulation (im Englischen „social engineering“) Personal, das mit dem System interagiert (wie Techniker), zu überzeugen, dass eine bestimmte Handlung vorgenommen werden muss, die im Interesse des Angreifers ist. Diese Art von Angriffen ist nicht im Fokus dieser Arbeit, wenngleich es auch ein großes Sicherheitsproblem darstellt [152, 111, 61].

2.3 FEHLERTOLERANTE SYSTEME

2.3.1 *Allgemein*

Ein fehlertolerantes System ist dadurch charakterisiert, dass das Verhalten des Systems auch in Anwesenheit von einer begrenzten Anzahl von Fehlern definiert bleibt. [40, 95, 48, 97] Die Art und Anzahl der Fehler, die toleriert werden müssen, das heißt, bei denen das Systemverhalten definiert bleibt, wird in dem sogenannten Fehlermodell definiert.

Die Vielzahl der bisher entwickelten Verfahren, lassen sich durch die im Folgenden beschrieben Eigenschaften kategorisieren:

- Genutzte Struktur und Ressourcen
- Die Art der Fehlererkennung
- Verfügbarkeit im Fehlerfall und Reaktion auf einen erkannten Fehler

2.3.2 *Struktur*

Bei der Struktur werden meistens zwei Klassen von Systemen unterschieden. [48] Statisch redundante Systeme, bei denen dauerhaft redundante Komponenten für die Fehlertoleranz genutzt werden und dynamisch redundante Systeme, die im Fehlerfall ihre Konfiguration verändern. Dies kann zum Beispiel bedeuten, dass im Fehlerfall Komponenten oder Ressourcen genutzt werden, die normalerweise andere nicht sicherheitsrelevante Aufgaben ausführen.

Es gibt jedoch auch hybride Systeme, die sowohl statische als auch dynamische Redundanz nutzen.

2.3.3 *Fehler*

Fehler lassen sich auch anhand ihres Verhaltes charakterisieren. Dabei werden oftmals drei Eigenschaften unterschieden, das Verhalten, die Dauer und das Auftreten.

Das Verhalten einer (Teil-)Komponente in der Anwesenheit eines Fehlers kann beliebig sein, aber auch auf bestimmte Verhaltensweisen beschränkt sein (z.B. beim sogenannten Anhalteausfall liefert eine Komponente entweder rechtzeitig einen korrekten Wert oder keinen).

Bei der Dauer eines Fehlers wird unterschieden, ob er ab dem ersten Auftreten dauerhaft aktiv ist (permanenter Fehler) oder ohne externe Intervention wieder verschwindet (temporärer Fehler).

Bei dem Auftrittsverhalten unterscheidet man zufälliges, stochastisches Auftreten, zum Beispiel bei Alterungseffekten, externen Störeinflüssen oder mangelnder Synchronisierung von nebenläufiger Software, von Fehlern mit deterministischen Auslösern, wie Fehler in der

Programmlogik. Diese werden auch als systematische Fehler bezeichnet.

2.3.4 *Fehlererkennung*

Für die Erkennung/Diagnose von Fehlern werden zwei Klassen von Tests verwendet.

2.3.4.1 *Relativtests*

Bei den Relativtests [48] werden im Prüfprozess m unabhängig voneinander ermittelte Ergebnisse miteinander verglichen um einen Fehler festzustellen. Bei deterministischen Prozessen reichen zum Erkennen eines fehlerhaften Zustands bei f aktiven Fehlern $f + 1$ Ergebnisse. Bei indeterministischen Prozessen ist dies nur mit zusätzlichen Randbedingungen möglich (z. B. Kenntnis der erlaubten maximalen Abweichung von gültigen Ergebnissen).

Da aufgrund der Fehlerannahme im Fehlermodell mindestens eines der $f + 1$ verglichenen Ergebnisse fehlerfrei ist, ist die Fehlererfassung bei Relativtests sehr gut. Bei deterministischen Prozessen wird garantiert, dass der Fehler erkannt wird oder die fehlerhaften Komponenten das gültige Ergebnis geliefert haben. Bei indeterministischen Prozessen wird garantiert, dass der Fehler erkannt wird oder die fehlerhaften Komponenten Ergebnisse geliefert haben, die in dem Intervall der erlaubten Abweichung um gültige Ergebnisse liegen.

Falls die fehlerfreien Ergebnisse stets in der Mehrheit sind ($m = 2 * f + 1$), dann kann der Relativtest nicht nur einen Fehler erkennen, sondern auch stets ein gültiges Ergebnis zurück liefern (siehe auch Kapitel 2.3.5).

2.3.4.2 *Absoluttests*

Bei den Absoluttests [48] wird im Prüfprozess nur ein Ergebnis auf Plausibilität geprüft. Die Prüfung soll im Vergleich zur Berechnung des zu prüfenden Wertes nur wenige Ressourcen benötigen um den Aufwand im Vergleich zu den mehrfach berechneten Ergebnissen bei einem Relativtest zu senken. Die Prüfung des Absoluttests kann dabei nur auf den Ausgabewerten operieren oder zusätzlich auch auf weitere Eingabewerte zurückgreifen.

Der erzielte Grad der Fehlererkennung von Absoluttests wird stark von der jeweiligen Anwendung und dem Fehlermodell bestimmt. Sehr gute Fehlererkennung lässt sich zum Beispiel mittels eines Cyclic Redundancy Check (CRC) (eine Form der statischen Informationsredundanz) unter der Annahme von zufälligen Verfälschungen erreichen. Auch geeignet sind Sensorwerte, da ein gültiger Wertebereich oftmals bestimmt werden kann und auch der maximale Gradient der Werteänderung durch den zugrunde liegenden Prozess bekannt ist.

2.3.5 *Verfügbarkeit im Fehlerfall und Reaktion auf einen Fehler*

Bei der Verfügbarkeit im Fehlerfall und Reaktion auf einen Fehler lassen sich zwei Klassen von Systemen unterschieden: Selbstsichernde Systeme (engl. fail safe) und fehlertolerierende Systeme (engl. fail operational).

Selbstsichernde Systeme haben für alle erkannten Fehler die gleiche Reaktion spezifiziert: Sofortiges Aufsuchen des sogenannten sicheren Zustands, in dem von dem System keine Risiken mehr ausgehen, die nicht akzeptabel sind. Das System wird dadurch inaktiviert.

Dies ist jedoch nur realisierbar, wenn der Übergang in den sicheren Zustand jederzeit möglich ist. Beispiele finden sich unter anderen im Eisenbahnverkehr (Ein Fehler in der Zugsteuerung führt zum Bremsen bis zum Stillstand; ein Fehler in der Stellwerksteuerung führt dazu, alle Signale auf Halt zu stellen) oder in der Automatisierungstechnik (Fehler in der Steuerung halten die Produktion an).

Ist das sofortige Aufsuchen des sicheren Zustands unmöglich (zum Beispiel bei einem Flugzeug während des Flugs oder in einem Atomkraftwerk, bei dem zumindest Teilsysteme weiter funktionieren müssen, um die Nachzerfallswärme unter Kontrolle zu halten) oder aus wirtschaftlichen Gründen zu vermeiden (Ein Ausfall eines wichtigen Stellwerks führt zu großflächigen Störungen im Bahnverkehr), so muss das System fehlertolerierend sein. Fehlertolerierend bedeutet, dass bei dem Auftreten der im Fehlermodell definierten Fehler die für den Nutzer relevanten Funktionen weiter erbracht werden. Dazu wird bei den fehlertolerierenden Systemen auf einen erkannten Fehler mittels Fehlermaskierung oder Rekonfiguration reagiert.

Bei der Fehlermaskierung wird das fehlerhafte Ergebnis verworfen und ein fehlerfreies Ergebnis weitergeleitet. Dazu muss die (Software-)Komponente, die das Ergebnis auswählt (häufig Maskierer genannt), zwischen mehreren Werten auswählen können, was wiederum statische Redundanz bedingt mit dem damit verbundenen hohen Aufwand. Die einfachste Implementierung sind Mehrheitsentscheider, bei denen so viele Werte anliegen müssen, dass die fehlerhaften Werte in der Minderheit sind.

Bei der Rekonfiguration – stets dynamische Redundanz – wird nach einem erkannten Fehler der Ablauf für die Dauer der Fehlerbehandlung unterbrochen. In diesem Zeitraum wird das System wieder in einem funktionsfähigen Zustand versetzt. Dies kann durch erneute Ausführung der beteiligten Softwarekomponenten mit den internen Parametern von einem als fehlerfrei angenommenen Sicherungspunkt (sog. Rückwärtsbehebung) oder durch die Wahl eines sicheren aber unter Umständen nicht optimalen Ersatzwertes, bis die Anwendung wieder die Funktion übernimmt (sogenannte Vorwärtsbehebung). Falls temporäre Fehler angenommen werden, kann die Fehlerbehebung auf denselben Knoten erfolgen, die bereits für die

Anwendung benutzt werden. Falls dagegen permanente Fehler angenommen werden, muss bei der Rekonfiguration ein anderer Knoten gewählt werden. Die Rekonfiguration benötigt Zeitredundanz, ist daher oftmals für Systeme mit harten Echtzeitanforderungen nicht geeignet.

2.3.6 *Annahmen in vielen verwendeten Designs*

Es gibt zwei wesentliche Annahmen, die in vielen fehlertoleranten System getroffen werden und die ein Angreifer (siehe Kapitel 2.2) ausnutzen kann. Die erste Annahme ist das stochastisch unabhängige Versagen von Einzelfehlerbereichen. Die zweite Annahme ist meist, dass Fehler zu zufälligem Verhalten führen und nicht gezielt Sicherungsmaßnahmen unterlaufen.

Zu Einzelfehlerbereichen werden beim Entwurf eines fehlertoleranten Systems alle Komponenten zusammengefasst, die durch einen Fehler gemeinsam ausfallen dürfen. Es wird angenommen, dass zeitgleiches Versagen von mehr als f Einzelfehlerbereichen ausreichend unwahrscheinlich ist, da jeder Einzelfehlerbereich für sich gemäß einem stochastischen Modell ausfällt und es keinen gemeinsame Fehler gibt.

Für die Annahme, dass Fehler nicht gezielt versuchen, Sicherungsmaßnahmen zu überwinden, lassen sich viele Beispiele finden. Besonders eignen sich verschiedenste Kommunikationsprotokolle wie die in AUTOSAR Profilen festgelegten Protokolle [10]. Die verwendeten Sicherungsmechanismen bieten gegen stochastische Fehler einen ausreichenden Schutz, lassen sich aber von einen intelligenten Angreifer leicht überwinden, da sie selbst mit der beschränkten Rechenkapazität eines Steuergeräts leicht per Brute-Force zu brechen sind.

2.4 KOMMUNIKATIONSSYSTEME

2.4.1 *Grundlagen*

Kommunikationssysteme bieten die Möglichkeit, zwischen zwei oder mehreren Knoten digital kodierte Daten auszutauschen. Ein Kommunikationssystem besteht dabei aus Hardware, die Verbindungen realisiert, und Kommunikationsprotokollen, die sowohl die Nutzung regeln als auch eine gewisse Dienstgüte für die darüber liegende Protokollschicht oder Anwendung realisieren. Ein in der Literatur sehr verbreitetes Model zur Beschreibung ist das aus sieben Ebenen bestehende OSI-Referenzmodel [44]. In der Praxis hat sich für viele Kommunikationssysteme und darauf aufbauende Anwendungen eine Struktur ergeben, die dem vierschichtigen DoD-Model [33] entspricht, das die Schichten eins und zwei sowie die Schichten fünf bis sieben zusammenfasst.

Im Rahmen dieser Arbeit werden Kommunikationssysteme in drei Klassen eingeteilt, die sich an der Topologie der Verbindungen zwischen den Knoten orientiert: Punkt zu Punkt Verbindungen, Bussysteme/Broadcast Kommunikation und paketvermittelte Verbindungen. Diese Einteilung erfolgt, da die sich daraus ergebenden Kommunikationssysteme charakteristische Eigenschaften aufweisen, die für die Angriffsmöglichkeiten (siehe Kapitel 2.4.2) von großer Relevanz sind.

PUNKT ZU PUNKT VERBINDUNGEN Bei den Punkt zu Punkt Verbindungen werden zwei Knoten direkt miteinander verbunden, ohne dass dritte Knoten involviert sind oder die Nachricht mitlesen können. Im DoD-Model fehlt dabei die Schicht 2 („Internet“), die es erlaubt, dass weitere Knoten erreicht werden können. Ein bekanntes Beispiel ist eine serielle Schnittstelle [16]. Eine Punkt zu Punkt Verbindung kann jedoch auch mit den Techniken, die typischerweise für Bussysteme oder paketvermittelte Verbindungen (wie zum Beispiel Ethernet [82]) betrieben werden, indem man die Zahl der angeschlossenen Knoten auf zwei beschränkt.

Dadurch, dass es keine Einflüsse von Dritten gibt, ist das zeitliche Verhalten der Verbindung sehr deterministisch, die Reihenfolge der gesendeten Daten wird nicht geändert und die Daten sind eindeutig einem Sender zuzuordnen. Daher fällt die Schicht 3 („Host-to-Host“) im DoD-Model meist sehr minimalistisch aus und besteht nur aus einer Integritätssicherung gegen Verfälschungen.

BROADCAST KOMMUNIKATION Die Kommunikation über Broadcast Kommunikationssysteme zeichnet sich nach [46] dadurch aus, dass ein gemeinsames Medium von einer größeren Anzahl von Knoten und Kommunikationsbeziehungen genutzt wird. Als gemeinsames Medium kann dabei sowohl ein Kabel als auch ein Funkkanal dienen. Bekannte Beispiele für Kabel sind CAN [132] und ZigBee [160] für Funk. Ein Signal, das von einem Knoten auf dem gemeinsamen Medium gesendet wird, wird von allen beteiligten Knoten empfangen. Da störungsfrei jeweils nur ein Knoten senden kann, werden Techniken benötigt, die festlegen welcher Teilnehmer zu welchem Zeitpunkt senden darf. Zur Unterscheidung der verschiedenen Nachrichten werden in der Schicht 1 des DoD-Models Identifikationsmerkmale genutzt, aber wie bei der Punkt zu Punkt Verbindung fällt typischerweise die Schicht 2 weg.

Durch das geteilte Kommunikationsmedium können sich verschiedene Kommunikationsströme gegenseitig beeinflussen. Das betrifft neben den auch bei den Punkt zu Punkt Verbindungen möglichen Verfälschungen auch das Zeitverhalten[4] und die Möglichkeit der Fehlleitung von Nachrichten (Maskerade). Daher fällt die Schicht 3 im DoD-Model meist schon etwas komplexer aus, um diese Fehlerbilder auch zu erkennen. Beispiele sind die AUTOSAR Kommunikationsprofile [10].

PAKETVERMITTELTE KOMMUNIKATION Bei der paketvermittelten Kommunikation teilen sich Sender und Empfänger (im Regelfall) kein physikalisches Medium, sondern die Kommunikation läuft über weitere Knoten, die jeweils die Daten auf das nächste zum Empfänger führende Kommunikationsmedium vermitteln. Die beteiligten Kommunikationsmedien können dabei jeweils Punkt zu Punkt Verbindungen als auch Broadcast Verbindungen sein. Da zwischen verschiedenen Kommunikationssystemen vermittelt wird, ist eine eindeutige globale Identifikation der Empfänger notwendig, die durch ein Schicht 2 Protokoll bereitgestellt wird. Bekanntestes Beispiel ist dafür das Internetprotokoll (IP) [124]. In jeden vermittelnden Knoten wird anhand von Topologieinformationen und der Schicht 2 Adresse für jedes Paket das darauf folgende Kommunikationsmedium bestimmt.

Dadurch, dass die digital kodierten Daten in Pakete zerlegt werden, die unabhängig voneinander durch das Netzwerk geleitet werden, ergibt sich neben den Fehlern bei der Broadcast Kommunikation zusätzlich das Phänomen, das die Reihenfolge der gesendeten Daten nicht mit der Reihenfolge der empfangenen Daten übereinstimmen muss, wobei auch Daten verloren gehen können und nicht nur verfälscht werden. Die Schicht 3 Protokolle müssen dieses Verhalten entsprechend berücksichtigen. Bekannte Beispiele für Schicht 3 Protokolle im DoD-Model sind UDP [124], Transmission Control Protocol (TCP) [32] und SCTP [142].

---

[4] Einzelne Kommunikationssysteme wie FlexRay [57] garantieren über Zeitfensterverfahren deterministische Transferdauern solange bestimmte Fehlerbilder ausgeschlossen werden.

2.4.2 *Angriffsmöglichkeiten*

Sobald ein Dritter Zugriff auf einen Kommunikationskanal erlangt hat[5], besteht die Möglichkeit diesen Zugang zu nutzen, um gezielt Fehlersituationen zu erzeugen. In der Literatur [75, 104, 127, 55, 56] werden dabei in Netzwerken insbesondere die folgenden drei Kategorien von Angriffen unterschieden:

2.4.2.1 *Dienstblockade*

Bei der Dienstblockade (im englischen Denial of Service oder als Abkürzung DoS) wird die Kommunikation zwischen den angegriffenen Knoten gestört. Im engeren Sinne wird von der Dienstblockade gesprochen, wenn die Kommunikation vollständig unterbunden wird. Im Regelfall wird jedoch auch von einer Dienstblockade gesprochen, wenn es den Angreifer gelingt, für die Kommunikation relevante Qualitätsparameter wie Paketverlustwahrscheinlichkeit, Übertragungsdauer und Jitter so negativ zu beeinflussen, dass es eine signifikante Auswirkung auf die darüber liegende Anwendung hat.

Die Anwendung kann aufgrund der Störung nicht mehr, oder nicht in den spezifizierten Zeitrahmen auf Änderungen reagieren, die der Sender dem Empfänger übermittelt.

BEISPIELE: Insbesondere bei Funknetzwerken und Bussystemen kann das Kommunikationsmedium direkt gestört werden, indem Störsignale auf das Medium gesendet werden [154, 123]. Sehr viele Kommunikationssysteme lassen sich auch durch Einfügung (siehe folgenden Abschnitt) von Steuernachrichten stören, die zu anderen Zwecken im Protokoll enthalten sind [151, 147, 69]. Eine weitere Möglichkeit der Dienstblockade ist die Überlastung des Kommunikationssystems mit extra für diesen Zweck generierten Nachrichten. Bei CAN [132] kann ein Angreifer hochpriore Nachrichten senden, die jeglichen anderen Verkehr verdrängen [84]. Alternativ sendet der Angreifer so viele Daten, dass an einem Flaschenhals im Netzwerk so viele Daten verworfen werden müssen, dass auch die legitimen Nachrichten inakzeptable Verlustraten erleiden. Beispiele sind dafür im Internet ICMP oder UDP Flutungsangriffe [1].

2.4.2.2 *Nachrichteneinfügung*

Bei der Nachrichteneinfügung (auch Maskerade genannt) wird vom Angreifer eine Nachricht an einen Empfänger gesandt, die vorgibt von einem gültigen Absender zu stammen. Dazu wird typischerweise das Identifikationsmerkmal, das den Nachrichteninhalt und/oder den Absender identifiziert, gefälscht.

---

[5] Bei Punkt zu Punkt Verbinden per Definition ausgeschlossen, sofern kein physikalischer Zugang gegeben ist.

Der Empfänger führt daraufhin die Aktionen aus, die bei einem Empfang der Daten vom vorgesehenen Absender erfolgen würden. Insbesondere in Steuerungssystemen kann ein Angreifer damit den Empfänger zu einer im aktuellen Zustand unpassenden Reaktion provozieren.

BEISPIELE: Neben den schon bei der Dienstblockade genannten Beispielen in der Verwaltung eines Kommunikationssystems gibt es sehr viele anwendungsspezifische Möglichkeiten. Beispielsweise kann versucht werden, in Steuerungssystemen unberechtigt Zugriff auf Aktuatoren zu nehmen [94].

2.4.2.3 *MITM-Angriff*

Bei dem man in the middle (MITM)-Angriff – selten auch Janusangriff oder Mittelsmannangriff genannt – manipuliert der Angreifer das Kommunikationssystem logisch oder physikalisch, so dass der Datenverkehr über den Angreifer geleitet wird. Dadurch kann der Angreifer die Daten, die zwischen zwei Teilnehmern ausgetauscht werden, mitlesen und insbesondere die übertragenen Daten manipulieren. Der Angreifer täuscht dabei jeden Teilnehmer vor, der jeweils andere Teilnehmer zu sein, teilweise mit Informationen, die er kurz zuvor vom anderen Teilnehmer erhalten hat. Aufgrund der direkten Verbindung bei Punkt zu Punkt Verbindungen und Broadcast Kommunikationssystemen ist der Angriff nur bei paketvermitelter Kommunikation möglich.

BEISPIELE: Der eigentliche Angriff [63] ist jeweils sehr anwendungsspezifisch (wie „SSL stripping“ [120]). Dazu wird im Regelfall auf einer der niedrigeren Kommunikationsschichten mittels Einfügung von Steuernachrichten [78, 105] der Verkehr manipuliert.

2.4.3 *Gegenmaßnahmen*

Gegenmaßnahmen gegen einen Angreifer lassen sich sowohl im Kommunikationssystem als auch durch Zusatzinformationen in der Nachricht die von Sender zu Empfänger transportiert wird realisiert werden. Letzteres wird im Allgemeinen als Ende-zu-Ende Absicherung bezeichnet.

2.4.3.1 *Im Kommunikationssystem*

Bei den Gegenmaßnahmen im Bereich des Kommunikationssystems sind die Gegenmaßnahmen rein konzeptuell identisch: Eine von dem zu kontrollierenden Knoten ausreichend[6] unabhängiger Knoten über-

---

[6] Bei der Annahme von zufälligen Hardwarefehlern wäre z. B. die Übermittlung einer Konfiguration, die durch eine Prüfsumme geschützt wird, als ausreichend anzuse-

wacht dessen Kommunikationsverhalten und unterbindet den weiteren Zugang zum Kommunikationsmedium wenn der Knoten Fehlverhalten zeigt. Die kontrollierenden Knoten können dezentral bei den sendenden Knoten angeordnet sein – häufig bei Broadcast Kommunikationssystemen vorgeschlagen unter den Begriff „Guardian" [26, 28, 57] – oder bei der paketvermittelten Kommunikation an den vermittelnden Knoten.

Diese Gegenmaßnahme ist im Bereich der Automatisierungstechnik oftmals durch die funktionale Sicherheit motiviert und damit ist oft nur eine fehlerbedingte Dienstblockade des Mediums (im englischen „babbling idiot") bei diesen Konzepten im Fokus. Weitergehende Sicherungsmaßnahmen gegen Nachrichteneinfügungen oder Dienstblockaden durch gezielten Missbrauch regulär genutzter Steuernachrichten des Kommunikationssystems werden nur in traditionell offenen paketvermittelten Kommunikationssystemen angeboten, und auch dort nur optional [7]. Diese sind auch nur dann gut umsetzbar, wenn für die kontrollierenden Knoten im Vorfeld die Information zur Verfügung gestellt werden kann, welcher Knoten welche Informationen senden darf. Dies ist insbesondere in dynamischen ad hoc Netzen problematisch.

2.4.3.2 *Ende-zu-Ende Absicherung*

Die Ende-zu-Ende Absicherung bietet den großen Vorteil, dass die Komponenten die zur Realisierung genutzt werden (Sender und Empfänger) dieselben sind, die auch die Funktion für die Anwendung erbringen. Dies ist für die funktionale Sicherheit vorteilhaft, da so die Einzelfehlerbereiche nicht vergrößert oder geändert werden müssen. Auch für den Schutz gegen Angreifer ist die Ende-zu-Ende Absicherung vorteilhaft, da keine Annahmen notwendig sind, an welcher Stelle im Kommunikationssystem ein Angreifer fehlerhafte Nachrichten einfügen kann.

Genutzte Techniken sind dabei Signaturen in verschiedenen Ausprägungen, um Nachrichtenverfälschungen und Nachrichteneinfügungen zu erkennen, sowie Zähler, um Vertauschungen von Nachrichten sowie Nachrichtenwiederholungen zu erkennen [10, 121, 130, 92].

Alle Ende-zu-Ende Techniken haben jedoch gemeinsam, dass sie keinen Schutz vor einer Dienstblockade auf dem Kommunikationssystem bieten können, da sie nur beim Empfang von Nachrichten geprüft werden können. Sie können teilweise sogar im Rahmen einer Dienstblockade im Sinne des Angreifers genutzt werden, da die Prüfroutinen, die bei einem Empfang der Nachricht ausgeführt werden, Ressourcen benötigen. Dadurch kann der Empfänger mit einer Vielzahl von eingefügten Nachrichten Ressourcen beim Empfänger belegen.

---

hen, bei einem Angreifer nicht, falls dieser mit vertretbaren Aufwand die Prüfsumme fälschen kann.

2.5 SOFTWARE

In diesem Kapitel werden die Schwachstellen von Software behandelt, die ein Angreifer ausnutzen kann, die Techniken um die Wahrscheinlichkeit für (gemeinsame) Schwachstellen zu reduzieren sowie Techniken zum Erkennen von bereits erfolgten Angriffen.

2.5.1 *Grundlagen*

Fehler in einer Softwarekomponente können an verschiedenen Punkten der Entwicklung hervorgerufen werden. Angefangen bei Fehlern in der Systemspezifikation, über Fehler im Softwareentwurf und bei der eigentlichen Softwareentwicklung bis zu Fehlern in verwendeten Tools (Bibliotheken, Compiler, Laufzeitumgebung) [13]. Da Software keiner Alterung[7] und auch keinen direkten Umwelteinflüssen[8] unterliegt, sind Softwarefehler per Definition systematische Fehler, das heißt, sie können (prinzipiell) jederzeit reproduziert werden. In der Literatur [70, 8, 45] wird dabei teilweise noch unterschieden zwischen Fehlern, die leicht zu reproduzieren sind, und Fehler, die nur in sehr bestimmten Zuständen sich reproduzieren lassen. Als systematischer Fehler betrifft ein Softwarefehler alle (identischen) Instanzen einer Softwarekomponente Ein Einzelfehlerbereich muss diese daher alle enthalten, jedoch wird dieser Einzelfehlerbereich oftmals nicht betrachtet (siehe Kapitel 2.3.6).

Ein Softwarefehler wird oftmals auch als „Bug“ bezeichnet. Falls ein Angreifer in der Lage ist, die Bedingungen zu beeinflussen, die den Softwarefehler auslösen, so kann er diesen ausnutzen. Die dazu verwendeten Schritte/Befehle oder Daten werden oftmals auch als „Exploit“ bezeichnet. In den beiden folgenden Kapiteln werden die beiden Gruppen von Softwarefehlern erläutert, die sich am häufigsten für Angriffe ausnutzen lassen. Die damit möglichen Angriffe können dabei von Störungen des Ablaufs („Denial of Service“) bis zur beliebigen Manipulation des Verhaltens der Softwarekomponenten reichen.

2.5.2 *Designfehler in Softwaresystemen*

2.5.2.1 *Problematik*

Beim Entwurf des Softwaresystems werden (oftmals implizite) unzutreffende Annahmen getroffen oder Auswirkungen einer Funktion auf andere Funktionen nicht vollständig untersucht. Diese Fehler wer-

---

[7] Teilweise wird in der Literatur von Alterung bei Software gesprochen, dabei ist aber kein „Verschleiß“ wie bei Hardware gemeint, sondern die oftmals sinkende Codequalität, nachdem sehr viele Änderungen nachträglich vorgenommen wurden.

[8] Durch Umwelteinflüsse hervorgerufene Hardwarefehler können natürlich Software beeinflussen.

den dann in die Spezifikation übernommen. Dadurch gibt es auch bei vollständig spezifikationskonformen Implementierung Schwachstellen, die ein Angreifer ausnutzen kann.

Designfehler gelten als besonders problematische Fehler, da sie oftmals nur mit extrem hohen Aufwand beseitigt werden können. Man denke an Fehler in genormten Protokollen [4], wo dann nicht nur eine Norm geändert werden muss, sondern auch sämtliche Implementierungen dieser Norm. Besonders gefährdet für diese Art von Fehlern sind Systeme mit einer hohen Lebensdauer, da sich die Umweltbedingungen im Laufe der Zeit stark wandeln können und damit implizite (oder teilweise auch explizite) Annahmen zum Entwurfszeitpunkt nicht mehr gültig sind.

2.5.2.2 *Beispiele*

- Im Produkt noch gut erreichbare Schnittstellen zur Fehlersuche während der Entwicklung. Diese Funktion war während der Entwicklung notwendig, aber die Sicherheitsimplikationen im Betrieb wurden nicht ausreichend evaluiert. Beispiele sind leicht zugängliche JTAG-Schnittstellen [76], über die problemlos beliebige Speicherinhalte gelesen und manipuliert werden können, oder auch versteckte Benutzeraccounts mit festem Passwort [118, 83], oder übers Netzwerk zugängliche Debug-Schnittstellen [35].

- Fehlende Authentifizierung von Sender und Empfänger im Netzwerk. Das ermöglicht einem Angreifer, sobald er Zugriff auf das Netzwerk hat, einzelne oder alle Funktionen zu nutzen und die Funktionen (wie zum Beispiel zum Ändern von Parametern oder Aktualisieren von Softwarekomponenten) selber auszuführen [159, 4]. Die fehlende Authentifizierung ist teilweise darin begründet, dass zum Entwurfszeitraum die entsprechenden Systeme vollständig isoliert waren und keine Schnittstellen mit öffentlichen Netzen hatten.

2.5.2.3 *Gegenmaßnahmen*

Die Gegenmaßnahmen gegen Sicherheitslücken durch Designfehler kann man grob in zwei Kategorien einteilen, zum einen die Fehlervermeidung und zum anderen in „prophylaktische Sicherung“.

Bei der Fehlervermeidung bieten sich Review-Verfahren, gezielte Sicherheitsuntersuchungen der Spezifikation oder auch diversitärer Spezifikationsentwurf mit Vergleich zum Aufdecken von Unterschieden und damit potentiellen Fehlern an. Bei der prophylaktischen Sicherung wird versucht, bei jeder Schnittstelle anzunehmen, dass ein Angreifer vorhanden sein kann. Entsprechend werden alle Eingabedaten validiert und Kommunikationsteilnehmer authentifiziert.

2.5.3 *Schwachstellen in Softwarekomponenten*

2.5.3.1 *Problematik*

Schwachstellen in Softwarekomponenten sind Folgen von Anweisungen, die zu einem von der Spezifikation abweichenden Verhalten führen können. Es handelt sich damit um Implementierungsfehler. Die möglichen Abweichungen reichen von leichten Abweichungen von der Spezifikation in Randfällen bis hin zu beliebigem (Fehl-)Verhalten.

Häufige Ursachen für Schwachstellen sind Fehlinterpretationen der Spezifikation, fehlerhafte Annahmen über Eingabedaten und menschliche „Flüchtigkeitsfehler“[9]. Da Schwachstellen – anders als Designfehler – Abweichungen von dem Verhalten implizieren, das die restlichen Komponenten des Systems erwarten, lassen sich Schwachstellen in einer Softwarekomponente in der Regel ohne Eingriffe in andere Komponenten beheben. Daher sind Schwachstellen – sofern erst einmal identifiziert – in der Regel relativ leicht auszubessern.

2.5.3.2 *Beispiele*

Entsprechend dem Fokus der Gesamtarbeit werden Schwachstellen betrachtet, die ein Angreifer ausnutzen kann. Es handelt sich dabei insbesondere um Schwachstellen, die durch fehlende oder fehlerhafte Prüfung von Eingabedaten entstehen.

- Angriffen über Pufferüberläufe [98, 60, 39] liegen Fehler in der Speicherverwaltung zu Grunde (beispielsweise wird für eine Eingabe nur ein fester Bereich an Speicher reserviert unabhängig von der tatsächlichen Datenmenge). Dadurch können Bereiche im Speicher der Softwarekomponente (oder auch Knotens), die für andere Zwecke als die Eingabedaten reserviert sind, überschrieben werden.
  Da Pufferüberläufe bereits seit den 1980er Jahren bekannt sind, wurden im Laufe der Zeit eine Vielzahl an Techniken entwickelt, welche einen erfolgreichen Angriff über einen Pufferüberlauf verhindern sollen (siehe auch Gegenmaßnahmen). Für die meisten Gegenmaßnahmen wurden jedoch Varianten der Angriffe gefunden, die weiterhin (zu einem ausreichend hohen Prozentsatz) funktionieren.

- Angriffe über Sonderzeichen (im englischen „format string attack“) [98, 34] nutzen aus, dass bestimmte Zeichen irgendwo in der Verarbeitungskette besondere semantische Bedeutung haben (beispielsweise bedeutet das Semikolon sowohl für SQL als

---

[9] Neben den bereits angesprochenen fehlerhaften Prüfungen der Eingabedaten zählen dazu oftmals auch fehlerhafte Synchronisierung verteilter Prozesse, die dann zu indeterministischen Fehlern – auch „Race-Conditions“ genannt – führen können.

auch für Unix-Kommandozeileninterpreter „nächster Befehl“). Wenn diese Sonderzeichen unbearbeitet an die Prozesskette weitergereicht werden, ermöglicht dies den Angreifer, eigene Befehle einzuschleusen oder die Semantik von bestehenden zu ändern. Am bekanntesten sind „SQL-Injections“ für Anwendungen, die zur Datenhaltung eine SQL-Datenbank nutzen, aber auch der Missbrauch der C-Funktion „printf“ fällt in diese Kategorie.

2.5.3.3 *Gegenmaßnahmen*

Bei den Gegenmaßnahmen lassen sich gut zwei Klassen von Ansätzen (die zueinander orthogonal sind) feststellen. Das Vermeiden von (Sicherheits-)Schwachstellen und das Verhindern, dass eine Schwachstelle durch einen Angreifer ausgenutzt werden kann.

Zur Vermeidung von Schwachstellen gehören Maßnahmen wie die gezielte Wahl einer „robusten“ Programmiersprache[10], sorgfältige Verifikation sämtlicher eingehender Daten an den Schnittstellen der Softwarekomponente als auch Tests auf bekannte problematische Muster im Quelltext zum Beispiel mit statischer Codeanalyse [34].

Zur Vermeidung der Ausnutzbarkeit einer Schwachstelle wird in der Regel versucht, die Softwarekomponente durch einen (impliziten) Absoluttest zu einer Softwarekomponente zu ändern, die bei einem Angriffsversuch einen Anhalteausfall erleidet. Beispiele sind ASLR, StackGuard, bei dem der Angreifer ein zufälliges Geheimnis zur Laufzeit richtig raten muss, damit der Angriff erfolgreich sein kann und nicht aufgrund ungültiger Speicheradressen erkannt wird.

Zusätzlich ist auch mit Diversität und Redundanz eine Toleranz von einzelnen Schwachstellen möglich. Siehe dazu die Kapitel 2.3 und Kapitel 2.5.5.

2.5.4 *Erkennen von ausgenutzten Softwarefehlern*

2.5.4.1 *Problematik*

Da ein Angreifer sich beliebig verhalten kann, nachdem er die Kontrolle über eine Softwarekomponente oder einen Knoten übernommen hat, kann er das Außenverhalten auch weiterhin spezifikationskonform halten. Die Fehlererkennungs- oder Fehlertoleranztechniken können den Angreifer erst erkennen, wenn sich in Folge des Angriffs das Verhalten ändert.

Ein Angreifer, der beliebig lange unerkannt im System verbleibt, ist problematisch, da er über einen großen Zeitraum hinweg weitere

---

[10] Insbesondere die Sprachen C und C++ sind aufgrund der manuellen Speicherverwaltung dafür bekannt, dass einem Programmierer leicht ausnutzbare Fehler unterlaufen können. Daher würden diese zum Beispiel durch Sprachen mit automatischer Speicherverwaltung ersetzt.

Softwarekomponenten angreifen kann oder gezielt auf verwundbare Systemzustände durch (zufällige) Fehler warten kann.

2.5.4.2 *Existierende Techniken zur Erkennung*

Es werden in der Literatur verschiedene Verfahren beschrieben, um von Angreifern übernommene Softwarekomponenten zu erkennen. Im Folgenden ist eine Auswahl von häufig genutzten Ansätzen aufgeführt:

- Netzwerkbasierte Systeme wie [6, 158, 72], die versuchen bereits Angriffsversuche anhand von den dabei entstehenden Verkehrsmustern zu erkennen.
- Lokal laufende Systeme wie [41, 136], die versuchen ausreichend schwer von einen Angreifer deaktivierbar zu sein, so dass die Wahrscheinlichkeit hoch ist, dass eine ausgenutzte Schwachstelle erkannt wird, bevor der Angreifer die Möglichkeit hat, das verhalten soweit zu ändern, dass die Erkennung nicht mehr funktioniert.
- Kombinierte Systeme, die lokale und netzwerkbasierte Teile aufweisen. Ein Beispiel dafür ist [137], bei dem die notwendige Zeit für die Berechnung von Prüfsummen über Teile des Speichers gemessen wird. Es lässt sich zeigen, dass unter bestimmten Annahmen, ein Angreifer die Prüfsummen zwar fälschen kann, aber aufgrund der dazu notwenigen höheren Zahl von Speicherzugriffen mehr Zeit benötigt und damit erkennbar wird. Dies setzt natürlich ein hohes Maß an Determinismus für die Übertragungszeit der Nachrichten voraus.

2.5.5 *Diversität*

Softwarediversität ist der gezielte Einsatz von Instanzen von unterschiedlich entwickelten Softwarekomponenten für die gleiche Funktionalität als Schutz vor systematischen Fehlern in einer Softwarekomponente. Bei Diversität handelt es sich also nicht um die Replikation von Instanzen einer Softwarekomponente. Vielmehr basiert jede Instanz auf einer eigenen Softwarekomponente, die von den anderen Softwarekomponenten unabhängig ist, um zu erreichen, dass ein möglicher Softwarefehler mit hoher Wahrscheinlichkeit nur in einer der Instanzen auftritt. Um einen systematischen Fehler in einer Instanz zu tolerieren, wird in dem selbstsichernden oder fehlertolerierenden System ein Vergleich/Mehrheitsentscheid über die Ergebnisse der Instanzen der diversitären Softwarekomponenten getroffen. [14, 45] Alternativ oder zusätzlich kann jedes einzelne Ergebnis auch einem Absoluttest unterzogen werden.

Der mögliche Zeitpunkt, indem Diversität in einem Entwicklungsprozess eingeführt wird, und das Vorgehen für die Entstehung von Diversität, werden im Folgenden kurz erläutert. Darauf folgt eine kurze Übersicht über vorhandene Studien, die Aufwand und Effektivität untersuchen.

2.5.5.1 *Zeitpunkt der Anwendung*

Softwarediversität kann bereits zum Zeitpunkt des Designs des Systems beginnen, oder erst ab den Zeitpunkt der Implementierung einer Softwarekomponente. [101, 89]

Diversität ab dem Designzeitpunkt führt zunächst zu mehreren voneinander unabhängigen Systemspezifikationen. Durch die unabhängige Erhebung der Anforderungen und daraus folgenden Anforderungen an das System wird das Risiko von gemeinsamen Designfehlern reduziert (siehe Kapitel 2.5.2). Da jedoch die Varianten der Softwarekomponenten im Betrieb interoperabel sein müssen, ist im Regelfall nach der unabhängigen Design- und Spezifikationsphase ein Zusammenführen und Vereinheitlichen der Spezifikationen notwendig. Durch diese Zusammenführung entsteht dann die Spezifikation für die Diversität zum Zeitpunkt der Implementierung. Eine diversitäre Entwurfsphase bietet, aufgrund der notwendigen Zusammenführung keine voneinander unabhängigen Ergebnisse, aber einen Mechanismus um Fehler in dem Design aufzudecken.

Diversität zum Zeitpunkt der Implementierung einer Softwarekomponente beruht auf einer gemeinsamen Spezifikation, aber voneinander (möglichst) unabhängigen Realisierungen der Spezifikation. [100, 67] Die gemeinsame Spezifikation kann dabei konventionell oder diversitär entwickelt worden sein. Diversität zum Zeitpunkt der Implementierung kann Schwachstellen in der Mehrheit der Softwarekomponenten vermeiden, bietet jedoch keinen Schutz gegen Designfehler. Mögliche Vorgehensweisen zur Erzielung von Diversität bei der Implementierung werden im folgenden Kapitel beschrieben.

2.5.5.2 *Art der Entstehung*

Zur Erzeugung der Unterschiede zwischen den verschiedenen Varianten einer Softwarekomponente gibt es drei Ansätze:

ZUFÄLLIGE DIVERSITÄT: Bei der zufälligen Diversität wird die Implementierung an voneinander isolierte Teams gegeben. Die Unterschiede ergeben sich im Wesentlichen aus den unterschiedlichen Präferenzen und Vorgehensweisen der Teams. [48]

GEZIELTE DIVERSITÄT: Die gezielte Diversität entspricht im Kern der zufälligen Diversität, nur wird versucht durch Vorgaben an die Teams die resultierende Diversität zu erhöhen. Dies kann auf verschiedenen Ebenen erfolgen. Es können beispielsweise unterschiedliche Programmiersprachen oder Entwicklungsvorgehen verordnet werden. Teilweise wird jedoch auch vorgeschlagen, einzelne Implementierungsdetails (z.B. Datenstrukturen) vorzugeben, was die Freiheit der Entwickler dann stark einschränkt und die (zufällige) Diversität reduzieren kann. [48]

AUTOMATISCHE DIVERSITÄT: Anders als bei den beiden vorhergehenden Ansätzen wird bei der automatischen Diversität nur eine Implementierung erstellt. Diese Implementierung wird dann automatisch in verschiedene Varianten transformiert (beispielsweise durch unterschiedliche Reihenfolgen von unabhängigen Codeblöcken, variierendes genutztes Speicherlayout). Die damit erzielbare Diversität ist jedoch begrenzt. Die Literatur [58, 31] schlägt es insbesondere zur Toleranz von „Heisenbugs" (Fehler, die nur in ganz bestimmten (internen-) Systemzuständen auftreten) und Angreifern vor. Die Angreifertoleranz wird in der Literatur damit begründet, dass die automatische Diversität ausreichend ist, damit mit hoher Wahrscheinlichkeit ein Angriffsvorgehen nur für eine Variante funktioniert[11].

2.5.5.3 *Effektivität/Wirksamkeit und Aufwand*

In der Vergangenheit wurden bereits einige Studien und Experimente [22, 49, 15, 67, 88, 64, 93, 138] zur Wirksamkeit von Diversität durchgeführt. Bei einigen wurde dabei auch der Aufwand gegenüber einer nicht diversitären Entwicklung untersucht. Die folgenden Erkenntnisse wurden jeweils in mehreren Experimenten bestätigt, so dass man diese als relativ gesichert ansehen kann:

- Ein diversitär entwickeltes System weist gegenüber einem nicht diversitär entwickelten System eine signifikant geringere Fehlerwahrscheinlichkeit auf. Diese kann jedoch höher sein als bei rein unabhängigem Versagen der Varianten zu erwarten wäre. [67, 25]

- Der Entwicklungsaufwand, um diversitäre Varianten zu erstellen, steigt nicht linear mit der Zahl der Varianten. Zwei Varianten erhöhen die Kosten etwa um 60%-80%, drei Varianten verdoppeln in etwa die Kosten im Vergleich zu einer nicht diversitären Entwicklung. [73, 22, 96, 87]

- Gemeinsame Fehler in unterschiedlichen Varianten einer Softwarekomponente gehen oftmals auf Schwächen der Spezifikation zurück. Dabei kann die Spezifikation selbst fehlerhaft sein oder sie wird falsch interpretiert. [49, 15, 22, 67]

---

[11] Der Grund liegt darin, dass bei den verbreitetsten Angriffen über Pufferüberläufe ohne genau Kenntnisse über den Aufbau des Speichers der Erfolg sehr unwahrscheinlich ist.

Die Abwägung, ob die geringere Fehlerwahrscheinlichkeit den erhöhten Aufwand des diversitären Entwurfs rechtfertigt, fällt in verschiedenen Domänen unterschiedlich aus. Teilweise gibt es regulatorische Vorgaben (z. B. bei Atomkraft [36]), in anderen Fällen sind Sicherheitsnachweise mit und ohne Diversität möglich. Beispiele finden sich bei Eisenbahnstellwerken [106] oder Fly-by-Wire Steuerungen (Airbus [24] mit Diversität, Boing [156] ohne).

3

KRYPTOGRAPHIE

In diesem Kapitel werden zunächst die Begriffe und grundlegende Verwendungszwecke beschrieben, um dann die zwei grundlegenden Klassen von Techniken zu beschreiben. Am Ende wird noch exemplarisch gezeigt, wie für bestimmte Anwendungszwecke geeignete Kryptographieverfahren gewählt werden können und sich verschiedene Techniken kombinieren lassen.

Die für eine sichere Implementierung notwendigen Details sind aufgrund ihres Umfangs im Allgemeinen nicht Bestandteil dieser Arbeit.

3.1 GRUNDLAGEN

Zur Kryptographie gehören mathematische Methoden zum Schutz von Daten vor Dritten [140, 134]. Traditionell wurden mit dem Begriff nur Methoden bezeichnet, die durch Verschlüsselung die Vertraulichkeit der Daten garantieren. Mittlerweile werden auch die Felder der Sicherung der Integrität einer Nachricht und die Authentifizierung des Absenders als Themengebiet der Kryptographie betrachtet [27, 52].

Es wird dabei als großes Unterscheidungsmerkmal zwischen symmetrischer und asymmetrischer Kryptographie unterschieden.

3.1.1 *Begriffe*

Im weiteren Verlauf des Kapitels werden einige grundlegende Begriffe genutzt, die hier kurz vorgestellt werden.

SCHLÜSSEL: Parameter (k) für den kryptographischen Algorithmus, die vom Sender (k_s) und Empfänger (k_e) benutzt werden. Bei einem guten kryptographischen Algorithmus ist der Schlüssel das einzige, das vor einem Angreifer geschützt werden muss.

KLARTEXT: Die zu übertragenden Daten in dem ursprünglich vorliegenden Format, bevor eine kryptographische Funktion angewendet wurde.

GEHEIMTEXT: Die zu übertragenden Daten in dem Format, nachdem eine kryptographische Funktion angewendet wurde. Der Geheimtext lässt im Idealfall ohne Kenntnis des Schlüssels keine Rückschlüsse auf den Klartext zu.

3.1.2 *Symmetrische Kryptographie*

Bei der symmetrischen Kryptographie wird vom Versender der gleiche Schlüssel für die kryptographische Operation (Verschlüsseln oder Signieren) genutzt, den auch der Empfänger für die Umkehroperation (Entschlüsseln oder Signatur prüfen) benötigt ($k_s = k_e$). Der Schlüssel muss daher über einen sicheren Kanal ausgetauscht werden, bevor geheim kommuniziert werden kann. [134, 27]

Moderne Verfahren für symmetrische Kryptographie haben den Vorteil, dass sie sehr schnell sind, das heiß nur relativ wenig Rechenaufwand benötigen [143].

Zu den symmetrischen Algorithmen gehören unter anderen AES (Advanced Encryption Standard) [117], DES (Data Encryption Standard) [116] und IDEA (International Data Encryption Algorithm) [107]. Zu den symmetrischen Verfahren gehört auch das „One-Time-Pad" Verfahren [129], das einzige Verschlüsselungsverfahren, welches beweisbar absolut sicher ist. Es benötigt einen zufälligen Schlüssel, der dieselbe Länge hat wie der zu übertragende Klartext. Der Schlüssel darf nur einmal verwendet werden. Schlüssel und Nachricht werden beim Sender mit einem Exklusiv-Oder verknüpft, der Empfänger verknüpft den empfangenen Geheimtext auch mittels Exklusiv-Oder mit dem Schlüssel und erhält so wieder den Klartext.

3.1.3 *Asymmetrische Kryptographie*

Bei der asymmetrischen Kryptographie werden vom Versender und Empfänger jeweils unterschiedliche Schlüssel für die kryptographische Operation (Verschlüsseln oder Signieren) genutzt ($k_s \neq k_e$). Ein Schlüsselpaar besteht dabei aus zwei Teilen, dem sogenannten privaten Schlüssel (k_{privat}), der wie bei der symmetrischen Kryptographie geheim gehalten werden muss, und dem sogenannten öffentlichen (im Englischen public) Schlüssel (k_{public}), der keiner Geheimhaltung bedarf. Die asymmetrische Kryptographie wird daher häufig auch als „Public-Key-Kryptographie" bezeichnet und wurde erst um 1970 erfunden. [143, 134]

Der öffentliche Schlüssel (k_{public}) kann zum Verschlüsseln eines Klartexts genutzt werden oder zum Überprüfen, ob eine Signatur mit dem dazugehörigen privaten Schlüssel (k_{privat}) erstellt wurde. Der private Schlüssel (k_{privat}) kann zum Entschlüsseln eines Geheimtexts – der mit dem dazugehörigen öffentlichen Schlüssel (k_{public}) erstellt wurde – genutzt werden oder zum Erstellen einer Signatur.

Dadurch, dass der öffentliche Schlüssel nicht geheim gehalten werden muss, vereinfach sich signifikant die Schlüsselverwaltung. Bei n Teilnehmern, von denen jeder mit jedem kommunizieren kann, werden bei symmetrischer Verschlüsselung $\frac{n*(n-1)}{2}$ Schlüssel benötigt, die über einen sicheren Kanal verteilt werden müssen. Bei asymmetrischer Verschlüsselung müssen nur n Schlüsselpaare (k_{privat} und k_{public}) erzeugt werden. Sofern sichergestellt ist, dass der Angreifer die übertragenen öffentlichen Schlüssel nicht unbemerkt ändern kann[1], können die öffentlichen Schlüssel auch zentral im Netzwerk gespeichert werden. Als Nachteil von asymmetrischen Verfahren ist der deutlich höhere Rechenaufwand zu sehen. [27]

Zu den häufig genutzten asymmetrischen Algorithmen gehören unter anderen RSA (von Rivest/Shamir/Adleman) [131], ElGamal [50] und auf elliptischen Kurven basierende ECC (Elliptic Curve Cryptography) Verfahren [74].

3.2 ANWENDUNGEN

Die verschiedenen Techniken der Kryptographie lassen sich jeweils einem oder mehreren der folgenden drei kryptographischen Kernfunktionalitäten zuordnen.

3.2.1 *Einwegfunktionen / Hashfunktionen*

Eine Einwegfunktionen, auch Hashfunktionen genannt, ist eine Funktion (F), die Werte aus der Menge der Eingabewerte (E) auf eine begrenzte Menge von möglichen Hashwerten (H) abbildet [134]. Die Mächtigkeit der möglichen Eingabewerte übersteigt dabei die möglichen Hashwerte ($|E| > |H|$). Dadurch gibt es Mengen von Elemente aus der Eingabemenge, die den gleichen Hashwert haben ($e_1, e_2 \in E$, $e_1 \neq e_2$, $F(e_1) = F(e_2)$). Dies nennt man auch eine Kollision.

Da eine Kollision Unterschiede in den Eingabewerten nicht aufdeckt, sollte bei einer guten Hashfunktion die Zahl der Kollisionen möglichst gering sein. Dazu sind die folgenden drei Bedingungen zu erfüllen:

- Die Funktion ist surjektiv. Das heißt, es werden alle möglichen Hashwerte auch genutzt.
- Für einen zufälligen Eingabewert ist jeder Hashwert in etwa gleich wahrscheinlich. Zusammen mit der ersten Bedingung folgt daraus, dass für einen beliebigen Eingabewert ($e \in E$) und eine zufälligen Hashwert ($h \in H$) mit der Wahrscheinlichkeit $p \simeq \frac{1}{|H|}$ gilt $h = F(e)$.
- Auch bei kleinen Änderungen am Eingabewert, sollen sich die resultierenden Hashwerte mit großer Wahrscheinlichkeit ($p \simeq 1 - \frac{1}{|H|}$) sich voneinander unterscheiden.

Für kryptographische Hashfunktionen gilt zusätzlich noch die Anforderung, dass ein Angreifer nicht gezielt Kollisionen herbeiführen

---

[1] Zum Beispiel durch Einsatz von kryptographischen Signaturen, der dazugehörige Schlüssel muss dann aber bekannt sein.

kann. Dies bedeutet, dass die Umkehrung der Funktion, also die Bestimmung von möglichen Eingabewerten für einen gegeben Hashwert, nicht realistisch[2] möglich ist. Außerdem soll gelten, dass nicht gezielt zwei unterschiedliche Nachrichten erstellt werden können, die den gleichen Hashwert ergeben [43, 115, 134].

In der Nachrichtentechnik wird oftmals an Nachrichten das Ergebnis einer Hashfunktion (meistens ein CRC) angehängt. Dies dient der Erkennung von zufälligen Verfälschungen durch Absoluttests (siehe Kapitel 2.3.4), bietet jedoch keinen Schutz gegen einen Angreifer, da dieser nach der Manipulation der Nachricht mit demselben Algorithmus den nach der Veränderung gültigen Wert berechnen kann[3].

Im Bereich der Kryptographie werden kryptographische Hashfunktionen meistens genutzt, um für Signaturen die zu signierende Datenmenge zu reduzieren. Anstatt die Nachricht zu signieren wird nur ein Hashwert signiert und diese Signatur dann mit der ursprünglichen Nachricht übermittelt. Dadurch kann Bandbreite bei der Übertragung gespart werden und die rechenintensiven Signaturverfahren (insbesondere bei asymmetrischer Kryptographie) müssen nur für geringere Datenmengen ausgeführt werden.

3.2.2 *Verschlüsselung & Entschlüsselung*

Die Verschlüsselung von Daten dient dem Schutz des Klartexts (t) während der Übertragung oder Speicherung gegen Ausspähen durch einen Angreifer. Dazu wird beim Sender eine Verschlüsselungsfunktion (E) genutzt, die als Parameter den Klartext und den Schlüssel des Senders enthält, um den Geheimtext (c) zu erzeugen ($c = E(t, k_s)$). Der Empfänger nutzt eine Umkehrfunktion (D), um mit seinen Schlüssel aus dem Geheimtext wieder den Klartext zu erhalten ($t = D(c, k_e)$). Bei einem guten kryptographischen Verfahren lässt der Geheimtext ohne die Kenntnis des Schlüssels keine Rückschlüsse auf den Klartext zu. [134]

Durch den Aufbau der kryptographischen Algorithmen können bereits kleine Fehler im Geheimtext bei der Entschlüsselung zu großen Unterschieden im Klartext führen. Um sicherzugehen, dass der Klartext t des Senders identisch ist mit dem Klartext des Empfängers, werden zusätzliche Informationen benötigt (Beispielsweise definierte Bitmuster), die sich für einen Absoluttest nutzen lassen (wie in [81]). Der Absoluttest verifiziert implizit damit auch, dass der Sender den Schlüssel k_s verwendet hat. Dies kann insbesondere bei symmetrischen Verfahren auch eine Authentifizierung des Absenders darstel-

---

[2] Das heißt nur über Ausprobieren/Brute-Force oder vergleichbaren Aufwand

[3] Eine Ausnahme bilden Keyed-Hash Message Authentication Code (HMAC), die als Mischung aus Hashfunktion und symmetrischer Signatur angesehen werden kann. Werden im Kapitel Signaturen vorgestellt.

len, da andere Kommunikationsteilnehmer und der Angreifer per Definition nicht im Besitz des Schlüssels sein dürfen.

3.2.3 *Signaturen*

Eine Signatur ist der Beweis, dass eine übermittelte Nachricht vom vorgegebenen Absender stammt. Dazu muss sie nach [134] die folgenden Eigenschaften aufweisen:

- Die Signatur darf nicht durch den Angreifer zu fälschen sein.
- Die Echtheit der Signatur muss (durch den Empfänger) überprüfbar sein.
- Änderungen am Klartext müssen die Signatur invalidieren.

Der Beweis ist, unabhängig ob symmetrische oder asymmetrische Kryptographie verwendet wird, auf zwei Arten möglich [143]:

- Als direkte Signatur oder auch Signatur mit Nachrichtenrückgewinnung. Dabei wird der Klartext vollständig mit dem kryptographischen Schlüssel des Absenders k_s transformiert, so dass der Empfänger den Klartext zurückgewinnen kann.
  Bei der symmetrischen Kryptographie ist – wenn der Schlüssel nur zwischen zwei Kommunikationspartnern verwendet wird – damit auch die Authentifizierung gewährleistet. Bei der asymmetrischen Kryptographie verwendet der Sender dabei den privaten Schlüssel ($k_s = k_{privat}$) und kann so beweisen, der Inhaber des privaten Schlüssels zu sein, zu dem der Empfänger den passenden öffentlichen Schlüssel besitzt ($k_e = k_{public}$).
- Als Fingerabdruck oder Message-Digest. Dabei wird an den Klartext ein auf einem Hashwert des Klartext basierender Beweis angehängt, der garantiert, dass die Nachricht vom Absender stammt. Bei asymmetrischer Kryptographie wird der Hashwert wie bei der direkten Signatur mit dem privaten Schlüssel verschlüsselt. Bei der symmetrischen Kryptographie gibt es zwei Möglichkeiten. Zum einen kann der Hash verschlüsselt werden oder aber als Eingabe der Hashfunktion der Klartext mit dem Schlüssel kombiniert werden. Die letztere Variante wird auch HMAC genannt und hat den Vorteil sehr schnell zu sein, da nur die Hashfunktion berechnet werden muss und keine zusätzlichen kryptographischen Operationen notwendig sind [23].

3.2.4 *Kombinationen aus symmetrischer und asymmetrischer Kryptographie*

Da asymmetrische Kryptographie im Vergleich zur symmetrischen Kryptographie einen sehr hohen Ressourcenbedarf hat (siehe Kapitel 3.1.2 und Kapitel 3.1.3), werden oftmals für die Verschlüsselung beide Verfahren kombiniert (auch „Hybridverfahren“ genannt [134]). Aufgrund der einfacheren Schlüsselverwaltung wird ein asymmetrisches Verfahren für die Übermittlung eines zufällig gewählten Sitzungsschlüssels genutzt. Der zufällige Sitzungsschlüssel wird dann für ein symmetrisches Verfahren genutzt.

3.2.5 *Public-Key-Infrastruktur*

Um bei der Schlüsselverwaltung für asymmetrische Verschlüsselungsverfahren zu vermeiden, dass alle öffentlichen Schlüssel über vertrauenswürdige Kanäle ausgetauscht werden müssen, kann eine eine Public-Key-Infrastruktur (PKI) genutzt werden. Dazu wird an jeden Teilnehmer über einen vertrauenswürdigen Kanal einmalig ein öffentlicher Schlüssel übermittelt. Der dazugehörige private Schlüssel wird dann genutzt, um nach erfolgreicher Identifikation des jeweiligen Teilnehmers, ein sogenanntes Zertifikat zu erstellen. Das Zertifikat enthält dabei den öffentlichen Schlüssel des Teilnehmers und die notwendigen Informationen, um diesen zu identifizieren. Um das Zertifikat vor Änderungen oder Fälschungen durch Dritte zu schützen, wird es mit den privaten Schlüssel der PKI signiert. [27]

Andere Teilnehmer können statt des öffentlichen Schlüssel das Zertifikat anfordern und die Signatur mit dem gespeicherten vertrauenswürdigen öffentlichen Schlüssel der PKI prüfen. Damit sind MITM-Angriffe erkennbar. Dadurch ist es möglich, nachträglich neue öffentliche Zertifikate sicher zu verteilen, auch über nicht vertrauenswürdige Verbindungen.

4

EXISTIERENDE ANSÄTZE ZUR ANGREIFERTOLERANZ

4.1 ÜBERSICHT

Der Begriff Angreifertoleranz wird in der Literatur für zwei verschiedene Szenarien verwendet. Der Schutz eines Systems, das vollständig unter der Kontrolle des Betreibers steht, vor einem externen Angreifer ist die erste Interpretation von "Angreifertoleranz". Das zweite Szenario besteht darin, die Integrität einer Anwendung, die auf einer nicht vollständig vertrauenswürdigen Infrastruktur aufbaut, sicherzustellen. In diesem Fall wird angenommen, dass die Infrastruktur vom Betreiber der Anwendung nicht vollständig kontrolliert werden kann. Beispiele für diese Annahme sind Cloud-Services oder auch Sensornetze, bei denen ein Dritter, der auch Angreifer sein kann, ein Teil der Infrastruktur kontrolliert.

Für das zweite Szenario gibt es verschiedene Forschungszweige:

- Die Gewährleistung des zuverlässigen Betriebs trotz Byzantinischer Fehler, die durch einen Angreifer verursacht werden [103, 9, 157].
- Die Simulation eines vertrauenswürdigen Dritten um Entscheidungen treffen zu können [79, 99], ohne dem potentiell nicht vertrauenswürdigen Gegenüber ein Geheimnis verraten zu müssen.
- Die Speicherung von Daten, die für den Angreifer nicht rekonstruierbar seien sollen [128, 77].

Das zweite Szenario wird im Folgenden nicht weiter betrachtet, da die betrachtete Klasse von Systemen typischerweise vollständig unter der Kontrolle des Betreibers steht. Daher liegt dieser Arbeit das erste Szenario zugrunde, nämlich der Schutz eines Systems, das vollständig unter der Kontrolle des Betreibers steht. Die Literatur zum Stand der Wissenschaft in den folgenden Kapiteln wurde darauf beschränkt.

4.2 KRITERIEN FÜR DIE AUSWAHL DER VERFAHREN

Es gibt eine Vielzahl von Ansätzen, um sicherheitsrelevante Systeme vor Angreifern zu schützen. Sie lassen sich gut in zwei Gruppen aufteilen: Zum einen die Ansätze [18, 159, 121, 148], die durch verschiedene Maßnahmen (z. B. Firewalls, Zugriffsbeschränkungen, Trennung von Kommunikationsmedien, fehlerfreie Software) versuchen sicherzustellen, dass ein Angreifer nicht in der Lage ist, das sicherheitsrelevante System zu manipulieren. Da dies der Kernannahme dieser Arbeit (siehe Kapitel 1.2) widerspricht, werden diese Ansätze im Rahmen dieser Arbeit nicht weiter betrachtet.

Zu dem ersten Ansatz zählen auch die meisten Systeme, die kryptographische Signaturen zum Schutz vor Maskerade nutzen (wie in [92], oder in [5] für Car-to-Car Kommunikation vorgeschlagen). Kryptographisch starke Signaturen bieten zwar grundsätzlich einen guten Schutz vor Maskerade und ermöglichen damit, nur mit als vertrauenswürdig definierten Knoten zu kommunizieren. Jedoch werden systematische Fehler in dem Programmteil, der die Überprüfung der Signaturen vornimmt, nicht berücksichtigt. Außerdem werden mehrstufige Angriffe (mit mehreren Bugs) bei denen erst ein kryptographischer Schlüssel vom Angreifer in Besitz gebracht wird und danach mit gültigen Signaturen Anwendungen angegriffen werden, nicht betrachtet.

Der zweite Ansatz besteht darin anzunehmen, dass es trotz aller (weiterhin sinnvoller) Schutzmaßnahmen einem Angreifer gelingen kann, Zugriff auch auf die (geschützten) Netzbereiche zu erlangen. In diesem Fall müssen die möglichen Auswirkungen des Angreifers toleriert werden. In dem Papier [141] werden dabei die Ansätze zur Vermeidung von Schwachstellen weiter systematisiert und von der Angreifertoleranz im engeren Sinne abgegrenzt.

Zwischen der Verhinderung von Angriffen und der Tolerierung von Angriffen lässt sich auch noch die Angreifererkennung einsortieren. Dieser Ansatz wird in der Praxis gerade für viele Supervisory Control and Data Acquisition (SCADA)-Systeme [113, 158, 122, 62] genutzt. Dabei wird mit verschiedenen Techniken nach Anomalien (zum Beispiel atypische Verkehrsmuster im Netzwerk, unbekanntem Code auf einem Knoten) gesucht, die auf einen Angreifer hindeuten. In diesem Fall wird das System entweder passiviert und es obliegt dem Betreiber, wieder einen fehlerfreien Zustand herzustellen, oder es wird nur der Betreiber informiert. Die erste Variante lässt sich nur auf Systeme anwenden, die einen leicht erreichbaren sicheren Zustand aufweisen. Sie benötigt einen guten Absoluttest zur Angreifererkennung. Bei der zweiten Variante entstehen regelmäßig (alleine durch menschliche Reaktionszeiten) Zeiträume, in denen das System verwundbar ist, was für sicherheitsrelevante Systeme inakzeptabel ist. Da diese Ansätze auch keine Tolerierung von Angriffen ermöglichen, werden sie nicht näher betrachtet.

Die Ansätze, bei denen ein Angriff vom System selbst toleriert wird, werden im Folgenden vorgestellt. Dabei werden jeweils Fehlertoleranztechniken genutzt, um die Auswirkungen des Angreifers zu begrenzen. Ein Einsatz von Fehlertoleranzverfahren, insbesondere Diversität und Mehrheitsentscheide, zum Schutz gegen Angreifer wird bereits seit den 1980er Jahren untersucht. Es lassen sich dabei drei unterschiedliche Herangehensweisen erkennen, nach denen die Verfahren auch im Folgenden gruppiert werden (auch wenn im Einzelfall Kombinationen existieren). Die erste Herangehensweise besteht im Wesentlichen darin, mittels Diversität die Wahrscheinlichkeit ausreichend gering halten, dass ein Angreifer die Mehrheit der Knoten eines verteilten Systems übernimmt. Die zweite Herangehensweise geht davon aus, dass ein Angreifer immer eine gewisse Zeit benötigt, einen Knoten zu übernehmen, und durch ausreichend häufiges Zurücksetzen in einen definierten nicht manipulierten Zustand die Mehrheit frei von Angreifern bleibt. Die dritte Herangehensweise nutzt Absoluttests, um bereits übernommene Knoten auszugrenzen und durch Rekonfiguration möglichst lange ein funktionales System zu erhalten.

4.3 ANGREIFERTOLERANZ DURCH DIVERSITÄT

Die Angreifertoleranz durch Diversität wird seit 1988 [85] diskutiert. Als Kern für die Angreifertoleranz durch Diversität wird ein fehlererkennendes/fehlertolerierendes System (siehe Kapitel 2.3) mit Komponenten aufgebaut, die diversitäre Eigenschaften (siehe Kapitel 2.5.5) aufweisen. Mit der Annahme, dass die Fehler in den diversitären Varianten der Komponenten voneinander unabhängig sind (zur Bewertung dieser Annahme siehe Kapitel 2.5.5 und Kapitel 9.2), ist ein systematischer Fehler wie ein Bug auf eine Variante beschränkt.

Ein Angreifer kann damit mit jedem bekannten Bug jeweils nur eine Variante einer Komponente erfolgreich angreifen. Das Angreiferverhaltensmodel erlaubt nach einem erfolgreichen Angriff meist beliebiges Fehlverhalten. Dies lässt sich jedoch mit konventionellen Fehlertoleranztechniken erkennen und/oder tolerieren.

Das Verfahren kann als gut untersucht und wirksam gelten. Jedoch ist durch die notwendige vollständige Diversität der Komponenten ein erhöhter Entwicklungsaufwand zu leisten.

4.4 ANGREIFERTOLERANZ DURCH ZURÜCKSETZEN

Bei der Angreifertoleranz durch Zurücksetzen werden die Knoten in regelmäßigen Abständen in einen angreiferfreien Zustand zurückgesetzt. Beispiel ist das Starten von einen ROM. Mit den folgenden Annahmen, ist dann sichergestellt, dass stets die Mehrheit der Knoten fehlerfrei ist.

- Das Zurücksetzen auf den garantiert fehlerfreien Zustand inkl. wieder Inbetriebnahme im Gesamtsystem braucht signifikant weniger Zeit als ein Angriff. Damit lässt sich dann das Zurücksetzen so planen, dass während eines Zyklus ein Angreifer nur f Knoten übernehmen kann.

- Der Angreifer kann die Knoten nur sequentiell angreifen.

Das Verfahren bietet den Vorteil, dass eine Implementierung in vielen Fällen (insbesondere wenn nach dem Start nicht aufwändig konsistente Sichten sichergestellt werden müssen) keinen großen Aufwand erfordert. Es werden allerdings für die Integrationen zur Laufzeit zusätzliche Ressourcen benötigt. Die Erfüllung der Annahmen zu garantieren ist dagegen für viele Systeme sehr schwer.

4.5 ANGREIFERTOLERANZ DURCH REKONFIGURATION

Bei der Angreifertoleranz durch Rekonfiguration wird das System so rekonfiguriert, dass es bestimmte Knoten nicht mehr benutzt, sobald ein Absoluttest oder Relativtest anzeigt, dass die Instanz einer Softwarekomponente auf diesen Knoten durch einen Angriff in einen fehlerhaften Zustand versetzt worden ist. Dieses Vorgehen beruht im Wesentlichen auf zwei Annahmen. Die erste Annahme ist, dass sich der Angriff anhand des Verhaltens des Knotens zeitnah erkennen lässt, also der übernommene Knoten nicht lange weiterhin fehlerfreie Ergebnisse liefert. Die zweite Annahme ist, dass es dem Angreifer nicht gelingt, unbemerkt Instanzen zu übernehmen (nahezu perfekter Absoluttest oder alternativ, dass der Angreifer zwischen zwei Relativtests nicht die Mehrheit kontrollieren kann). Diese Annahme stützt sich teilweise auf das Design, welches mehrere „Sicherheitsstufen“ beinhaltet. Zusätzlich (insbesondere bei fehlertolerierenden Systemen) muss es genug Knoten geben, dass alle vom Angreifer übernommenen Knoten ausgegliedert werden können, bevor durch Wartungsmaßnahmen neue fehlerfreie Knoten hinzugefügt werden. Dies impliziert, dass der Angreifer nur eine stark begrenzte Anzahl von Knoten in einer definierten Zeit übernehmen kann.

Das Verfahren bietet den Vorteil, dass eine Implementierung in einigen Fällen (z. B. in Systemen, die sehr viele Instanzen zur Lastverteilung haben) nur einen geringen Aufwand verursacht, falls der Test leicht zu implementieren ist. Erfüllung der Annahmen zu garantieren ist dagegen für viele Systeme sehr schwer.

4.6 BEISPIELE

„A fault tolerance approach to computer virus“ - Ein fehlertoleranter Ansatz gegen Computerviren

Bereits 1988 wurde von Mark K. Joseph und Algirdas Avižienis in [85] das Zusammenwirken von Diversität und Fehlertoleranz zur Tolerierung von Schadcode untersucht. Die genannten Autoren schlagen vor, die Programme durch einen erweiterten Programmablaufmonitor und Diversität zu schützen. Der Programmablaufmonitor soll Fehler zur Programmlaufzeit erkennen und damit auch potentielle Infektionen von Viren. Nach einer Infektion wird versucht durch einen „Rollback“ wieder in einen sicheren Zustand zu gelangen. Die Diversität wird als einfaches n Varianten-Verfahren mit Mehrheitsentscheid implementiert. Es handelt sich damit um Angreifertoleranz durch Diversität mit Anleihen bei der Diversität durch Zurücksetzen.

Abweichend von den klassischen Ansatz zur Fehlertoleranz liegt der Fokus jedoch nicht darauf, das eigene Programm diversitär zu schreiben, sondern eine diversitäre Werkzeugkette zu nutzen. Die diversitäre Werkzeugkette ist damit begründet, dass sich Viren über manipulierte Compiler einschleichen könnten, die auch bei sorgfältigsten Code-Reviews nicht erkannt werden können. Es können damit weiterhin unentdeckte systematische Fehler in allen Varianten befinden, die ein Angreifer ausnutzen kann, um alle Varianten einer Softwarekomponente simultan anzugreifen.

ITUA - Intrusion Tolerance by Unpredictable Adaptation

Das ITUA [38, 42] Projekt definiert eine adaptive Verteidigungsstrategie für mehrstufige Angriffe. Mehrstufig in diesem Projekt meint, dass der Angreifer verschiedene Sicherheitszonen nacheinander überwinden muss. Um die Überwindung der Grenzen (die zum Beispiel durch Firewalls realisiert werden können) einem Angreifer zu erschweren, sollte das System nicht immer deterministisch auf die bereits erkannten Angriffe reagieren, da sich der Angreifer darauf einstellen kann. Es handelt sich damit um Angreifertoleranz durch Rekonfiguration.

Die Kernannahmen sind, dass der mehrstufige Angriff ausreichend Zeit benötigt, dass eine Angriffserkennung (zum Beispiel durch Absuluttests in der Form von Intrusion Detection Systemen, siehe auch Kapitel 2.5.4) in der Zwischenzeit den Angriff erkennen können und dass der Angreifer in einzelnen Sicherheitszonen das zu schützende Gesamtsystem nicht gefährden kann. Die Funktion des Gesamtsystems lässt sich sicherstellen, indem die Funktionen in verschiedene Sicherheitszonen repliziert werden und dabei so entworfen werden, dass sie einzelne fehlerhafte Replikate tolerieren. Neben der Replikation sind für die Sicherheitszonen jeweils „Gruppenmanager“ und die Angreifererkennungstests notwendig. Diese erhöhen den Entwicklungsaufwand und benötigen zur Laufzeit auch Ressourcen.

Neben dem zusätzlichen Aufwand durch die Gruppenverwaltung ist für ein konkretes System zu überprüfen, ob die Annahme der unabhängigen Sicherheitszonen praktikabel anwendbar ist, da zwischen denen nur streng kontrollierter Informationsaustausch erlaubt ist. Benötigt eine Anwendung zum Beispiel ein Übereinstimmungsprotokoll zwischen allen Instanzen einer Softwarekomponente, ist damit weitgehende Kommunikation zwischen den unterschiedlichen Sicher-

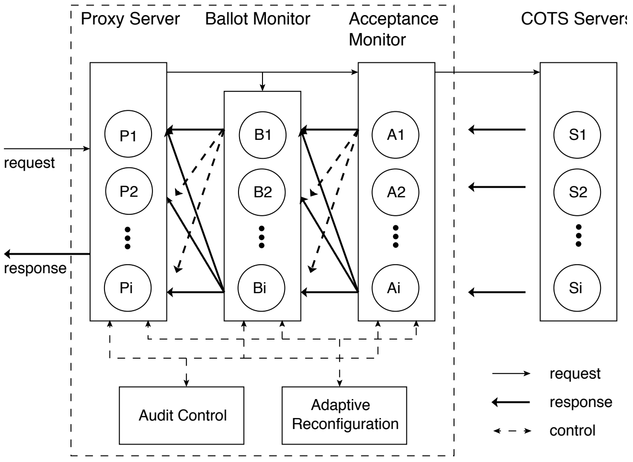


Abbildung 4: SITAR Architektur [150]

heitszonen notwendig und damit die Erfüllung der Annahme gefährdet.

MAFTIA - Malicious- and Accidental-Fault Tolerance for Internet Applications

In dem MAFTIA Projekt [126] wird ein generisches Framework für verschiedene internetbasierte (d. h. über öffentliche Netze erreichbare) Dienste entwickelt. Im Bereich der Angreifertoleranz werden die verschiedensten Techniken und Konzepte (Diversität, Mehrheitsentscheide, Erkennung von Angreifern, Ausgrenzen von fehlerhaften Knoten, Rekonfigurierung) erwähnt. Die Autoren bleiben jedoch so allgemein, dass keine konkreten Kombinationen von Techniken analysiert werden, sondern es den jeweiligen Entwicklern überlassen wird, eine geeignete Auswahl zu treffen und diese zu bewerten. Eine Zuordnung zu einem Ansatz ist aufgrund der Flexibilität nicht möglich.

SITAR - Scalable Intrusion Tolerant Architecture

Das SITAR [150] System realisiert einen Filter zwischen einem öffentlichen Netz, aus dem die Anfragen kommen, und einer Menge von Commercial off-the-shelf (COTS) Servern. Die Architektur ist in Abbildung 4 dargestellt. Man kann dabei gut die mehrstufige Architektur erkennen.

Anfragen werden von einem Proxyserver entgegen genommen. Der genutzte Proxyserver wählt gemäß der aktuellen Konfiguration die COTS Server aus, die diese Anfrage bearbeiten sollen. Es werden dabei mehrere gewählt, um Fehler, die zum Beispiel durch vorhergegangene Angriffe verursacht worden sind, bei einem COTS Server erkennen zu können. Bevor die Anfrage an die COTS Server geleitet wird, übernehmen die „Acceptance Monitors“ noch eine Prüfung auf bereits bekannte Angriffsmuster, die dann herausgefiltert würden. Die Antwort der COTS Server wird erneut an die „Acceptance Monitors“ gesendet, die mit Absoluttests nach Fehlern oder Anzeichen suchen, dass der COTS Server übernommen worden ist. Falls welche gefunden werden, wird diese Information an die für die Rekonfiguration zuständige Komponente weitergeleitet und führt damit zu einer Ausgliederung der betroffenen Komponente. Die Ergebnisse werden, nachdem sie die Acceptance Monitors erneut passiert haben, an „Ballot Monitors“ gesendet. Diese wählen durch ein Übereinstimmungsverfahren und Mehrheitsentscheid ein Ergebnis aus, das dann an den Proxy gesendet wird und von dort an den Client. Falls bei dem Mehrheitsentscheid Abweichungen gefunden werden – die auf einen Angriff hindeuten – führt dies wiederum zu einer Rekonfiguration. Das „adaptive reconfiguration module“ nimmt dynamisch Anpassungen an dem System vor. Als vom Angreifer übernommen erkannte Knoten werden ausgegliedert, weitere Komponenten werden bei Bedarf aktiviert.

Die internen Komponenten des SITAR Systems werden dabei diversitär entwickelt, um die Wahrscheinlichkeit gering zu halten, dass sie zeitgleich übernommen werden können.

Der hohe Aufwand im SITAR-System (drei Stufen diversitärer Systeme zuzüglich einer größeren Menge an COTS Servern und weitere Komponenten zur Verwaltung) erlaubt die Exposition von COTS Servern in einem öffentlichen Netz. Die dabei getroffenen Annahmen sind überschaubar:

- Sehr vereinzelt darf eine Anfrage fehlerhaft beantwortet werden, falls ein Proxy-Server die Anfrage an Server schickt, die unmittelbar vorher durch einen Angriff übernommen wurden, und die Übernahme von keinem Absoluttest erkannt worden ist.
- Übernommene Server werden nach kurzer Zeit erkannt, spätestens wenn sie beim Mehrheitsentscheid in der Minderheit sind.
- Der Angreifer kann nicht alle COTS Server auf einmal übernehmen.

Die erste Annahme ist für viele öffentliche Systeme durchaus vertretbar, insbesondere da die Alternativen in der Regel deutlich länger fehlerhafte Server exponieren. Die anderen beiden Annahmen betreffen Annahmen über den Angreifer. Ihre Erfüllung ist schwieriger sicherzustellen, da ein Angreifer die übernommenen Server sowohl eine längere Zeit normal weiter laufen lassen könnte, so dass das Verhalten nach außen unverändert ist, als auch mit einer Vielzahl von Angriffs-Anfragen zeitgleich die Mehrheit der COTS Server zugleich angreifen könnte (sofern der Angriff die Absoluttests übersteht).

4.7 RAUM FÜR EFFIZIENTERE VERFAHREN

Der generische Ansatz mit vollständiger Diversität (angelehnt an den „A fault tolerance approach to computer virus“ [85]) kombiniert mit einem effektiven Maskeradeschutz kann als „Goldstandard“ bezüglich Angreifertoleranz angesehen werden. Der große Nachteil dabei ist jedoch, dass von den Anwendungskomponenten jeweils n-Varianten mit einem entsprechend hohen Aufwand entwickelt werden müssen.

Die SITAR-Architektur hingegen bietet einen (größtenteils) wiederverwendbaren Schutzmechanismus, der generisch für COTS Server und Anfragen aus öffentlichen Netzen geeignet ist. Da die SITAR-Architektur mit Anfragen von nicht vertrauenswürdigen Knoten umgehen können muss und nicht auf Fehlertoleranztechniken in den COTS Servern aufbauen kann, ist ein hoher Aufwand bei der mehrstufigen – jeweils in sich redundanten – Filterhierachie notwendig (mit daraus resultierenden Latenzen).

Ein generischer (wiederverwendbarer) Ansatz für in sich geschlossene fehlertolerante Systeme wurde bisher noch nicht vorgestellt. Dabei würde es im Vergleich zu vollständig diversitär entwickelten Systemen den Entwicklungsaufwand signifikant reduzieren können (durch die Wiederverwendbarkeit). Im Vergleich zur SITAR-Architektur kann der Ansatz für geschlossene fehlertolerante Systeme deutlich einfacher aufgebaut werden, da zum einen durch die geschlossene Kommunikationsstruktur das System besser gegen den Angreifer isoliert werden kann. Zudem kann durch die Mitbenutzung der systeminhärent vorhandenen Fehlertoleranztechniken der zusätzliche Aufwand zur Erkennung und/oder Toleranz der vom Angreifer hervorgerufenen Fehlerbilder reduziert werden.

Die Entwicklung eines solchen Verfahrens, welches mit möglichst geringen zusätzlichen Mitteln vor Angriffen schützen kann und dabei die bereits vorhandenen Fehlertoleranztechniken mitbenutzt, ist Zielsetzung dieser Arbeit.

Teil III

KRYPTOGRAPHISCHE INSEL

In diesem Teil wird das Konzept der kryptographischen Insel vorgestellt, welches für fehlertolerante Systeme mit geringem Zusatzaufwand zusätzlich Toleranz gegenüber Angriffen ermöglicht. Es gliedert sich in zwei Bereiche. Zuerst wird das Konzept erläutert und verschiedene Optionen vorgestellt, mit denen es an verschiedene Nutzungszwecke angepasst werden kann. Danach findet eine Bewertung statt, welche den notwendigen Ressourceneinsatz betrachtet und mit anderen Lösungen vergleicht.

5

DER NEUE ANSATZ

5.1 IDEE

Fehlertolerante Systeme besitzen bereits Techniken, um mit fehlerhaften Knoten umzugehen. Die Idee ist, diese Systeme mit relativ geringem Entwicklungsaufwand so zu härten, dass ein externer Angreifer anders als in Kapitel 2.2 beschrieben nicht alle gleichartigen Komponenten übernehmen kann. Das Kernkonzept besteht darin, das sicherheitsrelevante fehlertolerante System ausreichend von der Umgebung zu isolieren, trotz der potentiell vorhandenen Kommunikationskanäle.

Dies wird realisiert, indem der Nachrichtenaustausch innerhalb des vertrauenswürdigen fehlertoleranten Systems kryptographisch signiert wird, damit Nachrichten von nicht autorisierten Komponenten sicher erkannt werden können. Da auch in dem Teil der Software ein Fehler liegen kann, der den Nachrichtenempfang inklusive Maskeradeprüfung realisiert, wird dazu eine diversitär entwickelte Softwareschicht genutzt. Diese Softwareschicht wird im Folgenden Kommunikationshypervisor genant. Die logische Isolierung der kryptographischen Insel durch den Maskeradeschutz wird in Abbildung 5 visualisiert. Durch eine geeignete Verteilung der Komponenten auf die verschiedenen Varianten des Kommunikationshypervisor sind dann die Auswirkungen eines Angriffs auf (normales) beliebiges Fehlverhalten der übernommenen Knoten beschränkt. Der Angriff wird also von den Auswirkungen auf ein Fehlerbild reduziert, das von vielen fehlertoleranten Systemen durch die ohnehin vorhan-

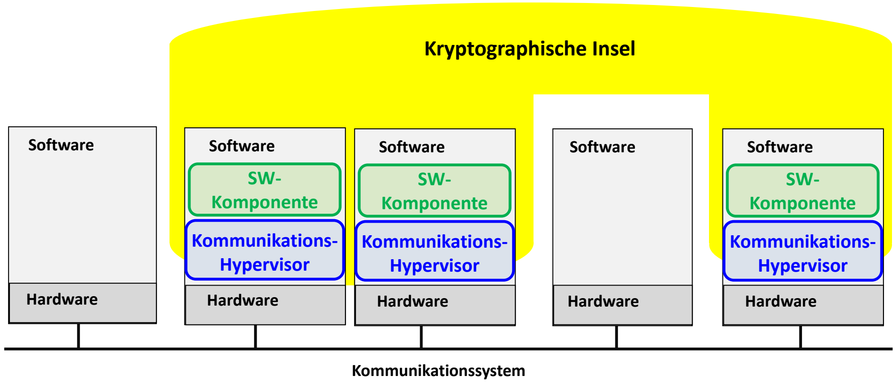


Abbildung 5: Struktur der logischen Isolierung durch die kryptographische Insel

denen Fehlererkennungs- oder Fehlermaskierungsverfahren toleriert wird.

Durch eine geeignete Wahl der Schnittstellen des Kommunikationshypervisor der kryptographischen Insel ist dieser nicht auf ein konkretes System beschränkt. Der Kommunikationshypervisor lässt sich damit wiederverwenden und verringert so den Aufwand im jeweiligen Projekt, nachdem die diversitär entwickelten Varianten initial erstellt worden sind. Dadurch kann ein Einsatz auch für neue Anwendungsfelder wirtschaftlich attraktiv werden.

5.2 EINSATZGEBIET

Aufgrund ihres Aufbaus (siehe Kapitel 5.1 und Kapitel 7) ist eine kryptographische Insel besonders geeignet für Systeme mit den folgenden Randbedingungen:

- Es steht eine ausreichende statische Redundanz zur Verfügung, um bei den übernommenen Knoten beliebiges Fehlverhalten zu tolerieren, oder dies zu erreichen erfordert nur einen geringen Mehraufwand. Je nach Architektur des Systems und seiner Funktionalität benötigt man dazu Mehrheitsentscheide über eingehende Signale von redundanten Komponenten oder Übereinstimmungsprotokolle (Beispiele in Kapitel 8).

- Die redundanten Softwarekomponenten müssen sich auf verschiedene Hardwarekomponenten verteilen lassen, damit ein Angreifer pro erfolgreichem Angriff jeweils nur eine Softwarekomponenten kontrollieren kann (siehe auch Kapitel 2.2 und Kapitel 5.3).

- Das zu schützende System besteht (zu größten Teilen) aus Komponenten, die nicht diversitär entwickelt wurden. In anderen Fällen ist das Konzept weiterhin einsetzbar, verliert jedoch den Vorteil, den Aufwand bei der Entwicklung zu senken.

- Das System hat (potentielle) Schnittstellen, über die ein Angreifer mit dem System kommunizieren kann. Es ist also nicht vollständig in sich geschlossen bzw. es kann nicht über die erwartete Betriebsdauer garantiert werden, dass es geschlossen bleibt.

Systeme mit diesen Eigenschaften finden sich oftmals in (industriellen) Steuerungssystemen. Für die funktionale Sicherheit werden die Softwarekomponenten im Bereich der Kernfunktionalität, aufgrund der getroffenen Maßnahmen, als mit ausreichender Wahrscheinlichkeit frei von systematischen Fehlern betrachtet, so dass die Redundanz üblicherweise nur gegen zufällige Fehler durch defekte Hardware oder externe Störeinflüsse ausgelegt ist.

5.3 ANNAHMEN

Für das Design und die Analyse der kryptographischen Insel wurden die folgenden Annahmen getroffen.

KOMMUNIKATIONSSYSTEM Die Anforderung an das Kommunikationssystem (siehe Kapitel 2.4) besteht seitens der kryptographischen Insel nur in zusätzlichem Bandbreitenbedarf durch die kryptographischen Signaturen. Für fehlertolerante Systeme, die bei Ausfall des Kommunikationssystems nicht in einen sicheren Zustand übergehen können, muss das Kommunikationssystem gegen DOS-Angriffe (siehe Kapitel 2.2 und Kapitel 2.4.2) geschützt sein. Dies kann durch die im Kapitel 2.4.3.1 erwähnten Maßnahmen oder durch eine geeignete Netztopologie erfolgen, die so eng vermascht ist, dass in jedem Fall – auch wenn übernommenen Knoten zeitgleich alle angeschlossene Verbindungen stören – ausreichend nicht gestörte Kommunikationsverbindungen übrig bleiben.

DIVERSITÄT Es wird angenommen, dass die diversitären Varianten des Kommunikationshypervisor keine Schwachstellen aufweisen, von der mehrere Varianten betroffen sind. Wie in Kapitel 2.5.5 dargelegt ist die Wahrscheinlichkeit eines gemeinsamen Bugs recht gering, jedoch nach aktuellen Stand der Forschung nicht auszuschließen. Es konnten keine Untersuchungen gefunden werden, bei den auch verglichen wurde, ob bei gemeinsamen (logischen) Fehlern[1] auch Angriffsvorgehen bestehen die bei beiden Varianten wirksam sind. Aufgrund der Vielzahl der Möglichkeiten, die zu einem Versagen bei Testfällen führen können, ist es wahrscheinlich, dass ein Angriffsvorgehen – selbst bei gemeinsamen Fehlern – nicht bei verschiedenen Varianten vollständig funktioniert. Da keine ausreichenden Daten für eine statistische Bewertung vorhanden sind, wird die Wahrscheinlichkeit daher für eine leichtere Nachvollziehbarkeit der Argumentationskette mit 0% angenommen (siehe dazu auch Kapitel 2.5.5 und Kapitel 9.2).

KONFIGURATION Es werden keine Konfigurationsfehler betrachtet. Das bedeutet insbesondere, dass die Verteilung der Softwarekomponenten gemäß den Regeln aus Kapitel 6.1. sichergestellt werden muss. Um Konfigurationsfehler auszuschließen wären automatische Selbsttests möglich, die jeweils bei Inbetriebnahme diese Bedingungen prüfen. Alternativ sind geeignete organisatorische Maßnahmen zu treffen.

---

[1] In den Studien werden gemeinsame Fehler in der Regel definiert, als ein Testfall bei der mehr als eine Variante versagt. Unabhängig davon ob die Ursache für das Versagen identisch oder unterschiedlich ist.

FÄHIGKEIT DES ANGREIFERS Entsprechend dem Fehlermodell (beinhaltet unter anderem die möglichen Fehler und die Fehleranzahl, bei der das System sich noch gemäß der Spezifikation verhalten muss) des zu entwerfenden selbstsichernden und fehlertolerierenden Systemen (siehe Kapitel 2.3) wird ein Angreiferverhaltensmodel erstellt:

- Der Angreifer ist nur im Besitz einer begrenzten Zahl von a Angriffsvorgehen. Als ein Angriffsvorgehen zählt der physikalische Zugang zu einer Komponente (a_p), bei der der Angreifer die Möglichkeit hat, Speicherinhalte auszulesen, die Komponente zu ersetzen oder zu zerstören oder direkt auf eine oder mehrere Schnittstellen der Komponente zur Umwelt einzuwirken. Ein anderes Angriffsvorgehen ist ein dem Angreifer bekannter Bug (a_b), der geeignet ist, die Kontrolle über eine Komponente zu übernehmen. Dabei gilt $a = a_p + a_b$.

- Es wird angenommen, dass der Angreifer nach einem erfolgreichen Angriff die vollständige Kontrolle über alle Softwarekomponenten, die auf der angegriffenen Hardwarekomponente laufen, besitzt. Dies ist eine pessimistische Annahme, die jedoch den großen Vorteil besitzt, unabhängig davon zu gelten, wie das konkrete System aufgebaut ist und ob es interne Schutzmaßnahmen gibt.

- Der Angreifer besitzt Zugang zum allgemeinen Kommunikationssystem, das die Softwarekomponenten nutzen, um untereinander Daten zu tauschen – sei es über den physikalischen Angriff auf einen Knoten ($a_p \geq 1$), über eine ungesicherte Kommunikationsschnittstelle oder über einen Angriff auf ein drittes System, welches mit demselben Kommunikationssystem verbunden ist wie das zu schützende System.

FEHLER UND ANGRIFFE Da die kryptographische Insel die Auswirkungen eines Angreifers auf (maximal) bösartige byzantinische Fehler reduziert, darf zu keinem Zeitpunkt die Zahl der genutzten Angriffsvorgehen ($a_{aktuell}$) und der aktiven Fehler ($f_{aktuell}$) größer werden[2] als die maximal zu tolerierende Fehlerzahl (f_{design}). Es gilt also $f_{design} \geq a_{aktuell} + f_{aktuell}$. Da ein fehlerhafter Systemzustand ($f \geq 1$) für die meisten Systeme die Ausnahme darstellt, ist diese Annahme solange tragbar, wie sichergestellt wird, dass ein Angreifer auch nur vergleichbare kurze Zeit (unbemerkt) im System verbleiben kann.

MONITORING DES SYSTEMS Es werden wirksame Techniken zur Erkennung von Angreifern (siehe Kapitel 2.5.4) genutzt, um bei An-

---

[2] Es gibt eine Ausnahme, wenn mehrere Fehler oder ein Angriff und ein Fehler in den gleichen Einzelfehlerbereich fallen. Dann bleibt die Zahl der betroffenen Einzelfehlerbereiche kleiner oder gleich f und das System bleibt in einen sicheren Zustand, obwohl $f_{design} < a_{aktuell} + f_{aktuell}$.

wesenheit entsprechender Anomalien zeitnah in einen sicheren Zustand zu wechseln, bis diese behoben wurden, oder es werden Maßnahmen ergriffen, um diese Anomalien zu beseitigen. Eine mögliche Maßnahme wäre eine Rekonfiguration des Systems, bei der die fehlerhaften/vom Angreifer übernommenen Komponenten ausgegliedert werden und durch neue Komponenten mit einer weiteren (bisher im System nicht genutzten) diversitär entwickelten Variante des Kommunikationshypervisors ersetzt werden.

Das Monitoring ist notwendig, damit die Annahmen über die Zahl der dem Angreifer bekannten Angriffsvorgehen und über das (nicht) zeitgleiche Auftreten von Fehlern und Angriffen gerechtfertigt werden können. Wäre keine zeitnahe Erkennung des Angreifers gegeben, hätte dieser ausreichend Zeit, im System nach weiteren ausnutzbaren Schwachstellen zu suchen. Alternativ könnte der Angreifer warten, bis genug normale zufällige Betriebsfehler aufgetreten sind, so dass $f_{design} < a_{aktuell} + f_{aktuell}$ gilt und damit das System in einen undefinierten Zustand geraten kann. Der Zeitraum, in dem der Angriff erkannt werden muss, ist abhängig von zwei Faktoren: Zum einen von der Abschätzung, welchen Zeitraum ein Angreifer benötigt, um weitere Angriffsvorgehen zu finden, zum anderen von der erwartete Rate an Betriebsfehlern. Die erwarteten Rate an Betriebsfehlern bestimmt den Zeitraum, ab dem das Risiko für die Anwendung zu groß ist, dass durch einen Fehler ($f_{aktuell}$) die Annahme $f_{design} < a_{aktuell} + f_{aktuell}$ verletzt wird, nachdem der Angreifer bereits mit a Angriffsvorgehen auf das System Einfluss genommen hat.

6

SYSTEMAUFBAU

6.1 VERTEILUNG DER SOFTWAREKOMPONENTEN ZUR ERZIELUNG DER ANGREIFERTOLERANZ

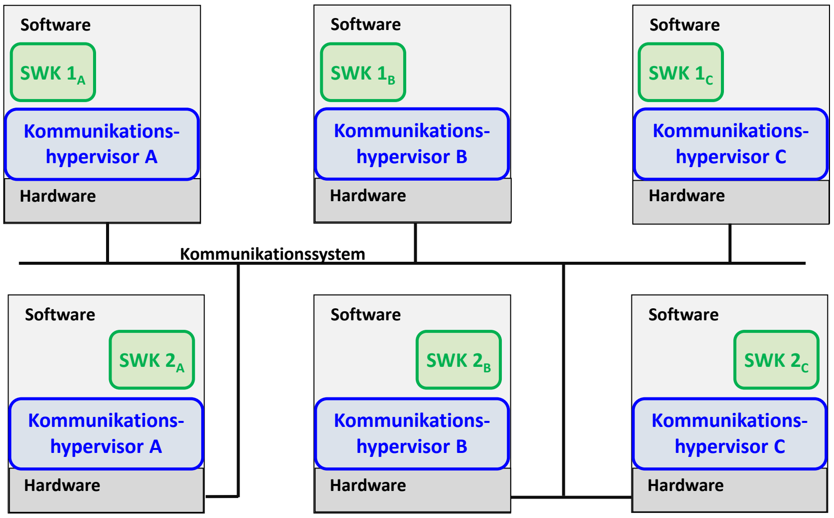
Bei der Platzierung der Softwarekomponenten ist neben den Anforderungen, die aus der Funktion resultieren[1], auch eine Zuordnung der redundantem Instanzen einer Softwarekomponente zu den diversitären Varianten des Kommunikationshypervisors festzulegen.

Gemäß des Angreiferverhaltensmodells (siehe Kapitel 5.3) sind dem Angreifer a Angriffsvorgehen bekannt. Ein Angriffsvorgehen entspricht dabei einem physikalischen Zugang, durch den ein Knoten unter Kontrolle des Angreifers gebracht werden kann oder die Kenntnis über eine Schwachstelle in einer Variante des Kommunikationshypervisor, die der Angreifer ausnutzen kann, um die Kontrolle über alle Knoten, auf denen eine Instanz des verwundbaren Kommunikationshypervisors läuft, zu erlangen. Aufgrund der Annahme, dass bei diversitär entwickelten Kommunikationshypervisoren keine gemeinsame Schwachstelle vorhanden ist (siehe Kapitel 5.3), betrifft jedes Angriffsvorgehen damit nur eine Variante des Kommunikationshypervisors. Damit kann der Angreifer maximal alle Knoten übernehmen, auf denen eine beliebige Kombination von a Varianten des Kommunikationshypervisors laufen.

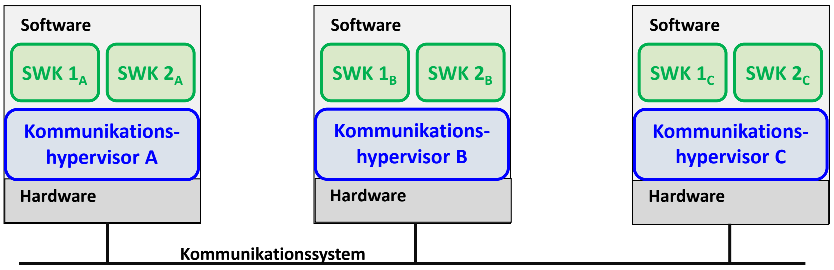
Angreifertoleranz wird daher erzielt, falls sichergestellt ist, dass auf jeder beliebigen Kombination von a Varianten des Kommunikationshypervisors maximal f Instanzen einer Softwarekomponente laufen. Für $a = f$ bedeutet dies, dass jede Instanz auf einer anderen Variante des Kommunikationshypervisors laufen muss. Der Angreifer kann danach für alle Instanzen von Softwarekomponenten, die auf den übernommen Knoten laufen, mit den dabei gewonnenen kryptographischen Schlüsseln beliebige Nachrichten mit gültigen Signaturen erzeugen. Da maximal f Instanzen einer Softwarekomponente auf den übernommenen Knoten laufen, können die vorhandenen auf f (beliebig) fehlerhafte Werte ausgelegten Fehlererkennungs- und Fehlertoleranztechniken das Verhalten des Angreifers tolerieren/erkennen. Für $a > 1$ muss die Fehlertoleranztechnik diversitär im Kommunikationshypervisor integriert sein, damit der Angreifer nicht nach dem ersten erfolgreichen Angriffsvektor Daten mit den erbeuteten gültigen Signaturen direkt an die Softwarekomponente senden kann und ei-

---

[1] z. B. Existenz bestimmter Schnittstellen zur Umwelt, Vorhandensein bestimmter Ressourcen wie Speicher oder Rechenkapazität



(a) Platzierung der Softwarekomponenten auf eigenen Knoten



(b) Platzierung der Softwarekomponenten auf gemeinsamen Knoten

Abbildung 6: Vergleich der Verteilung von Softwarekomponenten auf gemeinsame oder getrennte Knoten

ne dort vorhandene Schwachstelle in allen Instanzen ausnutzen kann (siehe Kapitel 6.2.2).

Da das Angreiferverhaltensmodell vorsieht, dass alle Instanzen eines Programms mit einem Bug (nahezu) zeitgleich angegriffen werden können, können Instanzen verschiedener Softwarekomponenten sowohl auf einem Knoten als auch auf verschiedenen Knoten mit derselben Variante des Kommunikationshypervisors laufen, ohne dass es für die Sicherheitsbewertung einen Unterschied macht. In Abbildung 6 sind zwei funktional äquivalente Systeme dargestellt, die jeweils zwei Softwarekomponenten (SWK 1 und SWK 2) haben, von denen jeweils drei Instanzen laufen. Die Instanzen sind jeweils auf die Varianten A, B und C des Kommunikationshypervisors verteilt. Gut zu erkennen ist, dass bei einem Angriff auf alle Knoten einer Variante des Kommunikationshypervisors jeweils nur eine Instanz jeder Softwarekomponente betroffen ist, unabhängig davon, ob sie gemeinsam auf einem oder verschiedenen Knoten laufen.

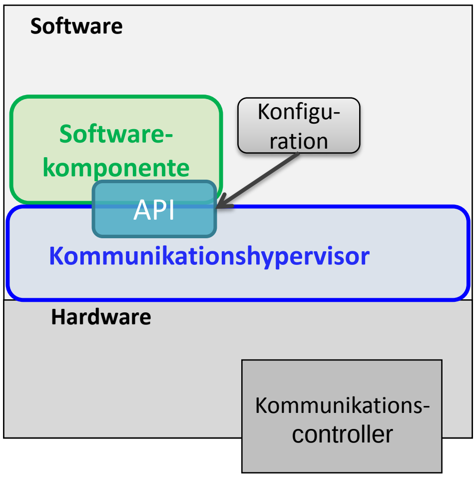


Abbildung 7: API und Konfiguration für den transparenten Kommunikationshypervisor

6.2 TRANSPARENZGRAD DES KOMMUNIKATIONSHYPERVISORS

Es werden zwei Varianten der kryptographischen Insel betrachtet. Die erste Variante geht von einem eingeschränkten Angreiferverhaltensmodell aus und kann für viele Anwendungen transparent realisiert werden. Die zweite Variante verlagert einen Teil der Fehlertoleranztechnik (Mehrheitsentscheide) in den Kommunikationshypervisor.

6.2.1 *Transparenter Kommunikationshypervisor*

Bei der Variante mit dem transparenten Kommunikationshypervisor wird eine bestehende Application Programming Interface (API) (siehe auch Kapitel 7.2 und Kapitel 7.4 für Beispiele für eine API.) für die Kommunikation durch den Kommunikationshypervisor nachgebildet. Der Kommunikationshypervisor stellt dabei nur den Maskeradeschutz sicher, in dem er ausgehende Nachrichten signiert und eingehende Nachrichten ohne gültige Signatur verwirft. Die Anwendung selbst ist für die Erkennung oder Toleranz eines fehlerhaften Eingabewertes verantwortlich. Die Konfiguration des Kommunikationshypervisor – insbesondere die Parameter zur Schlüsselverwaltung, die bestimmen, welche Signaturen von validen Sendern stammen – kann aufgrund der bestehenden API nicht durch die Anwendung zur Laufzeit erfolgen, sondern muss vorab für jede auf den Kommunikationshypervisor laufende Softwarekomponente konfiguriert werden. Der Aufbau ist in Abbildung 7 dargestellt.

Da eine bestehende API der Anwendungskomponenten angeboten wird, ist es möglich, bestehende selbstsichernde/fehlertolerante Anwendungen ohne Änderungen an den Softwarekomponenten mit Angreifertoleranz nachzurüsten. Beispiele für dieses Szenario sind bestehende Steuerungssysteme, bei denen sich nachträglich herausstellt, dass eine Isolierung der Kommunikationssysteme nicht mehr realisierbar ist, aber keine Änderungen an den Softwarekomponenten vorgenommen werden kann/soll. Das Vermeiden von Änderungen an den Softwarekomponenten kann verschiedene Gründe haben. Zum einen könnte das Risiko, bei Änderungen der Softwarekomponente sicherheitsrelevante Fehler einzuführen, als zu hoch angesehen werden, oder auch die Möglichkeit zur Änderung der Softwarekomponente nicht mehr gegeben sein[2].

Nach dem ersten erfolgreichen Angriff auf eine Variante des Kommunikationshypervisors, verfügt der Angreifer über einen gültigen kryptographischen Schlüssel. Mit diesem kryptographischen Schlüssel kann er nach dem erfolgreichen Angriff Nachrichten mit gültigen Signaturen zu erzeugen. Da Nachrichten mit gültiger Signatur beim Empfänger den (fehlerfreien) Kommunikationshypervisor passieren können, kann der Angreifer nach dem ersten erfolgreichen Angriff auf einen Kommunikationshypervisor Nachrichten an Softwarekomponenten senden, für die der erbeutete kryptographische Schlüssel einen gültigen Absender markiert. Mit der direkten Kommunikation wäre dann auch ein direkter Angriff auf Softwarekomponenten möglich.

Daher ist ein transparenter Kommunikationshypervisor nur für Systeme geeignet, wo die Anzahl der Angriffsvorgehen des Angreifers auf 1 limitiert ist ($a = 1$).

6.2.2 *Voting in dem Kommunikationshypervisor*

Um mehr als ein Angriffsvorgehen ($a \geq 1$) tolerieren/erkennen zu können, muss der Vergleich der verschiedenen Eingabewerte im Kommunikationshypervisor erfolgen. Wenn der Vergleich – beziehungsweise Mehrheitsentscheid – im Kommunikationshypervisor geschieht, kann anders als beim transparenten Kommunikationshypervisor der Angreifer mit einem gültigen kryptographischen Schlüssel nicht mehr die Anwendung direkt erreichen. Der Angreifer muss, um eine (schadhafte) Nachricht an eine Softwarekomponente zu senden, die Mehrheit der Eingangsnachrichten kontrollieren. Falls die Regeln aus Kapitel 6.1 eingehalten werden, kann der Angreifer jedoch erst dann die Mehrheit der Eingangsnachrichten kontrollieren, wenn er mehr Varianten des Kommunikationshypervisors übernommen hat, als die Fehlerannahme erlaubt ($a > f$), was den Annahmen aus Kapitel 5.3 widerspricht.

Die Softwarekomponente erhält bei dieser Variante über eine API das Ergebnis des Mehrheitsentscheids oder die Information, dass es

---

[2] Beispielsweise durch Fehlen von Werkzeugen, Know-how bei den Entwicklern oder Nicht-Verfügbarkeit des Quellcodes.

keine gültige Mehrheit gab. Die Softwarekomponente muss dazu diese API unterstützen, was bei neu zu entwickelnden Anwendungen sogar Vorteile bieten kann, da die Anwendungslogik nicht selber die Logik für den Mehrheitsentscheid implementieren muss. Außerdem kann die API auch so definiert werden, dass die Softwarekomponenten den Kommunikationshypervisor konfiguriert (erlaubte Absender in der Form von kryptographischen Schlüsseln, zu verwendende Technik für die Entscheidung). Bestehende Anwendungen (die selber Mehrheitsentscheide implementiert haben) für das Voting im Kommunikationshypervisor anzupassen ist potentiell sehr aufwendig, da unter Umständen tiefgreifende Änderungen notwendig sind.

Die Auswahl der Eingabewerte im Kommunikationshypervisor ist sowohl aus Gründen der Wiederverwendbarkeit als auch um die Komplexität gering zu halten[3] auf verbreitete generische Ansätze (zum Beispiel Mehrheitsentscheid, Medianentscheid oder Intervallentscheid [48]) beschränkt. Bestimmte Fehlererkennungs- oder Fehlermaskierungstechniken, die auf einen speziellen Anwendungsfall zugeschnitten sind[4], lassen sich daher nicht nutzen.

6.3 SCHNITTSTELLEN ZUR UMWELT

Die Schnittstellen des Systems zur Umwelt bedürfen genauer Planung und Analyse, da Schnittstellen zur Umwelt die in Kapitel 5.1 beschriebene Idee einer Isolierung des Systems vor nicht vertrauenswürdigen Systemen – durch die ein Angriff erfolgen könnte – unterlaufen können. Man muss daher für jede Schnittstelle prüfen, inwiefern ein Angreifer darüber das System angreifen kann und welche Gegenmaßnahmen notwendig sind. Problematisch sind dabei grundsätzlich Schnittstellen, die von (potentiell) unvertrauenswürdigen Quellen direkt eine Softwarekomponente erreichen, also Schnittstellen, die von der Umwelt auf das System einwirken. Für einige Arten von Schnittstellen lassen sich generische Aussagen treffen, während für andere eine Betrachtung im Einzelfall notwendig ist. Diese Unterscheidung wird in den beiden folgenden Kapiteln genauer analysiert.

6.3.1 *Analoge Schnittstellen*

Im Rahmen dieser Arbeit werden als analoge Schnittstellen alle Schnittstellen bezeichnet, die dem System Informationen über genau einen

---

[3] eine hohe Komplexität erhöht nach vielen Studien auch das Risiko für Bugs/Angriffsvorgehen. Siehe auch Kapitel 2.5.3.

[4] Ein Beispiel für Fehlererkennung wäre ein zu steuernder chemischer Prozess, bei dem die Mengen der hinzugefügten Stoffe und entnommenen Stoffe sowie der Füllstand überwacht werden. Mit dem Wissen über die Dichte der hinzugefügten und entstehenden Stoffe lässt sich überprüfen, ob die Masse konstant ist. Wird mehr Masse hinzugefügt als entnommen (oder andersherum), ohne dass sich der Füllstand ändert, muss ein Sensor fehlerhaft sein.

Parameter liefern, dessen Wert sich alleine über eine physikalische Größe (Spannung, Widerstand o.Ä.) ausdrückt. Ein direkter Angriff auf die Software über einen Exploit (siehe Kapitel 2.5.3) kann aufgrund der limitierten Informationsmenge, die ein Angreifer über eine analoge Schnittstelle bestimmen kann (und dessen interne Repräsentation vom Angreifer nicht kontrolliert werden kann), als ausreichend unwahrscheinlich angesehen werden.

Ein Angreifer, der eine analoge Schnittstelle kontrolliert, kann dadurch den Wert des Parameters bestimmen. Da der Parameter einen Einfluss auf den Programmablauf einer Softwarekomponente hat, ist dadurch prinzipiell die Möglichkeit gegeben, das Verhalten der Softwarekomponente(n), die diesen Parameter auslesen, zu beeinflussen, indem er einen anderen Zustand der Umwelt vortäuscht. Diese Art von Angriffen entspricht dem Fehlerbild von beliebigen Wertefehlern.

Angriffe über eine analoge Schnittstelle werden also toleriert, falls das System so entworfen ist, dass bei a Komponenten[5] beliebige Wertefehler toleriert werden. Dies ist beispielsweise bei Sensoren der Fall, falls alle Sensoren jeweils an eine andere Instanz der dazugehörigen Softwarekomponente angeschlossen sind, die auf einer jeweils unterschiedlichen diversitären Variante des Kommunikationshypervisors laufen und die Anwendung a fehlerhafte Eingabewerte toleriert[6]. Wichtig hierbei ist, dass keine Annahmen über das Verhalten des analogen Signals im Fehlerfall getroffen werden. Diese Annahmen könnte der Angreifer gezielt verletzen.

6.3.2 *Protokollbasierte Schnittstellen*

Bei protokollbasierten Schnittstellen besteht in der Regel die Möglichkeit, die übertragenen Daten recht frei zu definieren. Dadurch ist es möglich, dass der Angreifer mittels eines Exploits die Kontrolle über die empfangende Softwarekomponente übernimmt. Im Folgenden werden protokollbasierte Schnittstellen betrachtet, die direkt an die Softwarekomponenten Daten liefern können. Dies wird in Abbildung 8 dargestellt. Das Kommunikationssystem, welches auch protokollbasiert ist, ist im Folgenden nicht Untersuchungsgegenstand, da Nachrichten darüber vom Kommunikationshypervisor gefiltert werden. Es werden also protokollbasierte Schnittstellen, die nicht den Schutz der kryptographischen Insel unterliegen, betrachtet. Insbesondere bei einer Nachrüstung (siehe auch Kapitel 6.2.1) der kryptographischen Insel können einige Schnittstellen nicht die notwendige Bandbreite für die benötigten Signaturen bieten und daher ungesichert bleiben.

---

[5] Gemäß Kapitel 5.3 muss die Fehlertoleranz mindestens gleich der Angreifertoleranz sein.

[6] Zur Fehlererkennung also mindestens $a + 1$ Sensoren existieren und zur Fehlertoleranz mindestens $2 * a + 1$ Sensoren existieren.

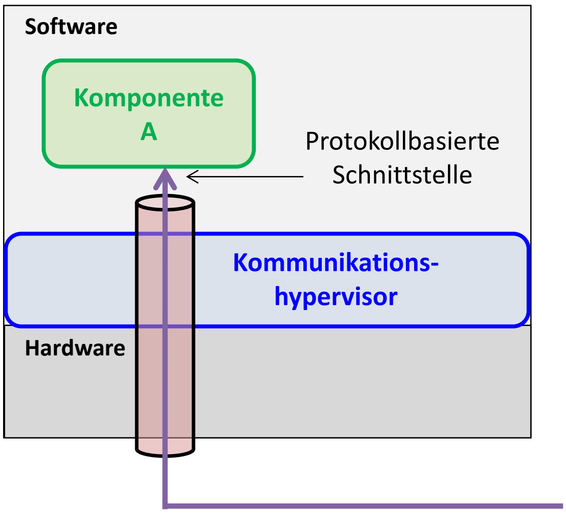


Abbildung 8: Direkter Informationsfluss zu einer Softwarekomponente

Wenn die protokollbasierte Schnittstelle mit nur einem Knoten des Systems verbunden ist[7], kann der Angreifer über jeden physikalischen Zugriff auf eine protokollbasierte Schnittstelle nur einen Knoten übernehmen. Gemäß der Annahme in Kapitel 5.3 (volle Kontrolle nach erfolgter Übernahme unabhängig vom der betroffenen Softwarekomponente im Knoten) ist in diesem Fall kein abweichendes Verhalten im Vergleich zu dem Fall möglich, dass der Angriff über einen Bug im Kommunikationshypervisor erfolgt.

Eine protokollbasierte Schnittstelle zur nicht vertrauenswürdigen Umwelt, die mit mehreren Knoten (z. B. allen Instanzen einer Softwarekomponente) verbunden ist, widerspricht dem Kerngedanken der Isolation, da damit alle Instanzen einer Softwarekomponente direkt angreifbar würden und damit das Systemverhalten vom Angreifer nach einem Angriffsvorgehen bereits kontrolliert werden kann. Dies ist daher sofern möglich zu vermeiden. Für den Fall, dass aus der Umwelt über eine protokollbasierte Schnittstelle Einfluss auf das (Gesamt-) System genommen werden muss (zum Beispiel weil die Informationen zu umfangreich für analoge Schnittstellen sind), ist diese sehr sorgfältig abzusichern, um trotz der Verletzung ein ausreichendes Schutzniveau sicherzustellen. Ein möglicher Ansatz ist eine mehrstufige Kaskade von diversitär entwickelten Firewalls, welche jeweils die übertragenen Informationen syntaktisch und semantisch prüfen und dazu noch die Repräsentationsform ändern. Ein Beispiel ist in Abbildung 9 dargestellt. Die Varianten des Kommunikationshypervisors werden dabei für den Maskeradeschutz zwischen den Firewallvarianten genutzt.

Dies kann es dem Angreifer erschweren, einen Exploit in der gewünschten Form direkt an die Softwarekomponenten zu senden. Die-

---

[7] Ein Beispiel dafür wäre ein Bussystem wie I2C oder LIN, an den verschiedene Sensoren angeschlossen sind.

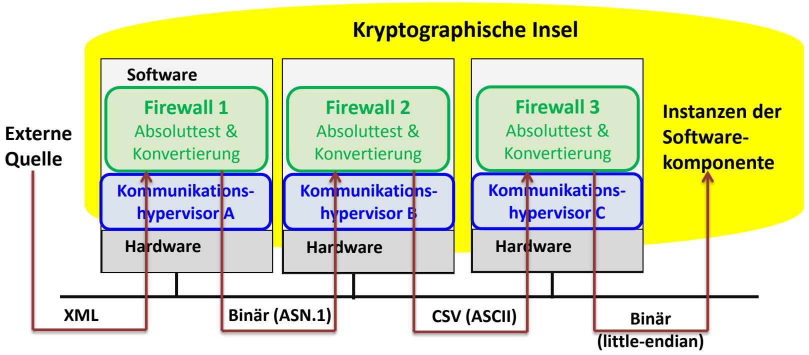


Abbildung 9: Beispiel für mehrstufigen diversitärer Firewallaufbau für externe Eingaben

se Erschwernis ist jedoch nur eine Reduktion der Wahrscheinlichkeit, dass es dem Angreifer gelingt. Daher ist zu beachten, dass bei der Existenz einer solchen protokollbasierten Schnittstelle keine Toleranz von a Angriffsvorgehen mehr garantiert werden kann.

7

EIGENSCHAFTEN DES KOMMUNIKATIONSHYPERVISORS

Für jede der mögliche Variante der kryptographischen Insel (siehe Kapitel 6.2), besitzt der Kommunikationshypervisor, der die kryptographische Insel implementiert, die gleichen Kerneigenschaften. Die Varianten unterschieden sich nur in der API zu den Softwarekomponenten und der implementierten Funktionalität[1].

7.1 UMFANG DES KOMMUNIKATIONSHYPERVISORS

Um einen Schutz gegen Angreifer gewährleisten zu können, muss der Kommunikationshypervisor alle Elemente beinhalten, die Schwachstellen enthalten könnten, welche über das Netzwerk ausgenutzt werden können. Der Kommunikationshypervisor muss daher alle Teile beinhalten, die Daten verarbeiten, die über das Netzwerk empfangen wurden, bevor die Daten den Kommunikationshypervisor verlassen. Zur Visualisierung dient die Abbildung 10. Der Kommunikationshypervisor beinhaltet mindestens die folgenden Funktionen:

- Die Verwaltung aller Nachrichtenpuffer bis zur Übergabe der geprüften Daten/Nachrichten an die Softwarekomponente, um die Auswirkungen von Pufferüberläufen und eventuell vorhandenen Race-Conditions in der Verwaltung der Puffer einzugrenzen.

- Die in Software ausgelagerten Teile des Kommunikationssystems (zum Beispiel TCP/IP), um Schwächen in der Implementierung auf eine Kommunikationshypervisorvariante zu beschränken. Die Schwächen können dabei sowohl Pufferüberläufe sein als auch Fehler in internen State-Machines, durch die entweder unzulässige Zustandsübergänge provoziert werden (z. B. bestimmte Prüfschritte übersprungen werden) oder undefinierte Zustände eingenommen werden und damit die Funktionalität nicht mehr sichergestellt ist.

- Die Implementierung der Algorithmen der kryptographischen Insel, um Schwächen in der Implementierung auf eine Kommunikationshypervisorvariante zu beschränken. Mögliche Schwächen sind insbesondere – wie bei der Software für das Kommunikationssystem – Pufferüberläufe und Fehler in internen State-Machines, durch die entweder unzulässige Zustandsübergänge

---

[1] Nur Maskeradeschutz oder auch Mehrheitsentscheider.

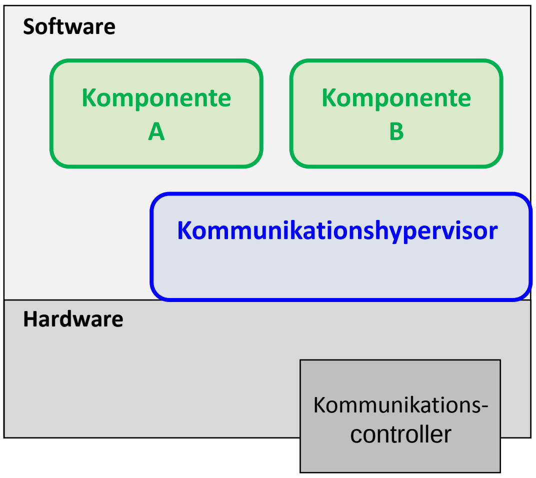


Abbildung 10: Umfang des Kommunikationshypervisors

provoziert werden (z. B. bestimmte Prüfschritte übersprungen werden) oder undefinierte Zustände eingenommen werden und damit die Funktionalität nicht mehr sichergestellt ist.

Diese Abgrenzung ist in der Praxis jedoch nicht immer so leicht zu treffen. Der in Abbildung 11 gezeigte schematische Aufbau des Linux Kernels verdeutlicht dies. Falls der Linux Kernel eine Grundlage für eine kryptographische Insel bieten sollte, müsste man analysieren, welche Teile von einem Angreifer über das Netzwerk angreifbar wären. Recht eindeutig wäre der grün hervorgehobene Teil, der die Netzwerkkommunikation regelt, da dieser Teil direkt mit den potentiell schädlichen Daten interagiert. Einige Teile (z. B. die Dateisysteme) müssten nicht diversitär entwickelt werden, da gemäß dem Angreiferverhaltensmodells dort keine Interaktionen stattfinden. Es gibt jedoch Module – wie der allgemeine Hardwarezugriff, Bus-Treiber und die Speicherverwaltung – die von dem Netzwerkstack genutzt werden. Dadurch können diese Bereiche potentiell auch durch Daten eines Angreifers beeinflusst werden. Es ist also im Einzelfall zu prüfen, wo genau die Grenze zu ziehen ist. Dabei kann auch eine Abwägung zwischen Sicherheit und Aufwand vorgenommen werden[2].

7.2 REALISIERUNG DER DIVERSITÄT

Um Fehler im Kommunikationshypervisor zu tolerieren, ist es notwendig, dass es von dem Kommunikationshypervisor diversitär entwickelte Varianten gibt. Der in Kapitel 7.1 in Abbildung 10 skizzierte Umfang kann in der Praxis auf vier unterschiedlichen Wegen erreicht werden. Diese werden in Abbildung 12 dargestellt. Die Funktionalität reicht dabei vom beschriebenen Minimalumfang bis hin zu vollstän-

---

[2] Beispielsweise implementiert man den Bus-Treiber nicht diversitär, basierend auf der Annahme, dass die Netzwerkkarte auf dem PCI-Bus sich spezifikationskonform

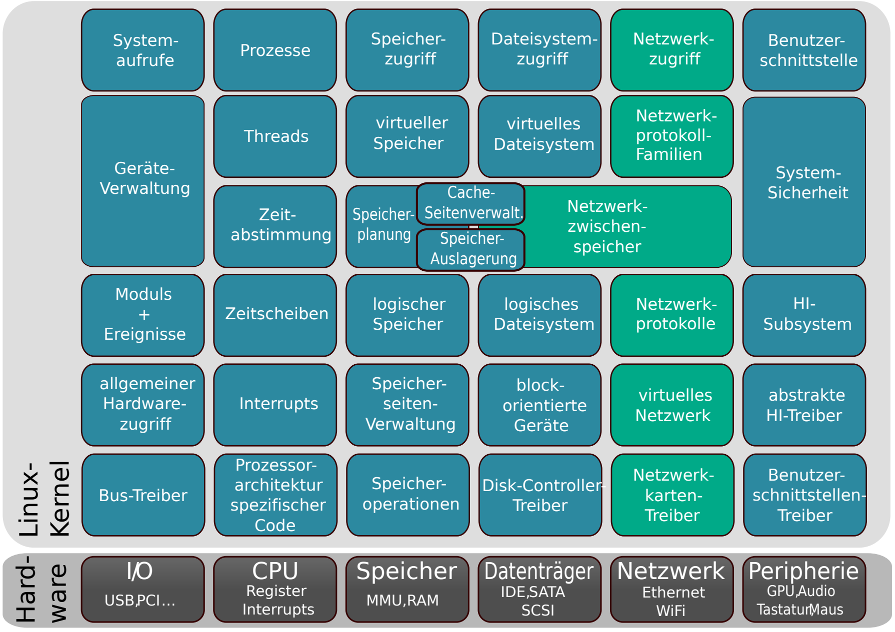
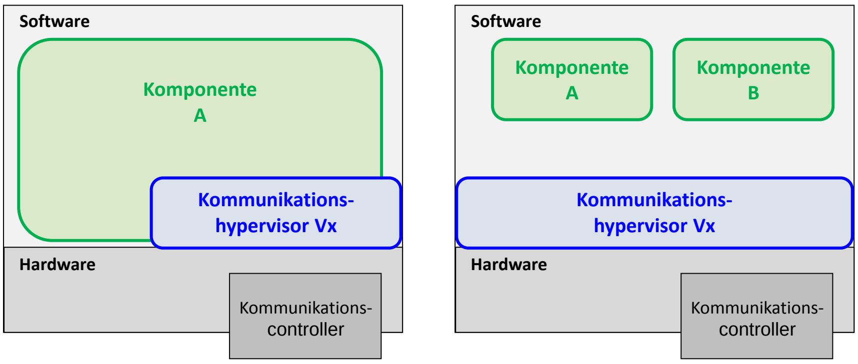
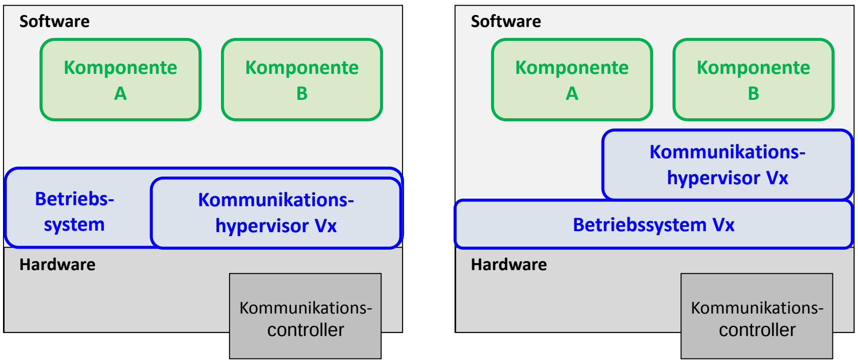


Abbildung 11: Schematischer Aufbau des Linux Kernels nach [146]



(a) Isolierter Kommunikationshypervisor

(b) Vollständiges Betriebssystem



(c) Betriebssystemkern mit diversitären Erweiterungen

(d) Diversitäre Betriebssysteme mit diversitärer Kommunikationsbibliothek

Abbildung 12: Mögliche Realisierungen des Kommunikationshypervisors digen Betriebssystemen und von eher monolithischen Ansätzen zu ausgeprägt modularen Varianten. Eine genauere Bewertung über den Aufwand und die Sicherheit des Kommunikationshypervisors findet sich in Kapitel 9.2.

ISOLIERTER KOMMUNIKATIONSHYPERVISOR Bei einem isolierten Kommunikationshypervisor wie in Abbildung 12a wird für die Anwendungskomponente (in der Abbildung Komponente A genannt) nur ein diversitär entwickelter Kommunikationshypervisor bereitgestellt. Alle Varianten (V_x mit $x \in \{V_1, \ldots, V_n\}$) besitzen zur Anwendung ein einheitliches Interface[3] und decken nur die in Kapitel 7.1 beschriebene Funktionalität ab. Der Vorteil dieses Ansatzes liegt darin, dass der Anwendung weiterhin möglichst weitreichende Zugriffsmöglichkeiten direkt auf die Hardware gegeben wird, was für einige Anwendungsszenarien vorteilhaft ist.

KOMMUNIKATIONSHYPERVISOR ALS VOLLSTÄNDIGES BETRIEBSSYSTEM Wird der Kommunikationshypervisor so erweitert, dass jede Variante entsprechend Abbildung 12b ein vollständiges Betriebssystem bildet, bietet das die folgenden Vorteile. Durch die vollständig diversitär entwickelte Softwareschicht zwischen Hardware und Anwendungskomponente, die das Betriebssystem bildet, ist das Risiko ausgeschlossen, dass vom Entwickler über den Kommunikationskanal manipulierbare Funktionen übersehen und daher nicht dem Kommunikationshypervisor zugeordnet wurden[4]. Zusätzlich besteht eine stärkere Abstrahierung von der Hardware, welche die Entwicklung der Anwendungskomponenten vereinfachen kann. Bei ausreichender Funktionalität des Betriebssystems können auch auf einem Knoten mehrere verschiedene Anwendungskomponenten genutzt werden. Als Nachteil ist zu sehen, dass die Entwicklung der Kommunikationshypervisorvarianten aufgrund der deutlich erweiterten Funktionalität komplexer und damit teurer wird.

BETRIEBSSYSTEM MIT INTEGRIERTEN DIVERSITÄREN KOMMUNIKATIONSHYPERVISOR Der Kommunikationshypervisor kann auch, wie in Abbildung 12c dargestellt, in ein Betriebssystem integriert werden. Dabei werden die für die Absicherung der Kommunikation relevanten Teile diversitär entwickelt, während die restlichen Teile des Betriebssystems nur einmal entwickelt werden. Diese Lösung hat den Vorteil, dass sie den Aufwand im Vergleich zum vollständig diversi-

---

verhält und der Angreifer erst in den höheren Layern in denen die Daten inhaltlich bearbeitet werden Fehler ausnutzen kann.

[3] Ein transparenter Kommunikationshypervisor gemäß Kapitel 6.2.1 ist bei solchen hardwarenahen Softwarekomponenten schwierig, da es selten einheitliche Schnittstellen gibt, die ersetzt werden können.

[4] Diese Funktionen könnten bei Fehlern in der Implementierung von einem Angreifer ausgenutzt werden.

tär entwickelten Betriebssystem reduziert und trotzdem die Vorteile der abstrahierten Hardware bietet. Sie kann auch genutzt werden, um bestehende Betriebssysteme mit der notwendigen Funktionalität für die Unterstützung der kryptographischen Insel zu erweitern. Durch eine bestehende API, die das Betriebssystem den Softwarekomponenten bietet, ist es möglich, einen transparenten Kommunikationshypervisor gemäß Kapitel 6.2.1 als zusätzlichen Schutzmechanismus zu nutzen.

DIVERSITÄRES BETRIEBSSYSTEM MIT ZUSÄTZLICHER DIVERSITÄRER KOMMUNIKATIONSBIBLIOTHEK Bei der in Abbildung 12d dargestellten Variante wird für jede Variante ein diversitär entwickeltes Betriebssystem genutzt und zusätzlich ein in der Funktionalität reduzierter diversitär entwickelter Kommunikationshypervisor. Das diversitär entwickelte Betriebssystem übernimmt die Verwaltung der Nachrichtenpuffer und der in Software realisierten Teile des Kommunikationssystems. Der reduzierte Kommunikationshypervisor kann als Softwarebibliothek realisiert werden, welche den Maskeradeschutz (und die Mehrheitsentscheide bei der Variante mit integriertem Voting) realisiert.

Trotz des auf den ersten Blick mindestens gleichen Aufwands wie die Lösung, den Kommunikationshypervisor als vollständiges Betriebssystem zu entwickeln, kann diese Variante in der Praxis deutliche Vorteile aufweisen. Diversitäre Betriebssystemvarianten sind für verschiedene Anwendungsbereiche als COTS erhältlich und daher im Vergleich zu eigenen Entwicklungen sehr preiswert. Es ist dabei jedoch sicherzustellen, dass die unterschiedlichen Betriebssystemvarianten ausreichend diversitär sind und nicht in relevanten Teilen gemeinsame Bestandteile haben. Beispiele wären gemeinsam genutzte Softwarebibliotheken. Für Standardanwendungen, welche die Portable Operating System Interface (POSIX) API nutzen, könnten als Betriebssysteme Windows, Linux und FreeBSD eingesetzt werden. Eine Untersuchung [64] hat verschiedene POSIX kompatible Systeme verglichen. Die Wahrscheinlichkeit für einen gemeinsamen Fehler war bei den untersuchten Systemkombinationen signifikant hoch, ohne jedoch Aussagen zu treffen, ob die Fehler vergleichbar auszunutzen waren (Siehe auch Kapitel 2.5.5). Für Anwendungen im automotive Umfeld existieren ebenfalls verschiedene Hersteller [11, 53, 149] für die AUTOSAR Plattform.

7.3 MASKERADESCHUTZ

Damit die Anwendungskomponenten vor Nachrichten von nicht autorisierten Sendern geschützt werden und damit auch vor netzwerkbasierten Angriffen, ist ein effektiver Maskeradeschutz notwendig. Bei den meisten Kommunikationssystemen ist ein sicherer Maskeradeschutz nicht oder nur mit in der Praxis nicht realisierbarem Aufwand (zum Beispiel sogenannte Guardians bei FlexRay) zu erzielen (siehe auch Kapitel 2.2 und Kapitel 2.4.2). Daher muss mittels kryptographisch sicherer Verfahren der Maskeradeschutz Ende-zu-Ende sichergestellt werden. Die dazu möglichen Techniken werden in Kapitel 9.1 diskutiert.

Für die Wirksamkeit der Implementierung ist neben dem gewählten kryptographischen Verfahren auch eine möglichst fehlerfreie Implementierung des Verfahrens notwendig. Standarisierte Verfahren (wie zum Beispiel der „Authentication Header“ von IPsec [91] für IP basierte Kommunikationssysteme) bieten den Vorteil, dass gut getestete COTS erhältlich sind. Andererseits sind je nach Anwendungsdomäne oftmals Anforderungen zu erfüllen (zum Beispiel harte Echtzeit oder sehr limitierte Ressourcen), die bei standarisierten generischen Ansätzen nicht ausreichend berücksichtigt werden.

7.4 WIEDERVERWENDBARKEIT

Der Kommunikationshypervisor sollte mit einer standarisierten API/ Schnittstelle zur Anwendung ausgestattet werden. Durch standarisierte Schnittstellen ist eine Nutzung für verschiedene Anwendungen möglich. Dadurch lassen sich die notwendigen Entwicklungskosten auf verschiedene Anwendungen verteilen, die sogar aus verschiedenen Domänen stammen können.

Um eine besonders leichte Integration für den Anwendungsentwickler zu ermöglichen, bietet es sich an, bereits in der jeweiligen Domäne, in der die Zielgruppe liegt, verwendete Schnittstellen zu nutzen. Beispiele für existierende Schnittstellendefinitionen finden sich in vielen Bereichen: Sowohl herstellerübergreifende Schnittstellendefinitionen wie das AUTOSAR [11] und POSIX als auch herstellerspezifische wie die SIMATIC S7-Serie [139] von Siemens. Diese ließen sich bei der in Kapitel 6.2.1 vorgestellten transparenten Variante des Kommunikationshypervisors nutzen.

8

BEISPIELE

Als Beispiele wurden zwei mögliche Systemaufbauten gewählt, die mit aufsteigender Komplexität verschiedene Möglichkeiten skizzieren. Sie stellen jedoch nur eine sehr kleine Teilmenge der möglichen Systemaufbauten dar.

8.1 ERKENNUNG EINES FEHLERS/ANGRIFFSVORGEHENS

Das Szenario besteht aus einem System zur Notabschaltung mit einem schnell erreichbaren (funktional) sicheren Zustand (durch Abschaltung der Energieversorgung)[1], das durch eine Menge von Sensoren ($S_1, \ldots, S_n$) überwacht wird. Die Sensoren liefern jeweils einen booleschen Wert der angibt, ob sich das überwachte Teilsystem in einem sicheren Zustand befindet. Um einen Sensorfehler zu tolerieren, gibt es von jedem Sensor jeweils zwei Exemplare (S_{xA} und S_{xB}), die die gleiche Funktion erfüllen. Damit bei einem Ausfall eines Knotens nicht beide Exemplare eines Sensors in Mitleidenschaft gezogen werden, sind diese an jeweils unterschiedlichen Knoten angebunden. Die Werte der Sensoren werden an die beiden Softwarekomponenten gesendet, die den Aktuator für die Notabschaltung ansteuern. Die Ansteuerung der Aktuatoren ist ebenfalls auf zwei Knoten verteilt, der Strom fließt seriell durch beide Aktuatoren, so dass ein Knoten ausreicht, um die Energieversorgung zu unterbrechen. Die beiden die Aktuatoren kontrollierenden Softwarekomponenten verknüpfen alle eingehenden Signale mit einem logischen „Und“ und werten ausbleibende Signale als „Falsch“. Wenn die Und-Verknüpfung nicht „Wahr“ ergibt, wird der Aktuator im fehlerfreien Fall angewiesen, die Energieversorgung zu trennen. Der logische Aufbau ist in Abbildung 13 dargestellt. Das System ist damit fehlererkennend und selbstsichernd für $f = 1$ Ausfälle. Die Schnittstellen zur Umwelt bestehen aus den Sensoren, den Aktuatoren und dem Kommunikationssystem.

Durch die Begrenzung auf $f = 1$ Fehler ist auch die Anzahl der maximal tolerierbaren Angriffsvorgehen auf $a = 1$ begrenzt. In dem Beispiel werden zwei Varianten eines Kommunikationshypervisors mit integriertem Kommunikationshypervisor genutzt. Das resultierende System kann den in Abbildung 14 skizzierten Aufbau besitzen. Die Daten der unterschiedlichen Exemplare der Sensoren werden dabei von Softwarekomponenten, die auf unterschiedlichen Varianten des Kommunikationshypervisors laufen, an die Softwarekomponenten, die die Aktoren ansteuern, übertragen. Diese Softwarekomponenten

---

[1] z.B. Automatisierungstechnik, Zugsicherung

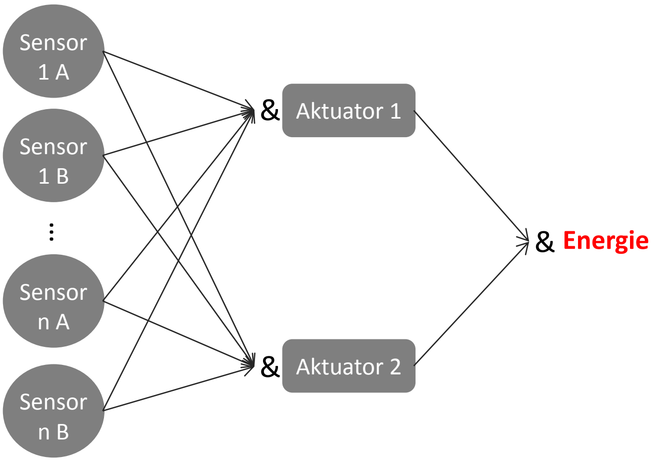


Abbildung 13: Logischer Aufbau der Notabschaltung (Ein Angriffsvorgehen erkennendes, selbstsicherndes System)

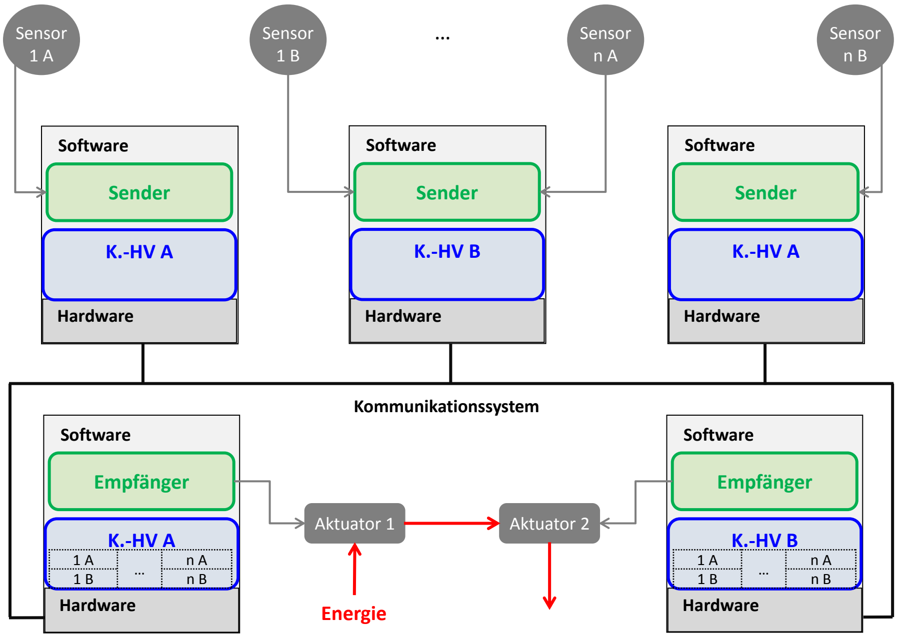


Abbildung 14: Ein Angriffssvorgehen erkennendes, selbstsicherndes System zur Notabschaltung

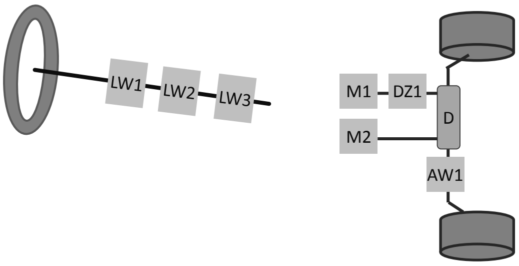


Abbildung 15: Mechanische Struktur „Steer by Wire“ System

laufen auf den beiden unterschiedlichen Varianten des Kommunikationshypervisors. Damit sind die Regeln zur Platzierung gemäß Kapitel 6.1 eingehalten. Ein Angreifer kann, entsprechend dem Modell des Angreiferverhaltens (siehe Kapitel 5.3), mit einem Angriffsvorgehen maximal alle Knoten mit derselben Variante des Kommunikationshypervisors übernehmen. Das bedeutet in diesem System maximal die Kontrolle über die Werte von jeweils einem Exemplar jedes Sensors und der Ansteuerung maximal eines Aktuators. Ein Abschalten des Systems durch Einwirken auf den Aktuator ist für die funktionale Sicherheit nicht relevant, da in dem Fall ein sicherer Zustand aufgesucht wird. Das Unterlassen einer Notabschaltung kann der Angreifer jedoch nicht hervorrufen, da durch die redundanten Sensoren – welche an Knoten mit der nicht betroffenen Variante des Kommunikationshypervisors angeschlossen sind – der nicht betroffene Kommunikationshypervisor der Softwarekomponente, welche den Aktuator ansteuert, in dem Fall als Ergebnis mitteilt, dass keine Mehrheit vorhanden ist. Dadurch evaluiert die Und-Verknüpfung zu „Falsch“ und der sichere Zustand wird aufgesucht.

8.2 TOLERIEREN EINES FEHLERS/ANGRIFFSVORGEHENS

Das Szenario besteht aus einem „Steer by Wire“ System, dessen Design im Fahrbetrieb einen Fehler tolerieren soll, da ansonsten ein Fehlverhalten zu einer gefährlichen Fahrsituation führen kann. Ein Angreifer sollte ebenfalls toleriert werden, da aufgrund der im Fahrzeug vorhandenen Kommunikationsschnittstellen keine Isolierung des Kommunikationssystems garantiert werden kann. Der geplante mechanische Aufbau ist in Abbildung 15 dargestellt und besteht aus einem Lenkrad, dessen Bewegungen von drei redundanten Lenkwinkelsensoren (LW1 bis LW3) erfasst werden. Die Ansteuerung der Räder geschieht über zwei Motoren (M1 und M2) mit integrierten Frequenzumrichtern über ein Differential (D). Die Drehzahl und Drehrichtung eines Motors wird über einen vom Motor unabhängigen Sensor (DZ1) erfasst. Die absolute Position des Lenkgestänges – und damit der Räder – wird über einen Sensor (AW1) erfasst. Die Motoren sind so ausgelegt, dass einer ausreichend ist, um die Funktion aufrecht

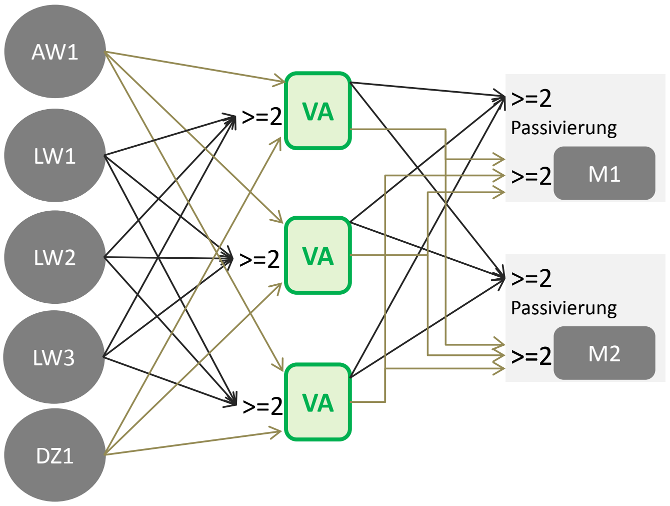
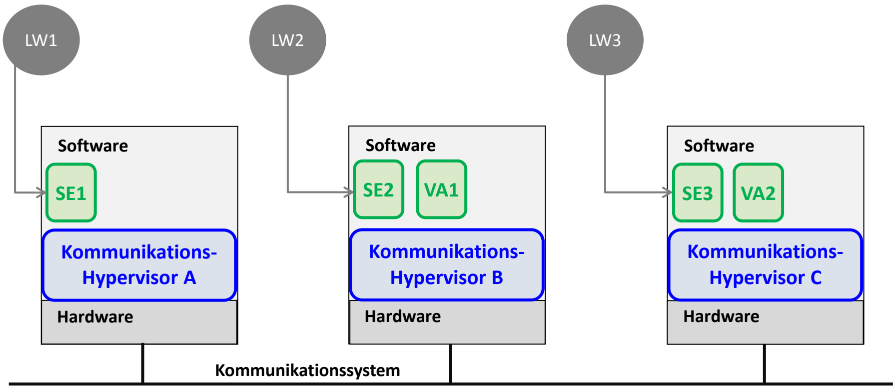


Abbildung 16: Logische Struktur „Steer by Wire“ System

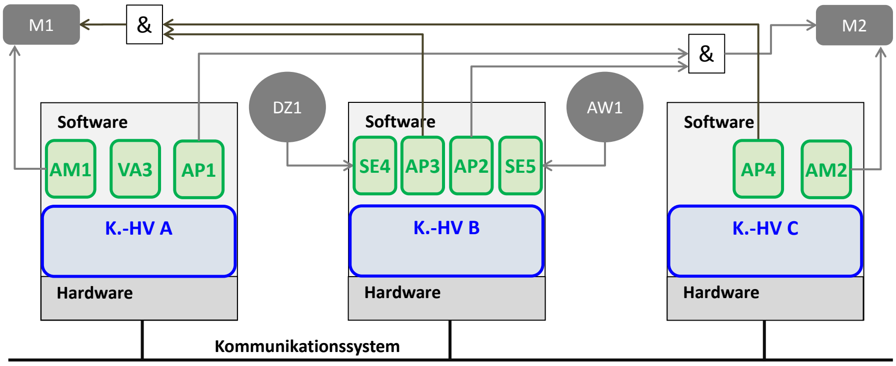
zu erhalten, und dass ein Motor nach der Passivierung blockiert und sich nicht weiterdreht, um die Funktionalität sicherzustellen. Bei dem System wird außerdem angenommen, dass ein fehlerhafter absoluter Positionswert zum Beginn des Fahrzyklus toleriert werden kann. Bei zu großen Diskrepanzen zwischen Radeinschlag und Lenkradposition wird angenommen, dass der Fahrer dies bei Fahrtbeginn bemerkt, bevor es zu gefährlichen Fahrsituationen kommen kann.

Die Fehlererkennung und Fehlerbehandlung während des Fahrzyklus wird für die Lenkwinkelsensoren über einen Mehrheitsentscheid in den Verarbeitungskomponten (VA1 bis VA3) realisiert, die mit den Lenkwinkelwerten die Sollwerte für die Motoren bestimmen. Bei den Motoren M1 und M2 wird in den Verarbeitungskomponenten (VA1 bis VA3) ein Soll-Ist-Vergleich zwischen den ausgesendeten Werten und den Werten der Sensoren DZ1 und AW1 gemacht. Weicht die von Sensor DZ1 gemeldete Drehzahl signifikant vom Sollwert für den Motor M1 ab, so ist entweder der Motor M1 oder der Sensor DZ1 defekt. Aufgrund der 1-Fehler-Annahme kann in beiden Fällen der Motor M1 passiviert werden. Weicht die absolute Positionsänderung des Lenkgestänges am Sensor AW1 von den erwarteten Wert ab, so ist entweder Motor M1, Motor M2 oder der Sensor AW1 defekt. Da ein Defekt von Motor M1 über DZ1 geprüft werden kann, kann bei Fehlerfreiheit von Motor M1 der Motor M2 passiviert werden, unabhängig davon, ob der Fehler in M2 oder AW1 liegt. Der logische Aufbau des verteilten Steuerungssystems ist in Abbildung 16 dargestellt. Alle Informationen, die über das Kommunikationssystem übertragen werden, werden periodisch gesendet, so dass jede Softwarekomponenten Mehrheitsentscheide über eingehende Nachrichten durchführen kann.

Die Verteilung der Funktionen auf verschiedene Softwarekomponenten, Knoten und Varianten des Kommunikationshypervisors ist



(a) Knoten und Softwarekomponenten am Lenkrad



(b) Knoten und Softwarekomponenten an den Motoren

Abbildung 17: Verteilung der Softwarekomponenten für ein „Steer by Wire“ System

in Abbildung 17 dargestellt. Das System besteht aus jeweils drei Knoten am Lenkrad und drei Knoten an den Motoren. Die Sensoren sind jeweils einer kleinen Softwarekomponente (SE1 bis SE5) zugeordnet, die den Sensorwert ausliest und an die Softwarekomponenten zur Verarbeitung (VA1 bis VA3) sendet. Für jeden Motor gibt es eine Softwarekomponente, welche über die Sollwerte der Verarbeitungskomponenten einen Mehrheitsentscheid durchführt und mit dem Ergebnis den Motor ansteuert. Um die Passivierung ohne einen vierten Knoten (und vierte Variante des Kommunikationshypervisors) zu realisieren[2], besteht diese abweichend vom logischen Aufbau, aus zwei Softwarekomponenten für jeden Motor (AP1 und AP2 oder AP3 und AP4), die jeweils einen Mehrheitsentscheid über die eingehenden Daten von VA1 bis VA3 durchführen und deren Ergebnisse mit einem logischen UND verknüpft werden. Das logische UND für die Passivierung wird durch ein logisches ODER in der Energieversorgung – zwei parallele Relais, welche die Energieversorgung gemeinsam kap-

---

[2] Die notwendig wären, um eine ausreichende Unabhängigkeit zu haben, falls man die Passivierung wie die Motorsteuerung in einer Softwarekomponente pro Motor realisieren wollte.

pen können – realisiert. Die Softwarekomponenten für die Sensoren und Aktuatoren (SE1 bis SE5, AM1 und AM2, AP1 bis AP4) laufen jeweils auf den Knoten, der mit der jeweiligen Schnittstelle verbunden ist. Die Softwarekomponenten zur Verarbeitung (VA1 bis VA3) können frei platziert werden, sofern die Platzierungsregel aus Kapitel 6.1 eingehalten wird, also jede Instanz auf einer anderen Variante des Kommunikationshypervisors. Die kryptographische Insel wird dabei durch einen transparenten Kommunikationshypervisor in drei Varianten realisiert.

Dass ein erfolgreicher Angriff auf die Kommunikationshypervisorvariante A oder C von dem System toleriert werden kann, erkennt man in der Abbildung 17. Der Angreifer hat jeweils die Minderheit der Sensoren und Verarbeitungskomponenten – deren fehlerhafte Werte werden in der jeweils nächsten Verarbeitungsstufe durch den Mehrheitsentscheid aussortiert – und eine Motoransteuerung unter seiner Kontrolle. Eine fehlerhafte Motoransteuerung kann jedoch durch die Sensoren detektiert werden und der Motor wird dann über die fehlerfreien Komponenten passiviert.

Da die Sensoren DZ1 und AW1 Softwarekomponenten zugeordnet sind, die auf der Variante B des Kommunikationshypervisors laufen, bekommt der Angreifer mit einen erfolgreichen Angriff auf die Variante B des Kommunikationshypervisors die kryptographischen Schlüssel, um die Werte von beiden Sensoren zu fälschen. Damit der Angreifer (oder auch ein Fehler) nicht das Lenksystem vollständig passivieren kann, indem er Daten sendet, die für beide Motoren vom Sollwert abweichen, muss in den Verarbeitungskomponenten die Logik für die Passivierungsaufforderung diesen Fall berücksichtigen. Dies geschieht, indem maximal für einen Motor die Passivierungsaufforderung gesetzt wird und das Entscheidungskriterium bei allen Verarbeitungskomponenten identisch ist. Der Angreifer kann dann durch fehlerhafte Werte nur noch einen Motor passivieren, was innerhalb der Spezifikation ist.

9

BEWERTUNG

Verschiedenen Optionen, die in den vorhergegangenen Kapiteln aufgezeigt wurden, eine kryptographische Insel zu implementieren, werden im Folgenden diskutiert. Dabei liegt der Schwerpunkt darauf, Aufwand und erzielbare Sicherheit einander gegenüber zu stellen für die Varianten der Kernkomponenten – dem kryptographischen Maskeradeschutz und der diversitären Implementierung des Kommunikationshypervisors. Dem schließt sich eine Abschätzung der wirtschaftlichen Eignung an, bei der die kryptographische Insel mit anderen etablierten Verfahren verglichen wird.

9.1 KRYPTOGRAPHISCHER MASKERADESCHUTZ

9.1.1 *Möglichkeiten*

Der in Kapitel 7.3 geforderte kryptographische Maskeradeschutz lässt sich grundsätzlich mit allen in Kapitel 3 genannten Verfahren implementieren, sowohl mit Signaturen als auch durch Verschlüsselung jeweils mit einem symmetrischen, asymmetrischen oder hybriden kryptographischen Verfahren.

Jedoch gibt es, gerade wenn man gemäß Kapitel 5.2 von verteilen Steuerungssystemen ausgeht, einige Punkte bei der Wahl zu beachten, die im Folgenden diskutiert werden.

9.1.2 *Sicherheit*

Bei der Sicherheit eines kryptographischen Verfahrens lassen sich zwei Ebenen unterscheiden. Zum einem die theoretische/kryptoanalytische Sicherheit des Algorithmus und zum anderen die real resultierende Sicherheit, bei der verschiedene Faktoren der Umgebung mit einfließen.

Die theoretische oder kryptoanalytische Sicherheit ist für etablierte Algorithmen gut untersucht und steigt mit der verwendeten Schlüssellänge. Die Auswahl von Algorithmus und Schlüssellänge hängt damit von zwei zu schätzenden Parametern ab:

1. Der Rechenkapazität, die dem Angreifer maximal zur Verfügung steht. Dabei ist insbesondere zu unterscheiden, ob der Angreifer mit der bekannten Rechenkapazität übernommener Knoten auskommen muss oder zur Analyse von Daten auf externe unbekannte Rechenkapazitäten zurückgreifen kann. Für

den maximalen Umfang der externen Rechenkapazität muss eine (idealerweise pessimistische) Schätzung[1] erfolgen.
Unter der Annahme dieser Arbeit, dass ein Angreifer Zugriff auf das Kommunikationssystem erlangt hat, ist die erste Annahme ("keine externen Ressourcen") nur schwer zu rechtfertigen.

2. Wie sich der Rechenaufwand während der Produktlebensdauer entwickelt. Zum einen steigt die verfügbare Rechenkapazität („Moore's Law“) und zum anderen besteht stets das Risiko, dass Mathematiker einen Weg finden, die zugrundeliegenden mathematischen Probleme zu vereinfachen und damit signifikant weniger Rechenkapazität für das gleiche Problem aufzuwenden ist. Insbesondere die Fortschritte in der Mathematik/ Kryptoanalyse lassen sich nur sehr schwer vorhersagen.

Da der Aufwand zum Brechen für die meisten Algorithmen mit der Schlüssellänge exponentiell zunimmt und die Schätzungen mit großen Fehlern behaftet sind, werden bei neuen Entwicklungen oftmals sehr große Sicherheitszuschläge gemacht, falls die Sicherheit über größere Zeiträume sichergestellt werden soll [17].

Neben dem theoretischen Sicherheitsaspekt gibt es in der Praxis noch das Problem, dass eine korrekte Implementierung eines kryptographischen Verfahrens tiefgreifende Kenntnisse über den eigentlichen Algorithmus hinaus benötigt, um Schwächen zu vermeiden, die ein Angreifer ausnutzen kann. Einige Beispiele:

- Für einige Algorithmen lässt sich der (private) Schlüssel signifikant leichter berechnen, falls es dem Angreifer gelingt, in den Besitz von mehreren Paaren von Geheimtext und Klartext zu kommen. Dies Problem kann bei Kommunikationsprotokollen auftreten, wenn der Angreifer aus anderen öffentlich verfügbaren Daten zum Beispiel den Aufbau von Headern vorhersagen kann. Dies wird im Englischen als „known plaintext“ Angriff bezeichnet. Beispiele sind die Wetterberichte bei der Verschlüsselung der Enigma [66] oder die Verschlüsselung des pkzip-Programms [21].

- Viele Verfahren nutzen Zufallszahlen. Bei Schwächen in den verwendeten Zufallszahlen werden die Schlüssel leicht(er) bestimmbar/erratbar. Das wohl bekannteste Beispiel ist die OpenSSL-Version in Debian, die von 2006 bis 2008 einen defekten Zufallszahlengenerator besaß, und daher leicht brechbare kryptographische Schlüssel erzeugte [3]. Ein anderes Beispiel war die Implementierung es ElGamal-Algorithmus in GNU Privacy Guard [119].

---

[1] Kann als Teil des Angreiferverhaltensmodells gesehen werden, welche Ressourcen ihm zur Verfügung stehen.

- Wenn bei der Implementierung eines Algorithmus nicht speziell darauf geachtet wird, lassen sich oftmals sogenannte Seitenkanalangriffe durchführen. Diese nutzen Informationen wie Stromverbrauch oder Zeitverhalten um Rückschlüsse auf interne Zustände und damit auf den (privaten) Schlüssel zu gewinnen. [86, 90]

Aufgrund der Vielzahl der Möglichkeiten, durch ungeschicktes Design der Protokolle und Implementierungen, die Sicherheit des Algorithmus zu untergraben, wird im allgemeinen empfohlen, in diesem Bereich auf gut analysierte COTS Bibliotheken zurückzugreifen [161, 80]. Dabei kann es allerdings das Problem geben, eine ausreichende Zahl an diversitär entwickelten Bibliotheken zu finden, da die Entwicklung einer solchen Bibliothek sehr aufwändig ist. Die Verfügbarkeit von Bibliotheken kann damit schon die Zahl der praktisch nutzbaren Verfahren reduzieren. Die COTS Bibliotheken unterstützen oftmals eine Vielzahl von kryptographischen Verfahren. Es muss durch geeignete Konfiguration sichergestellt werden, dass nicht unbemerkt unsichere Verfahren verwendet werden (sog. „downgrade“ Angriffe [112, 102]).

9.1.3 *Aufwand*

Der Aufwand, den ein kryptographischer Maskeradeschutz mit sich bringt, lässt sich in verschiedenen Dimensionen beschreiben. Der organisatorische Aufwand bei Planung, Inbetriebnahme und Wartung, der Initialisierungsaufwand bei jeden Start des Systems und der Aufwand zur Laufzeit.

9.1.3.1 *Organisatorischer Aufwand*

Der organisatorische Aufwand für einen kryptographischen Maskeradeschutz besteht aus der Schlüsselverwaltung und der Verwaltung der Berechtigungen, welcher Sender an welche Empfänger senden darf. Dieser tritt in der Regel nur bei der Inbetriebnahme und beim Austausch von Komponenten im Rahmen der Wartung auf. Der Aufwand für die Verwaltung der Berechtigungen, welche Sender für einen Empfänger gültige Absender sind, ist im Wesentlichen bei symmetrischen, asymmetrischen und hybriden Verfahren identisch, da diese Zuordnung nur von der Anzahl der Softwarekomponenten abhängt. Unterschiede ergeben sich jedoch bei der Schlüsselverwaltung.

Bei den asymmetrischen und hybriden Verfahren mit einer PKI (siehe Kapitel 3.2.5) muss auf den Knoten, neben dem öffentlichen Schlüssel der PKI, nur das Schlüsselpaar und das Zertifikat installiert werden. Es ist pro sendende Softwarekomponente nur ein Schlüsselpaar notwendig. Da die PKI ein Schwachpunkt ist, muss sie entsprechend gut gesichert werden. Ein Angreifer, der Zugriff auf den privaten

Schlüssel der PKI bekommt, kann sich beliebig viele gültige Zertifikate ausstellen und ist damit in der Lage, beliebige Maskeradeangriffe durchzuführen.

Bei den symmetrischen Verfahren ist der gemeinsam genutzte Schlüssel jeweils auf die sendende und empfangene Komponente zu verteilen. Ein Schlüssel darf nur für eine Kommunikationsbeziehung verwendet werden[2]. Würde analog den asymmetrischen Verfahren nur ein Schlüssel pro Sender genutzt, wäre der Angreifer nach dem ersten erfolgreichen Angriffsvorgehen ($a = 1$) mit den erbeuteten Schlüsseln in der Lage, gültige Signaturen von allen Sendern[3] zu erstellen, die mit dem übernommenen Knoten kommunizieren. Dies stellt für die meisten Systeme einen zu großen (Fehler)Bereich da, um diesen Angriff mit den Fehlertoleranzverfahren zu tolerieren zu können. Die Zahl der Kommunikationsbeziehungen ist in den meisten Steuerungssystemen bei n Softwarekomponenten zwar deutlich unter der oberen Grenze von $\frac{n*(n-1)}{2}$, aber die Zahl der manuell zu verwaltenden Schlüssel ist trotzdem signifikant höher als bei asymmetrischen und hybriden Verfahren.

9.1.3.2 *Initialisierungsaufwand*

Nennenswerter Initialisierungsaufwand beim Starten des Systems tritt nur bei asymmetrischen Systemen mit PKI und hybriden Systemen auf. Bei den asymmetrischen Systemen mit PKI muss in der Initialisierungsphase das Zertifikat geprüft werden, um danach den öffentlichen Schlüssel der Kommunikationsbeziehung zuzuordnen. Bei den hybriden Verfahren muss zusätzlich noch ein temporärer symmetrischer Schlüssel ausgehandelt werden.

Der Initialisierungsaufwand kann eine Rolle spielen, falls die Zeit zum Starten des Systems begrenzt ist und dadurch die Grenzen überschritten werden können.

9.1.3.3 *Aufwand zur Laufzeit*

Der Aufwand zur Laufzeit ergibt sich aus dem Bedarf an CPU-Zeit und der notwendigen Netzwerkbandbreite des Verfahrens.

Der Bedarf an CPU-Zeit oder auch Rechenkapazität bestimmt die notwendige Leistungsfähigkeit der Hardware eines Knotens für die kryptographische Insel. Da bei Steuerungssystemen Nachrichten oftmals im Abstand weniger Millisekunden gesendet werden und auch die maximal erlaubte Verzögerung im Bereich weniger Millisekunden liegen kann, muss das Verfahren mit relativ geringem Aufwand berechenbar sein, da ansonsten die resultierenden Hardwareanforderungen schwer zu erfüllen sind. Gemäß Kapitel 3 bedeutet dies, dass

---

[2] Eine Kommunikationsbeziehung meint einen Sender und einen Empfänger. Eine Information, die von einem Sender an vier Empfängerfließt, stellt vier Kommunikationsbeziehungen dar.

[3] beispielsweise alle redundanten Sensoren

aus Performancegründen bevorzugt symmetrische Verfahren (oder hybride Verfahren) verwendet werden sollten, und sofern möglich ein HMAC Signaturverfahren. Ein geringer Rechenaufwand zum Prüfen einer Signatur ist auch unter einem anderen Aspekt relevant: Es erschwert einem Angreifer, die Prüfroutine für einen Dienstblockadeangriff zu missbrauchen.

Der zusätzliche Bedarf an Netzwerkbandbreite – den das Kommunikationssystem übertragen muss – hängt vom gewählten kryptographischen Verfahren und der Architektur des Kommunikationssystems ab. Bei symmetrischen Signaturverfahren muss für jeden Empfänger eine Signatur erstellt und mit den Nutzdaten übertragen werden. Bei asymmetrischen Signaturverfahren ist nur eine Signatur für alle Empfänger notwendig. Die Länge einer Signatur (Bei einer SHA2-HMAC beispielsweise ab 28 Byte) ist bei einem Steuersystem teilweise länger als die kurze zu übertragenden Informationen. Bei Verschlüsselungsverfahren muss die Information für jeden Empfänger individuell – inkl. des Prüfmusters, das angibt, ob sie fehlerfrei entschlüsselt wurde – verschlüsselt werden[4].

Bei 1:1 Kommunikation (beispielsweise in vielen paketvermittelten Kommunikationssystemen) wird jede Nachricht nur um eine Signatur (oder das mitverschlüsselte Prüfmuster) verlängert. Bei kurzen Nutzdaten kann das bei modernen paketvermittelten Kommunikationssystemen wie Ethernet aufgrund der Mindestpaketgrößen (42 Byte bei Ethernet) zur Folge haben, dass Nutzdaten und Signatur zusammen unter der Mindestlänge bleiben und damit sich die effektive Kommunikationslast nicht ändert.

Bei einer Broadcast-Kommunikation müssen die Signaturen (oder verschlüsselte Informationen) für alle Empfänger mit der Information gemeinsam übertragen werden (jede einzelne Sender-Empfänger-Beziehung nutzt einen eigenen Schlüssel). Insbesondere bei einer hohen Anzahl von Empfängern und bei Anwendung von symmetrischen[5] Verfahren verlängert sich dadurch die Nachricht signifikant, so dass es bei bestehenden Systemen Probleme geben kann.

9.1.4 *Fazit*

Falls für ein konkretes System keine Gründe dagegen sprechen (z.B. der Bandbreitenbedarf oder der Zeitbedarf bei der Initialisierung), dürfte ein hybrides Verfahren mit HMAC die geeignete Wahl sein. Das

---

[4] Bei asymmetrischer Verschlüsselung gibt es hybride Verfahren, bei der die eigentliche Nachricht symmetrisch verschlüsselt wird und nur der für diese Nachricht zufällig generierte Schlüssel jeweils mit den öffentlichen Schlüsseln verschlüsselt wird, so dass der hohe Aufwand des asymmetrischen Verfahrens nur einen geringen Anteil des Gesamtaufwands ausmacht. Aufgrund der typischerweise sehr kurzen Nachrichten in Steuerungssystemen, bietet das hybride Verfahren keine Vorteile.

[5] Hybride kryptographische Verfahren nutzen auch symmetrische Verschlüsselung zur Laufzeit

bietet zur Laufzeit einen geringen Rechenaufwand und damit eine geringe Latenz und eine einfache Schlüsselverwaltung. Um auch gegen starke Angreifer geschützt zu sein, bieten sich nach den Empfehlungen aus [17] für den dynamisch ausgehandelten symmetrischen Sitzungsschlüssel, auch passend zur SHA2-HMAC, die Länge von 224 Bit an. Für die asymmetrische Verschlüsselung sollten RSA-Schlüssel mit 2048-Bit genutzt werden. Falls eine Verschlüsselung der Daten aus Datenschutzgründen gefordert wird, kann diese auch für den Maskeradeschutz genutzt werden, ist aber aufgrund des Aufwands nicht generell zu empfehlen. Signaturverfahren haben zusätzlich den Vorteil, dass die Daten auch von Softwarekomponenten ausgewertet werden können – da die Daten im Klartext vorliegen – die sich außerhalb der kryptographischen Insel befinden. Da diese externen Softwarekomponenten keine gültigen Signaturen erzeugen können, entsteht durch den Maskeradeschutz eine „Datendiode“, die genutzt werden kann, um Informationen aus dem fehlertoleranten System in die Umwelt[6] zu exportieren.

9.2 DIVERSITÄRER KOMMUNIKATIONSHYPERVISOR

Die meisten Aspekte des Kommunikationshypervisors für die kryptographische Insel wurden bereits im Teil iii dargestellt. Da eine Vielzahl von möglichen Varianten vorgestellt wurden, wird im Folgenden zentral die erzielbare Sicherheit und der Aufwand für die notwendige Diversität diskutiert.

9.2.1 *Sicherheit*

Die Sicherheit der kryptographischen Insel basiert darauf, dass ein Angreifer mit a Angriffsvorgehen nicht mehr als f Instanzen einer Softwarekomponente kontrollieren kann und die dem System inhärente Fehlertoleranz ein (beliebiges) Fehlverhalten von bis zu f Softwarekomponenten toleriert. Die Annahmen (siehe Kapitel 5.3) stellen sicher, dass ein Angreifer maximal alle Knoten kontrolliert, auf denen a Varianten des Kommunikationshypervisors laufen. Durch die Regeln zur Verteilung der Instanzen von Softwarekomponenten ist dann sichergestellt, dass maximal f Instanzen einer Softwarekomponente durch den Angreifer kontrolliert werden können. Dies kann dann wiederum per Definition von den vorhandenen Fehlertoleranztechniken toleriert werden. Dieser Schluss gilt für alle Varianten der möglichen Implementierung einer kryptographischen Insel, die sich aus Kapitel 6.2 und Kapitel 7 ergeben.

In Kapitel 9.1 wurde bereits die notwendige Sicherheit für den kryptographischen Maskeradeschutz diskutiert. In Kapitel 9.2.2 wird diskutiert, welche Maßnahmen zur Sicherstellung der Annahme des

---

[6] zum Beispiel ein nicht sicherheitsrelevantes Prozessüberwachungssystem

unabhängigen Versagens bei der diversitären Entwicklung eines Kommunikationshypervisors für eine kryptographische Insel möglich sind und an welchen Stellen eine Abwägung zwischen Sicherheit und Aufwand getroffen werden kann. Dabei wird dann auch diskutiert, inwieweit die Annahme, dass die Wahrscheinlichkeit eines gleichartigen Versagens bei den verschiedenen möglichen Realisierungen einer kryptographischen Insel identisch ist, haltbar ist.

Für ein konkretes System muss zudem geprüft werden, ob die Annahmen durch das System erfüllt werden. [125] Dies gilt insbesondere für das Angreiferverhaltensmodell[7] und die Annahme, dass das System beliebiges Versagen von f Softwarekomponenten toleriert[8].

9.2.2 *Aufwand für die notwendige Diversität*

Ergänzend zu den allgemeinen Betrachtungen des Aufwands diversitärer Entwicklung in Kapitel 2.5.5 wird im Folgenden für die Varianten in Kapitel 7.2 mit isolierten Kommunikationshypervisor und die diversitären Erweiterungen eines Betriebssystems der notwendige Umfang diskutiert. Die Eignung von COTS aus unterschiedlichen Quellen zur Gewinnung von Diversität wird darauf folgend diskutiert.

Minimal notwendige Diversität

Wie bereits im Kapitel 7.1 erwähnt, müssen alle Teile des Kommunikationshypervisors diversitär entwickelt sein, die gemäß des Angreiferverhaltensmodells vom Angreifer so beeinflusst werden können, dass evtl. vorhandene Fehler vom Angreifer ausgenutzt werden können. Neben der Software des Kommunikationshypervisors, kann unter Umständen auch Hardware betroffen sein, insbesondere wenn diese eingebettete Software (oftmals als Firmware bezeichnet) enthält. Ein Beispiel, bei der eine hohe Wahrscheinlichkeit von Beeinflussbarkeit des Systems über Hardwarekomponenten gegeben ist, sind COTS-Server, die ein über das Netzwerk ansprechbares Management-Interface haben, das weitreichende Zugriffe ermöglicht (IPMI).

Der minimal notwendige Umfang des diversitär entwickelten Kommunikationshypervisors wird anhand eines Beispiels diskutiert. Dabei wird die Abwägung aus Aufwand und resultierender Sicherheit aufgezeigt. Als Beispiel wird der schematische Aufbau des Linux-Kernels (wie in Kapitel 7.1) in Abbildung 18 verwendet, die Argumentation lässt sich jedoch auch auf andere Systeme wie den Aufbau

---

[7] Beschränkt u. A. den physikalischen Zugang zu den Knoten, was nicht bei allen Systemen sichergestellt werden kann.

[8] Keine Einschränkungen für die vom System tolerierten Fehlerbilder, da der Angreifer die nicht tolerierten Fehlerbilder sonst gezielt provozieren kann und damit das System in einen nicht definierten Zustand bringen kann.

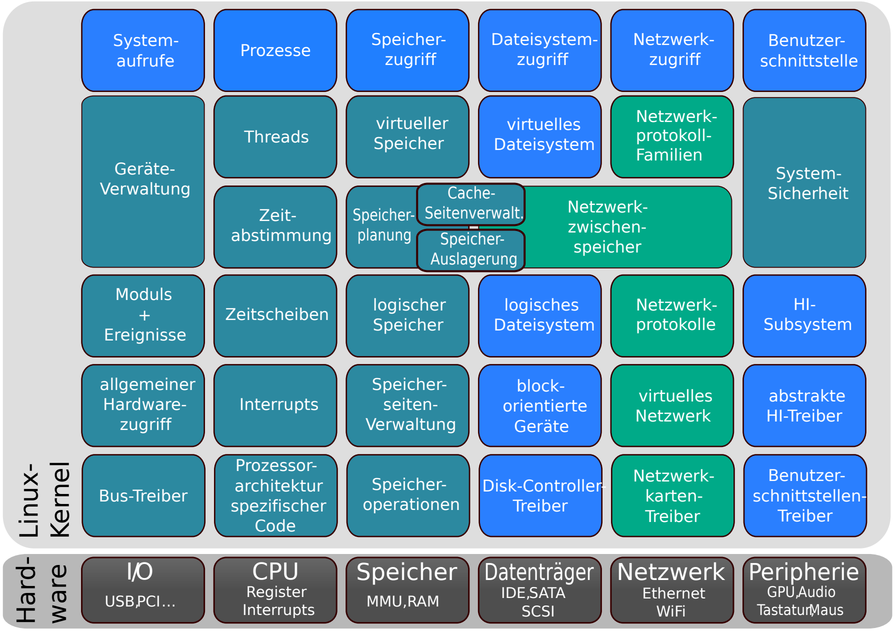


Abbildung 18: Schematischer Aufbau des Linux Kernels nach [146]

einer AUTOSAR Architektur [12] übertragen. Im Beispiel wird davon ausgegangen, dass der Kommunikationshypervisor in dem Modul „Netzwerkprotokoll-Familien“ implementiert wird.

In einem größeren Softwareprojekt wie ein Betriebssystem können beliebig komplexe Seiteneffekte niemals ausgeschlossen werden – und damit auch nicht, dass ein Angreifer sie ausnutzen kann. Beispiel wäre ein „use-after-free“ Bug [155] in einem logischen Dateisystem. Es kann dann nicht ausgeschlossen werden, dass Daten, die per Netzwerk empfangen werden, an genau diese (ja bereits freigegebene) Speicherseiten geschrieben werden. Falls dieser Bug Daten, die an die Anwendung weitergereicht werden, enthält und genau zu dem Zeitpunkt Daten gelesen werden, die signifikant den Programmablauf beeinflussen können (zum Beispiel ein Teil einer gerade zu ladenden Programmbibliothek), könnte theoretisch der Angreifer auf diesem Weg die Lücke im logischen Dateisystem ausnutzen. Durch die Vielzahl an notwendigen Bedingungen in diesem Beispiel (es muss genau der Zeitpunkt für den Empfang der Daten erwischt werden, bei dem sich die zu manipulierende Dateisystemoperation im verwundbaren Codeabschnitt befindet und dazu der interne Zustand der Speicherverwaltung dafür sorgen, dass bei den neu empfangenen Daten der gerade freigewordene Speicherbereich genutzt wird), die auch für den Angreifer typischerweise nicht gut zu beobachten sind, kann man bei der Abwägung von Sicherheit und Aufwand bei dem Beispiel gut argumentieren, dass Diversität an der Stelle nicht notwendig ist.

Der Autor schlägt vor, zur Analyse, welche Teile einer Software, die einen Kommunikationshypervisors realisiert, diversitär entwickelt werden sollen, in drei Schritten vorzugehen:

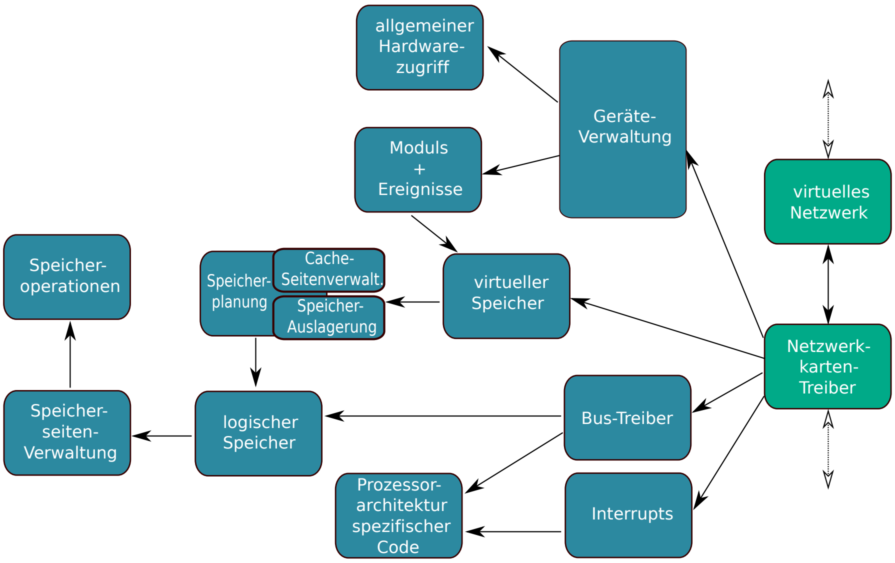


Abbildung 19: Beispielhafter Ausschnitt aus einem Abhängigkeitsgraph

1. Feststellen, welche Teile/Module der Software Operationen auf den vom Angreifer kontrollierbaren Daten durchführen, bevor diese den Kommunikationshypervisor verlassen. Da von Angriffen über das Netzwerk ausgegangen wird, wird der Teil oftmals als „Netzwerkstack“ bezeichnet. Für das Beispiel in Abbildung 18 betrifft dies die grün hinterlegten Module.

2. Ausgehend von den im ersten Schritt identifizierten Modulen, wird ein Graph erstellt, der die Abhängigkeiten der bereits identifizierten Module von weiteren Modulen (z. B. Speicherverwaltung, Bus-Treiber) und ebenfalls die daraus wieder resultierenden Abhängigkeiten zeigt. Ein kleiner (nicht vollständiger) Ausschnitt für den Linuxkernel anhand des Moduls des Netzwerkkartentreibers wird in Abbildung 19 dargestellt.

3. Für jedes Modul in dem Abhängigkeitsgraphen wird analysiert, inwiefern vom Angreifer kontrollierte Daten das Verhalten des Moduls beeinflussen können und damit das Risiko besteht, dass ein Bug vom Angreifer ausgenutzt werden kann. Bei den Modulen, die direkt mit den Daten operieren und im ersten Schritt identifiziert wurden, ist eine diversitäre Entwicklung notwendig. Module, die im Abhängigkeitsgraph aus dem zweiten Schritt nicht vorkommen, lassen sich, wie zuvor schon beschrieben, nur über komplizierte Seiteneffekte beeinflussen. Daher wird die Abwägung aus Aufwand und Sicherheit bei diesen nicht im Abhängigkeitsgraphen enthaltenen Modulen regelmäßig zur Entscheidung führen, diese Module nicht diversitär zu entwickeln. Module, die voraussichtlich nicht im Abhängigkeitsgraph auftauchen, sind in dem Beispiel in Abbildung 18 durch einen blauen Hintergrund kenntlich gemacht.

> Für die im Abhängigkeitsgraph enthaltenen Module wird das Vorgehen im folgenden Absatz diskutiert.

Das sicherste Vorgehen besteht darin, alle im Abhängigkeitsgraphen enthaltenen Softwaremodule diversitär zu entwickeln. Im Rahmen einer Abwägung von gewünschter Sicherheit und daraus resultierendem Aufwand bietet es sich an, über den Abhängigkeitsgraph zu analysieren, welche Menge an Informationen zu dem Modul fließt und in welchem Umfang diese Information durch den Angreifer bestimmbar ist. Beispielsweise kann der Angreifer bei der Interruptverwaltung das zeitliche Auftreten von Interrupts durch die Wahl des Sendezeitpunkts begrenzt beeinflussen. Jedoch keine darüber hinausgehende Informationen. Das Zeitverhalten alleine – das dazu noch durch Indeterminismus im Netzwerk verrauscht wird – transportiert so wenig Informationen, dass man eine Beeinflussung durch einen Angreifer auf diesem Weg als ausreichend unwahrscheinlich bezeichnen kann.

Bewertung von COTS zur Erzielung von Diversität

Wie in Kapitel 7.2 bereits dargestellt, ist die Verwendung von COTS attraktiv, da dies die Entwicklungskosten signifikant senken kann. Da man bei COTS keinen Einfluss auf den jeweiligen Entwicklungsprozess hat – und oftmals auch keinen Zugriff auf den Quellcode – lässt sich die erzielte Diversität nur schwer abschätzen. Grundsätzlich handelt es sich dabei um zufällige Diversität (siehe Kapitel 2.5.5), da die Hersteller voneinander unabhängig die gleiche API implementiert haben. Es besteht jedoch das Risiko, dass die COTS-Hersteller bestimmte Funktionen selbst als COTS von der gleichen Quelle beziehen und damit diese Teile identisch sind. Beispielsweise wurde eine kurze Zeit in zwei Windows-Versionen ein TCP/IP-Stack verwendet, der auf dem FreeBSD TCP/IP-Stack basierte [2].

Für die POSIX API wurde in [64] für verschiedene POSIX konforme Betriebssysteme verglichen, wie viele Varianten von gleichen Bugs betroffen waren. Dabei konnten weder die Ursachen für gemeinsame Bugs identifiziert werden[9], noch wurde analysiert, in wieweit diese Bugs mit einem Angriffsvorgehen ausnutzbar wären. Gemeinsames Versagen mehrerer Varianten war in der Untersuchung signifikant wahrscheinlicher als in anderen Experimenten zur diversitären Entwicklung und damit auch signifikant höher als bei unabhängigem Versagen zu erwarten wäre.

Sollen COTS eingesetzt werden, muss entweder argumentativ dargelegt werden, inwiefern diese COTS ausreichend unabhängig sind, oder eine höhere Wahrscheinlichkeit für gemeinsames Versagen – und damit geringere Sicherheit vor Angreifer – muss tolerabel sein.

---

[9] Ob es an der API-Spezifikation liegt, oder an gemeinsam genutzten Komponenten oder Zufall liegt.

9.3 WIRTSCHAFTLICHE EIGNUNG

Die wirtschaftliche Eignung der kryptographischen Insel wird für Systeme entsprechend dem Einsatzgebiet in Kapitel 5.2 gezeigt, die für eine ausreichende funktionale Sicherheit statische Redundanz zur Toleranz von Hardwarefehlern erfordern, bei der Software jedoch für die funktionale Sicherheit keine Diversität benötigen. Es wird dabei der Entwicklungsaufwand A für drei mögliche Realisierungen verglichen, von denen zwei Angreifertoleranz bieten:

1. Als Vergleichsmaßstab für den Aufwand A_{Rep} einer Implementierung ohne Angreifertoleranz dient ein n-fach redundantes System, das zur Toleranz von Hardwarefehlern replizierte Instanzen (keine Diversität) der Software nutzt.

2. Als Vergleichsmaßstab für den Aufwand A_{Div} einer konventionellen Implementierung mit Angreifertoleranz (siehe auch Kapitel 4.3) dient ein n-fach redundantes System, bei dem zusätzlich zur Vermeidung von systematischen Fehlern in der Software, die ein Angreifer ausnutzen könnte, die Software vollständig diversitär entwickelt ist (einschließlich Betriebssystem, Kommunikations- und Anwendungssoftware).

3. Der Aufwand A_{Insel} zur Implementierung des Systems mittels einer Variante der kryptographische Insel. Der Kommunikationshypervisor ist dabei diversitär entwickelt, um die Angreifertoleranz sicherzustellen. Die restliche Software, welche die Funktionalität des Systems implementiert, ist dabei wie in der ersten Realisierung repliziert.

Aufgrund der Vielzahl von möglichen Systemen – und auch möglichen Varianten der kryptographischen Insel – wird im Folgenden versucht, den Aufwand möglichst allgemeingültig zu vergleichen. Es wird dabei angenommen, dass die Kosten für die Hardware für alle Varianten vergleichbar sind und damit nur der Entwicklungskosten betrachtet werden müssen. Es wird zunächst der vollständige Entwicklungsaufwand für ein System betrachtet (Entwicklung auf der „grünen Wiese"). Daraufhin wird untersucht, welche Teile sich ohne Einschränkungen der Eigenschaften bezüglich Sicherheit und funktionaler Sicherheit wiederverwenden lassen und wie das den Aufwand für ein einzelnes System beeinflusst.

Der Aufwand A einer Softwaresystementwicklung wird auf die Realisierung ohne Angreifertoleranz normiert. Bei einer logischen Trennung des Softwaresystems in verschiedene Teile, kann der Aufwand entsprechend geteilt werden, beispielsweise in den Aufwand für die Hardwareabstraktionsebene (siehe Kapitel 2.1) A_{Ha} (z. B. Betriebssystem, Laufzeitumgebung) und den Aufwand für die Softwarekomponente A_{Sk} welche die systemspezifische Funktionalität implementiert. Dabei gilt: $A = A_{Ha} + A_{Sk}$.

Bei einer diversitären Entwicklung von n Exemplaren einer Software fällt gemäß Kapitel 2.5.5 nicht der n-fache Aufwand an. Der Faktor D_n bestimmt daher im Folgenden den Faktor, der den Aufwand für n diversitäre Software-Exemplare im Vergleich zu einer nicht diversitären Entwicklung angibt.

Für ein vollständig diversitär entwickeltes System ist der Aufwand dann:

$$\begin{aligned} A_{Div} =& D_n * A_{Rep} \\ =& D_n * (A_{Ha} + A_{Sk}) \\ =& D_n * A_{Ha} + D_n * A_{Sk} \end{aligned}$$

Der Aufwand für eine Realisierung der kryptographischen Insel A_{Insel} besteht immer aus mindestens zwei Teilen: Den diversitär zu entwickelnden Kommunikationshypervisor (*Khv*) und den restlichen nicht diversitären Softwareteilen. Da ein transparenter Kommunikationshypervisor möglich ist (Kapitel 6.2.1), kann der Aufwand A_{Sk} für die Softwarekomponente in allen hier verglichenen Systemen als identisch angesehen werden. Der Aufwand für die Hardwareabstraktionsebene besteht dabei aus dem Teil, den der Kommunikationshypervisor A_{Khv} implementiert, und optional weiteren Funktionen A_{HaOpt}. Durch den zusätzlichen Aufwand A_M zur Implementierung des Maskeradeschutzes im Kommunikationshypervisor, erhöht sich bei der kryptographischen Insel der Aufwand für die Hardwareabstraktionsebene. Bei vergleichbarer Aufteilung gilt daher:

$$\begin{aligned} A_{Khv} + A_{HaOpt} &\simeq A_{Ha} \\ A_{Khv} + A_M + A_{HaOpt} &\geq A_{Ha} \end{aligned}$$

Daraus ergibt sich für eine kryptographische Insel als gesamter Entwicklungsaufwand:

$$\begin{aligned} A_{Insel} =& D_n * (A_{Khv} + A_M) + A_{HaOpt} + A_{Sk} \\ =& D_n * A_{Khv} + D_n * A_M + A_{HaOpt} + A_{Sk} \\ =& (D_n - 1) * A_{Khv} + D_n * A_M + (A_{Khv} + A_{HaOpt}) + A_{Sk} \\ =& (D_n - 1) * A_{Khv} + D_n * A_M + A_{Ha} + A_{Sk} \\ =& (D_n - 1) * A_{Khv} + D_n * A_M + A_{Rep} \end{aligned}$$

Gut erkennbar ist bei beiden angreifertoleranten Realisierungen (vollständig diversitär und kryptographische Insel), der im Vergleich zur reinen Replikation erhöhte Aufwand. Der Unterschied zwischen vollständiger Diversität und kryptographischer Insel hängt, falls die Insel nur für ein System entwickelt wird, von dem Aufwand für den Maskeradeschutz im Verhältnis zum Aufwand für die Softwarekomponente ab. Eine allgemeine Aussage, welches Vorgehen günstiger ist, lässt sich dabei nicht treffen.

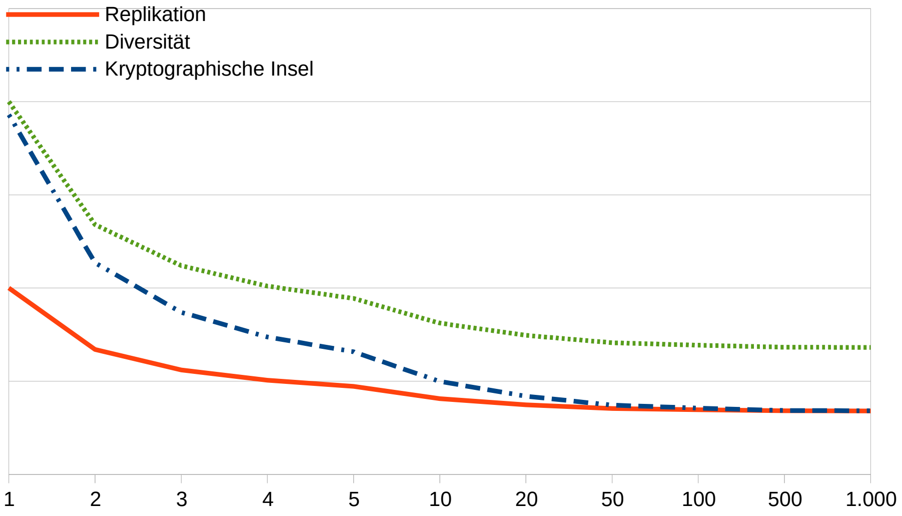


Abbildung 20: Entwicklungsaufwand für ein System bei Wiederverwendung der Basissysteme in m Softwaresystemen. (D_3=2, A_{Ha}=0,66, A_{Khv}=0,33, A_{HaOpt}=0,33, A_{Sk}=0,34, A_M=0,3)

Geht man jedoch davon aus, dass sich die Hardwareabstraktionsebene (Betriebssysteme und Kommunikationshypervisor) in m Softwaresystemen wiederverwenden lassen, kann der Aufwand für diese Teile auf m Softwaresysteme verteilt werden.

Die Formeln zur Bestimmung des Entwicklungsaufwands müssen dafür wie im Folgenden angepasst werden:

$$\begin{aligned} A_{Rep} &= \frac{A_{Ha}}{m} + A_{Sk} \\ A_{Div} &= \frac{D_n * A_{Ha}}{m} + D_n * A_{Sk} \\ A_{Insel} &= \frac{D_n * (A_{Khv} + A_M) + A_{HaOpt}}{m} + A_{Sk} \end{aligned}$$

Daraus geht hervor, dass sich der Entwicklungsaufwand A_{Insel} eines Softwaresystemens mit der kryptograpphischen Insel bei $m \to \infty$ dem Entwicklungsaufwand A_{Rep} der reinen Replikation annähert, während bei der vollständigen Diversität der Faktor D_n erhalten bleibt. Dies lässt sich auch an einem Beispiel in Abbildung 20 erkennen.

Geht man wie in Kapitel 7.4 von einer standarisierten API aus, ist eine Wiederverwendung des Kommunikationshypervisors gut möglich und wahrscheinlich, so dass dieser bei vergleichbarer Sicherheit gegen Angreifer deutliche wirtschaftliche Vorteile erzielen kann. Die Vorteile kommen in einem verteilten System schon innerhalb eines Projektes zum Tragen, da für die verschiedenen Teilsysteme große Teile wiederverwendet werden können. Daher ist ein Einsatz der kryptographischen Insel mit einer standarisierten API für Systeme vorteilhaft, bei denen Angreifertoleranz gewünscht oder benötigt wird und keine vollständige Diversität für die funktionale Sicherheit erforderlich ist.

Teil IV

ZUSAMMENFASSUNG & AUSBLICK

10

ZUSAMMENFASSUNG

In dieser Arbeit wurde eine kryptographische Insel entwickelt, um durch eine wiederverwendbare Softwareschicht effektiv und effizient eine Toleranz von Angreifern für Systeme mit statischer Redundanz in Form von Softwarereplikation zum Erkennen/Tolerieren von Hardwarefehlern zu ermöglichen. Diese Art von Systemen wird oftmals in Steuerungssystemen für cyber-physikalische Systeme eingesetzt, deren unkontrollierter Ausfall eine Gefährdung für die Umgebung darstellen würde. Die neuentwickelte kryptographische Insel schränkt die möglichen Interaktionen des Angreifers mit dem System durch einen diversitär implementierten Maskeradeschutz so ein, dass die im Systementwurf zur funktionalen Sicherheit vorhandenen Fehlererkennungs- und Fehlertoleranztechniken die Angriffe tolerieren können.

Für die Entwicklung der kryptographischen Insel wurden die zu betrachtenden Systeme definiert. Es wurde dabei auf häufig vorkommende Schwachstellen eingegangen, die ein Angreifer nutzen kann, um das Verhalten in seinem Sinne zu beeinflussen. Die Analyse der bestehenden Ansätze zur Angreifertoleranz ergab, dass die bisherigen Ansätze entweder in einem sehr hohen Entwicklungsaufwand resultieren, teilweise auch viele zusätzliche Komponenten im Betrieb benötigen, oder die möglichen Systeme und Systemstrukturen stark einschränken, um die Annahmen über das Angreiferverhalten zu vereinfachen. Es wurde dabei kein mit dieser Arbeit vergleichbarer Ansatz gefunden, bei dem die zur funktionalen Sicherheit vorhandenen Fehlererkennungs-/Fehlertoleranztechniken genutzt werden, um mit geringem zusätzlichen Aufwand eine Toleranz von Angreifern zu ermöglichen.

Das in dieser Arbeit entwickelte Konzept einer kryptographischen Insel basiert auf den folgenden Annahmen:

- Der Angreifer hat Zugriff auf die Kommunikationsinfrastruktur.
- Der Angreifer besitzt nur Kenntnisse über eine begrenzte Zahl von wirksamen Angriffsvorgehen (z. B. ausnutzbare Softwarefehler, physikalische Zugänge).
- Diversität bietet einen ausreichenden Schutz vor gleichartigen systematischen Fehlern, die ein Angreifer ausnutzen kann.
- Der Angreifer ist nicht in der Lage, kryptographisch starke Signaturen zu brechen.

Eine kryptographische Insel besteht stets aus einem diversitär entwickelten Kommunikationshypervisor, der einen Maskeradeschutz für die replizierte Anwendungssoftwarekomponente realisiert. Dadurch ist keine direkte Interaktion des Angreifers mit den Softwarekomponenten der Anwendung möglich und der Angreifer kann das System nur angreifen, sofern er einen wirksames Angriffsvorgehen für einen der Kommunikationshypervisor findet. Durch eine geeignete Verteilung der replizierten Softwarekomponenten kann ein Angreifer auch nach erfolgreichem Angriff nicht die Nachrichten der Mehrheit der Softwarekomponenten kontrollieren, so dass die vorhandenen Fehlererkennungs- und Fehlertoleranztechniken die falschen Eingaben des Angreifers tolerieren können.

Es wurden verschiedene Realisierungen einer kryptographischen Insel (transparent für die Anwendung, Toleranz mehrerer Angriffsvorgehen, verschiedene Implementierungsansätze) diskutiert und an Beispielen gezeigt. Die verschiedenen Realisierungsvarianten wurden unter den Aspekten Aufwand/Kosten und erzielte Sicherheit diskutiert.

Bei der Bewertung der Varianten konnte gezeigt werden, dass bei deutlich reduziertem Aufwand eine vergleichbare Sicherheit erzielbar ist wie bei der vollständig diversitären Entwicklung des gesamten Systems. Bei der Evaluation der wirtschaftlichen Eignung stellte sich heraus, dass die kryptographische Insel aufgrund der guten Wiederverwendbarkeit – sofern eine standarisierte API zur Softwarekomponente genutzt wird – schon bei einem einzigen zu entwickelnden System vorteilhaft sein kann. Durch hohe Wiederverwendung (z. B. als COTS) sinkt der Entwicklungsaufwand pro System mit zunehmender Anzahl von entwickelten Systemen asymptotisch gegen den Aufwand für reine Fehlertoleranz und ermöglicht damit Angreifertoleranz für Systeme, bei denen zuvor wirtschaftliche Erwägungen dies verhinderten.

In Anbetracht der zunehmend vernetzten Systeme wird auch Sicherheit für funktional sichere Systeme immer bedeutender. Eine Implementierung der kryptographischen Insel bietet daher für Systeme, die statische Redundanz zur Erkennung oder Tolerierung von Hardwarefehlern nutzen, eine effektive und effiziente Lösung, um auch Angreifertoleranz zu gewährleisten und damit die funktionale Sicherheit in Anwesenheit eines Angreifers sicherstellen zu können.

11

AUSBLICK

Eine der zentralen Annahmen der Arbeit besteht darin, dass die Wahrscheinlichkeit eines erfolgreichen Angriffsvorgehens bei mehreren diversitär entwickelten Varianten einer Softwarekomponente ausreichend gering ist. Diese Annahme – auf die auch alle anderen Verfahren zur Angreifertoleranz durch Diversität zurückgreifen – basiert auf den Ergebnissen der bekannten großen Diversitäts-Experimente (siehe auch Kapitel 2.5.5). Bei diesen wurde untersucht, wie wahrscheinlich ein simultanes Versagen mehrerer diversitär entwickelter Varianten ist. Als simultanes Versagen war dabei in der Regel ein Testfall definiert, bei dem mehr als eine Variante fehlerhafte Werte lieferte, unabhängig von der Ursache des fehlerhaften Verhaltens. Da sich das Verhalten eines Angreifers häufig deutlich außerhalb der Spezifikation bewegt, wäre eine sinnvolle Ergänzung, bei einem zukünftigen Experiment zur Diversität eine Sicherheitsanalyse bezüglich gemeinsamer Schwachstellen vorzunehmen, um die genutzte Hypothese zu validieren, dass gemeinsame systematischen Fehler zur Ausnutzung von Sicherheitslücken die gleiche Wahrscheinlichkeit aufweisen wie andere Implementierungsfehler.

Ein weiterer Punkt, der eine weiterführende Analyse verdient, ist die Frage, in wieweit die Annahme über die begrenzte Zahl der bekannten Angriffsvorgehen bei hoher Wiederverwendung der Kommunikationshypervisorvarianten haltbar ist. Einerseits wird in der Literatur oftmals empfohlen, verbreitete COTS zu nutzen, da diese durch ihre Verbreitung viel intensiver getestet wurden. Andererseits ermöglicht das einem Angreifer, sich im Vorfeld Kopien der eingesetzten Kommunikationshypervisorvarianten zu beschaffen und diese unentdeckt beliebig lange auf Schwachstellen zu untersuchen. Wenngleich keine gezielten Angreifertoleranztechniken genutzt wurden, zeigte Stuxnet, dass ein ausreichend potenter Angreifer dies ausnutzen kann und mit mehreren zuvor noch unbekannten Angriffsvorgehen das System erfolgreich anzugreifen kann.

Zusätzlich wäre eine Implementierung einer Variante der kryptographischen Insel mit einem Industriepartner eine sinnvolle Fortführung der Arbeit. Dabei könnte sowohl für die gegebene API untersucht werden, wie hoch der Anteil der Software ist, die diversitär entwickelt werden muss, und damit die in dieser Arbeit aufgezeigte Effizienz genauer quantifiziert werden. Es könnten auch die Aufwände im Betrieb – insbesondere die Schlüsselverwaltung – genauer bestimmt werden.

Die grundlegende Effektivität und Effizienz aller Varianten der kryptographischen Insel wird durch die weitergehenden Forschungsvorhaben nicht neu bewertet – und nicht in Frage gestellt. Diese Forschungsvorhaben dienen nur dazu, die die weit verbreitete und akzeptierte Annahmen der Effektivität von Diversität gegen Angreifer zu validieren und würde (an Beispielen) die in der Praxis zu erzielende Effizienz zu bestimmen. Daher ist eine Anwendung in der Praxis auch ohne die Ergebnisse der weitergehenden Forschung angezeigt und aufgrund der wirtschaftlichen Vorteile sinnvoll.

LITERATURVERZEICHNIS

[1] Muhammad Aamir and Mustafa Ali Zaidi. A Survey on DDoS Attack and Defense Strategies: From Traditional Schemes to Current Techniques. *Interdisciplinary Information Sciences*, 19(2):173–200, 2013. ISSN 1340-9050. doi: 10.4036/iis.2013.173. URL `http://jlc.jst.go.jp/DN/JST.JSTAGE/iis/2013.173?lang=en{&}from=CrossRef{&}type=abstract`. (Zitiert auf Seite 20.)

[2] Adamba. Microsoft, TCP/IP, Open Source, and Licensing || kuro5hin.org, 2001. URL `http://www.kuro5hin.org/story/2001/6/19/05641/7357`. (Zitiert auf Seite 82.)

[3] David Ahmad. Two years of broken crypto: Debian's dress rehearsal for a global PKI compromise. *IEEE Security and Privacy*, 6(5):70–73, 2008. ISSN 15407993. doi: 10.1109/MSP.2008.131. (Zitiert auf Seite 74.)

[4] Johan Akerberg and Mats Björkman. Exploring network security in profisafe. *Computer Safety, Reliability, and Security*, pages 67–80, 2009. URL `http://www.springerlink.com/index/hu54k44364253547.pdf`. (Zitiert auf Seite 24.)

[5] Mohammed Saeed Al-kahtani. Survey on security attacks in Vehicular Ad hoc Networks (VANETs). In *2012 6th International Conference on Signal Processing and Communication Systems*, pages 1–9. IEEE, dec 2012. ISBN 978-1-4673-2393-2. doi: 10.1109/ICSPCS.2012.6507953. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6507953`. (Zitiert auf Seite 38.)

[6] Abdulmohsen Almalawi, Xinghuo Yu, Zahir Tari, Adil Fahad, and Ibrahim Khalil. An unsupervised anomaly-based detection approach for integrity attacks on SCADA systems. *Computers and Security*, 46:94–110, 2014. ISSN 01674048. doi: 10.1016/j.cose.2014.07.005. URL `http://dx.doi.org/10.1016/j.cose.2014.07.005`. (Zitiert auf Seite 27.)

[7] Hayriye Altunbasak, Sven Krasser, Henry L Owen, Jochen Grimminger, Hans-peter Huth, and Joachim Sokol. Securing Layer 2 in Local Area Networks. In *4th International Conference on Networking*, pages 699–706. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-25338-9. doi: 10.1007/978-3-540-31957-3_79. (Zitiert auf Seite 22.)

[8] Paul E. Ammann and John C. Knight. Data diversity: an approach to software fault tolerance. *IEEE Transactions on Computers*, 37(4):418–425, 1988. ISSN 00189340. doi: 10.1109/12.2185. (Zitiert auf Seite 23.)

[9] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure Multiparty Computations on Bitcoin. In *IEEE Symposium on Security and Privacy 2014*, number c, pages 443–458, 2014. ISBN 978-1-4799-4686-0. doi: 10.1109/SP.2014.35. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6956580`. (Zitiert auf Seite 37.)

[10] AUTOSAR. Specification of SW-C End-to-End Communication Protection Library, oct 2010. URL `http://www.autosar.org/download/R4.0/AUTOSAR{_}SWS{_}E2ELibrary.pdf`. (Zitiert auf Seite 17, 19 und 22.)

[11] AUTOSAR. Project Objectives, 2013. URL `https://www.autosar.org/fileadmin/files/releases/4-1/main/auxiliary/AUTOSAR{_}RS{_}ProjectObjectives.pdf`. (Zitiert auf Seite 65 und 66.)

[12] AUTOSAR. Layered Software Architecture. Technical report, 2015. URL `http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/general/auxiliary/AUTOSAR{_}EXP{_}LayeredSoftwareArchitecture.pdf`. (Zitiert auf Seite 80.)

[13] Algirdas Avižienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, dec 1985. ISSN 0098-5589. doi: 10.1109/TSE.1985.231893. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1701972`. (Zitiert auf Seite 23.)

[14] Algirdas Avižienis and John P. J. Kelly. Fault Tolerance by Design Diversity: Concepts and Experiments. *Computer*, 17(8):67–80, aug 1984. ISSN 0018-9162. doi: 10.1109/MC.1984.1659219. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1659219`. (Zitiert auf Seite 27.)

[15] Algirdas Avižienis, Michael R. Lyu, and Werner Schütz. In search of effective diversity: a six-language study of fault-tolerant flight control software. In *The Eighteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*, volume 1, pages 15–22. IEEE Comput. Soc. Press, 1988. ISBN 0-8186-0867-6. doi: 10.1109/FTCS.1988.5291. URL `http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=5291`. (Zitiert auf Seite 29.)

[16] Jan Axelson. Serial Port Complete. Programming and Circuits for RS-232 and RS-485. *Networks*, pages 1–9, 1998. (Zitiert auf Seite 18.)

[17] Steve Babbage, Dario Catalano, Carlos Cid, Benne de Weger, Orr Dunkelman, Christian Gehrmann, Louis Granboulan, Tim Güneysu, Jens Hermans, Tanja Lange, Arjen Lenstra, Chris Mitchell, Mats Näslund, Phong Nguyen, Christof Paar, Kenny Paterson, Jan Pelzl, Thomas Pornin, Bart Preneel, Christian Rechberger, Vincent Rijmen, Matt Robshaw, Andy Rupp, Martin Schläffer, Serge Vaudenay, Fré Vercauteren, and Michael Ward. ECRYPT II Yearly Report on Algorithms and Keysizes (2011-2012). Technical report, European Network of Excellence in Cryptology II ECRYPT, sep 2012. URL `http://www.ecrypt.eu.org/ecrypt2/documents/D.SPA.20.pdf`. (Zitiert auf Seite 74 und 78.)

[18] Stewart Baker, Natalia Filipiak, and Katrina Timlin. In the Dark: Crucial Industries Confront Cyber Attacks. Technical report, McAfee and Center for Strategic and International Studies (CSIS), 2011. URL `http://www.mcafee.com/us/resources/reports/rp-critical-infrastructure-protection.pdf`. (Zitiert auf Seite 37.)

[19] Caroline Baylon, Roger Brunt, and David Livingstone. Cyber Security at Civil Nuclear Facilities Understanding the Risks Cyber Security at Civil Nuclear Facilities Understanding the Risks. Technical report, Chatham House, London, sep 2015. URL `https://www.chathamhouse.org/sites/files/chathamhouse/field/field{_}document/20151005CyberSecurityNuclearBaylonBruntLivingstone.pdf`. (Zitiert auf Seite 3.)

[20] Ron Bell. Introduction & background to IEC 61508. pages 1–15, 2007. (Zitiert auf Seite 3.)

[21] Eli Biham. A known plaintext attack on the PKZIP stream cipher. *Fast Software Encryption*, 1995. doi: 10.1007/3-540-60590-8_12. URL `http://www.springerlink.com/index/Y4249G25K128Q310.pdf`. (Zitiert auf Seite 74.)

[22] Peter G. Bishop, David G. Esp, Mel Barnes, Peter Humphreys, Gustav Dahll, and Jaakko Lahti. PODS - A project on diverse software. *IEEE Transactions on Software Engineering*, SE-12(9):929–940, 1986. ISSN 0098-5589. doi: 10.1109/TSE.1986.6313048. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6313048`. (Zitiert auf Seite 29.)

[23] J Black, S Halevi, H Krawczyk, T Krovetz, and P Rogaway. UMAC: Fast and secure message authentication. In *Advan-*

ces in Cryptology CRYPTO, pages 79–79. Springer, aug 1999. URL `http://www.springerlink.com/index/FT35C6HA1R8MGV8K.pdf`. (Zitiert auf Seite 35.)

[24] Dominique Brière and Pascal Traverse. AIRBUS A320/A330/A340 electrical flight controls - A family of fault-tolerant systems. In *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*, pages 616–623. IEEE Comput. Soc. Press, 1993. ISBN 0-8186-3680-7. doi: 10.1109/FTCS.1993.627364. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=627364`. (Zitiert auf Seite 30.)

[25] Susan S. Brilliant, John C. Knight, and Nancy G. Leveson. Analysis of faults in an N-version software experiment. *IEEE Transactions on Software Engineering*, 16(2):238–247, 1990. ISSN 00985589. doi: 10.1109/32.44387. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=44387`. (Zitiert auf Seite 29.)

[26] Ian Broster and Alan Burns. The babbling idiot in event-triggered real-time systems. In *Proceedings of the Work-In-Progress Session, 22nd IEEE Real-Time Systems Symposium, YCS*, volume 337, pages 25–28, 2001. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.5.5150{&}rep=rep1{&}type=pdf`. (Zitiert auf Seite 22.)

[27] Johannes Buchmann. *Einführung in die Kryptographie*. Springer-Lehrbuch. Springer, 2010. ISBN 9783642111860. (Zitiert auf Seite 31, 32, 33 und 36.)

[28] Giuseppe Buja, Alberto Zuccollo, and Juan Pimentel. Overcoming babbling-idiot failures in the FlexCAN architecture: A simple bus-Guardian. *Emerging Technologies and*, 2:461–468, 2005. doi: 10.1109/ETFA.2005.1612713. URL `http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=1612713`. (Zitiert auf Seite 22.)

[29] Bundesamt für Sicherheit in der Informationstechnik. Die Lage der IT-Sicherheit in Deutschland 2014. Technical report, Bundesamt für Sicherheit in der Informationstechnik, Bonn, 2014. (Zitiert auf Seite 4.)

[30] Eric Byres and P Eng. The Myths and Facts behind Cyber Security Risks for Industrial Control Systems The BCIT Industrial Security Incident Database (ISID). In *VDE Kongress*, pages 1–6, 2004. (Zitiert auf Seite 3.)

[31] Kevin M Carter, Hamed Okhravi, and James Riordan. Quantitative Analysis of Active Cyber Defenses Based on Temporal

Platform Diversity. jan 2014. URL `http://arxiv.org/abs/1401.8255`. (Zitiert auf Seite 29.)

[32] Vinton Cerf, Yogen Dalal, and Carl Sunshine. Specification of Internet Transmission Control Program, 1974. URL `http://tools.ietf.org/html/rfc675`. (Zitiert auf Seite 19.)

[33] Vinton G Cerf and Edward Cain. The DoD internet architecture model, 1983. ISSN 03765075. (Zitiert auf Seite 18.)

[34] Karl Chen and David Wagner. Large-scale analysis of format string vulnerabilities in Debian Linux. *Proceedings of the 2007 workshop on Programming languages and analysis for security - PLAS '07*, page 75, 2007. doi: 10.1145/1255329.1255344. URL `http://portal.acm.org/citation.cfm?doid=1255329.1255344`. (Zitiert auf Seite 25 und 26.)

[35] Cisco Systems. Cisco Security Advisory: Undocumented Test Interface in Cisco Small Business Devices, 2014. URL `http://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20140110-sbd`. (Zitiert auf Seite 24.)

[36] Committee on Application of Digital Instrumentation and Control Systems to Nuclear Power Plant Operations and Safety. *Digital Instrumentation and Control Systems in Nuclear Power Plants*. 1997. ISBN 030952444X. (Zitiert auf Seite 30.)

[37] George Coulouris, Jean Dollimore, and Tim Kindberg. *Verteilte Systeme - Konzepte und Design*. Pearson Studium, 2002. ISBN 3-8273-7022-1. (Zitiert auf Seite xi, 7 und 8.)

[38] Tod Courtney, James Lyons, Harigovind V Ramasamy, William H Sanders, Mouna Seri, Michel Cukier, Michael Atighetchi, Paul Rubel, Christopher Jones, Franklin Webber, Partha Pal, Ronald Watro, and Jeanna Gossett. Providing Intrusion Tolerance With ITUA. In *Supplemental Volume of the 2002 International Conference on Dependable Systems & Networks (DSN-2002)*, pages 1–3, Washington, 2002. (Zitiert auf Seite 41.)

[39] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 2, pages 119–129. IEEE Comput. Soc. ISBN 0-7695-0490-6. doi: 10.1109/DISCEX.2000.821514. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=821514`. (Zitiert auf Seite 25.)

[40] Flavin Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, feb 1991. ISSN 00010782. doi: 10.1145/102792.102801. URL `http://dl.acm.org/citation.cfm?id=102801`. (Zitiert auf Seite 14.)

[41] Ang Cui and S Stolfo. Defending embedded systems with software symbiotes. In Robin Sommer, Davide Balzarotti, and Gregor Maier, editors, *Recent Advances in Intrusion Detection*, pages 358–377, Menlo Park, CA, USA, sep 2011. Springer Berlin Heidelberg. ISBN 978-3-642-23644-0. doi: 10.1007/978-3-642-23644-0_19. URL `http://www.springerlink.com/index/X6QX60648376L108.pdf`. (Zitiert auf Seite 27.)

[42] Michel Cukier, James Lyons, Prashant Pandey, HariGovind V. Ramasamy, William H. Sanders, Partha Pal, Franklin Webber, Richard Schantz, Joseph Loyall, Ronald Watro, Michael Atighetchi, and Jeanna Gossett. Intrusion Tolerance Approaches in ITUA. In *Supplement of the 2001 International Conference on Dependable Systems and Networks*, pages B–64 to B–65, Göteborg, 2001. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.16.6011{&}rep=rep1{&}type=pdf`. (Zitiert auf Seite 41.)

[43] Ivan Bjerre Damgård. A Design Principle for Hash Functions. In *Advances in Cryptology - CRYPTO '89 Proceedings*, pages 416–427. Springer New York, New York, NY, 1990. ISBN 0387973176. doi: 10.1007/0-387-34805-0_39. URL `http://link.springer.com/10.1007/0-387-34805-0{_}39`. (Zitiert auf Seite 34.)

[44] John D. Day and Hubert Zimmermann. OSI REFERENCE MODEL., 1983. ISSN 00189219. (Zitiert auf Seite 18.)

[45] Yves Deswarte, Karama Kanoun, and Jean-Claude Laprie. Diversity against accidental and deliberate faults. *Proceedings Computer Security, Dependability, and Assurance: From Needs to Solutions (Cat. No.98EX358)*, pages 171–181, 1998. doi: 10.1109/CSDA.1998.798364. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=798364`. (Zitiert auf Seite 23 und 27.)

[46] DKE Deutsche Kommission Elektrotechnik Elektronik Informationstechnik. IEV-Woerterbuch, 2011. URL `http://www.dke.de/de/Online-Service/DKE-IEV/Seiten/IEV-Woerterbuch.aspx?search=351-32`. (Zitiert auf Seite 18.)

[47] Danny Dolev and Andrew C. Yao. On the security of public key protocols. In *22nd Annual Symposium on Foundations of Computer Science (sfcs 1981)*, number M, pages 350–357. IEEE, oct 1981. ISBN 0018-9448. doi: 10.1109/SFCS.

1981.32. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4568353`. (Zitiert auf Seite 11.)

[48] Klaus Echtle. *Fehlertoleranzverfahren*. Studienreihe Informatik. Springer, 1990. ISBN 978-3-540-52680-3. (Zitiert auf Seite 3, 14, 15, 28, 29 und 57.)

[49] Dave E. Eckhardt, Alper K. Caglayan, John C. Knight, Larry D. Lee, David F. McAllister, Mladen A. Vouk, and John P.J. Kelly. An experimental evaluation of software redundancy as a strategy for improving reliability. *IEEE Transactions on Software Engineering*, 17(7):692–702, 1991. ISSN 00985589. doi: 10.1109/32.83905. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=83905`. (Zitiert auf Seite 29.)

[50] Taher Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *Proceedings of CRYPTO 84 on Advances in cryptology*, I:10–18, 1984. (Zitiert auf Seite 33.)

[51] Yuval Elovici and Lior Rokach. Reaction to New Security Threat Class. pages 1–13, jun 2014. URL `http://arxiv.org/abs/1406.3110`. (Zitiert auf Seite 11.)

[52] W Ertel. *Angewandte Kryptographie*. Carl Hanser Verlag GmbH & Company KG, 2012. ISBN 9783446431966. (Zitiert auf Seite 31.)

[53] ETAS GmbH. Arctic Core - the open source AUTOSAR embedded platform, 2014. URL `http://www.arccore.com/products/arctic-core/arctic-core-for-autosar-4-x/`. (Zitiert auf Seite 65.)

[54] Nicolas Falliere, LO Murchu, and Eric Chien. W32. stuxnet dossier. *White paper, Symantec Corp., Security ...*, 4(February): 1–69, 2011. URL `http://www.h4ckr.us/library/Documents/ICS{_}Events/StuxnetDossier(Symantec)v1.4.pdf`. (Zitiert auf Seite 4.)

[55] Yousef Farhaoui. Performance method of assessment of the intrusion detection and prevention systems. *International Journal of Engineering Science and Technology*, 3(7):5916–5928, 2011. (Zitiert auf Seite 20.)

[56] Samer Fayssal, Salim Hariri, and Youssif Al-Nashif. Anomaly-Based Behavior Analysis of Wireless Network Security. *2007 Fourth Annual International Conference on Mobile and Ubiquitous Systems: Networking & Services (MobiQuitous)*, pages 1–8, 2007. doi: 10.1109/MOBIQ.2007.4451054. (Zitiert auf Seite 20.)

[57] FlexRay Consortium. FlexRay Communications System - Protocol Specification Version 2.1, dec 2005. URL `http://flexray.com/index.php?pid=47{&}lang=de`. (Zitiert auf Seite 19 und 22.)

[58] Stephanie Forrest and A Somayaji. Building diverse computer systems. *Systems, 1997., The Sixth*, 1997. URL `http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=595185`. (Zitiert auf Seite 29.)

[59] Ian Foster, Andrew Prudhomme, Karl Koscher, and Stefan Savage. Fast and Vulnerable: A Story of Telematic Failures. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, Washington, D.C., 2015. USENIX Association. URL `http://blogs.usenix.org/conference/woot15/workshop-program/presentation/foster`. (Zitiert auf Seite 4.)

[60] James C. Foster, Vitaly Osipov, Nish Bhalla, and Niels Heinen. *Buffer Overflow Attacks*. Syngress Publishing, Inc., Rockland, MA, 2005. ISBN 1932266674. URL `http://www.usenix.org/publications/library/proceedings/sec98/full{_}papers/cowan/cowan{_}html/node3.html`. (Zitiert auf Seite 25.)

[61] Christian Freckmann and Ulrich Greveler. IT-Sicherheitsaspekte industrieller Steuerungssysteme. In *Sicherheit*, pages 149–156, 2014. (Zitiert auf Seite 13.)

[62] Greveler Freckmann. ICS-Security-Kompendium. 2013. URL `https://www.bsi.bund.de/DE/Themen/weitereThemen/ICS-Security/Empfehlungen/empfehlungen.html`. (Zitiert auf Seite 38.)

[63] Subodh Gangan. A Review of Man-in-the-Middle Attacks, apr 2015. (Zitiert auf Seite 21.)

[64] Miguel Garcia, Alysson Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. OS diversity for intrusion tolerance: Myth or reality? In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 383–394. IEEE, jun 2011. ISBN 978-1-4244-9232-9. doi: 10.1109/DSN.2011.5958251. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5958251`. (Zitiert auf Seite 29, 65 und 82.)

[65] Felix C. Gärtner. Byzantine Failures and Security: Arbitrary is not (always) Random. In Rüdiger Grimm, Hubert B. Keller, and Kai Rannenberg, editors, *GI Jahrestagung (Schwerpunkt SSicherheit - Schutz und Zuverlässigkeit")*, pages 127–138. Swiss Federal Institute of Technology (EPFL), School of Computer and Communication Sciences, GI, 2003. ISBN 3-88579-330-X. URL

http://lpdwww.epfl.ch/upload/documents/publications/neg--2092223713IC{_}TECH{_}REPORT{_}200320.pdf.gz. (Zitiert auf Seite 12.)

[66] James J. Gillogly. CIPHERTEXT-ONLY CRYPTANALYSIS OF ENIGMA. *Cryptologia*, 19(4):405–413, oct 1995. ISSN 0161-1194. doi: 10.1080/0161-119591884060. URL http://www.tandfonline.com/doi/abs/10.1080/0161-119591884060. (Zitiert auf Seite 74.)

[67] L. Gmeiner and U. Voges. Software diversity in reactor protection systems: An experiment. In R. Lauber, editor, *Safety of Computer Control Systems (Proceedings of IFAC Workshop on safety of computer control systems)*, pages 75–79. Pergamon Press, 1979. ISBN 0-08-024453-X. (Zitiert auf Seite 28 und 29.)

[68] Dieter Gollmann, Pavel Gurikov, Alexander Isakov, Marina Krotofil, Jason Larsen, and Alexander Winnicki. Cyber-Physical Systems Security. In *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security - CPSS '15*, number JANUARY, pages 1–12, New York, New York, USA, 2015. ACM Press. ISBN 9781450334488. doi: 10.1145/2732198.2732208. URL http://dl.acm.org/citation.cfm?doid=2732198.2732208. (Zitiert auf Seite 12.)

[69] F. Gont. ICMP Attacks against TCP. Technical report, Internet Engineering Task Force (IETF), 2010. (Zitiert auf Seite 20.)

[70] Jim Gray. Why Do Computers Stop and What Can Be Done About It? *Office*, 3(June):3–12, 1986. doi: 10.1.1.59.6561. URL http://www.hpl.hp.com/techreports/tandem/TR-85.7.pdf. (Zitiert auf Seite 23.)

[71] Andy Greenberg. Hackers Remotely Kill a Jeep on the Highway - With Me in It. *WIRED*, 2015. URL http://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/. (Zitiert auf Seite 4.)

[72] Hadeli Hadeli, Ragnar Schierholz, Markus Braendle, and Cristian Tuduce. Leveraging Determinism in Industrial Control Systems for Advanced Anomaly Detection and Reliable Security Configuration. In *Proceedings of the 14th IEEE International Conference on Emerging Technologies and Factory Automation*, pages 1–8. IEEE, 2009. ISBN 9781424427277. doi: 10.1109/ETFA.2009.5347134. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5347134. (Zitiert auf Seite 27.)

[73] Gunnar Hagelin. ERICSSON Safety System for Railway Control. In Udo Voges, editor, *Software Diversity in Computerized*

Control Systems, pages 11–21. Springer, 1988. ISBN 978-3-7091-8934-4. doi: 10.1007/978-3-7091-8932-0_2. URL `http://www.springerlink.com/index/10.1007/978-3-7091-8932-0{_}2`. (Zitiert auf Seite 29.)

[74] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. 2003. ISBN 038795273X. doi: 10.1007/b97644. (Zitiert auf Seite 33.)

[75] Simon Hansman and Ray Hunt. A taxonomy of network and computer attacks. *Computers and Security*, 24:31–43, 2005. ISSN 01674048. doi: 10.1016/j.cose.2004.06.011. (Zitiert auf Seite 20.)

[76] Carl Hartung, James Balasalle, and Richard Han. Node compromise in sensor networks: The need for secure systems. Technical Report January, 2005. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.134.8146{&}rep=rep1{&}type=pdf`. (Zitiert auf Seite 24.)

[77] Maurice P. Herlihy and J. D. Tygar. How to Make Replicated Data Secure. In *CRYPTO '87 A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, volume 293, pages 379–391, 1988. ISBN 3-540-18796-0. (Zitiert auf Seite 37.)

[78] Amir Herzberg, Haya Shulman, and Ramat Gan. Antidotes for DNS Poisoning by Off-Path Adversaries. In *Seventh International Conference on Availability, Reliability and Security*, Prague, Czech Republic, aug 2012. The Institute of Electrical and Electronics Engineers, Inc. ISBN 9780769547756. doi: 10.1109/ARES.2012.27. (Zitiert auf Seite 21.)

[79] Martin Hirt and Ueli Maurer. Player Simulation and General Adversary Structures in Perfect Multiparty Computation. *Journal of Cryptology*, 13(1):31–60, jan 2000. ISSN 0933-2790. doi: 10.1007/s001459910003. URL `http://link.springer.com/10.1007/s001459910003`. (Zitiert auf Seite 37.)

[80] Taylor Hornby. Write Crypto Code! Don't publish it!, 2014. URL `http://www.cryptofails.com/post/75204435608/write-crypto-code-dont-publish-it`. (Zitiert auf Seite 75.)

[81] R. Housley. Cryptographic Message Syntax (CMS). *RFC 5652*, pages 1–57, 2009. URL `http://tools.ietf.org/html/rfc5652`. (Zitiert auf Seite 34.)

[82] IEEE 802 Working Group. IEEE 802.3: ETHERNET, 2008. URL `http://standards.ieee.org/about/get/802/802.3.html`. (Zitiert auf Seite 18.)

[83] JC CREW. Undocumented Backdoor Access to RuggedCom Devices, apr 2012. URL `http://seclists.org/fulldisclosure/2012/Apr/277`. (Zitiert auf Seite 24.)

[84] Michael Jenkins and SM Mahmud. Security needs for the future intelligent vehicles. In *2006 SAE World Congress*, number 724, Detroit, MI, USA, 2006. URL `http://www.eng.wayne.edu/{~}smahmud/PersonalData/PubPapers/SAE{_}2006-01-1426.pdf`. (Zitiert auf Seite 20.)

[85] Mark K. Joseph and Algirdas Avižienis. A fault tolerance approach to computer viruses. In *Proceedings. 1988 IEEE Symposium on Security and Privacy*, pages 52–58. IEEE Comput. Soc. Press, 1988. ISBN 0-8186-0850-1. doi: 10.1109/SECPRI.1988.8097. URL `http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=8097`. (Zitiert auf Seite 39, 40 und 44.)

[86] Abdel Alim Kamal. A Scan-Based Side Channel Attack on the NTRUEncrypt Cryptosystem. In *Seventh International Conference on Availability, Reliability and Security*, Prague, Czech Republic, aug 2012. The Institute of Electrical and Electronics Engineers, Inc. ISBN 9780769547756. doi: 10.1109/ARES.2012.14. (Zitiert auf Seite 75.)

[87] Karama Kanoun. Real-world design diversity: a case study on cost. *IEEE Software*, 18(August):29–33, 2001. ISSN 0740-7459. URL `http://dl.acm.org/citation.cfm?id=626289`. (Zitiert auf Seite 29.)

[88] Heinz Kantz and Christian Koza. The ELEKTRA railway signalling system: field experience with an actively replicated system with diversity. *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pages 453–458, 1995. ISSN 07313071. doi: 10.1109/FTCS.1995.466954. (Zitiert auf Seite 29.)

[89] John P.J. Kelly, Thomas I. McVittie, and Wayne I. Yamamoto. Implementing design diversity to achieve fault tolerance. *IEEE Software*, 8(4):61–71, jul 1991. ISSN 0740-7459. doi: 10.1109/52.300038. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=300038`. (Zitiert auf Seite 28.)

[90] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side Channel Cryptanalysis of Product Ciphers. *Building*, 8(2–3):1–18, 1998. ISSN 0926-227X. (Zitiert auf Seite 75.)

[91] S Kent and R Atkinson. Request for Comments: 4301 - Security Architecture for the Internet Protocol Status, 2005. URL `http://tools.ietf.org/pdf/rfc4301.pdf`. (Zitiert auf Seite 66.)

[92] Thorsten Kimmeskamp. *Entfernte Redundanz - Beschreibung, Analyse und Bewertung eines neuartigen Architekturkonzeptes für fehlertolerante Steuerungssysteme*. Dissertation, Universität Dusburg-Essen, sep 2011. (Zitiert auf Seite 3, 22 und 38.)

[93] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, SE-12 (1):96–109, jan 1986. ISSN 0098-5589. doi: 10.1109/TSE.1986.6312924. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6312924`. (Zitiert auf Seite 29.)

[94] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental Security Analysis of a Modern Automobile. In *2010 IEEE Symposium on Security and Privacy*, pages 447–462. IEEE, 2010. ISBN 978-1-4244-6894-2. doi: 10.1109/SP.2010.34. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5504804`. (Zitiert auf Seite 4 und 21.)

[95] Jean Claude Laprie. DEPENDABLE COMPUTING AND FAULT TOLERANCE : CONCEPTS AND TERMINOLOGY. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'.*, page 2. IEEE, 1995. ISBN 0-8186-7150-5. doi: 10.1109/FTCSH.1995.532603. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=532603`. (Zitiert auf Seite 14.)

[96] Jean Claude Laprie, Jean Arlat, Christian Beounes, and Karama Kanoun. Definition and analysis of hardware- and software-fault-tolerant architectures. *Computer*, 23(7):39–51, 1990. ISSN 00189162. doi: 10.1109/2.56851. (Zitiert auf Seite 29.)

[97] Nancy G. Leveson. A systems-theoretic approach to safety in software-intensive systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):66–86, 2004. ISSN 15455971. doi: 10.1109/TDSC.2004.1. (Zitiert auf Seite 14.)

[98] Kyung S. Lhee and Steve J. Chapin. Buffer overflow and format string overflow vulnerabilities. *Software - Practice and Experience*, 33(5):423–460, 2003. ISSN 00380644. doi: 10.1002/spe.515. (Zitiert auf Seite 25.)

[99] Yehuda Lindell and Benny Pinkas. Secure multiparty computation for privacy-preserving data mining. *Journal of Privacy and Confidentiality*, (860):59–98, 2009. URL `http://repository.cmu.edu/cgi/viewcontent.cgi?article=1004{&}context=jpc`. (Zitiert auf Seite 37.)

[100] Michael Lyu and Algirdas Avižienis. Assuring design diversity in N-version software: a design paradigm for N-version programming. In *2nd Dependable Computing for Critical Applications*, Tucson, Arizona, 1992. doi: 10.1.1.46.273. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.46.2733{&}rep=rep1{&}type=pdf`. (Zitiert auf Seite 28.)

[101] Michael R. Lyu, Jia-Hong CHen, and Algirdas Avižienis. Experience in Metrics and Measurements for N-Version Programming. *International Journal of Reliability, Quality and Safety Engineering*, 01(01):41–62, 1994. ISSN 0218-5393. doi: 10.1142/S0218539394000052. URL `http://www.worldscientific.com/doi/abs/10.1142/S0218539394000052`. (Zitiert auf Seite 28.)

[102] Jonas Magazinius. [Cryptography] OpenPGP SEIP downgrade attack, oct 2015. URL `http://www.metzdowd.com/pipermail/cryptography/2015-October/026685.html`. (Zitiert auf Seite 75.)

[103] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, oct 1998. ISSN 01782770. doi: 10.1007/s004460050050. URL `http://link.springer.com/10.1007/s004460050050`. (Zitiert auf Seite 37.)

[104] Adil Mudasir Malla. Security Attacks with an Effective Solution for DOS Attacks in VANET. *International Journal of Computer Applications*, 66(22):45–49, 2013. (Zitiert auf Seite 20.)

[105] Guillermo Mario Marro. *Attacks at the Data Link Layer*. PhD thesis, UNIVERSITY OF CALIFORNIA, 2003. (Zitiert auf Seite 21.)

[106] Ulrich Maschek. *Elektronische Stellwerke - ein internationaler Überblick*. Diplomarbeit, Technischen Universität Dresden, 1996. (Zitiert auf Seite 30.)

[107] James L. Massey and Xuejia Lai. DEVICE FOR THE CONVERSION OF A DIGITAL BLOCK AND USE OF SAME, 1992. URL `http://worldwide.espacenet.com/publicationDetails/biblio?locale=de{_}EP{&}CC=US{&}NR=5214703`. (Zitiert auf Seite 32.)

[108] Alexandre Maurer, Sébastien Tixeuil, and Xavier Défago. Reliable Communication in a Dynamic Network in the Presence of Byzantine Faults. pages 1–15, feb 2014. URL `http://arxiv.org/abs/1402.0121`. (Zitiert auf Seite 3.)

[109] Bill Miller and Dale Rowe. A survey SCADA of and critical infrastructure incidents. *Proceedings of the 1st Annual conference on Research in information technology - RIIT '12*, page 51, 2012. ISSN 9781450316439. doi: 10.1145/2380790.2380805. URL `http:`

//dl.acm.org/citation.cfm?doid=2380790.2380805. (Zitiert auf Seite 4 und 11.)

[110] Charlie Miller and Chris Valasek. Remote Exploitation of an Unaltered Passenger Vehicle, 2015. (Zitiert auf Seite 4.)

[111] Kevin D. Mitnick and William L. Simon. The Art of Deception: Controlling the Human Element in Security. *BMJ: British Medical Journal*, page 368, 2003. doi: 0471237124. URL http://www.bmj.com/content/347/bmj.f5889. (Zitiert auf Seite 13.)

[112] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE Bites : Exploiting The, sep 2014. URL https://www.openssl.org/{~}bodo/ssl-poodle.pdf. (Zitiert auf Seite 75.)

[113] Thomas Morris, Bradley Reaves, and Drew Richey. On SCADA control system command and response injection and intrusion detection. *2010 eCrime Researchers Summit*, pages 1–9, 2010. doi: 10.1109/ecrime.2010.5706699. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5706699. (Zitiert auf Seite 38.)

[114] Max Mühlhäuser and Alexander Schill. *Software Engineering für verteilte Anwendungen*. Springer Berlin / Heidelberg, 1992. ISBN 3-540-55412-2. (Zitiert auf Seite 7.)

[115] Moni Naor and Moti Yung. Universal one-way hash functions and their cryptographic applications. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing - STOC '89*, pages 33–43, New York, New York, USA, 1989. ACM Press. ISBN 0897913078. doi: 10.1145/73007.73011. URL http://portal.acm.org/citation.cfm?doid=73007.73011. (Zitiert auf Seite 34.)

[116] National Institute of Standards and Technology (NIST). 46-3: Data Encryption Standard (DES), oct 1999. (Zitiert auf Seite 32.)

[117] National Institute of Standards and Technology (NIST). 197: Announcing the advanced encryption standard (AES), nov 2001. URL http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf. (Zitiert auf Seite 32.)

[118] National Institute of Standards and Technology (NIST). Vulnerability Summary for CVE-2004-1921, 2004. URL https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2004-1921. (Zitiert auf Seite 24.)

[119] Phong Q Nguyen. Can We Trust Cryptographic Software? Cryptographic Flaws in GNU Privacy Guard v1.2.3. *Advances*

in Cryptology - EUROCRYPT 2004, pages 555–570, 2004. ISSN 03029743. doi: 10.1007/978-3-540-24676-3_33. (Zitiert auf Seite 74.)

[120] Nick Nikiforakis, Yves Younan, and Wouter Joosen. HProxy: Client-side detection of SSL stripping attacks. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6201 LNCS, pages 200–218, 2010. ISBN 3642142141. doi: 10.1007/978-3-642-14215-4_12. (Zitiert auf Seite 21.)

[121] Paul Oman, E Schweitzer, and Jeff Roberts. Safeguarding IEDs, substations, and SCADA systems against electronic intrusions. In *2001 Western Power Delivery Automation Conference*, pages 1–18, 2001. URL `http://www2.selinc.com/techpprs/6118.pdf`. (Zitiert auf Seite 22 und 37.)

[122] Andreas Paul, Franka Schuster, and K Hartmut. Towards the Protection of Industrial Control Systems - Conclusions of a Vulnerability Analysis of Profinet IO. In Konrad Rieck, Patrick Stewin, and Jean-Pierre Seifert, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment, 10th International Conference*, pages 160–176, Berlin, 2013. Springer Berlin Heidelberg. doi: 10.1007/978-3-642-39235-1_10. (Zitiert auf Seite 38.)

[123] Konstantinos Pelechrinis, Marios Iliofotou, and Srikanth V. Krishnamurthy. Denial of Service Attacks in Wireless Networks: The Case of Jammers. *IEEE Communications Surveys & Tutorials*, 13(2):245–257, 2011. ISSN 1553-877X. doi: 10.1109/SURV.2011.041110.00022. (Zitiert auf Seite 20.)

[124] Jon Postel. RFC 791 Internet Protocol, 1981. URL `http://www.ietf.org/rfc/rfc791.txt`. (Zitiert auf Seite 19.)

[125] David Powell. Failure mode assumptions and assumption coverage. In *Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing*, pages 386–395. IEEE, 1992. ISBN 0-8186-2875-8. doi: 10.1109/FTCS.1992.243562. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=243562`. (Zitiert auf Seite 79.)

[126] David Powell and Robert Stroud. Conceptual model and architecture of MAFTIA. Technical report, 2003. URL `http://www.cs.newcastle.ac.uk/research/pubs/trs/papers/787.pdf`. (Zitiert auf Seite 42.)

[127] Thomas H. Ptacek and Timothy N. Newsham. Insertion, Envasion, and Denial of Service: Eluding network intrusion detection. Technical report, SECURE NETWORKS INC, CALGARY ALBERTA, 1998. URL `http://oai.dtic.mil/oai/oai?verb=`

getRecord{&}metadataPrefix=html{&}identifier=ADA391565. (Zitiert auf Seite 20.)

[128] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, apr 1989. ISSN 00045411. doi: 10.1145/62044.62050. URL http://portal.acm.org/citation.cfm?doid=62044.62050. (Zitiert auf Seite 37.)

[129] Dirk Rijmenants. Is One-time Pad History ? *Cipher Machines & Cryptology*, pages 1–4, 2009. URL http://users.telenet.be/d.rijmenants. (Zitiert auf Seite 32.)

[130] Allen Risley, Jeff Roberts, and Peter LaDow. Electronic security of real-time protection and SCADA communications. *5th Annual Western Power Delivery Automation Conference*, apr 2003. URL http://www2.selinc.com/techpprs/6150.pdf. (Zitiert auf Seite 22.)

[131] R. L. Rivest, a. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. ISSN 00010782. doi: 10.1145/359340.359342. (Zitiert auf Seite 33.)

[132] Robert Bosch Gmbh. CAN specification version 2.0, sep 1991. (Zitiert auf Seite 19 und 20.)

[133] Mohammad Roohitavaf and Sandeep Kulkarni. Stabilization and Fault-Tolerance in Presence of Unchangeable Environment Actions. 2015. URL http://arxiv.org/abs/1508.00864. (Zitiert auf Seite 3.)

[134] Klaus Schmeh. *Kryptografie: Verfahren, Protokolle, Infrastrukturen*. dpunkt.Verlag, Heidelberg, 4. edition, 2009. ISBN 9783898646024. (Zitiert auf Seite 31, 32, 33, 34, 35 und 36.)

[135] Bruce Schneier. Attack Trees. *Dr Dobbs Journal*, 24(12):21–29, 1999. ISSN 1044-789X. URL http://www.schneier.com/paper-attacktrees-ddj-ft.html. (Zitiert auf Seite 11.)

[136] Mark M Seeger and Stephen D Wolthusen. Towards Concurrent Data Sampling using GPU Coprocessing. In *Seventh International Conference on Availability, Reliability and Security*, Prague, Czech Republic, aug 2012. The Institute of Electrical and Electronics Engineers, Inc. ISBN 9780769547756. doi: 10.1109/ARES.2012.92. (Zitiert auf Seite 27.)

[137] Aakash Shah, Adrian Perrig, and Bruno Sinopoli. Mechanisms to provide integrity in SCADA and PCS devices *. In *International Workshop on Cyber-Physical Systems*

- Challenges and Applications (CPS-CA '08), Santorini, Greece, jun 2008. URL `http://sparrow.ece.cmu.edu/group/pub/shah-perrig-sinopoli-cps-2008.pdf`. (Zitiert auf Seite 27.)

[138] Timothy J. Shimeall and Nancy G. Leveson. An empirical comparison of software fault tolerance and fault elimination. *IEEE Transactions on Software Engineering*, 17(2):173–182, 1991. ISSN 00985589. doi: 10.1109/32.67598. URL `http://dx.doi.org/10.1109/32.67598`. (Zitiert auf Seite 29.)

[139] Siemens. SIMATIC S7 Steuerung Modular Controller, 2014. URL `http://w3.siemens.com/mcms/programmable-logic-controller/de/simatic-s7-controller/Seiten/Default.aspx`. (Zitiert auf Seite 66.)

[140] Markus Siepermann, Tobias Kollmann, Richard Lackes, and Jochen Metzger. Gabler Wirtschaftslexikon, Stichwort: Kryptographie, 2015. URL `http://wirtschaftslexikon.gabler.de/Archiv/5160/kryptographie-v10.html`. (Zitiert auf Seite 31.)

[141] Victoria Stavridou, Bruno Dutertre, R.A. Riemenschneider, and Hassen Saidi. Intrusion tolerant software architectures. In *Proceedings DARPA Information Survivability Conference and Exposition II. DISCEX'01*, volume 2, pages 230–241. IEEE Comput. Soc, 2001. ISBN 0-7695-1212-7. doi: 10.1109/DISCEX.2001.932175. URL `http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=932175`. (Zitiert auf Seite 38.)

[142] R Stewart. Stream Control Transmission Protocol, sep 2007. URL `http://tools.ietf.org/rfc/rfc4960.txt`. (Zitiert auf Seite 19.)

[143] Joachim Swoboda, Stephan Spitz, and Michael Pramateftakis. *Kryptographie und IT-Sicherheit*. Vieweg+Teubner Verlag, 2. edition, 2011. ISBN 978-3-8348-1487-6. (Zitiert auf Seite 32 und 35.)

[144] Andrew S. Tanenbaum and Marten van Steen. *Distributed Systems - Principles and Paradigms*. Pearson Education Inc, second edition, 2007. ISBN 0-13-613553-6. (Zitiert auf Seite 7.)

[145] Hugo Teso. Aircraft Hacking - Practical Aero Series, apr 2013. URL `http://conference.hitb.org/hitbsecconf2013ams/materials/D1T1-HugoTeso-AircraftHacking-PracticalAeroSeries.pdf`. (Zitiert auf Seite 4.)

[146] Thomei08. Wikimedia Commons - Linux Kernel Struktur.svg, 2008. URL `https://commons.wikimedia.org/wiki/File:Linux{_}Kernel{_}Struktur.svg`. (Zitiert auf Seite xi, 63 und 80.)

[147] Luis A. Trejo, Raúl Monroy, and Rafael López Monsalvo. Spanning Tree Protocol and Ethernet PAUSE Frames DDoS Attacks : Their Efficient Mitigation. 2. (Zitiert auf Seite 20.)

[148] Albert Treytl, Thilo Sauter, and Christian Schwaiger. Security measures for industrial fieldbus systems - state of the art and solutions for IP-based approaches. In *IEEE International Workshop on Factory Communication Systems, 2004. Proceedings.*, pages 201–209. IEEE, sep 2004. ISBN 0-7803-8734-1. doi: 10.1109/WFCS.2004.1377709. URL `http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=1377709`. (Zitiert auf Seite 37.)

[149] Vector Informatik GmbH. MICROSAR - Die AUTOSAR-Basissoftware von Vector, 2014. URL `http://vector.com/vi{_}microsar{_}de.html`. (Zitiert auf Seite 65.)

[150] Dazhi Wang, Bharat B. Madan, and Kishor S. Trivedi. Security analysis of SITAR intrusion tolerance system. In *Proceedings of the 2003 ACM workshop on Survivable and self-regenerative systems in association with 10th ACM Conference on Computer and Communications Security - SSRS '03*, pages 23–32, New York, New York, USA, 2003. ACM Press. ISBN 1581137842. doi: 10.1145/1036921.1036924. URL `http://dl.acm.org/citation.cfm?id=1036924`. (Zitiert auf Seite xi und 42.)

[151] Nicholas Weaver, Robin Sommer, and Vern Paxson. Detecting Forged TCP Reset Packets. *Proceedings of NDSS*, page 15, 2009. URL `http://www.icir.org/vern/papers/reset-injection.ndss09.pdf`. (Zitiert auf Seite 20.)

[152] Ira S Winkler and Brian Dealy. Information Security Technology?...Don't Rely on It A Case Study in Social Engineering. *Science Applications International Corporation*, (June):1–6, 1995. URL `http://static.usenix.org/publications/library/proceedings/security95/full{_}papers/winkler.pdf`. (Zitiert auf Seite 13.)

[153] Wayne Wolf. Cyber-physical Systems. *IEEE Embedded Computing*, pages 88–89, 2009. ISSN 0018-9162. doi: 10.1109/MC.2009.81. URL `http://www.cps-cn.org/Download/Cyber{_}physical{_}Systems.pdf`. (Zitiert auf Seite 3.)

[154] Anthony D. Wood and John A. Stankovic. Denial of service in sensor networks. *Computer*, 35:54–62, 2002. ISSN 00189162. doi: 10.1109/MC.2002.1039518. URL `http://dl.acm.org/citation.cfm?id=168607`. (Zitiert auf Seite 20.)

[155] Mark Yason. Use-after-frees: That pointer may be pointing to something bad, apr

2013. URL https://securityintelligence.com/use-after-frees-that-pointer-may-be-pointing-to-something-bad/. (Zitiert auf Seite 80.)

[156] Ying C. Yeh. Design considerations in Boeing 777 fly-by-wire computers. *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No.98EX231)*, pages 64–72, 1998. doi: 10.1109/HASE.1998.731596. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=731596. (Zitiert auf Seite 30.)

[157] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. *ACM SIGOPS Operating Systems Review*, 37(5):253, dec 2003. ISSN 01635980. doi: 10.1145/1165389.945470. URL http://portal.acm.org/citation.cfm?doid=1165389.945470. (Zitiert auf Seite 37.)

[158] Bonnie Zhu and Sastry Sastry. SCADA-specific intrusion detection/prevention systems: a survey and taxonomy. In *First Workshop on Secure Control Systems (SCS'10)*, Stockholm, Sweden, 2010. URL http://www.cse.psu.edu/{~}smclaugh/cse598e-f11/papers/zhu.pdf. (Zitiert auf Seite 27 und 38.)

[159] Bonnie Zhu, Anthony Joseph, and Shankar Sastry. A Taxonomy of Cyber Attacks on SCADA Systems. In *2011 International Conference on Internet of Things and 4th International Conference on Cyber, Physical and Social Computing*, pages 380–388. IEEE, oct 2011. ISBN 978-1-4577-1976-9. doi: 10.1109/iThings/CPSCom.2011.34. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6142258. (Zitiert auf Seite 24 und 37.)

[160] ZigBee Alliance. Zigbee Specification. Technical report, 2012. (Zitiert auf Seite 19.)

[161] Phil Zimmermann. An Introduction to Cryptography. Technical report, Network Associates, Inc., 2000. URL ftp://ftp.pgpi.org/pub/pgp/7.0/docs/english/IntroToCrypto.pdf. (Zitiert auf Seite 75.)

[162] Zeljka Zorz and Berislav Kucan. Hijacking airplanes with an Android phone, apr 2013. URL http://www.net-security.org/secworld.php?id=14733. (Zitiert auf Seite 4.)