Masterarbeit

Methodik zur Identifikation von Laufzeitabhängigkeiten für Smartcards

Stefan Orehovec, BSc

Institut für Technische Informatik
Technische Universität Graz

Begutachter: Ass.Prof. Dipl.-Ing. Dr. techn. Christian Steger
Betreuer: Ass.Prof. Dipl.-Ing. Dr. techn. Christian Steger
Dipl.-Ing. Johannes Loinig (NXP Semiconductors Austria GmbH)

Graz, im Oktober 2011

Kurzfassung

Immer wieder erfährt man aus den Medien, dass ein sicherheitskritisches System zum Opfer einer Timing Attacke wurde. Timing Attacken gehören zu den Seitenkanalattacken, wobei die Zeit als Seitenkanal dient. Diese nutzen Informationen von Seitenkanälen aus, um einen bestehenden Sicherheitsmechanismus zu kompromittieren. Dabei wird nicht der zu Grunde liegende Algorithmus angegriffen, sondern die Implementation des Algorithmus.

Der Hardware/Software Codesign Flow in dieser Masterarbeit soll dem Entwickler helfen, ein System zu entwerfen, dass keine potentiellen Timing-Probleme bei sicherheitskritischen Funktionen aufweist. Dafür werden bei der Entwicklung zeitkritische Bereiche als sensibel markiert. Diese Markierungen dienen zum Messen der Laufzeit. Ein Hardware/Software Codesign Flow ist eine Entwurfsmethode bei der die Hardware- und Softwarekomponenten gemeinsam entwickelt werden. Dabei wird untersucht welche Funktionen in Hardware und welche Funktionen in Software implementiert werden, um einen guten Kompromiss zwischen den Anforderungen zu erhalten.

Für die Evaluierung des Fortschritts und der Funktionalität des Hardware/Software Systems werden Anwendungs- und Testfälle verwendet. Bei ihrer Abarbeitung werden die Laufzeiten der einzelnen Bereiche aufgenommen und im Anschluss statistisch analysiert. Werden dabei Timingprobleme detektiert, so erfolgt ein Warnhinweis für die entsprechende Funktion. Dies soll das System bereits in der Entwicklung vor der Möglichkeit schützen, durch eine Timing Attacke kompromittiert zu werden.

Abstract

The mass media reports from time to time of timing attacks. This attack belongs to the side channel attacks. The side channel attacks use information of side channels like power consumption or time to compromise a security system. They do not attack the security algorithm but the implementation of the algorithm. The timing attack uses the time as side channel. Timing attacks also require some assumptions of the implementation of the algorithm. This is enough to break into a security system.

A Hardware/Software Codesign Flow is a concurrent development of hardware and software. The decision to build a component in hardware or software is made during developing a system by its requirements. The requirements could be costs, performance, size and so on. It also could be combinations of them.

The Hardware/Software Codesign Flow in this master thesis helps the developer of a system to prevent the system of timing attacks. The developer has to mark every timing critical section in the source code. These marks are used to measure the execution time. The time measurement takes place during the execution of use cases.

The evaluation uses the timing results to verify the timing behaviour of the system. Every timing critical section is viewed separately by statistical analysis. The timing report consists of the results from timing analysis. This report helps the developer to find timing problems and to reduce attacks to the system.

Keywords: Hardware/Software Codesign Flow, Smartcard, Time Measurement, Timing Attack.

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz,am ....

(Unterschrift)

Inhaltsverzeichnis

1 Einleitung **9**
1.1 Motivation . . . 9
1.2 Zielsetzung . . . 10
1.3 Gliederung . . . 10

2 Grundlagen **12**
2.1 Seitenkanalattacken . . . 12
2.2 Smartcard . . . 14
2.2.1 Hardware Sicherheitsmechanismen gegen Attacken auf Smartcards . 15
2.3 Die Timing Attacke . . . 17
2.3.1 Gegenmaßnahmen bei Timing Attacken . . . 22
2.4 Statistische Grundlagen . . . 27
2.4.1 Normalverteilung . . . 28
2.4.2 Stichprobe . . . 28
2.4.3 Signifikanz . . . 29
2.4.4 Anwendung von Signifikanztests bei der Timing Attacke . . . 32

3 Design **35**
3.1 Funktionalität des Hardware/Software Codesign Flow . . . 37
3.2 Abhängigkeit der Prozesse des Hardware/Software Codesign Flow . . . 39
3.3 Entwicklungsansicht . . . 41
3.4 Physikalische Ansicht . . . 43

4 Implementierung **44**
4.1 Anforderungen . . . 44
4.1.1 Smartcard Hardware . . . 45
4.1.2 Modell der Smartcard Hardware . . . 49
4.1.3 Smartcard Betriebssystem . . . 49
4.1.4 Grundlegende Überlegungen zur Implementation aufgrund der Anforderungen . . . 52
4.2 Die Zeitmessung . . . 54
4.2.1 Aufbau der Messeinheit . . . 55
4.2.2 Integration in das Betriebssystem . . . 59
4.3 Übertragungssoftware für die Messdaten . . . 60
4.3.1 Die Messdatentransfer-Software . . . 61
4.3.2 Die Messdatenauslese-Anwendung . . . 64

4.4 Die Auswertesoftware 66
4.4.1 Anwendungsfeld der Auswertesoftware 66
4.4.2 Ansatz zur Laufzeitanalyse der Auswertesoftware 67
4.4.3 Implementation der Auswertesoftware 68

5 Ergebnisse 70
5.1 Messverhalten der Zeitmesseinheit 70
5.1.1 Bestimmung der Genauigkeit der Messergebnisse 70
5.1.2 Bestimmung des Verhaltens beim Messen einer konstanten Laufzeit . 75
5.2 Beispiel 77
5.2.1 Spezifikation der Beispielanwendung 78
5.2.2 Implementierung der Prepaid-Karten Anwendung 78
5.2.3 Messergebnisse 80

6 Zusammenfassung 92
6.1 Ausblick 93

Abkürzungsverzeichnis 94

Literaturverzeichnis 95

Abbildungsverzeichnis

2.1 Kryptografisches Modell mit Seitenkanälen 13
2.2 Synchrone Smartcard oder Speicherchipkarte 15
2.3 Asynchrone Smartcard oder Prozessorchipkarte 15
2.4 Normalverteilung von Mittelwerten μ mit Vertrauensintervall angegeben in Sekunden 30

3.1 Herkömmlicher Hardware/Software Codesign Flow 36
3.2 Hardware/Software Codesign Flow gegen Timing Attacken 36
3.3 Logische Ansicht 38
3.4 Prozessansicht 40
3.5 Entwicklungsansicht 42
3.6 Physikalische Ansicht 43

4.1 Schichten eines Smartcard Systems 45
4.2 Architektur des 8051 Mikrokontrollers 47
4.3 Auftreten und abarbeiten eines Interrupts 48
4.4 Aufbau des Smartcardsystems 50
4.5 Aufbau der Speicherverwaltung des Smartcard Betriebssystems 51
4.6 Erweiterung des 16 Bit Hardwaretimer auf 32 Bit 56
4.7 Format der im EEPROM gespeicherten Messdaten 58
4.8 Schematischer Vorgang des Datentransfers zwischem Smartcard und Computer 61
4.9 Ablauf des Übertragungsvorgangs der Messdatentransfer-Software 62
4.10 MessdatenGröße - Kommando für die Bestimmung der zu Übertragen Datengröße 64
4.11 HoleMessdaten - Kommando für den Datentransfer von der Smartcard zum Computer 65
4.12 Klassenaufbau der Auswertesoftware 68
4.13 Grafische Benutzerschnittstelle für die Auswertesoftware 69

5.1 Verteilung der Messergebnisse bei einer konstanten Laufzeit von zirka 0,5 Sekunden 76
5.2 Programmabläufe der Prepaid-Kartenanwendung 77
5.3 Laufzeitverhalten der naiven Implementierung 83
5.4 Laufzeitergebnisse der 100 Anwendungsfälle bei der naiven Implementierung 83
5.5 Laufzeitverhalten der zustandsabhängigen Implementierung 85

5.6 Laufzeitergebnisse der 100 Anwendungsfälle bei der zustandsabhängigen Implementierung 87
5.7 Laufzeitverhalten der bedachten Implementierung 87
5.8 Laufzeitverhalten der Dual Rail Implementierung 90
5.9 Messergebnisse der zustandsunabhängigen Implementierung 91

Tabellenverzeichnis

2.1 Auftrittswahrscheinlichkeit der möglichen Kombinationen 31

5.1 Messergebnisse des Simulators, des Emulators und der Zeitmesseinheit (ZME) 74
5.2 Vergleich der Länge und Anzahl an übereinstimmenden Zeichen beim gültigen Authentifizierungscode '1234' 82
5.3 Ergebnisse der Laufzeitanalyse der naiven Implementierung anhand der Anwendungsfälle 84
5.4 Ergebnisse der Laufzeitanalyse der zustandsabhängigen Implementierung anhand der Anwendungsfälle 86
5.5 Ergebnisse der Laufzeitanalyse der bedachten Implementierung anhand der Anwendungsfälle 88
5.6 Ergebnisse der Laufzeitanalyse der Dual Rail Implementierung anhand der Anwendungsfälle 89
5.7 Ergebnisse der Laufzeitanalyse der zustandsunabhängigen Implementierung anhand der Anwendungsfälle 91

Kapitel 1

Einleitung

1.1 Motivation

In den letzten Jahren gewannen Smartcard-Systeme immer größere Bedeutung. Sie finden in vielen unterschiedlichen Bereichen Anwendung und werden zum Beispiel für Kreditkarten, Reisepässe, Tickets und noch vieles mehr eingesetzt. Dabei ist ihre Aufgabe das Identifizieren eines Nutzers, das sichere Speichern seiner persönlichen Daten und der Aufbau einer sicheren Kommunikation.

Um diese Aufgaben bewältigen zu können, ist ein gutes Sicherheitskonzept erforderlich. Der Aufwand, der dabei betrieben wird, hängt natürlich stark von der Applikation, dem Anwendungsgebiet und dem Schaden, den ein Fehler verursachen kann, ab.

Das bedeutet, eine jede Anwendung hat ihre charakteristischen Merkmale. Diese Merkmale bestimmen welche Sicherheitsmechanismen eingesetzt werden. Zum Beispiel ein Ticket-System für eine U-Bahn verlangt kurze Wartezeiten für den Sicherheitscheck. Wird hingegen eine gefälschte Fahrkarte als gültige Fahrkarte erkannt, ist dies zwar auch kritisch, aber vergleichsweise unbedeutend. Der Grund dafür liegt auf der Hand, eine U-Bahnfahrt ist relativ günstig und vergleichsweise nur durch einen hohen Aufwand zu fälschen. Im Vergleich dazu verursachen lange Wartezeiten oder gar ein Stau im Ein- und Ausgangsbereich der U-Bahn, auf Grund von zu komplexen Sicherheitsmechanismen, Unmut unter den Nutzern, was dazu führen könnte, dass diese die U-Bahn nicht mehr nutzen. Diese negativen Folgen überwiegen beim subjektiven Empfinden des Nutzers.

Bei einem Kreditkarten-System sieht das ganz anders aus. Hier würde der entstehende Schaden durch illegales Benutzen des Bankkontos sehr groß werden. Die Motivation eines potentiellen Angreifers ist ebenfalls sehr hoch, da das Überwinden des Sicherheitsmechanismuses vergleichbar ist mit barem Geld. Im Gegensatz dazu ist die Dauer eher von untergeordneter Rolle, da der Zahlvorgang ohnedies länger dauert.

Ein gutes Sicherheitskonzept setzt voraus, dass nicht nur die richtigen Sicherheitsmechanismen eingesetzt werden, sondern auch, dass diese richtig implementiert sind. Ein Fehler hierbei könnte das gesamte Sicherheitskonzept gefährden.

Für eine sichere Implementierung ist aber nicht nur ein mathematisch korrekt implementierter Sicherheitsmechanismus erforderlich, sondern auch eine Resistenz gegen Seitenkanalattacken. Seitenkanalattacken sind Attacken, bei denen spezielle Eigenschaften der Hardware und Software, wie zum Beispiel das Laufzeitverhalten, die Leistungsaufnahme

oder die elektromagnetische Abstrahlung, genutzt werden, um eine Attacke auf das System erfolgreich durchzuführen. Zu den bekanntesten Seitenkanalattacken gehören die Timing Attacke, Simple Power Analysis (SPA) und Differential Power Analysis (DPA).

Die Timing Attacke beruht auf der einfachen Annahme, dass für das Ausführen eines Codesegments unterschiedlich viel Zeit benötigt wird. Diese Zeitunterschiede sind an den meisten Stellen des Codes unproblematisch und von geringer Bedeutung. Betrifft das unterschiedliche Zeitverhalten eine sicherheitstechnische kritische Codestelle, so besteht die Gefahr, dass dieser Zeitunterschied für einen Angriff genutzt wird.

Für die Timing Attacke ist natürlich etwas mehr notwendig, als die Messung der Zeitunterschiede. Es müssen auch Annahmen über das zu Grunde liegende System gemacht werden, sowie den verwendeten Algorithmus.

SPA und DPA verfolgen einen ähnlichen Ansatz im Bereich der Stromaufnahme. Weiters lassen Seitenkanalattacken auch hervorragend kombinieren. Dadurch wird die Seitenkanalattacke effizienter und das System kann schneller gebrochen werden. In dieser Masterarbeit wird ausschließlich eine Möglichkeit zum Erkennen von Timing Probleme vorgestellt.

1.2 Zielsetzung

Diese Masterarbeit umfasst den Aufbau eines Hardware/Software Codesign Flows, der es ermöglicht Timing Probleme während der Entwicklung zu erkennen. Dafür muss eine Möglichkeit zur Zeitmessung implementiert werden.

In der Designphase des Systems wird vorgegeben, welche Funktionen oder Bereiche als sicherheitskritisch einzustufen sind. Bei der Implementierung des Systems werden diese Codesegmente im Programmcode markiert. Im Anschluss wird dann die Funktionalität des gesamten Systems oder eines Teilsystems überprüft. Dafür werden die Anwendungsfälle auf dem System ausgeführt und falls ein sicherheitskritisches Codesegment des Programmcodes durchlaufen wird, wird die Ausführzeit gemessen. Die Messung der Ausführzeit erfolgt allerdings nur in der Implementierungsphase und Testphase des Systems. Im fertigen System wird die Messung der Ausführzeit deaktiviert.

Die Auswertungssoftware berechnet, ob die gemessenen Zeiten ein mögliches Sicherheitsproblem darstellen oder nicht. Für den Fall, dass ein Timing Problem festgestellt wird, gibt die Software im Timing Protokoll eine Alarmmeldung ab. Auf diesem Weg erfährt der Entwickler, ob die von ihm implementierte Funktion ein Timing Problem besitzt oder nicht. Für den Fall eines Timing Problems werden Vorschläge diskutiert, um den Code Timing unabhängig zu machen.

1.3 Gliederung

Kapitel 2 beschäftigt sich mit Smartcards, ihren Sicherheitsmechanismen und Seitenkanalattacken. Es werden dabei unterschiedliche Seitenkanalattacken vorgestellt. Weiters werden auch Maßnahmen besprochen, welche bei der Entwickelung getroffen werden können, um diese später zu verhindern. Die Timing Attacke wird dabei genauer behandelt, da es in dieser Arbeit um einen Hardware/Software Codesign Flow geht, der Timing Probleme schon in der Entwicklungsphase erkennt und somit Timing Attacken verhindern soll.

Im Kapitel 3 wird der Entwurf des Hardware/Software Codesign Flows gegen Timing Attacken vorgestellt. Es wird dabei beschrieben, wie man Timing Probleme erkennt. Kapitel 4 beschäftigt sich mit der Implementation des Hardware/Software Codesign Flows. Ein wichtiger Teil des Kapitels beschäftigt sich mit der Implementierung der Zeiterfassung.

Im Kapitel 5 wird anhand eines Praxisbeispiels die Funktionsweise des Hardware/Software Codesign Flows beschrieben. Dafür wird eine kurze Smartcard Anwendung implementiert und untersucht. Es wird dabei gezeigt wie sich unterschiedliche Implementierungen verhalten und wie man dafür sorgen kann, dass ein Programm ein konstantes Laufzeitverhalten erreicht. Kapitel 6 beinhaltet eine Zusammenfassung. Es wird dabei auch auf offene Fragen eingegangen und ein Ausblick auf weiterführende Arbeiten gegeben.

Kapitel 2

Grundlagen

2.1 Seitenkanalattacken

Seit geraumer Zeit ist Sicherheit bei Computern und in Kommunikationstechnologien von großer Bedeutung und steht somit immer wieder im Zentrum von zahlreichen Forschungsarbeiten. Dabei stellen kryptografische Algorithmen eine Art Grundwerkzeug zur Verfügung, mit denen man Sicherheitsaspekte in das Design aufnehmen kann. Diese Grundwerkzeuge werden dann in Sicherheitsmechanismen integriert. Ein Beispiel dafür ist ein Netzwerksicherheitsprotokoll wie TLS (Transport Layer Security), das für eine sichere Kommunikation und Authentifizierung zwischen den Teilnehmern sorgt.

In der Praxis wird bei einem Sicherheitsmechanismus nur der zu verwendende Algorithmus spezifiziert. Es wird dabei aber nicht darauf eingegangen, wie der zu verwendenden Algorithmus zu implementieren ist.

Das kann dazu führen, dass die Implementierung des Verschlüsselungsalgorithmus, auf unterschiedlichste Weise erfolgt. Es wäre zum Beispiel möglich, dass es sich einmal um eine Softwarelösung handelt, die auf einem handelsüblichen Prozessor abgearbeitet wird und das andere Mal, wird eine zusätzliche Hardwareeinheit integriert, die die Daten weiterverarbeitet. Diese Unterschiede zwischen den Sicherheitsmechanismen und ihrer Implementierung ermöglichen eine theoretische Analyse des Designs und des Sicherheitsprotokolls.

Typischer Weise wird bei der Umsetzung eines Sicherheitskonzeptes davon ausgegangen, dass es sich um eine ideale Blackbox handelt. Das heißt, man weiß von außen nicht, was in ihr passiert und es gibt auch keine Möglichkeit, ihre internen Zustände zu beobachten oder diese gar zu verändern. Durch diese Annahmen entspricht der Sicherheitsgrad der eingesetzten Sicherheitsmechanismen im Großen und Ganzen dem des mathematischen Modells der Sicherheitsmechanismen.

In der Praxis existiert diese ideale Blackbox aber nicht. Es wäre auch fahrlässig anzunehmen, dass es keine andere Möglichkeit gibt ein Sicherheitssystem zu attackieren, als die Brute-Force-Variante. Ein Großteil der bisherigen Angriffe auf Sicherheitssysteme, erfolgte auf Grund von Schwächen in der Entwicklung und in der Implementierung der Mechanismen in Verbindung mit ihren kryptografischen Algorithmen. Sie erlauben es dem Angreifer eine Überbrückung des Sicherheitssystems oder eine Reduktion der theoretischen Komplexität. Dadurch werden Angriffe möglich, die davor mehrere Jahre gedauert hätten.

Nach der Veröffentlichung eines kryptografischen Algorithmus, wird dieser während

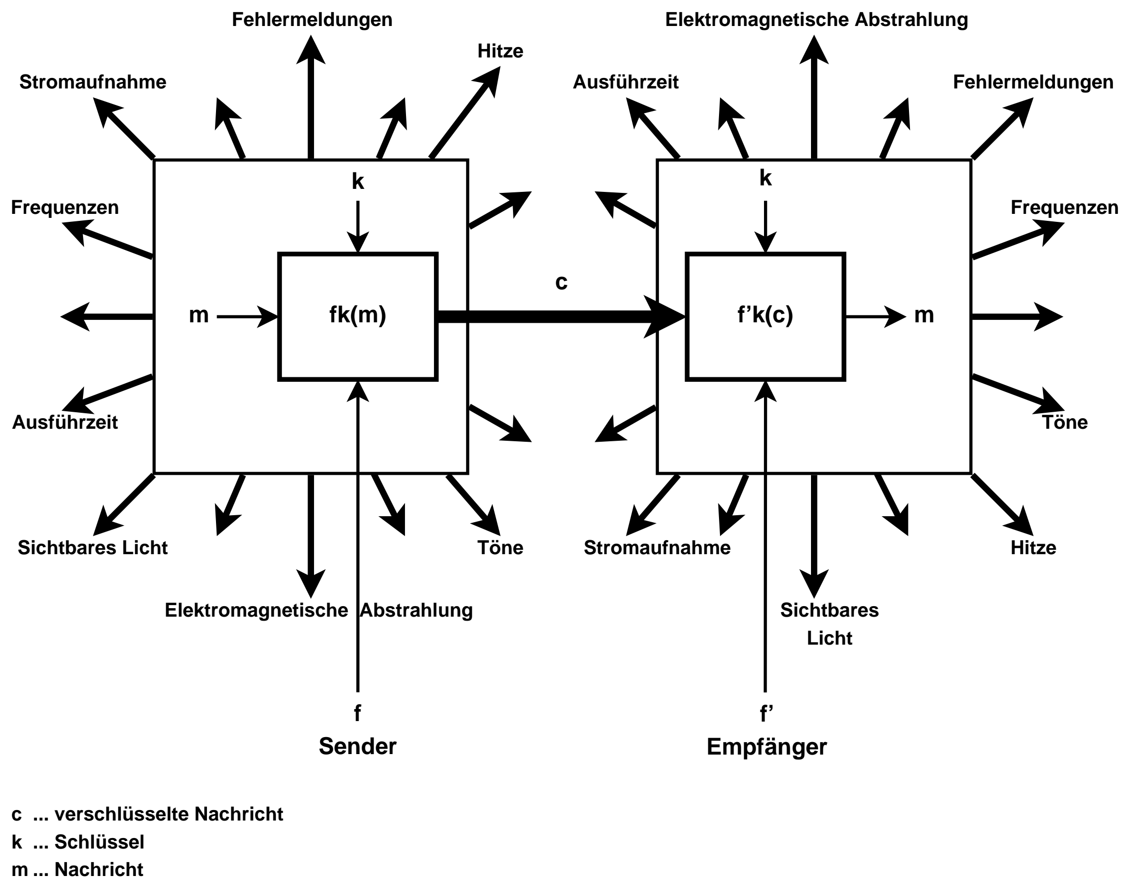

Abbildung 2.1: Kryptografisches Modell mit Seitenkanälen

seiner gesamten Lebensdauer immer wieder von Sicherheitsexperten analysiert und attackiert. Dabei versucht man den kryptografischen Algorithmus auf mathematischem Weg zu brechen. Bis dies gelingt, gilt er als "Sicher".

Geht man nun davon aus, dass ein kryptografischer Algorithmus sicher ist, bedeutet das nicht, dass auch die Implementierung des Algorithmus sicher ist. Gelingt es einen Zusammenhang zwischen extern messbaren Daten und den internen Zuständen, die auf den Schlüssel schießen lassen, zu finden, so ist es möglich über diese Daten, den Schlüssel zu errechnen. Dies ist die Grundidee der Seitenkanalattacken. Solche Seitenkanäle können zum Beispiel die Ausführungszeit, die Leistungsaufnahme, elektromagnetische Strahlung und ähnliches sein. In Abbildung 2.1 wird das kryptografisches Modell mit diesen Seitenkanälen dargestellt. Es beschreibt dabei den Vorgang des Verschlüsseln und Entschlüsseln einer Nachricht auf einem System. Dabei sieht man den Informationsverlust durch die einzelnen Seitenkanäle.

Die erste offiziell bekannte Seitenkanalattacke wurde 1965 vom britischen Geheimdienst

MI5 durchgeführt. Dabei wurde versucht verschlüsselte Nachrichten der ägyptischen Botschaft in London zu entschlüsseln. Diese verwendeten zum Verschlüsseln ihrer Nachrichten eine Rotor Chiffriermaschine [ZF05].

Da die ihnen zur Verfügung stehende Rechenleistung nicht ausreichte, um die Nachrichten ohne externe Hilfe zu entschlüsseln, musste man sich alternative Ansätze zum Überwinden der Verschlüsselung einfallen lassen.

Dabei entstand die Idee, ein Mikrophon neben der Rotor Chiffriermaschine zu platzieren, um auf diese Weise zu hören, welche signifikanten Geräusche bei der Verschlüsselung entstehen. Von diesen Geräuschen versprach man sich Aufschluss über die Funktionsweise und die verschlüsselten Daten. Da die Chiffriermaschine jeden Morgen auf ihren Anfangszustand zurückgesetzt wurde, war es möglich, die Position der zwei Kernrotoren zu bestimmen.

Dadurch wurde die Komplexität der Verschlüsselung so stark reduziert, dass sie mit der verfügbaren Rechenleistung gebrochen werden konnte. Auf diese Weise konnte der MI5 die Kommunikation der ägyptischen Botschaft über Jahre hinweg belauschen [ZF05].

Richtig bekannt wurden Seitenkanalattacken aber erst 1996 durch Paul Kocher. Dieser schrieb eine Seminararbeit über "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". Dabei wurde eine Timingattacke durchgeführt die zeigte, dass es möglich ist, über das Zeitverhalten eines Geräts auf dessen Schlüssel zu schließen. Nach bekannt werden dieser Attacke wurden natürlich sofort Gegenmaßnahmen gegen diese Art von Angriff ergriffen.

Die Idee ist aber der Grundstein für eine große Anzahl von anderen Attacken, die auf einem ähnlichen Ansatz beruhen. Diese nutzen andere Seitenkanäle (wie zum Beispiel die Leistungsaufnahme, elektromagnetische Abstrahlungen, Temperaturabstrahlung) oder kombinieren die Informationen aus mehreren Seitenkanälen, um somit noch effizientere Angriffe durchzuführen.

2.2 Smartcard

Smartcards werden heutzutage in vielen Bereichen des Lebens eingesetzt. Sie dienen meist zur Identifizierung oder zur Speicherung von Daten. Das bekannteste Beispiel für eine Smartcard ist, die "Bankomatkarte".

Eine Smartcard besteht aus einem Mikrochip und einem oder mehreren Kommunikations- und Stromversorgungsinterfaces. Alle Komponenten sind in eine 0,76 Millimeter dicke Plastikkarte eingegossen. Der Mikrochip, der in Smartcards eingesetzt wird, bestimmt ihren Anwendungsbereich. Es gibt zwei grundsätzlich unterschiedlich aufgebaute Smartcardtypen: Synchrone Smartcards (Abbildung 2.2) und Asynchrone Smartcards (Abbildung 2.3). Ihr Aufbau könnte nicht unterschiedlicher sein.

Synchrone Smartcards sind im Grunde genommen nur eine Logik zum Ansprechen ihres Speichers. Dieser Speicher kann gelesen und geschrieben werden. Man findet sie häufig in Telefonwertkarten und Krankenkassenkarten. Man verwendet diesen Smartcardtyp nur für Vorgänge mit geringer Komplexität. Dies ist auch der Grund dafür, warum sie heute kaum noch eingesetzt werden.

Bei Asynchronen Smartcards hingegen wird ein Mikroprozessor eingesetzt. Dieser kann vergleichsweise komplexe Aufgaben bewältigen. Auf den Speicher kann man nur über

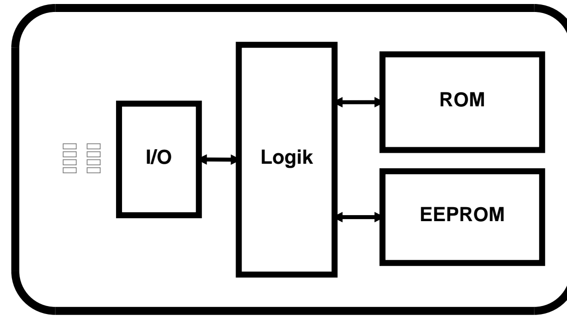

Abbildung 2.2: Synchrone Smartcard oder Speicherchipkarte

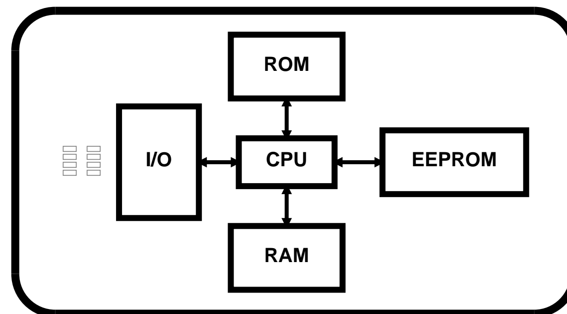

Abbildung 2.3: Asynchrone Smartcard oder Prozessorchipkarte

den Mikroprozessor zugreifen. Es ist dadurch möglich, die Daten die auf der Karte gespeichert werden zu verschlüsseln, was die Sicherheit erhöht. Ihr Anwendungsgebiet ist größer. Typische Anwendungen für Asynchrone Smartcards sind zum Beispiel Kreditkarten, SIM-Karten oder ID-Karten (Erkennungskarten).

Die Sicherheit ist auf Grund ihres Anwendungsgebiets als Bankomatkarte oder als ID-Karte ein sehr wichtiges Thema. Der Grund dafür liegt auf der Hand, keine Bank würde eine Bankomatkarte anbieten, wenn der Kunde den Kontostand verändern könnte oder es möglich wäre, den PIN der Karte auszulesen oder gar zu verändern. Dasselbe gilt auch für ID-Karten, bei denen sich der Nutzer selbst Rechte für das Betreten von Räumen oder Anlagenteilen freischalten könnte. Aus diesem Grund sind physikalische Sicherheitsmechanismen in Smartcards integriert worden, die Manipulationen verhindern sollen. Beschreibung von Smartcards.

2.2.1 Hardware Sicherheitsmechanismen gegen Attacken auf Smartcards

Es gibt unterschiedliche Typen von Attacken auf Smartcards. Im Wesentlichen unterscheidet man zwischen invasiv oder nicht invasiven Attacken. Bei einer invasiven Attacke wird die Smartcard physikalisch angegriffen. Das bedeutet, sie wird geöffnet, mit UV-Lichtblitzen beschossen oder auf irgendeine Art extern beeinflusst.

Das Öffnen der Smartcard kann zum Beispiel durch wegätzen oder abfräsen der Kunststoffschicht erfolgen. Auf diese Weise erhält man Zugang zum inneren Aufbau der Smartcard. Es ist dadurch möglich, dass man zum Beispiel die Busverbindung abhört, Register ausliest und sogar Teile des Rechenwerks beim Arbeiten beobachtet.

Als Schutz gegen das Entfernen der Plastikschicht wird eine Kapazitätsmessung oder ein optischer Sensor verwendet. Wird nun eine Veränderung der Kapazität oder der Helligkeit gemessen, so kann die Software beziehungsweise das Betriebsystem der Smartcard davon ausgehen, dass die Smartcard geöffnet wurde. Das bedeutet das System wird angegriffen und es werden keine weitern Operationen ausgeführt.

Durch Blitzlicht oder Laserattacken auf bestimmte Bereiche des Speichers können einzelne Bits oder Bytes verändert werden. Diese kann zum Beispiel zum Zurücksetzten eines Sicherheitsbits verwendet werden. Die einzige Voraussetzung, die dafür nötig ist, ist dass die Speicherzelle einen genügend großen Abstand zu den restlichen Speicherzellen aufweist. Gegen die Blitzlicht und Laserattacken helfen ebenfalls die vorhin beschriebenen optischen Sensoren.

Weiters reagieren Smartcards empfindlich auf eine außerhalb des Normalbetriebs liegende Betriebsspannung oder Temperatur. Diese ist besonders beim Beschreiben des EEPROMs kritisch, da eine abgesenkte Betriebsspannung dazu führen kann, dass der Wert im EEPROM fehlerhaft ist.

Als Gegenmaßnahme hierfür wird ein Spannungssensor in die Smartcard integriert, der bei einer nicht der Norm entsprechenden Betriebsspannung ein Sicherheitsbit setzt oder gar einen Reset auslöst. Ein ähnlicher Ansatz existiert auch für eine Betriebstemperatur, die außerhalb der üblichen Parameter liegt.

Eine weitere Angriffsmöglichkeit stellt die Änderung des Systemtaktes da. Diese Attacke kann zum schrittweisen Abarbeiten der Smartcard Anwendung führen. Als Gegenmaßnahme wird der Takt der Smartcards hardwaremäßig überprüft. Dies erfolgt zum Beispiel über einen Frequenz-Spannungskonverter oder RC Elemente.

Nicht invasive Attacken setzen voraus, dass keinerlei Manipulationen an der Smartcard erfolgen. Angriffe auf Smartcards beschränken sich allerdings nicht nur auf die mathematische Analyse der verwendenden Algorithmen und Protokolle, sondern beinhalten auch die in Kapitel 2.1 vorgestellten Seitenkanalattacken.

Bei der Simple Power Analysis Attacke wird der Energieverbrauch des Mikroprozessors bei kritischen Operationen (zum Beispiel beim Ver- oder Entschlüsseln) aufgezeichnet. Der Grund dafür ist, dass nicht jeder Befehl gleich viel Energie benötigt. Der Energiebedarf hängt zusätzlich auch noch vom Schlüssel ab. Über die Variation des Energieverbrauchs lässt sich so auf den Schlüssel schließen.

Es gibt unterschiedliche Ansätze wie Smartcards vor diesen Angriffen geschützt werden. Einer davon ist das Verstecken der Seitenkanalinformation im Rauschen. Dafür muss die für das Ausführen des Befehls nötige Energie nur so weit schwanken, das man nicht eindeutig sagen kann, um welchen Befehl es sich handelt. Dieser Ansatz erschwert zwar den Angriff, allerdings kann er ihn nicht verhindern, da sich durch eine größere Anzahl von Messungen und eine statistische Analyse die ursprüngliche Ausführzeit bestimmen lässt.

Eine weitere Gegenmaßnahme ist, dass kritische Teile des Mikroprozessors oder des Kryptocoprozessors nach dem Dual Rail Prinzip aufgebaut sind. Das bedeutet, dass alle Signalleitungen zweimal ausgeführt sind und zwar in einer komplementären Logik. Somit macht es bei der Berechnung keinen Unterschied, welche Daten der Algorithmus bearbeitet, denn es wird immer die Berechnung für 0 und 1 durchgeführt. Dadurch ist sichergestellt, dass immer derselbe Energiebedarf bei einem Befehl benötigt wird.

Eine andere Gegenmaßnahme ist, dass kritische Daten nicht direkt verwendet werden. Dieser Ansatz kann in Software als auch in Hardware implementiert werden und

wird Masking genannt [AGM+09] [PGH+04]. Dafür werden die Originaldaten d mit einer zufällig bestimmten Maske m_d XOR-Verknüpft (siehe Formel 2.1), das Ergebnis d_m für die weiterführenden Berechnungen verwendet.

$$d_m = d \oplus m_d \tag{2.1}$$

Da d und m_d paarweise unabhängig von d_m sind, ist die einzige Information die öffentlich bekannt wird, die von d und m_d in Kombination. Natürlich muss der Algorithmus für die weiteren Berechnungen angepasst werden. Dies erschwert zwar das Bestimmen des Schlüssels, es ist aber kein hundertprozentiger Schutz gegen Power Analysis Attacken [Man10]. Eine weitere Attacke ähnelt der Power Analysis Attacke und beruht auf der Analyse der elektromagnetischen Abstrahlung, kurz EMA [Jan08]. Dabei wird das elektromagnetische Feld der Smartcard aufgezeichnet. Im Weiteren erfolgt eine Auswertung der aufgezeichneten Daten und kann so auf den verwendeten Schlüssel zurückschießen. Wie schon bei der Power Analysis Attacke ist auch hierbei eine simple und eine differentielle Auswertung der elektromagnetischen Abstrahlung möglich.

Die Gegenmaßnahmen bei der SEMA (Simple Electromagnetic Analysis) und DEMA (Differential Electromagnetic Analysis) Attacke sind um ein vielfaches komplexer als die bei SPA und DPA Attacken [Jan08]. Der Grund dafür ist, dass die SPA (Simple Power Analysis) und DPA (Differential Power Analysis) Attacke nur die Stromaufnahme des gesamten Chips aufnimmt. Im Vergleich dazu ist es mit der SEMA und DEMA Attacke möglich gezielte Bereiche des Chips zu vermessen. Durch gezieltes Chipdesign lässt sich die elektromagnetische Abstrahlung reduzieren, das vermindert die Gefahr bei einer EMA Attacke.

Eine andere Möglichkeit ist, dass man durch Randomisierung einen möglichst großen Anteil an Zufallsinformation in die Messung einbringt, die die Nutzinformation überdeckt. Dadurch wird das Rückgewinnen der Nutzinformation schwieriger bis hin zu unmöglich. Dieser Ansatz stellt eine Software Lösung dar, die auch gegen SPA und DPA einsetzbar ist. Es ist dabei aber auch wichtig, dass die ausgeführte Anwendung keine Datenabhängigkeiten aufweisen durch die Daten unterschieden werden können. Nur im Zusammenhang mit Hartwaregegenmaßnahmen entfalten die Softwaremaßnahmen ihre volle Wirkung [Jan08].

Eine weitere Attacke ist die Timing Attacke. Diese Attacke betrifft meist das zeitliche Verhalten von Funktionen und Funktionseinheiten. Da es in dieser Arbeit um einen Hardware/Software Codesign Flow geht der vor Timing Attacken schützen soll, wird dieser Attacke das Unterkapitel 2.3 gewidmed.

2.3 Die Timing Attacke

Wie bereits im Kapitel 2.1 erwähnt, wird bei Seitenkanalattacken nicht der Algorithmus direkt angegriffen, sondern der Angriff erfolgt vielmehr auf die Implementierung des Algorithmus. Wenn man nun davon ausgeht, dass der Algorithmus sicher ist, bleibt nur noch die Umsetzung, die eine Schwachstelle aufweisen kann, welche ausgenutzt werden kann um das Sicherheitsdesign zu überwinden.

Bei Timing Attacken wird das Zeitverhalten der vermeintlich sicheren Funktion untersucht. Stellt man dabei eine unterschiedliche Laufzeit fest und ist man zudem noch in der

Lage diese zeitlichen Unterschiede in Verbindung mit dem Schlüssel oder den Nutzdaten zu bringen, so ist man in der Lange den Sicherheitsmechanismus zu überwinden.

Die älteste bekannte Timing Attacke ist die auf den Login des Betriebsystem TENEX [Lam83]. Hierbei wurden Schwächen in der Umsetzung des Passwort Konzepts gemacht. Diese ermöglichten es jedem Nutzer des Systems das Passwort eines beliebigen Nutzers in Erfahrung zu bringen.

TENEX ist ein Betriebssystem, das für DEC-10 Systeme in den 1970 Jahren entwickelt wurde. Es beherrscht unter anderem Paged Memory und Passwortchecks für Files. Die Passwortchecks wurden nicht über Systemcall implementiert.

Beim Passwortcheck wurde Byteweise vergleichen, ob das eingegebene Passwort mit dem gespeicherten Passwort übereinstimmt. Trat keine Übereinstimmung auf, so wurde die Passwortüberprüfung sofort abgebrochen und der Zugriff verweigert.

Ein weiteres Problem dieses Konzepts war, dass nur benötigte Daten geladen wurden. Dies führte zu dem Nachteil, dass nur bereits verglichene Bytes des Passworts im Speicher waren. Wenn das erste unbekannte Byte erfolgreich erraten wurde, wurde das darauffolgende Byte in die aktive Page geschrieben. Dieser Vorgang dauerte signifikant länger als ein Vergleich von zwei Variablen die sich bereits auf der aktiven Page befinden. Daher war jeder Nutzer des Systems in der Lage, Passwörter in linearer Zeit zu erraten, in dem man einfach alle Möglichkeiten eines einzelnen des Bytes ausprobierte. Erriet man ein Byte, so merkte man den Nachladevorgang und wusste, dass das soeben eingetippte Byte korrekt war.

Eine weitaus bekanntere Timing Attacke ist die von Paul Kocher. Der genaue Aufbau der Timing Attacke wird in der Arbeit [Koc96] beschrieben und in dem folgenden Abschnitt kurz zusammengefasst.

Es wird dabei beschrieben, dass die meisten kryptografischen Systeme leicht in ihrer Laufzeit variieren. Die Variation der Laufzeit ist dabei vom Schlüssel, sowie vom verwendeten Klartext abhängig.

Der Grund für diese Laufzeitunterschiede kommt von Performanceoptimierungen, die nicht benötigte Abschnitte des Programms überspringen, Verzeigungen im die nicht gleich viel Zeit benötigen, RAM Cache Hits und Misses, Prozessoroperationen die unterschiedlich lange Ausführzeiten haben, wie zum Beispiel Multiplikationen oder Divisionen und noch einige mehr.

Es wird an Hand des Beispiels des diskreten Exponenzierers gezeigt, wie sich Zeitunterschiede durch Verzweigungen und Performanceoptimierungen ergeben. Dafür teilt die Arbeit [Koc96] die Attacke in zwei Teile auf. Der erste Teil beschäftigt sich mit der Kryptoanalyse eines diskreten Exponenzierers und der zweite Teil beschäftigt sich mit der Umsetzung der Timing Attacke.

Kryptoanalyse eines diskreten Exponenzierers

Für die Attacke setzt Kocher voraus, dass bei Formel 2.2 die beiden Variablen n und y bekannt sind. Dies begründet er dadurch, dass n allgemein bekannt ist und y könnte man durch einen einfachen Lauschangriff in Erfahrung bringen. Das Ziel des Angriffes ist es den geheimen Schlüssel x in Erfahrung zu bringen. Um dies zu erreichen wird vorausgesetzt, dass Formel 2.2 für unterschiedliche Werte y berechnet wird und man die Ausführzeit misst.

$$R = y^x \mod n \tag{2.2}$$

Weiters wird davon ausgegangen, dass der Angreifer über das Design des angegriffenen Systems Bescheid weiß. Diese Information kann aber laut Kocher aus den Timing Informationen abgeleitet werden. Eine weitere Voraussetzung für die Timing Attacke ist, dass die Implementation des Algorithmus keine konstante Ausführzeit besitzt. Die Implementierung des Algorithmus sieht in der Arbeit [Koc96] wie folgt aus.

```
{RSA}
Let s(0) = 1
For k = 0 upto w-1
   If  (bit k of x) = 1 then
      Let R(k) = (s(k) * y) * mod n
   Else
      Let R(k) =s(k)
   Let s(k+1) = R(k)*R(k) * mod n
EndFor
Retrun ( R(w-1) )
```

Die Idee hinter der Attacke ist, dass der Angreifer Bit für Bit des Schlüssels errät. Die Attacke erlaubt einem Angreifer, der eine gewisse Anzahl von Bits (0 bis $b-1$) des Exponenten kennt, das darauffolgende Bit b zu bestimmen. Man startet mit b ist gleich 0 und wiederholt die Attacke bis der gesamte Exponent bekannt ist.

Wenn die ersten b Bits bekannt sind, berechnet der Angreifer die ersten b Iterationen der For-Schleife. Daraus ergibt sich der Wert s_b. Der nächste Iterationsschritt verlangt den Wert des ersten unbekannten Bits. Dieser kann entweder 0 oder 1 sein, was zu einer Verzweigung in der For-Schleife führt. Ist das Bit b gleich 1, so wird $R_b = (s_b * y) \mod n$ berechnet. Ist das Bit b dagegen 0, so wird $R_b = s_b$ beziehungsweise $R_b = 0$ gesetzt.

Wie man sieht, ist der auszuführende Code bei $b = 1$ wesentlich komplexer als bei $b = 0$. Dies führt jetzt zu der Annahme, dass $b = 1$ länger benötigen wird. Auf Grund des Zeitunterschieds beim Berechnen des Iterationsschrittes wird dann bestimmt, ob das gesuchte Bit b gleich 1 oder 0 ist.

Timing Attacke auf den diskreten Exponenzierer

In diesem Abschnitt beschreibt die Arbeit [Koc96] die Timing Attacke, als eine Art von Signal Erkennungsproblem. Dabei ist die Variation der Laufzeit in Abhängigkeit des jeweiligen Bits des Exponenten das Signal. Die Messunsicherheit und die Variation der Laufzeit beim mehrfachen Messen desselben Bits stellen das Rauschen dar.

Um das Signal vom Rauschen unterscheiden zu können, müssen mehrere Messungen j gemacht werden mit dem gleichen Exponent x. Eine Messung besteht zum Einen aus der Nachricht y_n und zum Anderen aus der dafür benötigten Zeit T_n. Die gemessene Zeit $T = e + \sum_{i=0}^{w-1} t_i$ besteht aus der Zeit t_i, die für jede Iteration (jedes Bit i) benötigt wird und den Fehler e. Der Angreifer versucht nun das Exponentenbit x_b für jede Nachricht y zu schätzen, um so die benötigte Zeit $T = \sum_{i=0}^{b-1} t_i$ zu finden. Ist diese Schätzung korrekt, dann wird versucht die Zeit für die noch unbekannten Bits zu finden $e + \sum_{i=0}^{w-1} t_i - \sum_{i=0}^{b-1} t_i = e + \sum_{i=b}^{w-1} t_i$.

Es wird davon ausgegangen, dass der Fehler e und die Zeiten, die für die diskreten Multiplikationen benötigt werden, von einander unabhängig sind. Die Varianz von $e + \sum_{i=b}^{w-1} t_i$ über alle Nachrichten ist $Var(e) + (w - b) * Var(t)$. Geht man nun davon aus, dass nur die ersten c Bits der b Bit des Exponenten richtig sind, so ergibt sich eine Varianz von $Var(e) + (w - b + 2 * c) * Var(t)$.

Wenn die vorangegangenen Iterationen korrekt nachgeahmt wurden, stellt man eine Verringerung der Varianz $Var(t)$ fest, wird hingegen eine Erhöhung der Varianz festgestellt, so kann davon ausgegangen werden, dass das zuletzt bestimmte Bit nicht stimmt.

Anhand dieses Konzeptes werden weitere Attacken beschrieben, die sich gegen die Montgomery Multiplikation, das Chinese Remainder Theorem und den Digital Signatur Standard richten.

Die Timing Attacke von Kocher war der Anfang einer Serie von Side Channel Attacken und natürlich auch von weiteren Timing Attacken. Die Arbeit [Hey98] beschreibt eine Timing Attacke auf einen RC5 Symmetric Block Cipher. Dieser Algorithmus wurde für eine effiziente Softwareimplementierung entwickelt. Er benötigt dafür nur 3 Grundoperationen: Exklusive Oder, die Addition und eine datenabhängige Rotation.

Die letzte Grundoperation ist der Grund für die in der Arbeit beschriebene Attacke. Sie setzen dabei aber natürlich voraus, dass die Rotation nicht in konstanter Zeit erfolgt. Sie nehmen dabei an, dass es sich um eine einfache Hardwareimplementierung oder eine Softwareimplementierung auf einem zum Beispiel 8 Bit Mikroprozessor handelt.

Bei der Hardwareausführung könnte dies ein einfacher Shifter sein, der die Daten jeweils nur um ein Bit verschieben kann. Durch wiederholtes Ausführen dieser Shift-Operation ist es jedoch möglich, jeden beliebige Rotationsweite zu erreichen. Diese Implementierung würde zu unterschiedlichen Laufzeiten führen.

Das Problem daran ist, dass die Rotationsweite von den Daten abhängt. Dadurch verrät die Laufzeit der Rotation Informationen über die bearbeiteten Daten. Durch mehrfaches Messen der Laufzeiten lässt sich der gewünschte Schlüssel bestimmen. Dies funktioniert nur, wenn genügend genaue Zeitmessungen durchgeführt werden.

In der Arbeit [BB03] wird eine Timing Attacke beschrieben, die zeigt, dass auch Netzwerkserver von diesen Attacken betroffen sind. Der Grund dafür war, dass viele Kryptobibliotheken ohne Rücksicht auf Timing Attacken implementiert wurden, unter anderem auch die Implementierung von OpenSSL. Zur Demonstration nutzten sie einen Apache Webserver der das SSL-Modul nutzt. Zwischen dem Angreifer und dem angegriffenen Webserver befanden sich 3 Router und mehrere Switches. Das Ziel der Attacke war der private Schlüssel, der über einen Angriff auf die RSA-Endschlüsselung bestimmt wurde. Dies gelang durch die sehr stark optimierte Implementierung der RSA-Entschlüsselung der OpenSSL Bibliothek. Sie nutzte das Chinese Remainder, Sliding Windows, die Montgomery Multiplikation und den Karatsuba's Algorithmus [BB03].

Diese Kombination ermöglichte es zwei Datenabhängigkeiten in der OpenSSL Implementation zu finden. Die erste war die Anzahl der Montgomery Reduktionen und die zweite war die Zeitdifferenz durch die Wahl des Multiplikationsalgorithmuses (Karatsuba's vs. Normal). Sie variierten bei ihren Analysen mit mehreren Parametern, wie zum Beispiel der verwendeten Hardware und deren Optimierungsgrad, Interprozesses vs Local Network oder gar unterschiedlichen Schlüsseln. Trotzdem zeigte sich die Attacke immer als erfolgreich, allerdings mit unterschiedlichem Aufwand. Das heißt, die Anzahl der benötigten Messungen variierte.

Ähnlich wie bei dieser Attacke wurde auch in der Arbeit [CRW09] eine Timing Attacke auf einem Webserver vorgestellt. Der Unterschied zur vorangegangen Arbeit [BB03] ist, dass die Timing Attacke auf einen Webserver im Internet erfolgt. Es wurde ebenfalls ein Apache Webserver verwendet, der das SSL-Modul nutzt. Hierbei wurde allerdings nicht der Frage nachgegangen, ob eine Timing Attacke möglich ist, sondern wie genau eine Messung sein muss, um die Attacke noch ausführen zu können. Dabei wurde ein Filter verwendet, der den Jitter entfernen, beziehungsweise reduzieren sollte. Sie kamen zum Schluss, dass die genauesten möglichen Zeitdifferenzmessungen im Internet $20\mu s$ waren, was aber für eine Timing Attacke ausreicht. In lokalen Netzwerken sind sogar 100ns möglich. Es wurde auch beobachtet dass die Round Trip Time und die Anzahl der Hops keinen signifikanten Einfluss auf den Jitter haben. Somit kamen sie zum Schluss, dass die Distanz zwischen Angreifer und Angegriffen keinen Schutz vor einer Timing Attacke sei.

Eine weitere Timing Attacke beschäftigt sich mit AES und wird in den Arbeiten [Ber04], [OST05] und [BEPM10] beschrieben. Die Arbeit [Ber04] beschäftigt sich mit einer Timing Attacke auf die OpenSSL Implementation des AES-Algorithmus, auf einer General Purpose CPU.

Es wird erklärt, dass es leicht möglich sei den AES-Code auf konstante Zeit zu bringen, dies hätte allerdings den Nachteil, dass der Code für den AES-Algorithmus nicht mehr performant wäre. Der Grund für das nicht konstante Zeitverhalten liegt im Aufbau von AES. Der AES-Algorithmus verschlüsselt immer 16 Byte Eingangsdaten auf einmal. Um diese Daten zu verschlüsseln werden mehrere Arrays benötigt, auf die die eigentlichen Verschlüsselungsoperationen (Transformationen) angewendet werden.

Ihre Überlegung war es, dass sie für das Ansprechen der einzelnen Werte im Array, unterschiedlich viel Zeit benötigt. Auf diese Weise verliert der Algorithmus Timing Informationen, die für eine Timing Attacke genutzt werden kann.

Abschließend wurden noch mehrere Gründe beschrieben, warum der Code unterschiedliche Laufzeiten haben kann. Einer der Gründe war, dass Daten die sich im Cachespeicher befinden, schneller verfügbar sind und somit eine Timing Attacke ermöglichen. Das Konzept der Attacke wird in der Arbeit [OST05] beschrieben.

Die Attacke wurde anhand eines einfachen Beispiels beschrieben. Dabei wird erklärt, dass der Cachespeicher ein kleiner und schneller Zwischenspeicher ist, der der CPU Zugriffe auf den Hauptspeicher erspart, sofern die gewünschten Daten in ihm vorhanden sind.

Wenn nun die CPU das erste Mal Daten anfordert, so wird zuerst im Cache nachgesehen, ob diese Daten vorhanden sind. Dies ist nicht der Fall (Miss), so findet ein Zugriff auf den Hauptspeicher statt und die Daten werden an die CPU übertragen. In dem Moment werden sie aber auch für den Cachespeicher verfügbar gemacht. Fordert die CPU nun noch einmal denselben Wert an, so werden die Daten direkt aus dem Cachespeicher geladen (Hit). Dies verursache eine unterschiedliche Laufzeit in Abhängigkeit der benötigten Daten.

Weiter gilt zu erwähnen, dass der Cachespeicher moderner CPUs nicht ein Byte im Cache hält, sondern er speichert Blöcke aus mehreren Bytes. Diese Blöcke heißen Lines. Aus wie vielen Bytes eine solche Line besteht, hängt von der CPU ab. Bei einem Pentium III zum Beispiel besteht eine solche Line aus 32 Bytes. Auf diese Weise füllt sich der Cache relativ schnell. Ist der Cachespeicher voll, so werden von ihm gehaltene Daten durch neue Daten überschrieben. Versucht man danach auf die alten Daten zuzugreifen, so muss man diese wieder aus dem Hauptspeicher laden.

Diese Eigenschaften des Cachespeichers helfen natürlich die AES-Berechnungen zu beschleunigen, dies führt allerdings auch zu den verräterischen Zeitunterschieden, die bei der Timing Attacke ausgenützt werden.

Die AES-Verschlüsselung benötigt sequentiell 160 Tabellenzugriffe $l_0, l_1, ..., l_{159}$, wobei es vorkommen kann, dass ein Zugriff l_i gleich einem Zugriff l_j ist, dann befinden sich die Daten im Cachespeicher und müssen nicht aus dem Hauptspeicher geladen werden. Ist dies nicht der Fall, also $l_i \neq l_j$, so kann es sein, dass die gewünschten Daten nicht im Cache vorhanden sind und somit aus dem Hauptspeicher geladen werden müssen.

Die Annahme war es, dass nach der ersten Runde der AES-Verschlüsselung, die individuellen Tabellenzugriffe (Lookups) in einer zufälligen Sequenz erfolgen. Diese Annahmen stellten den Cachespeicher sehr vereinfacht dar. Trotzdem wurden ihre Annahmen durch ihre Messergebnisse bestätigt und zeigten somit, dass Cache Hits und Misses einen Side Channel für die Timing Attacke öffneten.

Die Arbeit [BEPM10] beschreibt eine Timing Attacke auf AES. Dabei ist ebenfalls die Nutzung des Cachespeichers der Grund für die Laufzeitvariationen. Die Idee hinter der Attacke war es, das bestimmte Paare von Klartexten verwendet werden, bei denen fünf AES S-Boxen (eine davon in Runde 2 und vier davon in Runde 3) gleich oder paarweise verschieden waren, bei zwei aufeinanderfolgenden AES-Berechnungen. Wenn bei den Klartexten die fünf S-Boxen gleich sind, so nennt man es Wide Collision. Ihre Attacke bestand aus der Durchschnittslaufzeitmessung jeder zweiten AES-Berechnung eines Klartextpaares. Dabei war für sie die Anzahl der durchschnittlichen Kollisionen von c bei der S-Box Operation der zwei Paare interessant. Wenn eine Wide Collision für ein Klartextpaar auftritt, so ist der Wert von c fünffach höher, als wenn dies nicht der Fall wäre. Sie hofften darauf, dass sie deutlich mehr Wide Collisions finden würden als bei herkömmlichen Berechnungen.

Danach erstellten sie ein System, welches aus vier nichtlinearen Gleichungen bestand, womit Teile des Schlüssels bestimmt wurden. Die fehlenden Teile des Schlüssels wurden durch eine Brute-Force Attacke gefunden.

Möglich wurde diese Attacke durch Maximum-Distance-Separable-Code (MDS-Code), der bei AES verwendet wird, um die diffuse Matrix zu erstellen, welche für die Mix-Columns Operation benötigt wird. MDS-Code verwendet eine lineare Transformation, welche zu einer maximalen Anzahl von Verzweigungen führt. Dies ist im Fall von AES fünf.

Sie stellten dabei fest, dass die ausgezeichneten kryptografischen Eigenschaften von AES bei der Erstellung der diffusen Matrix, welche sie gegen die lineare und differentielle Kryptoanalysis schützt der Grund war, der ihre Cache Timing Attacke ermöglichte.

2.3.1 Gegenmaßnahmen bei Timing Attacken

Wie man sieht ist die Timing Attacke durchaus eine ernstzunehmende Seitenkanalattacke, der im Laufe der Zeit immer wieder Systeme zum Opfer fallen. Dabei spielt es keine Rolle, ob es sich um einen Standard PC oder ein Eingebettetes System handelt.

Sobald das Zeitverhalten von den Eingangsdaten in kritischen Bereichen abhängt, besteht die Gefahr, dass eine Timing Attacke erfolgreich ist. Als kritischer Bereich versteht man einen Bereich, der sicherheitsrelevante Operationen ausübt, zum Beispiel: ein Passwort überprüfen oder eine Signatur berechen.

Man sieht, wie einfach die Lösung des Problems der Timing Attacke sein könnte. Man muss nur das System dazu bringen, dass das Zeitverhalten unabhängig von den

Eingangsdaten ist. Dafür gibt es zwei unterschiedliche Möglichkeiten:

- Die erste Möglichkeit ist, dass das Zeitverhalten des Systems so stark variiert wird, dass bei einer Analyse der Eingangsdaten, des Systemaufbaus und der Laufzeiten, kein Zusammenhang zwischen diesen drei Größen gefunden wird.

- Die zweite Möglichkeit ist, dass die Laufzeit des Systems konstant gehalten wird. Das ist vergleichbar mit einem System, bei dem es egal ist, welche Eingangsdaten verwendet werden, denn es weist immer die Worst Case Laufzeit auf.

So trivial diese Möglichkeiten klingen, so problematisch können sie in der Umsetzung sein. Die erste Möglichkeit weist das Problem auf, dass sie die Timing Attacke nicht zur Gänze verhindern kann. Sie macht es dem Angreifer nur schwerer, eine erfolgreiche Timing Attacke auszuführen.

Es ist vergleichbar mit einer Messung der Zeit, durch ein ungenaues Messinstrument. Geht man nun von einer Normalverteilung des Rauschens bei der Messung aus, so lässt sich das Messergebnis verbessern. Dafür muss man die Messung unter den gleichen Bedingungen, mit dem gleichen Messinstrument mehrfach ausführen. Anschließend muss man nur den Mittelwert der Messergebnisse berechnen und erhält so ein genaueres Messergebnis.

Wenn eine sicherheitskritische Funktion eine variierende Laufzeit aufweisen soll, so muss diese ebenfalls durch eine zufällige Verzögerung entstehen. Tut sie das nicht, so kann man eine Funktion finden, die es einem ermöglicht, die zufällige Verzögerung zu berechnen. Wird hingegen eine echte zufällige Verzögerung verwendet, so hat man wiederum das Problem, dass durch mehrfaches Messen die wahre Laufzeit bestimmt werden kann [PTVF92, Lit05].

Daher wird in der Literatur meist, die zeitunabhängige Methode für Implementierungen empfohlen. Diese verspricht eine hundertprozentige Resistenz gegen Timing Attacken, da bei konstantem Zeitkonsum einer Funktion keine Zeitdifferenz in Abhängigkeit der Eingangswerte auftritt und somit existiert theoretisch zeitlich gesehen kein Seitenkanal.

Das Problem dieser Methode ist, wie bereits erwähnt, dass es immer eine Worst Case Laufzeit aufweist. Das bedeutet, dass jegliche Optimierungen wie man sie aus sicherheitsunkritischen Bereichen kennt, hier entfallen, es sei denn, sie deckt den gesamten kritischen Bereich ab, sodass sich durch die Optimierung keine unterschiedliche Laufzeit ergibt.

Im Grunde gibt es drei Möglichkeiten, um eine konstante Laufzeit in kritischen Bereichen zu erreichen:

- Zeitunabhängiges Programmieren

- Kryptohardware

- Das Design eines speziellen Algorithmus wird auf ein konstantes Laufzeitverhalten optimiert

Jede dieser Möglichkeiten hat ihre individuellen Vor- und Nachteile, wobei diese sich auf unterschiedliche Weise bemerkbar machen. Zum Beispiel lässt sich eine Kryptohardware nicht nur auf ein konstantes Laufzeitverhalten hin optimiert, sondern es ist dabei auch möglich die Leistungsaufnahme und Geschwindigkeit gegenüber einer Softwarelösung zu

verbessern. Dafür ist die Kryptohardware aber auch eine eigene Komponente im System, die zum einen Chipfläche benötigt und somit auch extra Kosten verursacht. Der Lösungsansatz über das Design des Algorithmus für eine konstante Laufzeit ist für Hardware und Software in gleicher Weise verfügbar. Es ist aber vergleichweise sehr aufwendig einen Algorithmus auf konstante Laufzeit zu bringen und zeitgleich auch dafür zu sorgen, dass die Ausführzeit nicht zu langsam ist. Ein weiterer wichtiger Punkt ist, dass alle Gegenmaßnahmen miteinander harmonieren müssen. Es darf dabei nicht vorkommen, dass zum Beispiel eine Kryptohardware verwendet wird, um Daten zu ver- und entschlüsseln, die Abfrage bei der Authentifizierung aber direkt nach dem ersten falschen Zeichen abbricht. Die folgenden drei Abschnitte beschreiben mögliche Gegenmaßnahmen gegen Timing Attacken anhand von Praxis bezogenen Beispielen.

Zeitunabhängiges Programmieren

Wie der Name schon sagt wird hierbei versucht, dass der Sourcecode im kritischen Bereich in konstanter Zeit ausgeführt wird. Es ist wichtig, dass im nicht kritischen Bereich, keine Operationen ausgeführt werden dürfen, welche durch ihr Zeitverhalten auf kritischen Bereich schließen lassen.

Um die geforderte Zeitunabhängigkeit zu schaffen, muss der Entwickler ein gewisses Grundwissen über die verwendete Hardware, den verwendeten Compiler und dessen Optimierungsverhalten beziehungsweise dessen Optimierungseinstellungen besitzen. Weiters muss der Entwickler auch wissen, welche Befehle eine konstante Laufzeit aufweisen und wie er Befehle ersetzen kann, die keine konstante Laufzeit aufweisen. Ein Beispiel für einen kritischen Befehl ist eine Verzweigung oder eine Schleife.

Der Grund dafür liegt auf der Hand, durch eine Verzweigung besteht die Gefahr, dass beide Pfade unterschiedlich lange benötigen. Dasselbe Problem könnte natürlich auch bei einer Schleife auftreten. Bis die Abbruchsbedingung erreicht ist, wird der Programmcode in der Schleife ausgeführt. Ist die Abbruchsbedingung erreicht, dann ist die Schleife zu Ende. Man sieht schon, dass die dafür benötigte Zeit vom Programmcode und der Abbruchsbedingung abhängt. Anhand des Beispiels von `ArrayCompare` kann gezeigt werden, wie fatal sich eine If-Verzweigung auf die Ausführzeit auswirken kann. Im Sinne der besseren Verständlichkeit wird angenommen, dass beide Arrays dieselbe Länge haben. Weiters sollte man noch erwähnen, dass es auch andere Attacken als die Timing Attacke gibt. Diese werden im folgenden Abschnitt beziehungsweise in den folgenden Codeblöcken nicht berücksichtigt. Ein Beispiel für eine andere Attacke ist, dass sobald es gelingt die for-Schleife zu überspringen, der Vergleich auf jeden Fall ein positives Resultat zur Folge hat und somit die Überprüfung der beiden Arrays er erfolgreich war.

```
bool ArrayCompare(Array A, Array B, int length) {
    for ( int index = 0 ; index <= length ; index++ ){
            if ( A[i] != B[i] )  return( FALSE );
        }
        return( TRUE );
}
```

Codeblock 2.1: Naive Implementierung von ArrayCompare

Die im Codeblock 2.1 dargestellte If-Verzweigung sorgt dafür, dass der Vergleichsvorgang abgebrochen wird, was zu einer unterschiedlichen Laufzeit führt. Dies ist ein Verhalten das von den meisten Programmierern von nicht zeitkritischen Programmen erwünscht ist, allerdings in diesem speziellen Fall kann man das Zeitverhalten ableiten, wie viele Stellen beim Vergleichen der Arrays übereinstimmten.

Eine Lösung für das Problem ist eigentlich relativ einfach, man benötigt eine zusätzliche Variable die den Zustand speichert und initialisiert diese auf TRUE. Danach muss man nur noch den Vergleich mit dem gespeicherten Zustand Und verknüpfen. Der Zustand wird am Ende von `ArrayCompare` zurückgegeben. Der Grund für die Initialisierung auf TRUE ist, dass der Zustand der Variablen so lange TRUE ist, bis die Variable das erste Mal mit einem FALSE Und-Verknüpft wird. Danach kann sie durch die Und-Verknüpfung nicht mehr auf TRUE gesetzt werden. Dies sieht man in Codeblock 2.2.

```
bool ArrayCompare(Array A, Array B, int length) {
    bool isOk = TRUE;
    for ( int index = 0 ; index <= length ; index++ ){
        if ( A[i] == B[i] )
            isOk = isOk & TRUE;
        else
            isOk = isOk & FALSE;
    }
    return( isOk );
}
```

Codeblock 2.2: Überlegte Implementierung von ArrayCompare

Wie man sieht verwendet dieser Ausdruck ebenfalls eine If-Verzweigung, allerdings benötigen beide Pfade der Verzweigung nun gleich lange (sofern man Pipelining außer Acht lässt). Das Einzige das jetzt noch die Ausführungsgeschwindigkeit des Programmcodes beeinflusst ist die Implementation der If-Verzweigung. Diese kann je nach Implementation einmal mit nur einer Sprunganweisung auskommen oder aber auch mit zwei. Das Problem stellt dabei die Implementation mit nur einer Sprunganweisung dar, denn diese weist beim Ausführen des If-Zweiges eine andere Laufzeit aus als beim Else-Zweig. Dieser minimale Laufzeitunterschied entsteht auf Grund eines zusätzlichen Sprungbefehls.

Verknüpft man den Zustand direkt mit dem Ergebnis des Vergleichs, so kann man sich die If-Verzweigung sparen. Der Grund dafür ist, dass der logische Ausdruck ausgewertet wird und das Ergebnis ist ein Boolescher Wert. Dieser wird dann mit dem Zustand Und-Verknüpft, was dazu führt, dass sobald ein Ergebnis FALSE ist, es immer FALSE bleibt. Der Codeblock 2.3 zeigt die so implementierte `ArrayCompare` Funktion.

```
bool ArrayCompare(Array A, Array B, int length) {
    bool isOk = TRUE;
    for ( int index = 0 ; index <= length ; index++ ){
        isOk = isOk & ( A[i] == B[i] ) ;
    }
    return( isOk );
}
```

Codeblock 2.3: If-Lose Implementierung von ArrayCompare

Wie man sieht ist die Funktion `ArrayCompare` nur noch von der Länge der zu verarbeitenden Daten abhängig, nicht aber vom Ergebnis der Zustandsvariablen. Dieses einfache Konzept lässt sich mit ein paar Anpassungen für viele Fälle anwenden, in denen sonnst eine If-Verzweigung benötigt würde.

Der Nachteil dieses ist, dass man sehr viele Informationen benötigt um sicherzustellen, dass der geschriebene Code wirklich zeitunabhängig ist. Zusätzlich sollte man beim Implementieren immer wieder Tests durchführen, die prüfen, ob der Code wirklich zeitunabhängig ist.

Kryptohardware

Hier wird eine Hardware verwendet, die es dem System ermöglichen soll, zeitkritische Berechnungen in konstanter Zeit zu absolvieren. Dies ist meist ein kryptografischer Coprozessor (Kryptocoprozessor), wie der in der Arbeit [ESM+06] beschriebene. Dieser ist in der Lage öffentliche und private Schlüssel Verschlüsselungsalgorithmen auszuführen, wie zum Beispiel DES, AES und ECC. Dafür wurden die Kernoperationen in die Hardware implementiert. Ein rekonfigurierbarer Programmspeicher beinhaltet die Befehlsabfolgen, die ausgeführt werden müssen, um einen bestimmten kryptografischen Algorithmen auszuführen. Ein Befehlsdecoder steuert die Kernoperationen an.

Ein weiters Beispiel wird in der Arbeit [GS02] vorgestellt. Es handelt sich dabei um eine Implementierung eines AES Public Key Kryptocoprozessors, der resistent gegen mehrere Seitenkanalattacken ist. Dafür wurden einige der AES Operationen in Hardware implementiert. Die Operationen RowShift und ByteSub werden hingegen von einer Standard CPU übernommen. Dies ist der Grund warum der Kryptocoprozessor nur in Kombination mit einer CPU arbeitet, wobei die CPU nicht sehr leistungsfähig sein muss. Es reicht zum Beispiel ein SLE66P, was ein 8051 basierender Sicherheitsmikrocontroller ist. Ein weiters Beispiel wird in der Arbeit [AAK09] vorgestellt. Es handelt sich dabei um einen Low Power Kryptocoprozessor, der ECC-Berechnungen mit 168 Bit und 192 Bit durchführen kann und dabei gerade einmal 23.1 μW und 26.2 μW benötigt bei einer Frequenz von einem Megaherz. Um diese geringen Leistungsaufnahmen zu erreichen wurde eine Finite State Machine verwendet, die das Rechenwerk für die ECC Berechnung ansteuert.

Eine Alternative zu Kryptocoprozessoren wird in der Arbeit [MH10] vorgestellt. Hierbei handelt es sich nicht um einen Kryptocoprozessor der die Berechnungen übernimmt, sondern es geht um die Nutzung eines Timers, der dafür sorgen soll, dass die Verschlüsselungsoperationen immer gleich lange benötigen. In der gewonnenen Zeit kann die Hardware andere Operationen ausführen, die natürlich nicht im Zusammenhang mit der Verschlüsselungsoperation stehen.

Man sieht schon wie unterschiedlich die Lösungsansätze für die Hardware Implementiertierung aussehen, um kryptografische Berechnungen sicherer zu machen. Man darf dabei nur nicht vergessen, dass diese Verbesserung der Sicherheit durch zusätzliche Chipflächen erkauft wird.

Das Design eines speziellen Algorithmus wird auf ein konstantes Laufzeitverhalten optimiert

In diesem Bereich gibt es viele spezielle Lösungen, die für einen entsprechenden Algorithmus zugeschnitten sind. Dabei wird nicht immer der gesamte Algorithmus verändert,

sondern es werden nur die Bereiche verändert, die die unterschiedlichen Laufzeiten verursachen. Weiters handelt es sich da nicht immer um kryptografische Algorithmen, sondern es werden Algorithmen wie zum Beispiel der Aufbau eines zeitunabhängigen `ArrayCompare` beschrieben, dass unabhängig von der gewählten Programmiersprache und Plattform immer in konstanter Zeit ausgeführt wird.

Es wird teilweise auch dafür gesorgt, dass die gemessenen Zeitunterschiede in keinem Zusammenhang mit den verwendeten Daten stehen, das heißt es wird nicht dafür gesorgt, dass der Algorithmus immer mit konstanter Laufzeit ausgeführt wird, sondern es werden die verwendeten Daten vom Zeitverhalten entkoppelt.

Einige Möglichkeiten dafür wird in der Arbeit [Koc96] von Kocher vorgestellt. Es geht dabei um das sogenannte Blinding. Damit wird verhindert, dass der Angreifer die Eingangsdaten für die diskrete Exponentialfunktion kennt. Um dies zu erreichen, benötigt man eine bidirektionale Funktion, die Eingangsdaten so transformiert, dass sie zufällig im Eingangsdatenbereich angeordnet sind. Nach der Transformation werden die transformierten Daten für die diskrete Exponentiation verwendet. Anschließend findet die Rücktransformation der Daten statt.

Die so durchgeführte Verschlüsselung wird nicht in konstanter Laufzeit ausgeführt, was aber kein Problem darstellt, denn der Angreifer kann keinen Zusammenhang zwischen den Laufzeiten und Eingangsdaten herstellen.

Ein anderes Beispiel betrifft die Implementierung des AES Algorithmus und wird in der Arbeit [BR09] vorgestellt. Dabei wird der exzessive Zugriff auf die Lookup Table, beim Rijndael Algorithmus, als Grund für die Timing Attacke gesehen. Diese Zugriffe führen in Verbindung mit Cachespeicher zu unterschiedlichen Laufzeiten, die durch die Timing Attacke ausgewertet werden können.

Ihr Lösungsansatz ist die Verwendung des Dynamic Flush Algorithmus. Dieser löscht den Cachespeicher in unbestimmten Intervallen. Das führt dazu, dass das Zeitverhalten unvorhersehbar wird. Dieser Ansatz soll sich auch für andere Algorithmen mit ähnlichen Problemen eignen.

In den Arbeiten [BEPM10, BR09, Ber04] wird die Möglichkeit erwähnt, dass sich die Timing Attacke gegen AES ganz leicht verhindern lässt, wenn man den Cachespeicher nicht verwendet. Dies führt dann aber zu signifikanten Performanceeinbusen.

2.4 Statistische Grundlagen

Die statistischen Grundlagen sind für die folgenden Kapitel dieser Masterarbeit von Bedeutung, da diese die Grundlage für die Analyse der Laufzeiten bildet. Sie werden daher in diesem Abschnitt kurz erklärt. Dabei werden Grundlagen wie das Minimum, das Maximum, der arithmetische Mittelwert und die Standardabweichung vorausgesetzt und daher nicht extra behandelt. Für ein besseres Verständnis wird teilweise auf die Verwendung von Beispielen zurückgegriffen, da sich dadurch Zusammenhänge anschaulicher darstellen lassen. Weiters stammen alle Informationen aus diesem Abschnitt aus den Büchern [Nac06, BZ09, WMMY06] sowie der Arbeit [CKNS01].

2.4.1 Normalverteilung

Die Normalverteilung oder auch Gaußverteilung ist eine Verteilungsdichtefunktion. Das bedeutet, sie gibt an wie Wahrscheinlichkeit des Auftretens eines zufällig gewählten Wertes in der Verteilung ist. Der zufällige Wert könnte zum Beispiel der Ist-Wert einer Stichprobe oder der Messung sein, die einen Soll-Wert aufweisen sollte. Dabei sollte der Soll-Wert der Mittelpunkt der Verteilung sein, sofern sich die Unterschiede beim Ist-Wertes durch eine Zufälligkeit ergeben. Die Normalverteilung ist eine symmetrische Verteilung die einen Maximalpunkt besitzt [BZ09]. Sie beschränkt sich dabei nicht auf einen Wertebereich in dem die Zufallswerte vorkommen können, vielmehr ist jeder Wert möglich, wobei seine Auftrittswahrscheinlichkeit mit zunehmendem Abstand zum Mittelwert abnimmt. Als Maß für die Distanz vom Mittelwert wird die Standardabweichung s verwendet. Dabei entspricht eine Distanz von plus und minus einer Standardabweichung bei einer Stichprobe ungefähr 68,28 Prozent aller Fälle. Erhöht man die Distanz auf zwei Standardabweichungen, so liegen 95,44 Prozent aller Stichproben innerhalb dieses Bereiches.

2.4.2 Stichprobe

Eine Stichprobe ist das Entnehmen einer Teilmenge aus einer größeren Gesamtmenge. Die Teilmenge wird dabei anhand verschiedenster Kriterien gewählt. Das Ziel der Stichprobe ist es, eine möglichst repräsentative Teilmenge aus der Gesamtmenge zu ziehen. Es gibt verschiedenste Herangehensweisen, wie die Entnahme einer Stichprobe zu erfolgen hat, um dieses Ziel zu erreichen. Die üblichsten sind:

- zufälliges Auswählen von Stichprobenelementen
- systematisches Auswählen von Stichprobenelementen

Im Fall des zufälligen Auswählens einer Probe, kann jedes Element der Gesamtmenge mit der gleichen Wahrscheinlichkeit gezogen werden. Daher deckt diese Form der Probenahme die Anforderung eine möglichst repräsentative Stichprobe zu erhalten am besten ab. Der Grund dafür ist, dass keinerlei Annahmen über die Elemente der Gesamtmenge gemacht werden. Diese ist der wesentlichste Unterschied zur systematischen Auswahl der Elemente einer Stichprobe, denn hierbei existieren Informationen über die Gesamtmenge, die die Wahl der Stichprobenelemente bestimmt.

Ein Beispiel für ein zufälliges Ziehen einer Stichprobe ist eine Lottoziehung. Hierbei werden zufällig sechs Kugeln aus einer Gesamtmenge von 45 Kugeln gezogen. Ein Beispiel für systematisches Auswählen von Stichprobenelementen wäre das Testen von Hardware und Software. Hierbei werden Annahmen über das zu prüfende System gemacht. Zum Beispiel, dass bei einer Software der Lese- oder Schreibzugriff auf den Randbereich eines Speicherblocks sehr viel problematischer ist, als ein Zugriff auf die Mitte des Speicherblocks. Der Grund dafür liegt auf der Hand, in der Mitte des Speicherblocks besteht keine Gefahr, dass auf einen anderen Speicherblocke zugegriffen wird, während hingegen im Randbereich eine kleine Ungenauigkeit dazu führen kann, dass ein fremder Speicherblock gelesen oder beschrieben wird. Daher werden diese Stellen der Software als mögliche Problemstelle betrachtet. Weiters wird davon ausgegangen, dass sich diese Zugriffe einheitlich verhalten. Das bedeutet, wenn die Daten in einem eindimensionalen Feld gespeichert

sind und der Zugriff auf die Speicherzelle über eine einzelne Indexvariable realisiert ist, so müsste sich jeder Zugriff auf jede beliebige Speicherzelle, die sich in der Mitte des Speicherblocks befindet, gleich verhalten. Anhand dieser Annahme wird versucht, die potentiell problemanfälligen Bereiche des Systems zu überprüfen.

Zieht man mehrmals hintereinander aus derselben Gesamtmenge eine Stichprobe, wobei die Elemente der Stichproben in der Gesamtmenge erhalten bleiben, so wird man feststellen, dass der Mittelwert über die einzelnen Stichproben schwankt. Der Grund dafür ist, dass bei der Entnahme einer Stichprobe unterschiedliche Elemente aus der Gesamtmenge gezogen werden. Die Wertigkeit der gezogenen Elemente kann dabei variieren, was dazu führt kann, dass sich die Mittelwerte von zwei gezogenen Stichproben nicht gleichen. Je geringer diese Schwankung ist, desto ähnlicher sind sich die gezogenen Stichproben und desto repräsentativer sind die gezogenen Stichproben. Hierbei sollte auch noch erwähnt werden, dass die Stärke der Schwankungen hauptsächlich von der Anzahl der Elemente, die für die Stichproben gezogen weden abhängt. Das bedeutet, je mehr Elemente gezogen werden, umso geringer fallen die Schwankungen der Mittelwerte aus, da eine Stichprobe mit einer größeren Anzahl an Elementen, die Gesamtmenge besser und repräsentativer abdeckt.

Wird aus einer Gesamtmenge mehrmals eine Stichprobe gezogen, die aus mindestens 30 Elementen besteht, so ist die Verteilung der Mittelwerte der einzelnen Stichproben normalverteilt [Nac06]. Der Grund dafür ist, dass sich viele kleine unsymmetrische Effekte aufsummieren und sich somit eine Normalverteilung einstellt. Dieses Prinzip wird auch in der Messtechnik eingesetzt, wenn durch mehrfaches Messen eine höhere Genauigkeit erzielt wird, als durch eine Einzelmessung möglich wäre.

Vertrauensintervall bei Stichproben

Bei Stichproben ist das Vertrauensintervall beziehungsweise das Konfidenzintervall eine wichtige Kenngröße, die Auskunft über erhobenen Daten gibt. Typischer Weise wird das Vertrauensintervall in Prozent angegeben, wobei typische Werte 95% oder 99% sind. Sie drückt die Wahrscheinlichkeit aus, dass ein erhobenes Element der Stichprobe innerhalb eines bestimmten Bereichs zum wahren Wert befindet. Dieser Bereich wird durch den Mittelwert der Strichprobe und die Standardabweichung des Mittelwerts festgelegt. Die Abbildung 2.4 stellt eine Normalverteilung um den Mittelwert dar. Für diesen Mittelwert μ wird das Vertrauensintervall angegeben, wobei in Standardabweichung s gemessen wird.

2.4.3 Signifikanz

Als Signifikant wird ein Ergebnis einer Stichprobe oder Messung bezeichnet, wenn es sich aus der Masse der zufälligen Ergebnisse deutlich hervorhebt. Daher liegt die Vermutung nahe, dass es dafür einen Grund geben könnte. Dies könnte zum Einen auf Grund eines oder mehrerer bekannter Parameter erfolgen oder zum Anderen auf Grund eines oder mehrer nicht bekannter Parameter oder es könnte aber auch einfach nur Zufall sein. Die Signifikanz gibt Aufschluss darüber wie wahrscheinlich es ist, dass ein Ergebnis durch Zufall zustande kommt.

Ein Beispiel für eine Signifikanz in einer Menge wäre, wenn bei der Produktion von Holzkugeln allen 10000 Stück eine Stichprobe von 10 Stück gemacht wären, deren Größe

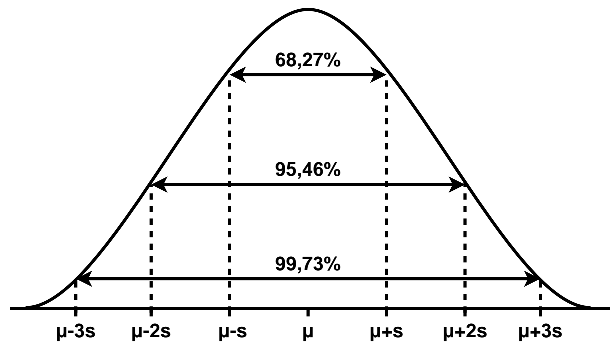

Abbildung 2.4: Normalverteilung von Mittelwerten μ mit Vertrauensintervall angegeben in Sekunden

passen muss. Die Schwankung der Größe der Kugeln ist normalverteilt. Im Schnitt sind bei 10000 Holzkugeln ungefähr 10 Holzkugeln dabei die zu groß oder zu klein sind.

Würde die Stichprobe 10 Holzkugeln der falschen Größe enthalten, so wäre dieses Ergebnis signifikant, da es ziemlich unwahrscheinlich ist, dass man genau die 10 Kugeln aus der Menge gezogen hat, die eine nicht passende Größe aufweisen. Es wäre aber dennoch möglich, dass dieser Fall eintritt. Es würde aber auch schon genügen, wenn bei 10 gezogenen Holzkugeln 5 Stück die falsche Größe aufweisen, um daraus einen falsche Information über die Gesamtmenge zu erhalten, denn auch dieses Ergebnis wäre signifikant. Wenn man davon ausgeht das die Stichprobe ein Maß für die Gesamtmenge darstellt. Dies würde bedeuten, dass ungefähr die Hälfte aller Holzkugeln keine korrekte Größe aufweisen. Das wären in diesem Beispiel circa 5000 Stück.

Das Vorgehen für die Bestimmung der Signifikanz einer Stichprobe oder Messung kann auf unterschiedlich Weise erfolgen, wie zum Beispiel der Signifikanztest anhand von Mittelwertsunterschieden. Dies ist der beliebteste Signifikanztest, es wäre aber auch möglich, dass der Signifikanztest eine bestimmte Forderung besitzt, die erfüllt sein muss um als signifikant zu gelten. Ein Beispiel hierfür wäre, dass 50 Prozent aller Messungen über einem bestimmten Wert liegen muss.

Der Signifikanztest

Wie bereits erwähnt, dient der Signifikanztest zum Auffinden von Signifikanzen. Er wird verwendet um die Richtigkeit einer Hypothese anhand von erhobenen Daten zu beweisen. Die Hypothese entsteht auf Grund einer Theorie, die eine Frage beantworten soll. Die Frage könnte zum Beispiel lauten, besteht ein Zusammenhang zwischen dem binären Resultat einer Funktion und der benötigten Ausführzeit. Das binäre Resultat ist `WAHR` oder FASCH und könnte zum Beispiel beim Überprüfen jedes Zeichens eines Passworts auftreten. Die Theorie führt dann zu einer Hypothese, der Nullhypothese H_0. Diese könnte lauten, dass das Ergebnis von der Laufzeit abhängt. Das bedeutet, wenn das Resultat der Funktion `WAHR` ist, dann benötigt die Funktion mehr Zeit als wenn das Resultat der Funktion `FALSCH` ist oder umgekehrt. Als Gegenstück zur Nullhypothese gibt es die alternativ Hypothese H_1, die die Aussage der Nullhypothese negiert. Alternativ Hypothese für die vorhin beschriebe

Kombination	A	B	C	D	E	F	G	H	I	J	K
WAHR	0	1	2	3	4	5	6	7	8	9	10
FALSCH	10	9	8	7	6	5	4	3	2	1	0
p [%]	0,1	1,0	4,4	11,7	20,5	24,6	20,5	11,7	4,4	1,0	0,1

Tabelle 2.1: Auftrittswahrscheinlichkeit der möglichen Kombinationen

Nullhypothese wäre, dass das Resultat der Funktion unabhängig von der Laufzeit der Funktion ist. Das heißt, dass die Schwankung der Laufzeit nicht im Zusammenhang mit dem Ergebnis steht.

Würde man davon ausgehen dass die Laufzeitschwankungen zufällig sind und somit nicht vom Ergebnis der Funktion abhängen, so wäre eine längere Ausführungszeit für beide möglichen Zustände des Resultats gleich wahrscheinlich. Es würden gleich viele Resultate deren Ergebnis `WAHR` ist länger benötigen, wie Resultate deren Ergebnis `FALSCH` ist.

Dies trifft allerdings erst ab einer gewissen Mindestanzahl an Messungen auf. Der Grund dafür ist, dass die Wahrscheinlichkeiten miteinander multipliziert werden. Werden zum Beispiel 2 Messungen durchführt, deren Ergebnis ein binärer Zustand ist, wobei jeder Zustand dieselbe Auftrittswahrscheinlichkeit ($p_{binaer} = 50\%$) hat, so ist es ist die Wahrscheinlichkeit das zwei auf einander folgende Ereignisse, dass selbe Resultat haben gleich 25 Prozent ($p_{binaer} * p_{binaer} = 0.25 = 25\%$). Wie man sieht, ist diese nicht sonderlich unwahrscheinlich. Erhöht man hingegen die Anzahl n der durchgeführten Messungen auf 10, so wären die in der Tabelle 2.1 dargestellten `WAHR` und `FALSCH` Kombinationen möglich. Ihre Auftrittswahrscheinlichkeit p wird in Prozent angegeben und sind Binomialverteilt.

Wie man sieht, ist die Auftrittswahrscheinlichkeit von 10-mal demselben Ergebnis äußerst unwahrscheinlich und je näher man dem Mittelwert kommt umso wahrscheinlicher werden die Resultate. Daher eignet sich der Mittelwert auch gut als Unterscheidungsmerkmal um Veränderungen festzustellen. Ist die Auftrittswahrscheinlichkeit für ein Ereignis nicht gleich groß, wie die Wahrscheinlichkeit dass es nicht eintritt, so muss dieser Unterschied für die weiteren Betrachtungen berücksichtigt werden.

Für einen Signifikanztest müssen zwei unterschiedliche Gruppen von Daten erhoben werden. Die erste Gruppe ist dabei die Gruppe, die unbeeinflusst ist. Sie stellt somit den Urzustand eines Systems dar. Die zweite Gruppe ist eine Gruppe, die auf Grund einer Veränderung einen Effekt aufweisen kann, der zuvor nicht vorhanden war. Was dabei exakt verändert wurde, muss nicht bekannt sein, denn durch Hypothese wird eine Vermutung geäußert, die durch einen Signifikanztest bewiesen oder widerlegt wird. Ein Beispiel hierfür wäre, dass durch zugeben einer zusätzlichen Substanz bei der Herstellung eines Akkus die Lebensdauer eines Akkus gesteigert wird. Dabei wäre dann in der ersten Gruppe Akkus ohne die Substanz und in der zweiten Gruppe wären die Akkus die die Substanz enthalten. Diese beiden Gruppen werden dann im Signifikanztest miteinander verglichen. Es muss zuvor aber überlegt werden, welche Art des Signifikanztests zum Einsatz kommt.

Wird ein Signifikanztest durchgeführt, der sich zum Beispiel anhand von Mittelwerten unterscheidet, so gibt es unterschiedliche Typen von Hypothesen, die gerichtete oder ungerichtete Hypothese und spezifische oder unspezifische Hypothese. Eine ungerichtete Hypothese trifft nur die Aussage, dass ein Unterschied auftreten muss. Das bedeutete das der Mittelwert der ersten Gruppe nicht gleich dem Mittelwert der zweiten Gruppe ist

($\mu_1 \neq \mu_2$). Im Gegensatz dazu, wird bei der gerichteten Hypothese eine Aussage darüber gemacht, in welche Richtung sich der Mittelwert der zweiten Gruppe entwickelt (zum Beispiel: $\mu_1 < \mu_2$). Der Unterschied zwischen spezifischer und unspezifischer Hypothese ist, dass bei der spezifische eine Aussage über die Größe des Unterschieds der Mittelwerte gemacht wird. Dieses Unterscheidungen betreffen die Formulierung der Hypothese, lassen sich auch auf andere Arten des Signifikanztest übertragen, wie zum Beispiel dem Signifikanztest, der auf Basis des Korrelationskoeffizienten unterscheidet. Das Ergebnis eines Signifikanztests ist der p-Wert der über einer vorgegeben Schranke liegen muss. Diese Schranke wird Signifikanzniveau α bezeichnet und ist üblicherweise 1% oder 5% groß. Das bedeutet dass man bei Normalverteilung sich in dem Bereich aufhält, der nur noch eine 1% oder 5% Auftrittswahrscheinlichkeit besitzt. Liegt der durch den Signifikanztest berechnete p-Wert innerhalb des Signifikanzniveaus ($p - Wert \leq \alpha$) so wird die Nullhypothese H_0 ausgewählt, liegt er außerhalb, so wird die alternativ Hypothese H_1 ausgewählt.

Dieses Vorgehen kann, neben der richtigen Entscheidung, aber auch zu Fehlern beim Entscheiden einer Hypothese führen. Dies tritt ein, wenn eine Entscheidung auf Grund der Stichprobe oder Messung getroffen wurde, diese aber nicht stimmt. Dabei wird unterschieden ob es sich um einen $\alpha - Fehler$ oder $\beta - Fehler$ handelt. Der $\alpha - Fehler$ ist dabei der Fehler der gemacht wird, wenn fälschlicherweise die alternativ Hypothese H_1 gewählt wird, wobei die Nullhypothese richtig gewesen wäre. Er entspricht dabei dem Signifikanzniveau. Der $\beta - Fehler$ ist der Fehler der auftritt, wenn die Nullhypothese H_0 fälschlicherweise beibehalten wird, obwohl die alternativ Hypothese richtig gewesen wäre. Dieser lässt sich nur schwer abschätzen, da er von mehreren Größen abhängt, wie zum Beispiel dem Signifikanzniveau und der Mittelwertverteilung. Dabei sollte noch erwähnt werden, dass der $\beta - Fehler \neq 1 - \alpha - Fehler$. Die Begründung dafür ist:

$$\begin{aligned} \beta &= P(Entscheidung für H_0 | H_1 gilt) \\ &\neq P(Entscheidung für H_0 | H_0 gilt) = 1 - \alpha \end{aligned} \text{[Nac06]} \quad (2.3)$$

Der $\beta - Fehler$ wird allerdings dennoch sehr stark vom Signifikanzniveau α beeinflusst. Der Zusammenhang ist dabei einfach ausgedrückt, je kleiner das Signifikanzniveau α gewählt wird, desto größer wird der $\beta - Fehler$. Daher ist es ratsam, wenn man nicht will, dass die alternativ Hypothese H_1 fälschlicherweise gewählt wird, dass man das Signifikanzniveau α sicherheitshalber klein wählt.

2.4.4 Anwendung von Signifikanztests bei der Timing Attacke

Für eine Timing Attacke werden, wie schon im Abschnitt 2.3 beschrieben, die Laufzeit einer Anwendung beziehungsweise eines System vermessen. Die daraus resultierenden Messdaten werden nach Abschluss des Messvorgangs statistisch analysiert. Dabei werden die Messdaten in Abhängigkeit der Eingangsdaten γ_i betrachtet. Die Eingangsdaten sind zum Beispiel, Schlüssel, Nachrichten sowie deren Antwort. Die Analyse erfolgt mittels Prüfung der Signifikanz. Für den Signifikanztest S werden jeweils zwei Eingangsdaten γ_1 und γ_2 inklusive ihrer Laufzeitergebnisse $T(\gamma_1)$ und $T(\gamma_2)$ bearbeitet. Es sollte dabei noch angemerkt werden, dass die Laufzeitmessung für jede der beiden Eingangsdaten γ_1 und γ_2 mehrmals durchgeführt wird. Die Anzahl der Messdurchläufe n muss dabei ausreichen, um ein Aussagekräftiges Ergebnis des Signifikanztests zu erhalten. Der Signifikanztests S wird

auf alle Laufzeitergebnisse $T(\gamma_1) = t_1[1], t_1[2], \ldots, t_1[n]$ und $T(\gamma_2) = t_2[1], t_2[2], \ldots, t_2[n]$ angewandt. Das Ergebnis des Signifikanztests $S(T(\gamma_1), T(\gamma_2))$ gibt Aufschluss über die Wahrscheinlichkeit α, dass ein Zusammenhang zwischen den beiden Eingangsdaten und ihren Laufzeitmessungen besteht.

$$\alpha = S(t_1[1], t_1[2], \ldots, t_1[n], t_2[1], t_2[2], \ldots, t_2[n]) \tag{2.4}$$

Zeigt sich dabei ein Zusammenhang zwischen den beiden Eingangsdaten und dazugehörigen gemessenen Laufzeiten, so war die Timing Analyse erfolgreich und es müssen weitere Überprüfungen vorgenommen werden. Zeigt sich kein Zusammenhang, so bedeutet dies nicht, dass kein Zusammenhang zwischen den Daten besteht. Es bedeutet nur, dass der gewählte Signifikanztest keine Signifikanz feststellen konnte. Ein anderer Signifikanztest, der noch nicht ausprobiert wurde, könnte zum Beispiel eine Signifikanz feststellen.

Mit anderen Worten, das Testen mittels Signifikanztests ist vergleichbar mit einem See, bei dem es heißt, dass darin keine Fische existieren. Würde man jeden Kubikmeter des Sees absuchen und dabei keinen Fisch finden so wäre die Aussage, dass er keine Fische beinhaltet, bestätigt. Würde man hingegen einen einzigen Fisch finden, so könnte man die Suche abbrechen, da man ja schon einen Fisch gefunden hat und dadurch wäre bewiesen dass die Aussage falsch war.

Da das Durchsuchen des ganzen Sees sehr aufwendig ist, könnte man natürlich einen einfacheren und bequemeren Weg wählen. Dabei würde man nur eine Angel ins Wasser halten und warten, dass ein Fisch anbeißt. Wenn dann ein Fisch anbeißt, wäre bewiesen, dass die Aussage, dass kein Fisch im See ist, falsch ist. Beißt hingegen kein Fisch an kann man nicht daraus schließen, dass keinen Fisch im See lebt. Es könnte ja sein, das man einen zu großen Haken verwendet hat oder einen falschen Köder oder das man irgendetwas anderes falsch gemacht hat [CKNS01].

Daher ist es ratsam mehrere Signifikanztests für die Timing Analyse zu verwenden. Der Vorteil besteht dabei darin, das unter Umständen ein signifikantes Verhalten der Laufzeiten erkannt wird, die sonst nicht erkannt worden wäre und somit in die weiteren Untersuchungen einfließt. Die drei gängigsten Signifikanztests für Seitenkanalattacken werden in der Arbeit [CKNS01] erwähnt und verwendet. Das sind:

- Distanz der Mittelwerte
- Anpassungstests
- Wilcoxon-Rangsummentest

Findet einer der verwendeten Signifikanztests eine Signifikanz, so wird diese in Block oder eine Gruppe eingeteilt. Diese Blöcke beschreiben dabei eine mögliche Eigenschaft der Eingangsdaten im Zusammenhang mit ihrer Laufzeit. Für eine weitere Untersuchung dieser möglichen Eigenschaft wird ein Zufälligkeitstest verwendet, der auf eine binär Zeichenkette angewendet wird. Das Ergebnis der Zufälligkeitstests ist eine Wahrscheinlichkeit. Diese trifft eine Aussage darüber, wie wahrscheinlich eine erkannte Eigenschaft ist. In der Arbeit [CKNS01] wurden die zwei aufgelisteten zufälligkeitsüberprüfenden Testverfahren empfohlen, da diese einfach zu Implementieren sind und eine ausreichende Sensibilität zur Verfügung stellen. Diese sind:

- Frequency Test
- Run Test

Wie man sieht ist für die Anzahl der Laufzeitmessungen für die Timing Analyse wichtig. Daher ist es erforderlich, dass man eine bestimmte minimale Anzahl von Messungen durchführt. Diese variiert dabei mit der Eindeutigkeit der Messergebnisse. Das bedeutet, sind die Messergebnisse bei einer bestimmten Anzahl an Messungen nicht aussagekräftig, so kann man durch Erhöhen der Anzahl der Messergebnisse, unter Umständen, die Auswertung verbessern. Allerdings darf man hierbei nicht beliebig lange messen und nur darauf warten, dass sich signifikantes Verhalten einstellt [Nac06]. Der Grund dafür ist, dass als Abbruchsbedingung ein signifikantes Verhalten vorausgesetzt wird. Dies wäre ein Grund für ein nicht aussagekräftiges Ergebnis des Signifikanztests. Gründe für nicht aussagekräftige Messergebnisse sind zum Beispiel, zu wenige Laufzeitmessungen, zu stark verrauschte Laufzeitergebnisse oder dass die zu vermessende Funktion einer Anwendung in konstanter Zeit abgearbeitet wird. Letzteres kann auch der Grund für verrauschte Laufzeitmessungen sein, da bei einer konstanten Ausführungszeit nur die Ungenauigkeit des Messergerätes vermessen wird. Diese weist meist eine Normalverteilung auf.

Kapitel 3

Design

Dieser Hardware/Software Codesign Flow soll dem Entwickler helfen, ein System zu entwerfen, bei dem es möglich ist, Bereiche zu definieren und zu markieren, die aus zeitlicher Sicht, als kritisch gelten. Dadurch sollen Produkte, die aus diesem Hardware/Software Codesign Flow hervorgehen, gegenüber Timing Attacken residenter sein.

Ein Hardware/Software Codesign Flow wird für den Entwurf eines Hardware/Software Systems eingesetzt. Es wird dabei nicht wie beim klassischen Design eines Systems zuerst die Hardware entworfen und dann die dazugehörige Software, vielmehr führt man den Entwurf der Hardware und Softwarekomponenten gleichzeitig aus.

Der Vorteil daran ist, dass man nicht eine starre Hardware zur Verfügung hat für die es gilt eine Software zu schreiben. Vielmehr ist es die Möglichkeit für die einzelnen Komponenten beziehungsweise Funktionen des Systems zu entscheiden, ob sie in Hardware oder in Software auszuführen sind. Dieser Schritt wird Partitionierung genannt und kann iterativ mehrfach wiederholt werden. Dadurch soll das System so entworfen werden, dass es seine Anforderungen optimal erfüllt.

Unter Anforderungen versteht man in diesem Fall die Parameter des Produkts, zum Beispiel die maximal benötigte Chipfläche, die maximalen Kosten, den maximalen und mittleren Leistungsbedarf oder die Flexibilität des Produktes und noch viele mehr. Dabei sind die gewählten Parameter maßgeblich dafür verantwortlich, wie die Partitionierung ausgeführt wird. Die Abbildung 3.1 zeigt eine schemenhafte Darstellung eines herkömmlichen Hardware/Software Codesign Flows. Im Vergleich dazu zeigt die Darstellung 3.2 den Hardware/Software Codesign Flow, der Timing Attacken verhindern soll.

Dafür ist es nötig kritische Bereiche des Systems während der Entwicklung zu überwa“-chen, so dass zeitliche Probleme so schnell wie möglich erkannt werden. Kritische Bereiche sind Bereiche in denen sicherheitsrelevante Operationen ausgeführt werden. Dies kann zum Beispiel das Überprüfen eines Schlüssels, Verschlüsseln von Daten und vieles mehr sein.

In der System Spezifikation werden Funktionalitäten die kritische Bereiche enthalten gekennzeichnet. Dadurch ist der Entwickler des Systems in der Lage die entsprechenden Stellen im Quelltext des Systems zu markieren. Die Markierungen werden bei der Simulation des Systems dafür genutzt, eine Zeitmessung durchzuführen. Die Zeitmessung darf dabei das Verhalten des Systems nicht beeinflussen. Der Grund dafür ist, dass eine zeitliche oder funktionale Beeinflussung des Systems dazu führen kann, dass die gemessenen Zeiten eine unnatürliche Schwanken aufweisen und dass dadurch die gesamten Messergebnisse

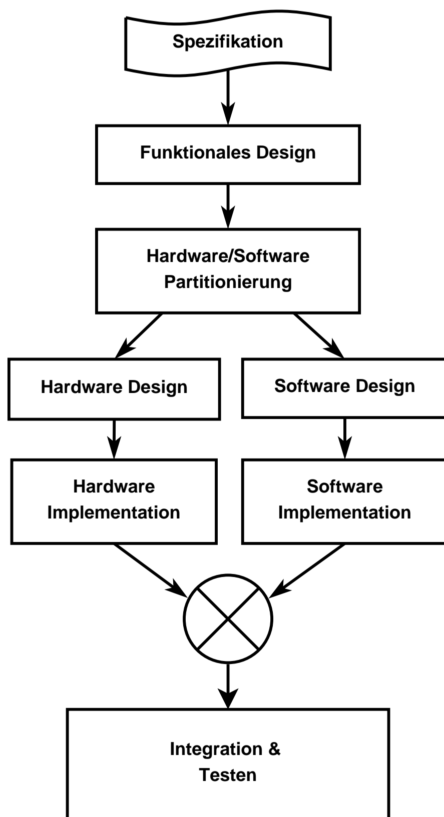

Abbildung 3.1: Herkömmlicher Hardware/Software Codesign Flow

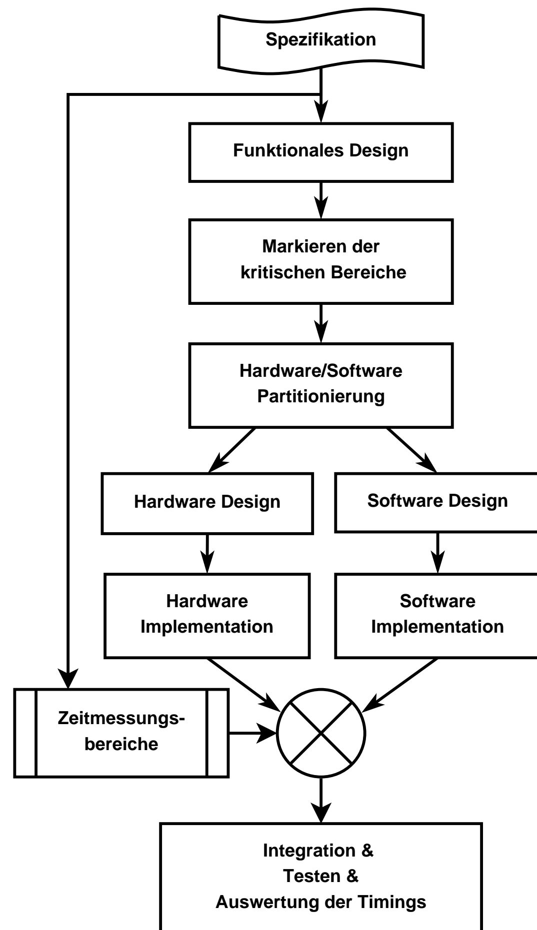

Abbildung 3.2: Hardware/Software Codesign Flow gegen Timing Attacken

der Messung unbrauchbar werden.

Beim Ausführen von Anwendungsfällen und Testfällen auf dem entworfenen System werden zu den funktionalen Ergebnissen so auch timing Ergebnisse produziert. Die Ergebnisse werden ausgewertet und an den Entwickler übergeben. Auf diese Weise hat er eine Möglichkeit zu prüfen ob die Implementierung der Spezifikation entspricht. Für eine genauere Beschreibung des Hardware/Software Codesign Flow werden mehrere Ansichten verwendet.

Zuerst wird die Logische Ansicht des Hardware/Software Codesign Flow in Abschnitt 3.1 beschrieben. Diese beschreibt die Funktionalität des Hardware/Software Codesign Flow aus der Sicht des Anwenders. Dann folgt die Prozessansicht im Abschnitt 3.2. Sie beschreibt die Abhängigkeit der einzelnen Prozesse voneinander wodurch sich die Übergänge von einem auf den anderen Prozess besser darstellen lassen. Danach kommt die Entwicklungsansicht die im Abschnitt 3.3 beschrieben wird. Sie stellt die Interaktion der am Hardware/Software Design Flow involvierten Personen beziehungsweise Personengruppen dar. Abschließend wird im Abschnitt 3.4 die Physikalische Ansicht des Hardware/Software Codesign Flow beschrieben. Diese beschäftigt sich mit den einzelnen Arbeitsschritten, die von der Spezifikation bis hin zum fertigen Produkt führt.

3.1 Funktionalität des Hardware/Software Codesign Flow

Die logische Ansicht zeigt die Funktionalität des Hardware/Software Codesign Flow aus der Sicht des Anwenders. Sie wird in Abbildung 3.3 gezeigt.

Der Ausgangspunkt ist die Customer Requirements Spezifikation (CRS oder Lastenheft), welche sich links in der Abbildung 3.3 findet. Dieses Dokument beschreibt die Anforderungen des Kunden an das zu entwickelnde Produkt. Die Anforderungen beinhalten Funktionalitäten und Sicherheitskriterien. Ergänzend zur funktionalen Beschreibung werden auch Beispiele verwendet, um zu zeigen, wie das fertige Produkt funktionieren soll.

Aus dem CRS wird das Design entwickelt. Dieses beschreibt den Aufbau des zu entwickelnden Produkts. Dafür wird das zu entwickelnde Produkt in mehrere kleinere funktional sinnvolle Blöcke aufgeteilt. Dies steigert die Übersichtlichkeit des Entwurfs. Die Blöcke werden dann genauer definiert und mit den im CRS beschriebenen Sicherheitskriterien versehen.

Während des Designvorgangs werden zwei Dokumente erstellt, dass Functional Requirements Spezifikation (FRS) Dokument und das Timing Requirements Spezifikation (TRS) Dokument. Das FRS beschreibt dabei die funktionalen Blöcke des Produkts so genau, dass die Umsetzung dieser Beschreibung zu einem funktionierenden System führt. Das TRS Dokument existiert in einem herkömmlichen Hardware/Software Codesign Flow nicht. Es integriert die Informationen zum Laufzeitverhalten in den Hardware/Software Codesign Flow. Es spezifiziert dabei, welcher der Blöcke, die im FRS spezifiziert sind, ein konstantes oder zufälliges Zeitverhalten aufweisen soll.

Aus den Spezifikationen wird ein Modell des Produktes erzeugt. Dieses beinhaltet das System sowie die dazugehörigen Annotationen. Annotationen sind Meta-Informationen die Eigenschaften eines Teils des Produktes genauer beschreiben. Diese Meta-Informationen unterscheiden sich von herkömmlichen Kommentaren dadurch, dass sie beim Umsetzen der Quellen eine definierte Bedeutung haben. Die Annotationen dieses Hardware/Software

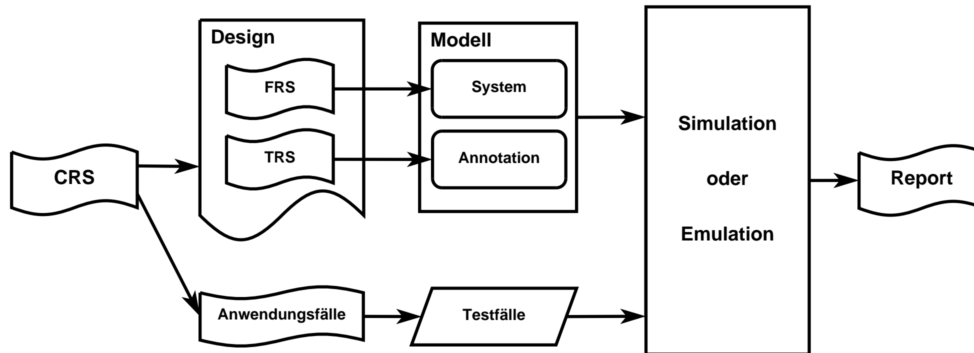

Abbildung 3.3: Logische Ansicht

Codesign Flows werden für die Messung der Zeit benötigt.

Das System hingegen beinhaltet die Hardware und Software. Diese sind in der Logischen Ansicht zusammengefasst. Der Grund dafür ist, dass der Hardware/Software Codesign Flow übersichtlich bleibt, eine Aufspaltung würde seine Darstellung nur komplexer wirken lassen. Für die Entwicklung der Hardware wird eine Hardwarebeschreibungssprache verwendet. Dies hat den Vorteil, dass das Modell der Hardware relativ einfach verändert werden kann. Das so entstehende Modell der Hardware kann durch einen Computer simuliert oder auf einem FPGA emuliert werden.

Bei der Software hingegen sieht es anders aus, diese benötigt auf jeden Fall einen Bezug zur Hardware beziehungsweise zum zugrunde liegenden System in Form eines Simulators oder Emulators. Durch das in der Hardwarebeschreibungssprache beschriebene Modell ist man in der Lage, die entwickelte Software auf der Hardware laufen zu lassen.

Mit Hilfe des CRS werden die Anwendungsfälle generiert. Diese beschreiben durch Szenarien die Nutzung des Produktes. Dadurch soll gezeigt werden, was das Produkt kann und wie es einzusetzen ist. Der Detailliertheitsgrad der skizzierten Szenarien muss so genau sein, dass die Anwendungsfälle den Nutzungsbereich des Produkts abdecken. Wie detailliert dabei auf das System eingegangen wird, ist im beschriebenen Design Flow nicht vorgegeben.

Aus den Anwendungsfällen werden die Testfälle generiert, die das Produkt auf ihre Korrektheit untersuchen sollen. Es werden daher mehr Testfälle als Anwendungsfälle benötigt, da nicht nur die externe Interaktion mit dem Produkt untersucht wird. Vielmehr werden auch Testfälle generiert die auch die Richtigkeit der einzelnen internen Funktionen abprüfen.

Die Anzahl der Testfälle hängt dabei stark vom Funktionsumfang des Produktes beziehungsweise von der Komplexität der einzelnen funktionalen Bereiche ab. Es ist wichtig, dass die Testfälle eine gute Abdeckung der Anforderung erzielen. Daher wird für die Erzeugung der Testfälle meist ein kontrollflussorientiertes Testverfahren eingesetzt. Dieses versucht durch die Nutzung des Kontrollflussgraphen, alle (möglichen) Pfade des Graphen abzudecken. Es werden dabei zwar nicht immer alle möglichen Konstellationen durchprobiert, aber man kann zumindest davon ausgehen, dass jeder Bereich des Produktes einmal getestet wurde.

Bei der Simulation oder Emulation werden die Testfälle auf das Modell angewendet.

Hierbei kann, wie oben beschrieben, entweder eine reale Hardware zum Einsatz kommen oder ein Simulationsmodell der Hardware. Das Ergebnis der Simulation oder Emulation ist der Report.

Der Report beinhaltet detaillierte Informationen über das getestete Modell. Diese Informationen beschreiben die Korrektheit des Systems im Bezug auf die Anwendungsfälle und Testfälle. Dadurch wird sichergestellt, dass die Funktion des Produktes gewährleistet ist.

3.2 Abhängigkeit der Prozesse des Hardware/Software Codesign Flow

Die Prozessansicht zeigt die Abhängigkeit der Prozesse des Hardware/Software Codesign Flow. Sie verdeutlicht dabei die Übergänge und Verteilungen. Das Flussdiagramm in Abbildung 3.4 stellt die Prozessansicht dar.

Der Ausgangspunkt für die Prozessansicht ist die Spezifikation. Diese stellt die zentrale Vereinbarung für die Umsetzung des Produktes dar. Sie beschreibt das Produkt formal. Dabei werden überprüfbare Eigenschaften definiert, die abschließend zur Prüfung des Produktes dienen.

Das Design beschreibt die Entwicklung von der Spezifikation zu einer Implementation des Produktes. Dieses setzt sich aus zwei Komponenten zusammen, der Hardware und die Software. Beide zusammen bilden das in der logischen Ansicht beschriebene System. Die Aufteilung des Produktes in Hardware und Software ist ein komplexes Thema, das im System Entwurf durchgeführt wird. Es wird in dieser Masterarbeit nicht weiter behandelt. Ein Beispiel für einen mögliche Partitionierungsvorgäng findet man in der Arbeit [LSG10].

Der nächste Schritt nach dem Design ist die Implementation der Hardware und Software. Sie setzt die Ideen, die im Design beschriebenen wurden, um. Dabei erfolgt die Implementation der Hardware und Software zeitgleich. Die Annotation kann zeitgleich zur Implementation erfolgen, allerdings wurde sie aus Gründen der Übersichtlichkeit erst nach der Implementation eingezeichnet. Bei der Annotation werden die Teile des Systems, die ein zeitkritisches Verhalten besitzen, markiert. Die Markierungen dienen dabei als Unterscheidungsmerkmal zwischen kritischen und nicht kritischen Bereichen. Die Integration fasst die Hardware und Software zu einem System zusammen.

Nach dem Abschluss der Integration, werden die Tests auf das implementierte System angewendet. Dies ermöglicht eine Verifikation des Systems. Findet man bei der System Verifikation heraus, das es ein funktionales Problem gibt, so kann dies im schlimmsten Fall zum Redesign des Systems führen.

Natürlich bedeutet das nicht, dass jeder funktionale Fehler gleich ein erneutes verändern des Designs eines Systems zu Folge hat. Es kann auch ausreichen, dass eine Funktion des Systems erneut implementiert wird. Falls dies nicht reicht, kann es auch zu einem erneuten Designvorgang der Funktion kommen. Stellt man ein Timingproblem fest, so ist die Lösung ähnlich wie beim funktionalen Problem. Es wird versucht die Teile die schlechtes Zeitverhalten aufweisen, zu korrigieren. Ist dies nicht möglich, so kann dies zu einem erneuten Designvorgang führen, bei dem ein Teil oder das gesamte Systems verändert wird.

Nach der erfolgreichen Überprüfung des Systems, werden die Annotationen entfernt.

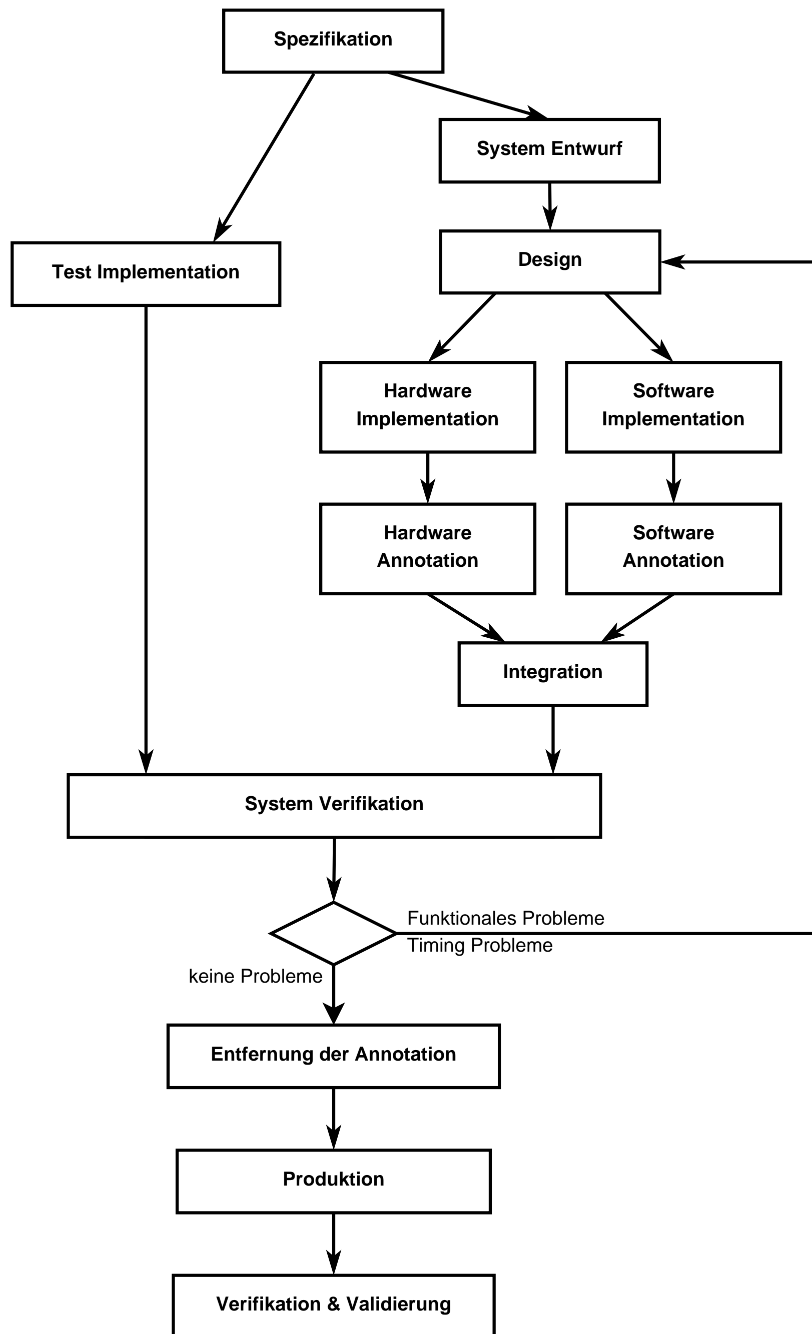

Abbildung 3.4: Prozessansicht

Dies ist der letzte Schritt in der Entwicklung des Systems. Der Grund dafür ist, dass man im fertigen Produkt keine Annotationen benötigt, da diese nur als Endwicklungshilfe dienten. Falls sie im fertigen Produkt verbleiben, so besteht die Gefahr, dass dadurch Probleme auftreten, die es sonst nicht gäbe. Zum Beispiel könnte sie ein Angriffspunkt für eine Attacke sein oder die Zeitmessfunktion könnte durch einen Fehler in der Kommunikation aktiviert werden, und dadurch ständig die Laufzeit messen. Dadurch würde dann der freie EEPROM Speicherplatz des Smartcard Systems sehr schnell voll werden. Probleme dieser Art sind natürlich nicht erwüscht.

Danach wird die Produktion des Systems eingeleitet. Dabei wird eine kleine Charge des Produkts hergestellt. Der Grund dafür ist, dass bei der ersten Fertigung des Systems Probleme auftreten können, die zuvor nicht bemerkt wurden. Um sicherzustellen dass das System korrekt funktioniert, findet eine abschließende Überprüfung des Produktes statt. Auf diese Weise kann sichergestellt dass es keine Probleme bei der Fertigung gibt.

Diese Überprüfung ist die Validierung und Verifizierung. Dabei wird überprüft ob das Produkt dem spezifizierten Verhalten entspricht. Dies ist die letzte Möglichkeit eventuelle Fehler zu finden, bevor das Produkt vertrieben wird.

3.3 Entwicklungsansicht

Die Entwicklungsansicht wird in Abbildung 3.5 dargestellt und beschreibt die Interaktion aller beteiligten Akteure. Dadurch wird gezeigt, wer welche Tätigkeit auszuführen hat.

Die Entwicklungsansicht beginnt mit dem Kunden, der seine Vorstellungen des Produktes in Form der Spezifikation definiert. Diese wird dann an den System-Entwickler und den Test-Entwickler ausgegeben. Der System-Entwickler entwirft die Hardware und Software Komponenten. Das Resultat ist dann das Hardware/Software-System. Der Test-Entwickler nützt die Spezifikation zum Erzeugen der Testfälle. Diese beinhalten Software und Hardware Testfälle.

Die Testfälle und das Hardware/Software-System werden an den Test-Ingenieur übergeben. Dieser führt die Testfälle auf dem Hardware/Software-System aus. Der Test-Ingenieur kann Teile oder das gesamte System auf seine Richtigkeit hin untersuchen. Damit ist gemeint, dass die Möglichkeit besteht, dass die Hardware getrennt von der Software überprüft werden kann. Das Ergebnis besteht aus zwei unterschiedlichen Bereichen. Der erste Bereich beschäftigt sich mit den funktionalen Ergebnissen und der zweite Bereich beschäftigt sich mit den Laufzeit Ergebnissen.

Die funktionalen Ergebnisse sind dabei der Output der von den Testfällen generiert wird. Dieser wird bei der Evaluierung ausgewertet. Dabei wird das Ergebnis mit den Vorgaben der Spezifikation verglichen. Wird hierbei ein Fehler gefunden, so wird die Information, um welchen Fehler es sich handelt, an den System-Entwickler weitergegeben. Dieser untersucht die vermeintlich fehlerhafte Komponente und versucht anschließend die korrekte Funktionsweise wieder herzustellen.

Die Timing Ergebnisse hingegen sind die Ausführzeit, die die einzelnen Bereiche benötigen. Diese werden bei der Evaluierung auf Übereinstimmung mit den Vorgaben überprüft. Wird hierbei ein Fehler festgestellt, so wird dieser ebenfalls an den System-Entwickler weitergeleitet. Dieser überprüft dann, ob die Funktion auch ein funktionales Problem aufweist. Ist dies der Fall wird zuerst der funktionale Fehler beseitigt und im Anschluss findet

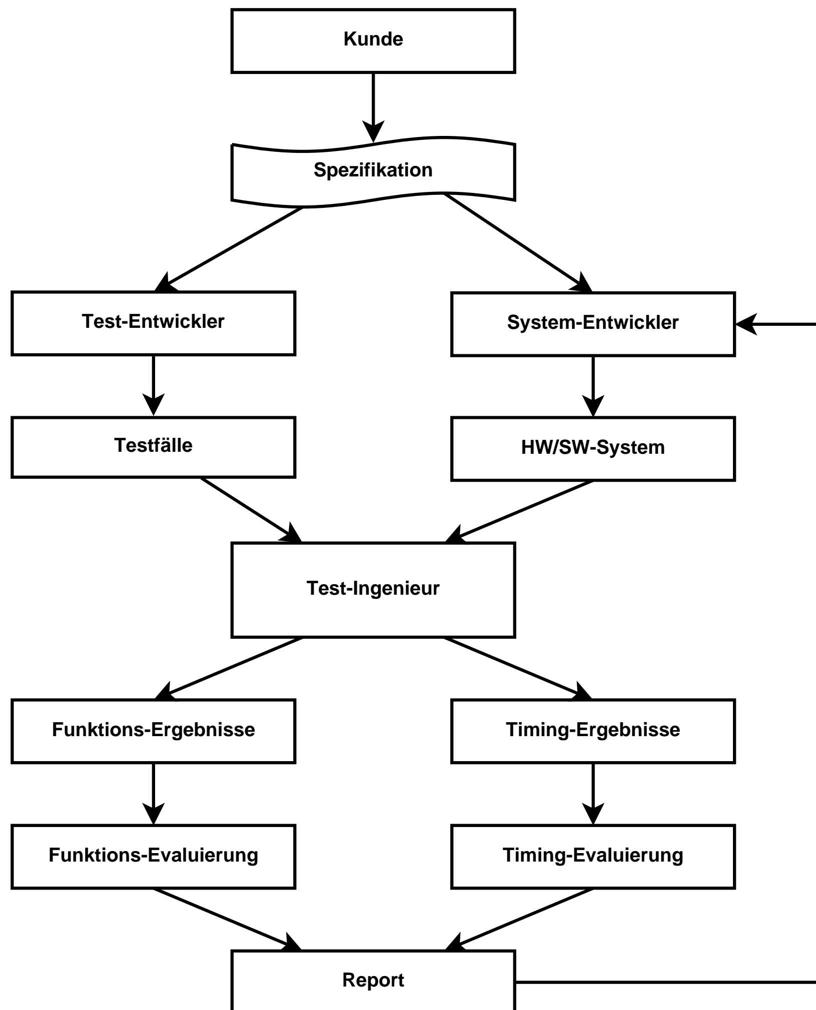

Abbildung 3.5: Entwicklungsansicht

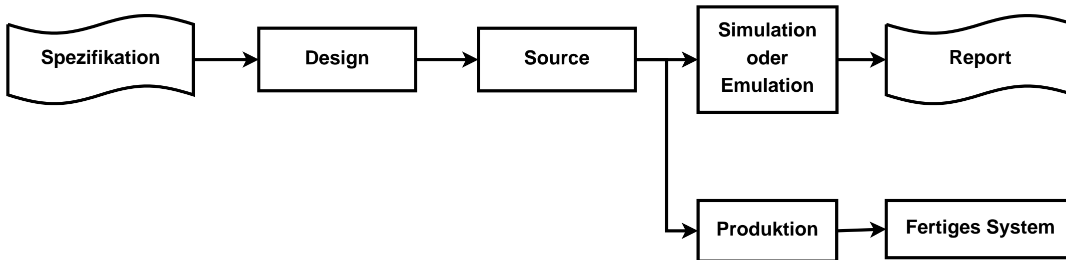

Abbildung 3.6: Physikalische Ansicht

eine erneute Überprüfung statt.

3.4 Physikalische Ansicht

Die Abbildung 3.6 stellt die physikalische Ansicht dar. Sie verdeutlicht die einzelnen Arbeitsschritte von der Definition des Produkts bis hin zum fertigen Produkt. Ihr Ausgangspunkt ist die Spezifikation, die ganz links in der Abbildung zu sehen ist.

Aus der Spezifikation entsteht das Design. Diese ist ein Dokument, welches das Produkt detailliert beschreibt. Diese Beschreibung erfüllt exakt die Spezifikation und dient als Grundlage für die Entwicklung des Produkts.

Das Design wird verwendet um daraus den Quelltext des Produktes zu erstellen. Der Quelltext ist die Grundlage für das Hardware und Software System, sowie für die Testfälle, die für die Überprüfung eingesetzt werden.

Danach gibt es zwei mögliche Entwicklungsschritte. Der erste ist der, der die Entwicklung des Produkts beschreibt. Das System wird dabei getestet und auf seine korrekte funktionsweise hin untersucht. Dafür muss der Quelltext auf einem Simulator oder Emulator ausgeführt werden. Dieser generiert als Ergebnis den Bericht, der zeigt ob ein Fehler bei der die Überprüfung gefunden wurde oder nicht.

Wurde die Prüfung erfolgreich absolviert, dass heißt es sind alle Funktionen, wie in der Spezifikation beschrieben, umgesetzt, so kann der zweite Schritt ausgeführt werden. Der Quelltext wird an die Produktion übergeben. Dadurch wird die Produktion des fertigen Systems eingeleitet.

Das fertige System wird natürlich ebenfalls auf seine Funktionstüchtigkeit überprüft. Dies ist allerdings nicht mehr Teil des Hardware/Software Codesign Flow, da dieser bereits abgeschlossen ist. Die Überprüfung findet daher im Zuge der Qualitätssicherung statt.

Kapitel 4

Implementierung

Dieses Kapitel beschäftigt sich mit der praktischen Umsetzung des Hardware/Software Codesign Flows. Dabei werden im ersten Unterkapitel die Vorgaben für den in der Masterarbeit implementierten Hardware/Software Codesign Flow beschrieben. Es geht dabei um die Smartcard-Hardware, das Smartcard-Betriebsystem sowie die Kommunikation zwischen der Smartcard und dem Smartcard-Reader. Die Abbildung 4.1 zeigt dabei den Aufbau eines Smartcard Systems. Sie dient als Überblick für die Aufteilung der Vorgaben und wird im Unterkapitel 4.1 erläutert.

Das Unterkapitel 4.2 beschäftigt sich mit der Zeitmessung und der Implementierung der Zeitmesseinheit. Dabei wird die Funktionsweise der Zeitmesseinheit beschrieben, wie diese implementiert wurde und wie sie in den Hardware/Software Codesign Flow integriert ist. Im nächsten Unterkapitel wird die Übertragungssoftware beschrieben. Diese setzt sich aus einer On-Card Anwendung und einer Off-Card Anwendung zusammen. Ist eine Anwendung eine On-Card Anwendung, so bedeutet dies, dass die Abarbeitung der Anwendung auf der Smartcard erfolgt.

Im Unterkapitel 4.4 wird die Auswertesoftware beschrieben. Diese liest die Datei ein und analysiert die gemessenen Daten. Wird dabei eine Auffälligkeit im Zeitverhalten festgestellt, so wird dies im Auswertungsbericht festgehalten. Der Aufbau des Auswertungsberichts wird in diesem Abschnitt ebenfalls behandelt.

4.1 Anforderungen

Für die Implementation des Hardware/Software Codesign Flow wurde die Entwicklung eines Smartcard Systems gewählt. Die Zeitmessung erfolgt bei diesem System nicht auf Modellebene, da dafür ein zu genaues zeitliches Verhalten simuliert werden müsste. Stattdessen wurde eine Lösung gewählt, die aus einer Kombination von Hardware und Software besteht. Diese kann, ohne Anpassungen an der Zeitmessung, auf dem Modell oder einem Emulator arbeiten. Für die Simulation muss allerdings wieder die Simulationsgenauigkeit des Modells erhöht werden.

Das Smartcard System besitzt noch zahlreiche weitere Vorgaben, die für die Implementation einzuhalten sind. Dieses Unterkapitel erläutert die Vorgaben und teilt sich in vier Abschnitte auf. Der erste Abschnitt beschreibt die Hardware der Smartcard. Im zweiten Abschnitt wird das Modell der Smartcard Hardware beschrieben. Der dritte Abschnitt

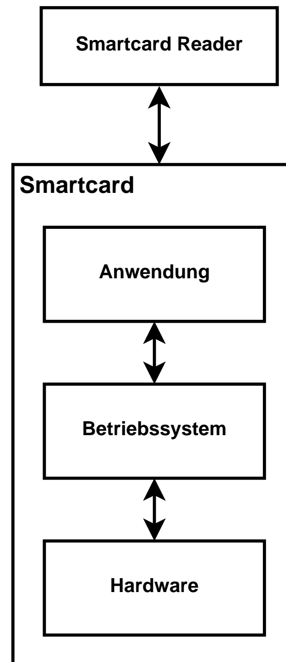

Abbildung 4.1: Schichten eines Smartcard Systems

beschreibt das Betriebssystem, dass auf der Smartcard eingesetzt wird. Der vierte und letzte Abschnitt erläutert Überlegungen, die auf Grund der Vorgaben gemacht wurden.

4.1.1 Smartcard Hardware

Das Smartcard System verwendet eine asynchrone Chipkartenarchitektur. Diese Architektur wird im Kapitel 2 beschrieben. Sie setzt einen Mikroprozessor oder zumindest einen Mikrokontroller voraus. Auf diesem wird das Smartcard Betriebssystem abgearbeitet. Dadurch ist die asynchrone Chipkartenarchitektur in der Lage komplexere Aufgaben zu bewältigen als eine synchrone Chipkartenarchitektur, die nur die Aufgabe hat, Daten zu speichern und zu laden.

Für die Implementierung des Hardware/Software Codesign Flows zum Entwickeln von Smartcard Systemen wurde ein Mikroprozessor gewählt, der zu einem 8051 kompatibel ist. Das bedeutet, dass der Mikroprozessor den Befehlssatz des 8051 beherrscht. Dies ermöglicht den Einsatz eines bereits existierendes Smartcard Betriebssystem. Dieses setzt die Funktionalität des 8051 voraus. Der Vorteil beim Verwenden eines existierenden Smartcard Betriebssystems ist, dass viele Standardkomponenten, wie zum Beispiel das Kommunikationsprotokoll oder die Speicherverwaltung, bereits implementiert sind. Dadurch reduziert sich der Aufwand für die Implementation des Hardware/Software Codesign Flow zum Vorbeugen von Timing Attacken. Die Vorgabe des 8051 kompatiblen Mikroprozessors betrifft nur diese konkrete Implementation des Hardware Software Codesign Flows. Alternative Implementierungen können auch auf anderen Prozessoren oder Betriebssystemen beruhen, allerdings muss dafür die Zeitmessung und die Software zum

Übertragen der Messdaten angepasst werden.

Der Mikrokontroller 8051

Der 8051 ist ein 8 Bit Mikrokontroller. Dieser wurde 1980 entwickelt und ist seit dem einer der beliebtesten Mikrokontroller. Durch seine vielseitigen Einsatzmöglichkeiten wurde er quasi zu einem Industriestandard. Die Abbildung 4.2 zeigt den Aufbau der 8051 Architektur. Die Basisvariante des 8051 verfügt über internes RAM mit einer minimalen Speicherkapazität von 128 Byte. Durch externe RAM- und ROM-Speichermodule ist eine Erweiterung des verfügbaren Speicherplatzes möglich. Diese werden nach einem einheitlichen Schema adressiert, wobei der interne RAM-Speicherplatz schneller adressiert werden kann als der Externe. Auf Grund der geringen Kapazität und der schnelleren Adressierung sollte man beim Entwickeln einer Anwendung mit dem internen RAM sehr sparsam umgehen, um eine optimale Rechenleistung zu erhalten.

Der 8051 verfügt über 4 Ports zum Anbinden von Peripherien. Die Peripherien können dabei sehr vielseitig ausfallen, angefangen von Speichererweiterungen, über Bussysteme und Schnittstellen, bis hin zu Coprozessoren ist alles möglich. Man muss dabei nur beachten, dass die angeschlossene Peripherie nicht zu schnell für den 8051 ist, denn dieser benötigt zum Ausführen eines Maschinenbefehls zwischen 1 und 3 Maschinenzyklen.

Ein Maschinenzyklus benötigt zwölf Taktzyklen. Das bedeutet, wenn der Mikrokontroller mit einem Takt von 6 MHz betrieben wird, dann benötigt der schnellste Befehl 0,5 μ Sekunden und der langsamste 1,5 μ Sekunden. Der Mikrokontroller kann beim Abarbeiten eines Befehls nur durch einen Reset unterbrochen werden. Beim Auftritt eines Interrupts wird der gerade ausgeführte Befehl abgeschlossen. Danach reagiert der Mikrokontroller erst auf das Interrupt. Der 8051 verfügt über mindestens fünf Interrupt Quellen die jeweils zwei unterschiedliche Prioritäten aufweisen können. Interrupt Quellen höherer Priorität haben dabei immer Vorrang. Eine Interrupt Quelle ist zum Beispiel einer der beiden 16-Bit Hardwaretimer des Mikrokontrollers.

Hardwaretimer und Interrupts des 8051

Für den Hardware/Software Codesign Flow zum Vorbeugen von Timing Attacken ist es wichtig, dass eine Bestimmung der Zeit möglich ist. Der Hardwaretimer des 8051 stellt eine derartige Möglichkeit dar.

Der Hardwaretimer ist ein in Hardware ausgeführter Aufwärtszähler, der unabhängig von der restlichen Hardware arbeitet. Das bedeutet, so lange der Timer aktiviert ist, zählt er konsequent die Taktzyklen und der Mikrokontroller kann parallel dazu beliebige Befehle abarbeiten. Der Hardwaretimer zählt dabei so lange aufwärts, bis er gestoppt wird oder bis ein Overflow auftritt. Als Overflow wird ein Überlaufen des Timer bezeichnet. Dies bedeutet, dass der Timer wieder von Null zu zählen beginnt. Weiters wird auch ein Interrupt ausgelöst, dass dem auf dem Mikrokontroller arbeitenden Programm beziehungsweise Betriebssystem mitteilt, dass ein Überlauf des Timers stattgefunden hat.

Der Hardwaretimer besitzt zwei unterschiedliche Betriebsmodi, den Betriebmodus Timer und den Betriebsmodus Counter. Der Unterscheid zwischen den beiden ist, dass beim Timer ein interner Takt gezählt wird und beim Counter werden externe Ereignisse gezählt beziehungsweise es kann auch ein externer Takt gezählt werden. In welchem Betriebsmodus der Hardwaretimer sich befindet, hängt von der Konfiguration ab.

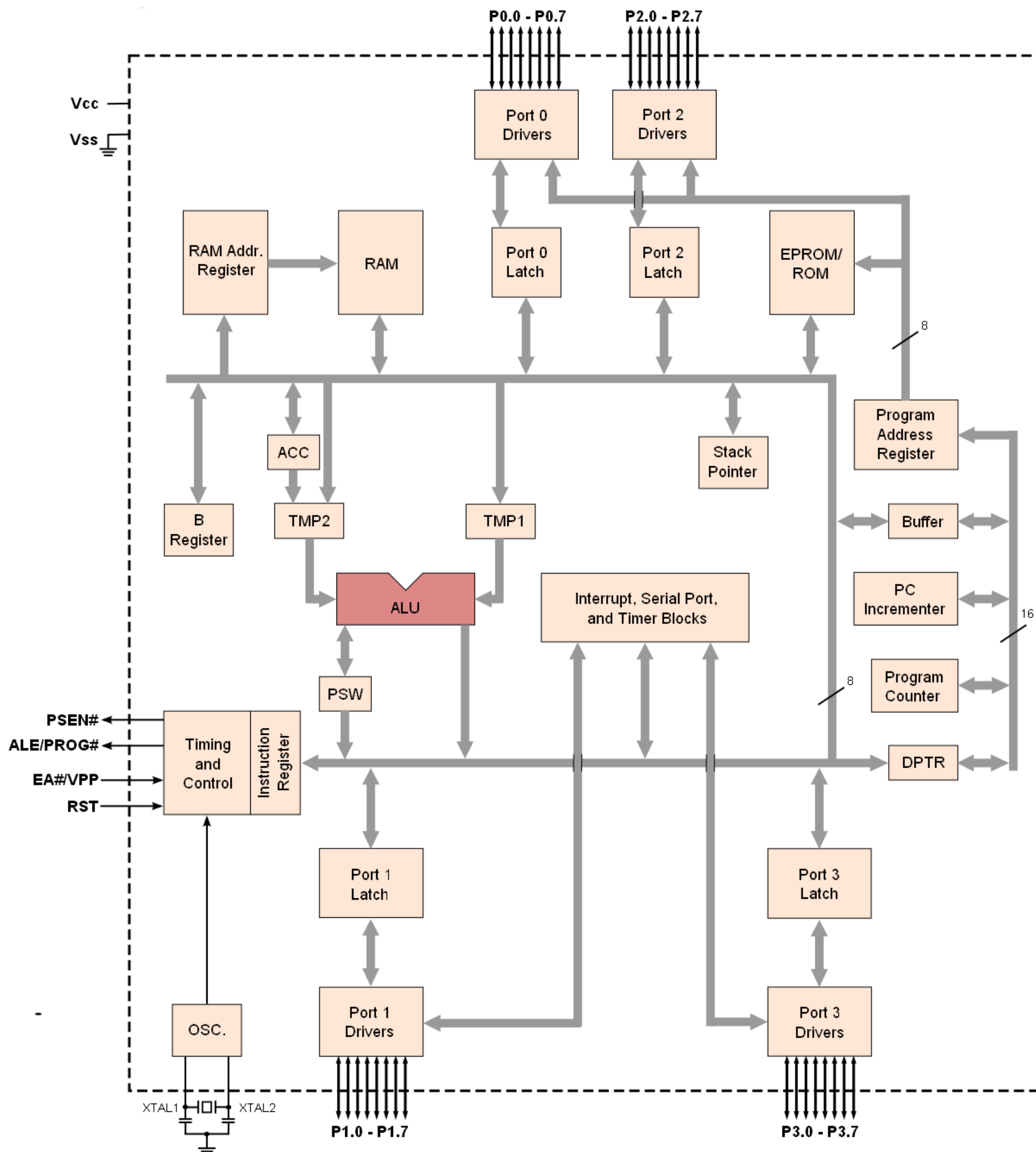

Abbildung 4.2: Architektur des 8051 Mikrokontrollers
[Wik11]

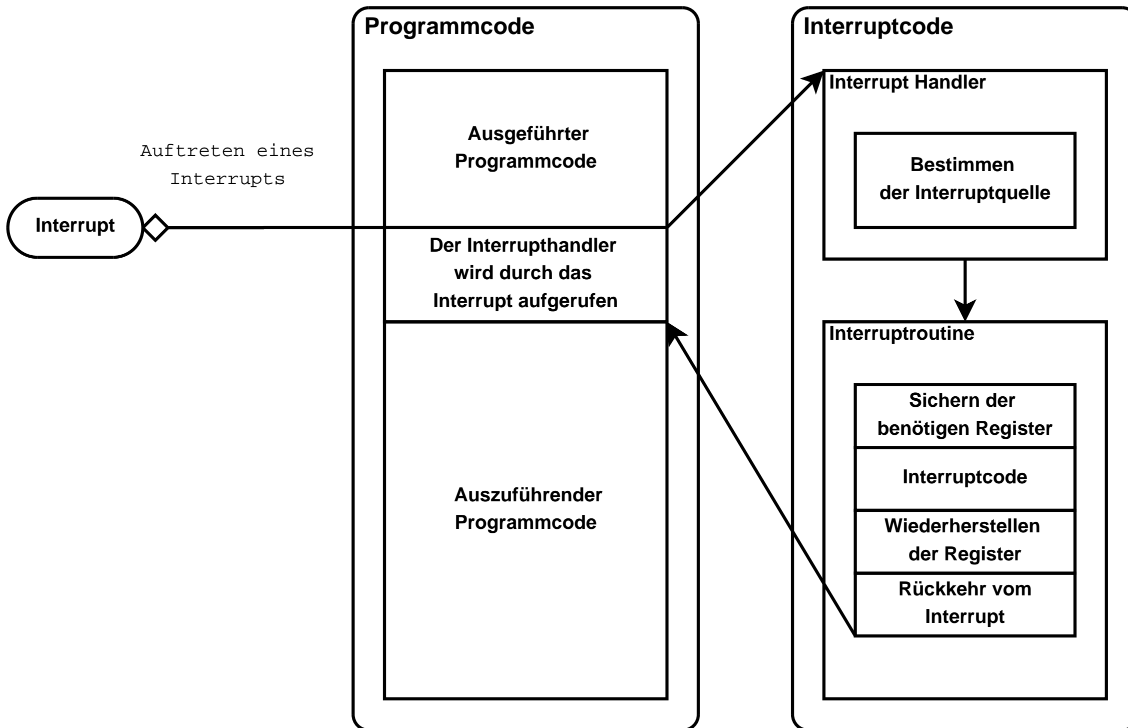

Abbildung 4.3: Auftreten und abarbeiten eines Interrupts

Ein Interrupt ist eine kurze Unterbrechung des auszuführenden Programmcodes. Es wird in Abbildung 4.3 dargestellt. Der Interrupthandler sorgt dafür, dass die richtige Interruptroutine ausgeführt wird. Die Funktion der Interruptroutine wird dabei vom auszuführenden Programmcode definiert. Im Fall eines Interrupts, der durch einen Timer ausgelöst wird, könnte der Programmcode der Interruptroutine zum Beispiel eine Wartefunktion oder eine Erweiterung des Timers implementieren. Natürlich sind auch komplexere Aufgaben denkbar, aber man darf dabei nicht vergessen, dass der Zustand der CPU Register vor und nach dem Interrupt ident sein muss. Daher muss jedes verwendete Register gesichert werden. Das ist natürlich mit zeitlichem Aufwand verbunden und benötigt zudem noch Speicherplatz. Nach Abarbeitung der Interruptroutine wird das Programm an der Stelle wieder aufgenommen, an der es vom Interrupt unterbrochen wurde.

Moderne 8051 Mikrokontroller

Moderne Mikrokontroller die zu einem 8051 kompatibel sind, weisen die hier beschriebenen Eigenschaften auf. Sie verfügen aber auch über zahlreiche Verbesserungen. Angefangen von einer höheren Taktung, über mehr RAM und ROM Speicher bis hin zu neuen Befehlen, die den Mikrokontroller für bestimmte Anwendungen optimieren. Weiters kommen auch Speicherkontroller zum Einsatz, um größere Datenmengen schneller und einfacher verarbeiten zu können.

4.1.2 Modell der Smartcard Hardware

Das Modell der Smartcard Hardware ist ein existierendes SystemC Modell. Es lässt sich nach Belieben erweitern und verfügt über dieselben Eigenschaften wie die Smartcard Hardware. Der Vorteil des Modells liegt in der Tatsache, dass sich Änderungen an der Hardware schnell und einfach umsetzen lassen. Dadurch ist es möglich verschiedene Hardware und Software Partitionierung zu simulieren, ohne dass die Kosten und der Zeitverlust durch die Fertigung einfließen. Weiters verfügt es über eine Kommunikationsschnittstelle, die die Anbindung eines ebenfalls existierenden, virtuellen Smartcard Readers erlaubt. Dadurch ist eine Anwendung des Smartcard Betriebsystems ohne Anpassung des Modells möglich. Der einzige Unterschied zwischen der hardware- und modellbasierenden Implementierung ist, die benötigte Zeit für das Abarbeiten der Anwendungen. Hierbei ist das Modell langsamer als die hardwarebasierende Lösung.

4.1.3 Smartcard Betriebssystem

Das Betriebssystem einer Smartcard verbindet die Hardware mit der darüber liegenden Anwendung. Dies wird in Abbildung 4.4 dargestellt. Das Smartcard Betriebssystem stellt die Treiber zur Verfügung, mit denen die einzelnen Funktionseinheiten der Hardware angesprochen werden. Dies könnte zum Beispiel ein AES-Krypto-Coprozessor oder die Kommunikationseinheit sein.

Eine grundsätzliche Vorraussetzung für das Smartcard Betriebssystem ist ein 8051 kompatibler Mikrokontroller oder Mikroprozessor. Der Grund dafür ist, dass ein Teil des Betriebssystems in Assembler geschrieben wurde. Dies bietet die Möglichkeit zeitkritische Funktionen per Hand zu optimieren und Hardwarekomponenten des Mikrokontrollers direkt anzusprechen. Ein Beispiel für eine zeitkritische Funktion wäre das Timing der Kommunikationseinheit. Benötigt diese zu viel oder zu wenig Zeit, so schlägt die Kommunikation fehl, dies kann dazu führen, dass ein Hardware Reset der Smartcard ausgelöst wird. Ein Nachteil der Nutzung von Assemblercode ist, dass man sich näher an die Hardware bindet. Wird der Mikrokontroller durch einen nicht kompatiblen Typ ausgetauscht, so muss der Assemblercode porträtiert beziehungsweise neu geschrieben werden.

Das zur Verfügung stellen der Treiber ist aber nicht die einzige Funktion des Betriebssystems, es sorgt auch für die Speicherverwaltung, implementiert eine Programmierschnittstelle (API) und noch vieles mehr. Da sich die Hardwareausstattung einer Smartcard im Betrieb nicht mehr ändert, müssen auch keine Treiber ausgetauscht werden. Daher ist das Betriebssystem einer Smartcard vergleichbar mit einer Virtuellen Maschine, die direkt auf der Hardware arbeitet. Die API dient zum Implementieren einer Anwendung. Die Anwendung stellt die gewünschte Funktionalität des Smartcard Systems her. Die Anwendung ruft dafür die einzelnen API Befehle auf, die dann vom Betriebssystem der Smartcard abgearbeitet werden. Dadurch ist es möglich, Anwendungen zu entwerfen, die unabhängig von der verwendeten Hardware sind. Der Vorteil daran ist, dass die Smartcard Hardware beim Entwurf einer Anwendung noch nicht bekannt sein muss.

Die Smartcard Hardware kann während des Entwicklungsprozesses näher definiert werden. Dabei wird entschieden welche Geschwindigkeit die Smartcard Hardware haben muss, wie groß der benötigte Speicher sein muss, welche kryptografischen Funktionen unterstützt werden, welche Schnittstellen unterstützt werden und noch vieles mehr. Dadurch ist es

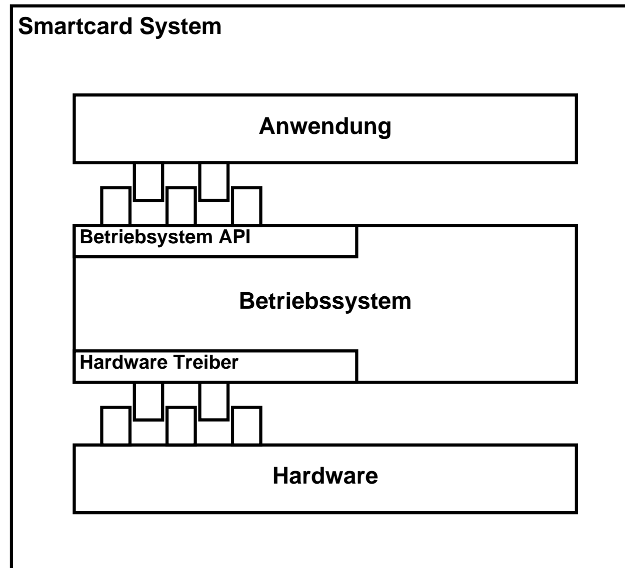

Abbildung 4.4: Aufbau des Smartcardsystems

möglich eine Hardware zu verwenden, die auf eine ganz spezifische Anwendung zugeschnitten ist.

Speichermanagement des Smartcard Betriebssystems

Das Speichermanagement ist eine zentrale Aufgabe des Smartcard Betriebssystems. Es stellt den exklusiven Zugriff auf eine physikalische Speicherzelle beziehungsweise einen physikalischen Speicherblock sicher. Es spielt auch eine zentrale Rolle in der Implementation der Zeitmesseinheit, da jedes benötigte Byte vom Speichermanagement berücksichtigt wird. Beispiele in denen die Zeitmesseinheit auf Speicherblöcke zugreift, sind die Timererweiterung oder der Speichervorgang der gemessenen Laufzeiten. Ein genauere Erklärung finde im Abschnitt 4.2.1 statt, in der der Aufbau der Zeitmesseinheit beschrieben wird. Die Abbildung 4.5 zeigt den Aufbau der Speicherverwaltung.

Sie nutzt eine Adresstabelle um den belegten und den freien Speicherbereich zu verwalten. Die physikalische Adresse, über die die Daten direkt angesprochen werden, wird dabei durch eine virtuelle Adresse ersetzt. Eine virtuelle Adresse entspricht dabei nicht der physikalischen Adresse. Der Grund dafür ist, dass sichergestellt werden soll, dass niemand direkt auf eine Speicherzelle oder einen Speicherblock zugreift, ohne dass die Speicherverwaltung involviert ist.

Für Speicherverwaltung sind mehrere Befehle des Betriebssystems als Systemcall implementiert. Diese ermöglichen das Lesen und Schreiben der einzelnen Speicherzellen oder Speicherblöcke. Die Systemcalls sind in allen Ebenen des Betriebssystems implementiert, wodurch eine lückenlose Nutzung der Speicherverwaltung möglich ist. Wird ein Systemcall aufgerufen, so wird überprüft ob die virtuelle Adresse gültig ist. Diese bestimmt, an welcher Stelle, in der Adresstabelle, sich die Informationen über den Datenblock befinden. Diese Informationen beinhalten die physikalische Adresse des Speicherblocks, sowie seine

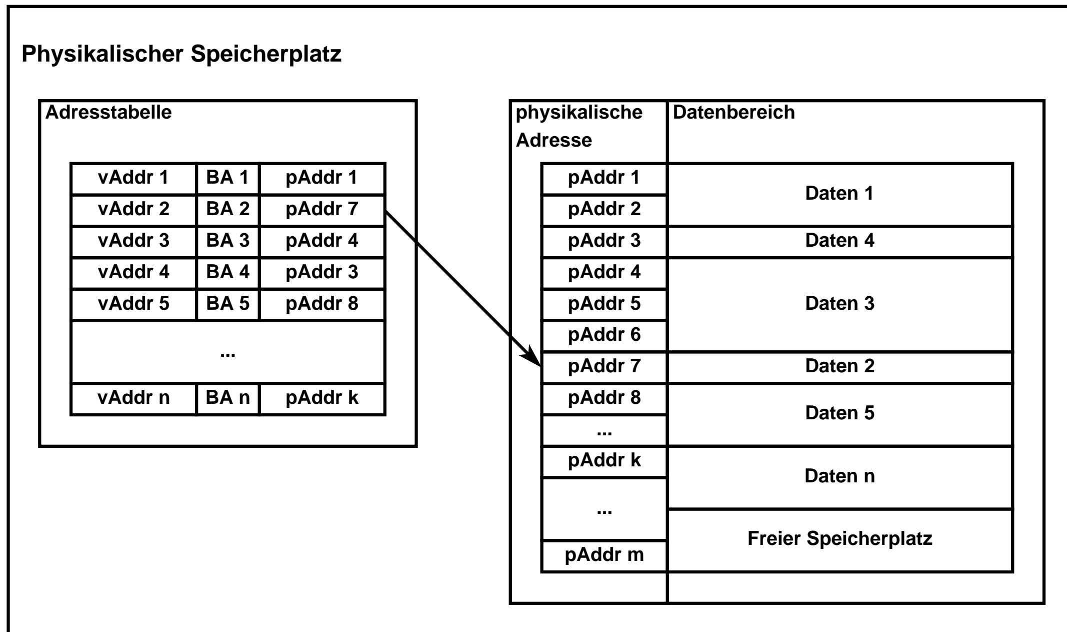

Abbildung 4.5: Aufbau der Speicherverwaltung des Smartcard Betriebssystems

Attribute. Die Attribute des Speicherblocks beinhalten Informationen wie zum Beispiel welcher Anwendung ein Speicherblock zugewiesen wurde, wie groß der Speicherblock ist und so weiter. Diese Attribute werden natürlich ebenfalls überprüft. Dadurch wird sichergestellt, dass man nicht über den zugewiesenen Speicherbereich hinaus schreibt oder gar Daten, die der Anwendung nicht zugewiesen wurden, überschreibt. Sind alle angegebenen Parameter korrekt, so greift der Systemcall auf die physikalischen Speicher zu und führt die gewünschte Operation aus.

Systemvoraussetzungen durch das Smartcard Betriebssystem

Beim eingesetzten Betriebssystem handelt es sich um ein proprietäres Smartcard Betriebssystem. Es benötigt einen gewissen Speicherplatz im RAM, ROM und EEPROM um seine Funktionalität zu erfüllen. Daher muss beim Designen und Implementieren eines Smartcard Systems darauf geachtet werden, dass ausreichend Speicherplatz zur Verfügung steht, um die gewünschte Anwendung implementieren zu können. Weiters verfügt der Mikrokontroller über einen vom Betriebssystem nicht genutzten Hardwaretimer. Dieser darf von keinem Entwickler außerhalb des Betriebssystems verwendet werden, da er ein Interrupt auslösen kann und dadurch zu einem Sicherheitsrisiko wird.

Das Betriebssystem unterstützt mehrere Kommunikationsschnittstellen. Diese sind aber nicht immer alle in Hardware ausgeführt. Je nach Anwendung muss der Entwickler entscheiden, ob er eine oder mehrere Kommunikationsschnittstellen benötigt. Er muss entscheiden ob die zu entwickelnde Anwendung eine kontaktbehaftete, eine kontaktlose

oder Kombination aus beiden Kommunikationsschnittstellen benötigt. Ein typisches Beispiel für eine Anwendung mit einer kontaktbehafteten Kommunikationsschnittstelle wäre eine Bankomatkarte. Ein Beispiel mit kontaktloser Kommunikationsschnittstelle wäre ein elektronisches Schlüsselsystem. Die Entscheidung welche der üblichen Kommunikationsschnittstelle im Smartcard System eingesetzt wird, ist keine Frage des Betriebssystems, vielmehr ist es eine Frage des Designs der Hardware.

4.1.4 Grundlegende Überlegungen zur Implementation aufgrund der Anforderungen

Für die Implementation des Hardware/Software Codesign Flows wurde die Entwicklung eines Smartcard System gewählt. Die Anwendung eines Smartcard Systems wird in Abbildung 4.1 gezeigt. Dabei werden die unterschiedlichen Abstraktionsschichten des Systems dargestellt.

Für die Implementation des Hardware/Software Codesign Flows zum Vorbeugen von Timing Attacken ist es wichtig, dass man die Laufzeit von Funktionen und Codeabschnitten messen kann. Der Grund für die Laufzeitmessung ist, das Erkennen von signifikantem Laufzeitverhalten. Das bedeutet, man misst die einzelnen Laufzeiten und versucht im Anschluss einen Zusammenhang zwischen den gemessenen Laufzeiten herzustellen.

Die folgenden drei Abschnitte beschreiben die zu Grunde liegenden Ideen zur Umsetzung des im Hardware/Software Codesign Flows. Dabei wird im ersten Abschnitt die Erweiterungsfähigkeit der Smartcard Hardware sowie des darauf arbeitenden Betriebssystems behandelt. Der zweite Abschnitt behandelt die Zeitmessung und im dritten Abschnitt wird der Datentransfer zur Laufzeitanalyse behandelt.

Erweiterungsfähigkeit des Smartcard System

Das zu implementierende Smartcard System muss einen 8051 kompatiblen Kern besitzen. Die weiteren Funktionalitäten der Smartcard Hardware könnten variiert werden. Für diesen 8051 kompatiblen Kern des Systems existiert bereits ein Software Model. Dieses lässt sich um zusätzliche Hardware Funktionen erweitern.

Die Erweiterung der Hardware um eine zusätzliche Komponente müssen auch im Smartcard Betriebssystem berücksichtigt werden. Daher ist es notwenig, eine Anpassung der Treiber durchzuführen. Der Aufwand für die Anpassung hängt davon ab, ob die Hardwarekomponente bereits vom Betriebssystem unterstützt wird oder ob es sich um eine neue Hardwarekomponente handelt. Nach der Anpassung ist das Betriebssystem in der Lage, die entsprechende Hardwarekomponente anzusprechen. Soll eine bestehende Funktion des Betriebssystems die neue Komponente verwenden, so muss lediglich der Treiberaufruf im Programmcode des Betriebssystems, an der entsprechenden Stelle, eingefügt werden. Wird eine neue Funktion des Betriebssystems integriert, so muss die bestehende API des Systems um einen Befehl erweitert werden. Der API-Befehl implementiert die entsprechende Funktion, in dem er zum Beispiel eine spezielle Hardwarekomponente nutzt. Dieser Zusammenhang wird in Abbildung 4.4 dargestellt.

Daraus ergibt sich der Vorteil, dass ein neu entwickeltes Smartcard System schon in einem frühen Entwicklungsstadium funktionsfähig ist. Dies ermöglicht eine funktionale und zeitbasierende Überprüfung des Systems während der Entwicklung.

Ansätze zur Messung der Zeit

Die Zeitmessung kann auf unterschiedlichste Arten erfolgen, wobei die Zeitmessung vom Modell bis hin zur Produktion möglich sein soll. Daher ist eine rein Modell basierende Zeitmessung nicht zielführend. Weiters sollte die Zeitmesseinheit nicht mehrmals implementiert werden müssen. Das bedeutet, dass im Modell und auf der Hardware dasselbe System für die Laufzeitmessung im Einsatz ist. Der Grund dafür ist, dass unterschiedliche Implementierungen, dazu führen können, dass die gemessenen Laufzeiten von einander abweichen. Der ungünstigste Fall wäre, wenn sich nach der Auswertung der beiden gemessenen Datensätze herausstellt, dass ein Datensatz ein signifikantes Zeitverhalten aufweist und der andere nicht, denn dann würde sich die Frage stellen, welches der beiden Ergebnisse ist richtig.

Eine mögliche Implementation der Zeitmessung wäre eine hardwarebasierende Zeitmesseinheit. Diese muss die Möglichkeit bieten, dass mehrere Zeitmessungen gleichzeitig durchgeführt werden können. Die Ergebnisse der Messungen müssen auf der Smartcard gespeichert werden oder direkt über ein eigenes Interface extern zugänglich sein. Der Grund dafür ist, dass während der Abarbeitung eines Kommandos keine zusätzliche Kommunikation erlaubt ist. Diese Anforderungen machen eine rein in Hardware ausgeführte Zeitmesseinheit sehr komplex.

Würde man hingegen die Zeitmessung in einer Kombination aus Hardware und Software ausführen, so wäre die Komplexität wesentlich geringer. Dafür muss aber das Betriebssystem angepasst werden, um so den Speicherplatz (falls vorhanden) und das Kommunikationssystem des Smartcard Systems zu verwenden. Eine weitere Anpassung ist notwendig um die Hardware der Zeitmesseinheit ansprechen zu können.

Dieser Ansatz kann dadurch verbessert werden, dass die zur Verfügung stehende Hardware für die Zeitmessung verwendet wird. Dies wäre zum Beispiel durch Nutzung einer Timer- oder Counter-Funktion des Betriebssystems möglich. Der Vorteil daran wäre, dass keine zusätzliche Hardware in das System integriert werden muss.

Der letzte, der oben beschrieben Ansätze wurde für die Implementation der Zeitmessmessung im Hardware/Software Codesign Flows eingesetzt. Da aber keine Zeitmessfunktion im verwendeten Smartcard Betriebssystem vorhanden war, musste diese Funktion im Zuge dieser Masterarbeit implementiert werden. Dafür wurde ein nicht benötigter Timer der Hardware verwendet. Der Aufbau dieser Laufzeitmesseinheit wird im Abschnitt 4.2.1 beschrieben.

Übertragen der Messdaten

Die Implementation der Zeitmesseinheit besteht allerdings nicht nur aus der Implementation einer Funktion für die Laufzeitmessung. Es muss auch eine Möglichkeit geben, dass die auf der Smartcard gespeicherten Messdaten an einen Computer übertragen werden, der die Laufzeitergebnisse analysiert.

Wie im vorangegangen Abschnitt bereits erwähnt, ist es nicht gestattet während der Abarbeitung eines Kommandos eine response APDU zu versenden. Dies begründet sich durch das APDU-Protokoll, das besagt, dass nur am Ende eines Kommandos eine response APDU erfolgen darf. Daher werden die Messergebnisse auf der Smartcard gespeichert.

Weiters können Kommandos nur an die selektierte Anwendung der Smartcard gesendet werden. Daher muss die zu vermessende Anwendung entweder die Kommandos für den

Datentransfer implementieren oder es muss eine zweite Anwendung auf die Smartcard installiert werden, die den Datentransfer durchführt. Jeder der beiden Ansätze hat seine eigenen Vor- und Nachteile.

Der Vorteil des ersten Ansatzes ist, dass man nur eine Anwendung auf der Smartcard installieren muss. Die Messdaten können nach Abschluss jeder APDU ausgelesen werden. Daher wäre es zum Beispiel möglich, Zwischenberichte der Laufzeitmessungen zu erstellen. Der Nachteil daran ist, dass jede Smartcard Anwendung ihre eigenen Auslesekommandos implementieren muss. Das Generieren der Auslesekommandos könnte man zwar automatisieren, aber die dafür benötigten Kommandocodes wären auf jeden Fall belegt und müssten nach Abschluss der Tests entfernt werden.

Der Vorteil des zweiten Ansatzes ist, dass die zu vermessende Anwendung klar von der Anwendung zum Datentransfer getrennt ist. Dadurch lässt sich nach Abschluss aller Überprüfungen die Anwendung zum Datentransfer einfach entfernen. Weiters steht jeder beliebige Kommandocode der Anwendung zur Verfügung. Der Nachteil dieses Ansatzes ist, dass die Anwendung für den Datentransfer explizit zu selektieren ist und man keine Messdaten während der Überprüfung auslesen kann, die zum Beispiel für die Generierung eines Zwischenberichts genutzt werden.

Für die Umsetzung im Hardware/Software Codesign Flow wurde der zweite Ansatz gewählt. Dieser ist flexibel einsetzbar und verlangt keine zusätzlichen Änderungen der Smartcard Anwendung. Die Anwendung zum Auslesen der Messdaten muss nur einmal implementiert werden und kann für jede Weiterentwicklung auf dieser Smartcard Plattform eingesetzt werden.

Eine Off-Card Software steuert den Datentransfer, indem sie Kommandos für den Datentransfer an die Smartcard sendet. Die response APDU enthält die angeforderten Messdaten. Dieser Vorgang funktioniert auch im Modell, da ein Modell des Smartcard Readers existiert. Wurden alle Messdaten erfolgreich übertragen, wird die Kommunikation beendet.

Die Off-Card Software steuert nicht nur den Datentransfer der Messdaten, sie speichert diese auch in eine Datei. Als Dateiformat wurde das CSV Format gewählt. Dieses ermöglicht das Importieren der Messdaten in unterschiedliche Mathematikanwendungen. Dadurch ist eine schnelle Analyse der gemessen Daten möglich. Die finale Auswertesoftware für den Hardware/Software Codesign Flow wurde in Java implementiert. Sie setzt die Erkenntnisse, die aus den Mathematikanwendungen gewonnen wurden, um.

4.2 Die Zeitmessung

Eine zentrale Aufgabe dieses Hardware/Software Codesign Flows ist die Messung der Ausführzeit einzelner Funktionen und Funktionseinheiten. Es ist dabei wichtig, dass die Zeitmessung ohne Beeinflussung des Systems erfolgt. Das bedeutet, dass das zeitliche und funktionale Verhalten mit und ohne die Laufzeitmessung ident sein muss.

Eine funktionale Beeinflussung des Smartcard Systems durch die Zeitmesseinheit könnte dazu führen, dass eine Funktion ein nicht korrektes Verhalten aufweist. Dies könnte wiederum dazu führen, dass einer der Anwendungsfälle oder Testfälle, als Ergebnis, einen Fehler im System erkennt, der nicht existiert beziehungsweise der nur durch die Zeitmessung auftritt.

Beim Messen der Laufzeit ergibt sich ein ähnliches Problem. Die für die Messung der Laufzeit benötigte Zeit darf nicht in die Laufzeitmessung einfließen. Falls eine nicht konstante Laufzeit $t_{Laufzeit}$ gemessen wird, könnten die Zeitschwankungen möglicherweise durch die Berechnung der Laufzeit entstehen und nicht durch die zu vermessende Funktion selbst. Idealerweise benötigt das Durchführen der Zeitmessung keine Laufzeit.

Dies lässt sich aber in der Praxis nicht immer garantieren, daher ist es wichtig, dass das Starten und Stoppen der Laufzeitmessung immer eine konstante Zeit benötigt. Die daraus entstehenden Ungenauigkeiten lassen sich zum einen korrigieren und zum anderen beeinflussen sie nur die gesamte Laufzeitmessung, nicht aber die Zeitdifferenz. Der Grund dafür ist, dass bei der Berechnung der Differenz, die Zeitmessung einmal gestartet und einmal gestoppt werden muss. Die für das Starten der Messung benötigte Zeit wird durch t_{Start} und die für das Stoppen benötigte Zeit wird durch t_{Stopp} ausgedrückt.

Daraus ergibt sich für eine Laufzeitmessung $t_{Gemessen} = t_{Start} + t_{Laufzeit} + t_{Stopp}$. Für die Berechnung der Zeitdifferenz Δt werden zwei Zeitmessungen benötigt, die im Anschluss voneinander subtrahiert werden $\Delta t = t_{Gemessen_1} - t_{Gemessen_2}$. Setzt man nun für die gemessenen Zeiten $t_{Gemessen_x} = t_{Start_x} + t_{Laufzeit_x} + t_{Stopp_x}$ ein, so ergibt sich für die Zeitdifferenz $\Delta t = (t_{Start_1} + t_{Laufzeit_1} + t_{Stopp_1}) - (t_{Start_2} + t_{Laufzeit_2} + t_{Stopp_2})$. Da das Starten und das Stoppen des Messvorgangs immer gleich lange Dauert ($t_{Start_1} = t_{Start_2}$ und $t_{Stopp_1} = t_{Stopp_2}$), fällt er bei der Berechnung der Differenz weg. Wie man sieht ist das Ergebnis für die Differenz $\Delta t = t_{Laufzeit_1} - t_{Laufzeit_2}$.

Weiters muss das Ergebnis der Laufzeitmessung möglichst exakt sein, denn je ungenauer das gemessene Ergebnis ist, desto schwieriger, wenn nicht gar unmöglich ist es, durch die gemessenen Laufzeiten einen Aufschluss über das Laufzeitverhalten der Funktion zu erlangen. Hierbei empfiehlt es sich, die Genauigkeit so zu wählen, dass es möglich ist, zwei Operationen mit unterschiedlicher Laufzeit zu unterscheiden. Ist dies nicht der Fall, so weist die Analyse des Laufzeitverhaltens nur die Genauigkeit, des kleinst möglichen messbaren Laufzeitunterschiedes auf. Die Genauigkeit der Messergebnisse sind aber nicht das Einzige das für eine erfolgreiche Timing Analyse benötigt wird. Die Anzahl der Messergebnisse spielt dabei eine genauso große Rolle wie auch der gewählten Signifikanztest.

4.2.1 Aufbau der Messeinheit

Die Zeitmesseinheit nutzt, wie schon in der Einleitung des Kapitels erklärt, die vorhandene Hardware und das Betriebssystem der Smartcard zum Realisieren der Zeitmessung. Dafür wurde ein existierender 16 Bit Hardwaretimer verwendet, welcher im bereits implementierten Betriebssystem nicht verwendet wird.

Timererweiterung

Da ein 16 Bit Timer, der bei einem Takt von 4 MHz betrieben wird, bereits nach 98,304 Millisekunden überläuft, musste eine Timererweiterung verwendet werden. Die verwendete Erweiterung hat die Aufgabe, dass auch länger als 98,304 Millisekunden dauernde Funktionen und Anwendungen vermessen werden können.

Es stellte sich die Frage, wie groß sollte eine Timererweiterung sinnvollweise sein. Mehr als 2 Byte sind dabei nicht sinnvoll, da man mit 2 Byte und einem Takt von 4 MHz fast zwei Stunden messen könnte. Dies ist eine Zeitspanne, die von einer realen Anwendung nie

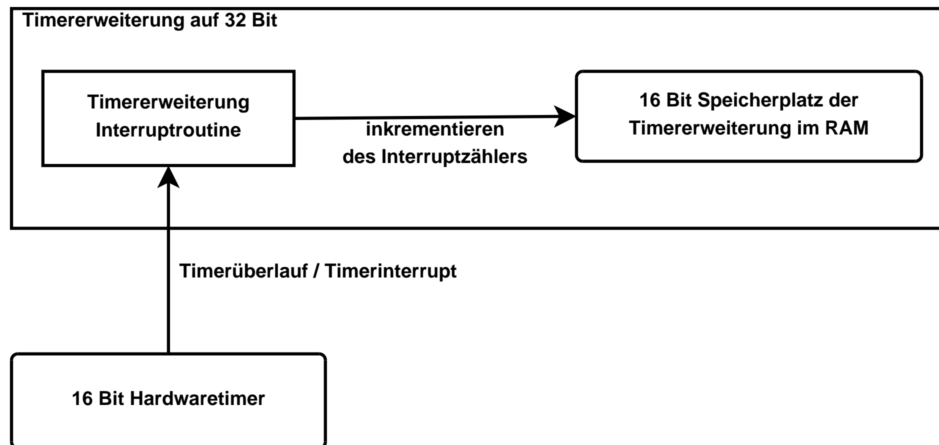

Abbildung 4.6: Erweiterung des 16 Bit Hardwaretimer auf 32 Bit

erreicht wird. Eine Nutzungsdauer über mehrere Minuten ist dabei unrealistisch. Würde man den Takt der Smartcard von 4 MHz auf fiktive 200 MHz anheben, so würde die Timererweiterung bereits nach etwas mehr als zwei Minuten überlaufen. Dies sollte aber immer noch ausreichen, um jede Smartcard Anwendungen zu vermessen.

Würde man hingegen nur 1 Byte für die Timererweiterung nutzen und der Takt wäre 4 MHz, so würde die Timererweiterung bereits nach etwas mehr als 25 Sekunden überlaufen. Dabei wäre dann an eine Erhöhung des Taktes nicht mehr zu denken, da bereits bei einer Verdopplung des Taktes die Messzeit halbiert wird. Das bedeutet, dass nur Anwendungen vermessen werden können, die nicht länger als 13 Sekunden benötigen. Dies könnte für manche Anwendungen problematisch werden, daher wurde eine Größe von 2 Byte für die Timererweiterung gewählt.

Die Timererweiterung nutzt die Interruptroutine, um den Timer von 16 Bit auf 32 Bit zu erweitern. Dabei werden die 2 Byte der Erweiterung an den bestehenden Hardwaretimer angehängt. Beim Aufruf der Interruptroutine werden die 2 Byte wie ein 16 Bit Wert betrachtet und inkrementiert. Der Aufruf der Interruptroutine wird in konstanter Zeit bewältigt. Da die Inkrementierung der 2 Byte ebenfalls in konstanter Zeit erfolgt, wird die gesamte Interruptroutine in konstanter Zeit abgearbeitet. Das hat den Vorteil, dass der Einfluss der Zeitmesseinheit berechenbar bleibt und somit eine Korrektur der Messergebnisse möglich ist. Die Abbildung 4.6 stellt den Erweiterungsvorgang des Timers grafisch dar.

Gleichzeitiges vermessen mehrerer Messpunkte

Zum gleichzeitigen Vermessen mehrerer zeitkritischer Bereiche eignet sich die so implementierte Zeitmesseinheit (Timer/Softwaretimer) allerdings noch nicht. Daher wird der 16 Bit Hardwaretimer und die Timererweiterung nicht direkt zum Messen der Laufzeiten eingesetzt. Das heißt der Timer wird nicht direkt bei jedem Anfang einer Messung gestartet und nach ihrem Abschluss gestoppt. Dies hätte zwar den Vorteil, dass das Messergebnis direkt in den Timerregistern und der Timererweiterung stehen und nicht gesondert berechnet

werden muss. Der Nachteil dieses Ansatzes ist, dass der 16 Bit Hardwaretimer immer nur für eine Messung zu einem Zeitpunkt verwendet werden kann. Das bedeutet, man müsste beim Entwerfen der Testfälle darauf achten, das keine Überschneidungen zwischen zwei kritischen Bereichen auftreten. Dies wäre zum einen sehr umständlich beim Markieren der kritischen Bereiche, da man darauf achten müsste wie sich die einzelnen Funktionen aufrufen und welche Parameter sie übergeben. Zum anderen könnte es vorkommen, dass kritische Bereiche nicht vermessen werden, da es Überschneidungen von Messbereichen kommt.

Durch die Verwendung des Timers zum Generieren der Zeitstempel ergibt sich dieses Problem nicht. Das lässt sich dadurch erklären, dass das Hohlen der Zeitstempel sehr schnell ist und der Timer somit nur kurzfristig zum Auslesen der aktuellen Zeit im System benötigt wird. Bis zum Hohlen des zweiten Zeitstempels kann der Timer somit problemlos von anderen Messpunkten zum Generieren ihrer Zeitstempel verwendet werden. Dafür muss aber der Zeitstempel gespeichert werden.

Der erste Zeitstempel wird dabei temporär im RAM gespeichert und beim Hohlen des zweiten Zeitstempels wird die Differenz berechnet. Danach wird der gespeicherte Zeitstempel gelöscht. Das Messergebnis wird im Anschluss in das EEPROM geschrieben, wobei das Schreiben ins EEPROM erheblich länger dauert als das Schreiben ins RAM. Der Vorteil beim EEPROM ist, dass die meisten Smartcards mehr EEPROM Speicher besitzen als RAM und das die Speicherung persistent ist.

Verwaltung und Speicherung der Messdaten

Man sieht schon, dass das Speichern der Zeitstempel nicht ausreicht, denn man muss beim Berechnen der Differenz zwischen zwei Zeitstempel auch wissen, welcher Zeitstempel zum aktuellen Zeitstempel gehört. Daher ist es wichtig, dass jeder Messpunkt ein eindeutiges Erkennungsmerkmal in Form einer ID erhält. Die ID dient dabei nicht nur zum Verwalten der Zeitstempel, sondern sie wird auch dafür benutzt, um Messergebnisse einem Messpunkt zuzuordnen. Daher müssen nicht nur Messergebnisse gespeichert werden, sondern auch die zugehörige ID.

Da in Smartcard Systemen sehr wenig RAM und EEPROM Speicherplatz zur Verfügung steht, ist es wichtig, dass man ihn effizient nutzt und keinen Speicher verschwendet. Daher wurde die Anzahl maximal möglicher IDs begrenzt. Durch die Begrenzung der IDs wird natürlich auch die maximale Anzahl an Messpunkten begrenzt. Um die Nutzung der Zeitmesseinheit nicht einzuschränken, wurde die maximale Anzahl der zum Messen einsetzbaren IDs auf 63 beschränkt. Eine nicht zum Messen eingesetzte ID, wird zum Speichern eines Fehlers verwendet. Dadurch ist es möglich Probleme, wie zum Beispiel einen Timerüberlauf, zu signalisieren. Das bedeutet man kann die IDs beziehungsweise den Fehlercode problemlos in einem Byte speichern und hat immer noch 2 Bit zur Verfügung um zusätzliche Informationen zu speichern. Die 2 Bit wurden im Fall der Zeitmesseinheit dazu benutzt, denn Speicherbedarf im EEPROM zu verringern.

Die Idee dabei ist, dass nicht jedes Messergebnis gleich groß ist. Das bedeutet, wenn ein Ergebnis nicht den gesamten Speicherplatz, der ihm zur Verfügung steht sinnvoll nutzt, dann kann man einen Teil des nicht sinnvoll genutzten Speicherplatzes zurückgewinnen. Der nicht sinnvoll genutzte Speicherplatz sind führende Nullstellen. Diese ergeben sich auf Grund der Subtraktion der beiden 32 Bit Zahlen. Das Ergebnis der Berechnung kann

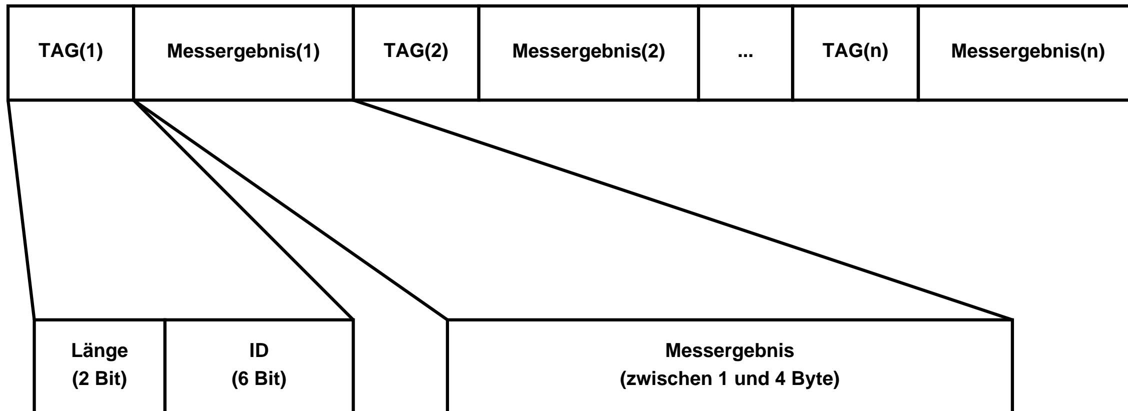

Abbildung 4.7: Format der im EEPROM gespeicherten Messdaten

daher nur kleiner oder gleich groß wie eine 32 Bit Zahl sein. Dies erklärt sich dadurch, dass der erste Zeitstempel immer kleiner als der zweite Zeitstempel ist. Dadurch ist es nicht möglich, dass ein negatives Ergebnis herauskommt. Dasselbe gilt auch für ein Ergebnis größer als 32 Bit.

Weiters sollte erwähnt werden, dass durchschnittliche Ergebnisse eine Länge in der Größenordnung von 2 bis 3 Byte aufweisen. Das bedeutet, dass das erste Byte eines gespeicherten Messergebnisses meist mit einer Null befüllt ist. In manchen Fällen trifft dies auch auf das zweite und das dritte Byte zu. Um diesen Speicherplatz zurückzugewinnen wird nun das Ergebnis byteweise untersucht. Dabei wird festgestellt, ob ein Byte zur Gänze mit einer führenden Nullstelle befüllt ist oder nicht. Die führenden Nullstellen werden danach beim Speichern des Ergebnisses nicht gespeichert.

Um die gemessenen Daten von einander unterscheiden zu können, benötigt man zur ID auch noch eine Information über die Länge der Daten. Diese Information wird dann in den 2 freien Bit der ID gespeichert. Die ID und die gespeicherte Länge der Daten werden als TAG bezeichnet und gemeinsam in einem Byte gespeichert. Wie man sieht, lassen sich durch 2 Bit nur vier Zustände kodieren, diese entsprechen der Länge des Ergebnisses angefangen bei 1 Byte bis hin zu 4 Byte. Es ist nicht sinnvoll jedes einzelne führende Nullstellen Bit zurückzugewinnen, da dafür erheblich mehr Rechenaufwand betrieben werden muss und zudem mehr Bit zum Speichern der Längenkodierung benötigte werden.

Diese Informationen sind aber nicht ausreichend um sicher zu stellen, dass man nur die Messdaten ausliest. Der Grund dafür ist, dass durch die Kodierung der Messdaten, fast jede beliebige Bitfolge einen Sinn im Bezug auf das Datenformat ergibt. Daher wird am Anfang der gespeicherten Daten auch noch die Anzahl der benötigten Byte im EEPROM gespeichert. Weiters wird am Ende der Messdaten ein Abschlusscode geschrieben. Dieser signalisiert dass die Messung erfolgreich beendet wurde. Weiters beinhaltet der Abschlusscode Informationen über nicht beendete Laufzeitmessungen. Den Aufbau der im EEPROM gespeicherten Daten wird in Abbildung 4.7 dargestellt.

Die im EEPROM gespeicherten Daten der Messeinheit werden nach Abschluss der Messungen ausgelesen. Der Auslesevorgang und die benötigten Software wird im Abschnitt

4.3 beschrieben.

4.2.2 Integration in das Betriebssystem

Das Smartcard Betriebssystem ist eine der Vorgaben für den Hardware/Software Codesign Flow. Es stellt die Verbindung zwischen der Anwendung und der Hardware her. Es ermöglicht die Integration unterschiedlichster Hardware- und Softwarekomponenten in das Smartcard System. Um Laufzeitmessungen an Hardware- und Softwarekomponenten durchführen zu können, war es notwendig, die Zeitmesseinheit in das Betriebssystem zu integrieren. Dafür wurde ein Treiber im Betriebssystem implementiert, der die Zeitmesseinheit verwendet. Die API Befehle nutzen wiederum die Funktionen des Treibers, wodurch die Zeitmesseinheit allen von der Plattform unterstützten Programmiersprachen zur Verfügung steht. Die Zeitmesseinheit besteht aus fünf in Assembler implementierten Hauptfunktionen. Diese sind:

- `starteZeitmesseinheit`
- `stoppeZeitmesseinheit`
- `beginneZeitmessung`
- `beendeZeitmessung`
- `TimererweiterungInterruptroutine`

Vier dieser Hauptfunktionen können vom Anwender aufgerufen werden. Die fünfte Hauptfunktion wird ausschließlich durch den Interrupthandler aufgerufen. Sie stellt die Implementierung der Timererweiterung dar und befindet sich in der Auflistung an unterster Stelle.

TimererweiterungInterruptroutine

Die `TimererweiterungInterruptroutine` erweitert den 16 Bit Hardwaretimer auf einen 32 Bit Timer. Dafür wird bei jedem Interrupt des 16 Bit Hardwaretimers der Interrupthandler aufgerufen, der wiederum die `TimererweiterungInterruptroutine` startet. Daher ist dies die einzige Funktion, die nicht als API Befehl ausgeführt wird.

starteZeitmesseinheit

Die Funktion `starteZeitmesseinheit` fordert den benötigten Speicherplatz an und startet den 16 Bit Hardwaretimer. Der benötigte Speicherplatz wird als ein Block bei der Speicherverwaltung des Betriebssystems über einen Systemcall angefordert. Danach wird der Speicherplatz initialisiert und den entsprechenden Variablen der Zeitmesseinheit zugeordnet. Abschließend startet die Funktion den 16 Bit Hardwaretimer.

stoppeZeitmesseinheit

Die Funktion `stoppeZeitmesseinheit` sorgt für das stoppen des 16 Bit Hardwaretimers, wodurch die noch aktiven Laufzeitmessungen gestoppt werden. Danach werden alle offenen Laufzeitmessungen beendet und der benötigte EEPROM-Speicherplatz wird ins EEPROM gespeichert. Zu guter Letzt wird die Abschlussstatusmeldung ins EEPROM gespeichert, was das Ende der Messdaten signalisiert. Danach wird der RAM Speicherplatz freigegeben und die Zeitmesseinheit kann nur durch einen erneuten Aufruf der `starteZeitmesseinheit` gestartet werden. Alle anderen Funktionen der Zeitmesseinheit bleiben inaktiv.

beginneZeitmessung

Die Funktion `beginneZeitmessung` ist eine von zwei Funktionen die einen Parameter benötigen. Der Parameter ist die ID des Messpunktes, anhand dieser unterschieden wird um welchen Messpunkt es sich handelt und welcher Speicherbereich ihm zur Verfügung steht. Die Funktion `beginneZeitmessung` ist für das Starten eines Messpunktes verantwortlich. Sie überprüft zuerst, ob der Messpunkt gerade aktiv ist. Ist dies der Fall, so wird die Fehlermeldung gespeichert und der Startzeitpunkt des Messpunktes bleibt unverändert. Ist hingegen der Messpunkt nicht gestartet, so wird der Zeitstempel gestartet und im RAM gespeichert.

beendeZeitmessung

Die Funktion `beendeZeitmessung` benötigt ebenfalls den Parameter ID. Sie ist wesentlich komplexer als die Funktion `beginneZeitmessung`. Der Grund dafür ist, dass diese Funktion mehrere Aufgaben besitzt. Zuerst überprüft sie ob der Messpunkt gestartet wurde. Ist dies nicht der Fall so wird eine Fehlermeldung gespeichert. Sonst wird der aktuelle Zeitstempel geholt und die Differenz zum Startzeitpunkt berechnet. Danach wird das Entfernen der führenden Nullstellen durchgeführt. Abschließend wird die Speicherung der Daten durchgeführt und der Messpunkt auf inaktiv gesetzt.

4.3 Übertragungssoftware für die Messdaten

Die bei der Abarbeitung der Anwendungsfälle gemessenen Laufzeitergebnisse werden auf der Smartcard gespeichert. Da das Smartcard System nicht für die Verarbeitung der Messdaten geeignet ist, werden die Messdaten an einen Computer übertragen. Der Datentransfer zwischen dem Smartcard System und dem Computer erfolgt über den Smartcard Reader. Dieser ist an einen Computer angeschlossen, der die empfangen Messdaten in eine Datei speichert. Die Abbildung 4.8 zeigt den grundsätzlichen Aufbau des Datentransfers.

Wie man sieht, benötigt man für den Transfer der Messdaten zwei Anwendungen. Eine ist dabei auf der Smartcard untergebracht und die andere befindet sich auf einem Computer. Die auf dem Computer verwendete Messdatentransfer-Software ist dafür verantwortlich, dass die Messdaten angefordert werden. Dies geschieht über APDUs die an die Smartcard als 'Sende Messdaten' Kommando gesendet werden. Die auf der Smartcard befindliche Messdatenauslese-Anwendung sendet die verfügbaren und angeforderten Messdaten an die Messdatentransfer-Software zurück.

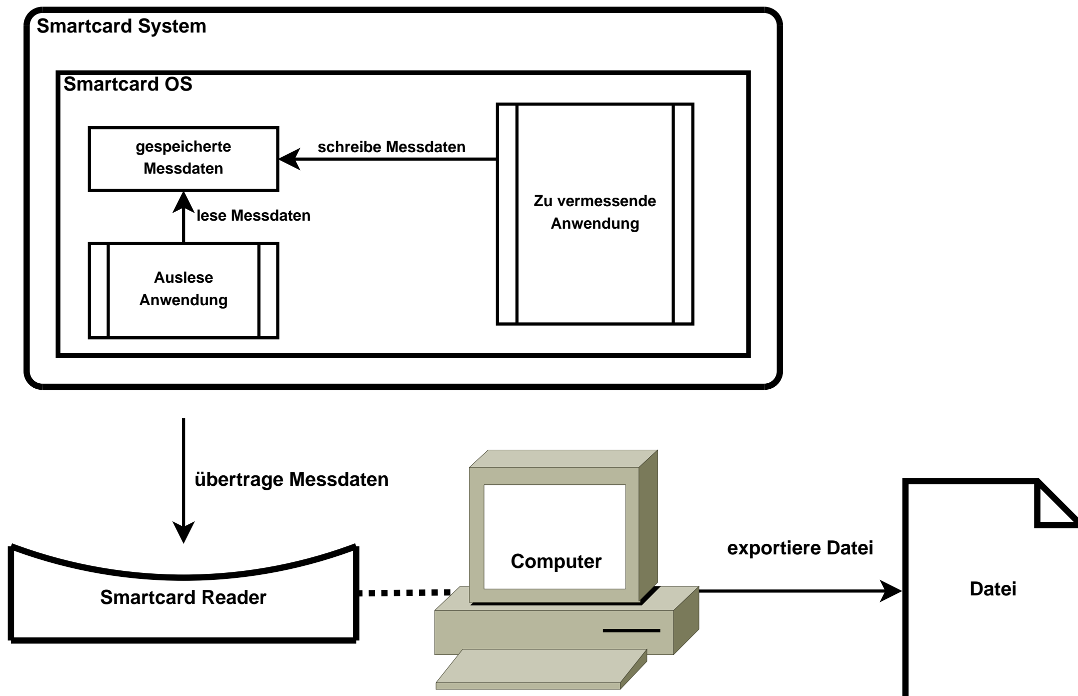

Abbildung 4.8: Schematischer Vorgang des Datentransfers zwischem Smartcard und Computer

Dabei werden die Messdaten nicht als Ganzes übertragen. Sie werden blockweise ausgelesen und transferiert. Der Vorteil daran ist, dass man die Messdaten in beliebig große Datenblöcke unterteilen kann. Dabei stellt die maximale Größe der APDU, die Obergrenze der zu versendenden Messdaten dar.

Die minimal mögliche Blockgröße der zu übertragenden Messdaten, beträgt ein Byte. Die Nachteile dieser Minimalkonfiguration sind, dass der Datentransfer mehr Zeit in Anspruch nimmt und dass der Verwaltungsoverhead steigt, da viele kleine APDUs versendet werden müssen. Im Gegensatz dazu, kann man die Messdaten auch in Datenblöcken von 255 Byte versenden. Dies hat den Vorteil, dass die Messdaten sehr schnell und unter der Verwendung weniger APDUs transferiert werden. Zwischen diesen beiden Extremen kann jede beliebige Blockgröße für den Datentransfer gewählt werden.

4.3.1 Die Messdatentransfer-Software

Die Messdatentransfer-Software fordert die Messdaten von der Smartcard an. Sie überprüft die übertragenen Messdaten auf Übertragungsfehler und ändert die Repräsentation der Messdaten in ein für Menschen lesbareres Format. Nachdem alle Messdaten empfangen wurden, speichert sie das Ergebnis in eine Datei. Die Abbildung 4.9 stellt dabei den Ablauf des Übertragungsvorgangs der Messdatentransfer-Software dar. Dabei wird gezeigt, in welcher Reihenfolge die Kommandos von der Messdatentransfer-Software an die Messdatenauslese-Anwendung gesendet werden.

Auf der Smartcard sind zwei Anwendungen installiert. Eine Anwendung ist die zu

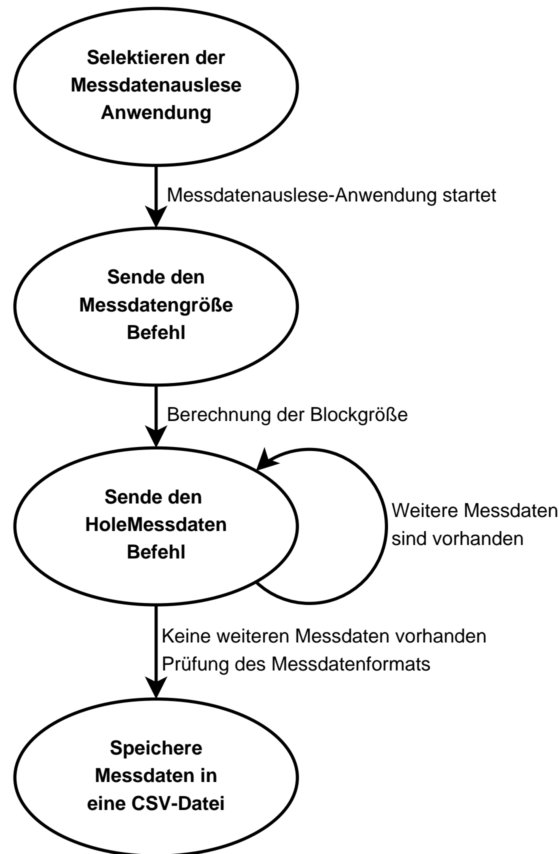

Abbildung 4.9: Ablauf des Übertragungsvorgangs der Messdatentransfer-Software

vermessende Anwendung. Sie beinhaltet die Aufrufe für die Zeitmesseinheit und speichert die Messergebnisse in den persistenten Speicher. Die zweite Anwendung ist die Messdatenauslese-Anwendung. Diese ist für die Übertragung der Messdaten verantwortlich. Sie wird im nachfolgenden Kapitel genauer behandelt. Bevor der Datentransfer erfolgen kann, muss die Messdatentransfer-Software die Messdatenauslese-Anwendung auf der Smartcard selektieren. Dies ist notwendig damit das gewünschte Kommando an die richtige Anwendung der Smartcard gesendet wird.

Nach dem Selektieren der Messdatenauslese-Anwendung wird sie gestartet und wartet auf weitere Kommandos. Das nächste an die Smartcard gesendete Kommando ist der Befehl zum bestimmen der Messdatengröße (`MessdatenGröße`). Die Smartcard sendet die Größe der verfügbaren Messdaten, über eine responseAPDU, zurück an die Messdatentransfer-Software.

Die Messdatentransfer-Software prüft, ob der Anwender eine bestimmte Blockgröße festgelegt hat. Dies kann der Anwender beim Starten der Messdatentransfer-Software über den Parameter Übertragungsblockgröße festlegen. Hat er keine Blockgröße vorgegeben, so versucht die Software die Blockgröße selbst zu bestimmen. Dafür wird zuerst die maximale Blockgröße zu senden versucht. Dies ist eine Blockgröße von 255 Byte. Schlägt dieser Versuch fehl, so versucht sie iterativ eine mögliche Blockgröße zu bestimmen, bei der die

Messdaten übertragen werden können. Dabei wird versucht, möglichst große Messdatenblöcke auf einmal zu übertragen um die Übertragungszeit zu minimieren.

Die Größe der Messdaten und die Blockgröße werden zur Berechnung der Anzahl der benötigten Blöcke verwendet. Dabei wird die Größe der Messdaten durch die Blockgröße dividiert. Wird für die Übertragung des letzten Blocks nicht die volle Blockgröße benötigt, so findet für diesen Datenblock eine Reduktion der Blockgröße statt. Für die Bestimmung der Größe des letzten Blocks wird die Messdatengröße modulo der Blockgröße berechnet. Das Resultat entspricht der noch zu übertragenden Daten. Dadurch ist es möglich nur die benötigten Daten zu übertragen und der Rest des Antwortblocks muss nicht ohne oder durch zufällige Daten aufgefüllt werden. So eine Markierung könnte zum Beispiel durch ein Auffüllen mit Nullen erfolgen.

Die Übertragung eines Datenblocks wird durch das Kommando `HoleMessdaten` gestartet. Es muss so oft an die Messdatenauslese-Anwendung gesendet werden, bis alle Messdaten transferiert übertragen wurden. Dabei wird der Startindex Parameter immer um die Blockgröße erhöht. Der Grund dafür wird im Abschnitt Messdatenauslese-Anwendung beschrieben. Nach Abschluss des Datentransfers werden die Messdaten in ein für Menschen lesbares Format gebracht.

Dabei erfolgt zeitgleich eine Prüfung des Datenformats, dass im Unterkapitel 4.2 durch die Abbildung 4.7 beschrieben wird. Dadurch müssen die Messdaten nur einmal bearbeitet werden, sofern kein Fehler auftritt. Wird eine Unstimmigkeit gefunden, die zum Beispiel durch ein umgekipptes Bit auftritt, so wird der betroffene Teil der Messdaten erneut übertragen. Dafür wird ein Datenblock mit maximaler Blockgröße verwendet. Weiters wird darauf geachtet, dass die Unstimmigkeit in der Mitte des angeforderten Datenblocks liegt. Dadurch soll es möglich sein einen Fehler zu finden, der vor dem erkannten Fehler auftrat. Ist der erneut übertragene Datenblock ident mit dem zuvor übertragenen Datenblock, so muss sich der Fehler vor dem übertragenen Datenblock befinden. In diesem Fall fordert die Software den vorangegangen Datenblock an. Dieser Vorgang wird so oft wiederholt, bis eine Abweichung erkannt wurde oder alle zuvor gespeicherten Daten übertragen sind.

Wurden alle Messdaten erneut übertragen und kein Übertragungsfehler gefunden, dann handelt es sich um einen Fehler der beim Speichern der Messdaten auftrat. Dieser Fehler kann nicht korrigiert werden, da der Auftrittspunkt des Fehlers nicht bekannt ist. Deshalb werden die Messdaten für ungültig erklärt. Um den Anwender trotzdem einen Einblick in die Ergebnisse zu gewähren, kann der Anwender die Rohdaten des Datentransfers speichern.

Abschließend werden die Messdaten in eine Datei gespeichert. Um diese Messergebnisse möglichst vielseitig verarbeiten zu können, wurde das CSV-Format (Comma-Separated Values Format) gewählt. Die CSV-Datei besteht dabei aus drei Spalten `ID`, `Laufzeit` und `Kommentar`. Die erste Spalte enthält die ID des Messpunkts an dem die Laufzeitmessung durchgeführt wurde. Die Zweite Spalte enthält das Laufzeitergebnis. Die dritte Spalte enthält den Kommentar. Dieser kann zum Beispiel ein Überlauf der 32 Bit Timererweiterung beinhalten. Die CSV-Datei kann in viele Mathematikprogramme importieren und verarbeitet werden, zum Beispiel in Matlab, PSPP oder Sarge. Es ist zudem möglich, durch Nutzung eines simplen Texteditors die in der CSV-Datei gespeicherten Messdaten zu betrachten.

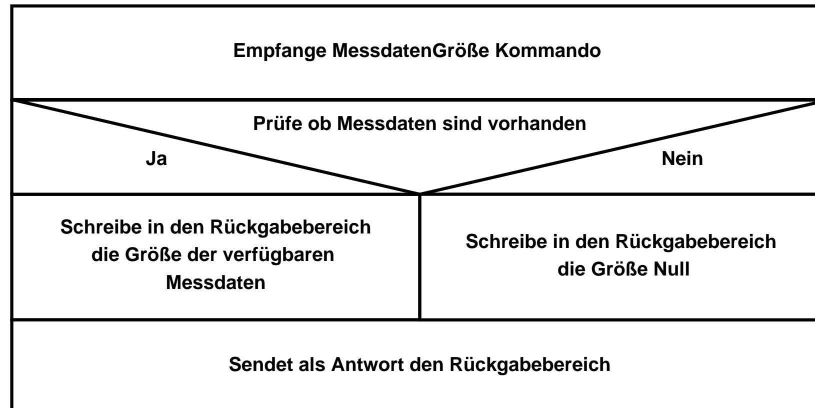

Abbildung 4.10: MessdatenGröße - Kommando für die Bestimmung der zu Übertragen Datengröße

4.3.2 Die Messdatenauslese-Anwendung

Die Messdatenauslese-Anwendung ist das Gegenstück zur Messdatentransfer-Software. Sie wird auf der Smartcard abgearbeitet und ist dafür verantwortlich, dass die angeforderten Blöcke, bestehend aus Messdaten, zurück gesendet werden. Dafür muss, wie bereits im vorangegangen Abschnitt erwähnt, die Messdatenauslese-Anwendung als erstes selektiert werden. Dadurch startet die Anwendung und wartet auf eine der beiden von ihr unterstützten Kommandos. Dies ist das `MessdatenGröße` Kommando welches in Abbildung 4.10 dargestellt wird. In Abbildung 4.11 wird das `HoleMessdaten` Kommando dargestellt.

Die Aufteilung des Datentransfers in diese zwei Kommandos hat mehrere Vorteile. Zum einen ist die Größe der zu übertragenden Messdaten bekannt. Dies erleichtert die blockweise Übertragung der Messdaten, da die Blockgröße exakt auf den Datentransfer angepasst werden kann. Zum anderen kann im Fall eines Übertragungsfehlers nur der fehlerhafte Datenblock erneut angefordert und übertragen werden.

Die Kommunikation zwischen der Messdatenauslese-Anwendung und der Messdatentransfer-Software gestaltet den Datentransfer simpel. Zuerst wird das Kommando `MessdatenGröße` gesendet und im Anschluss erfolgt das einmalige oder mehrmalige Senden des Kommandos `HoleMessdaten`. Tritt ein Fehler bei der Übertragung auf, so ist es möglich den vermutlich fehlerhaften Datenbereich erneut zu übertragen. Das genaue Vorgehen bei einem Fehler wird im Abschnitt der Messdatentransfer-Software behandelt.

Das Kommando für das Transferieren der Messdatengröße (`MessdatenGröße`) prüft, ob sich Messdaten im Speicher befinden. Ist dies der Fall so gibt es die Größe der gespeicherten Messdaten zurück. Ist dies nicht der Fall, so ist das Ergebnis Null. Null signalisiert der Messdatentransfer-Software, dass keine Messdaten vorhanden sind und somit auch keine weiteren Daten übertragen werden müssen.

Das `HoleMessdaten` Kommando ist wesentlich komplexer. Es besitzt zwei Parameter, die für eine erfolgreiche Übertragung wichtig sind. Der erste Parameter ist dabei der `Startindex`. Dieser gibt den Beginn der auszulesenden Messdaten an. Der zweite Parameter ist die Größe des auszulesenden Messdatenblocks `BlockGröße`. Beide zusammen

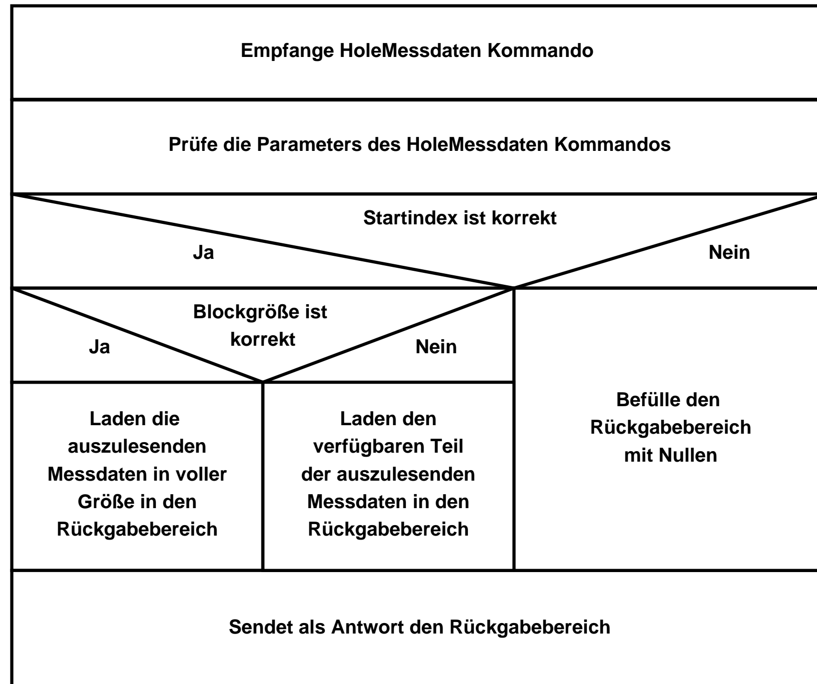

Abbildung 4.11: HoleMessdaten - Kommando für den Datentransfer von der Smartcard zum Computer

bestimmen den exakten Datenbereich der auszulesen und zu übertragen ist.

Die Gültigkeitsprüfung der beiden Parameter erfolgt separat. Damit ist gemeint, dass zuerst überprüft werden muss, ob sich der Startindex innerhalb der Größe der Messdaten befindet. Ist dies nicht der Fall, so können keine sinnvollen Messdaten zurückgesendet werden, da die Messdatenauslese-Anwendung keinen Startpunkt zufällig bestimmen kann, ab dem die Messdaten auszulesen sind. Die Antwort wird in der vollen Blockgröße mit Nullen befüllt und zurückgesendet. Befindet sich der Startindex hingegen innerhalb der Messdaten, so können die verfügbaren Messdaten ab dieser Position zurück gesendet werden.

Danach muss der Parameter für die Größe des Datenblocks auf seine Gültigkeit geprüft werden. Hierbei muss die Blockgröße addiert mit dem Startindex noch innerhalb der Messdaten liegen. Ist dies der Fall, so sind beide Parameter korrekt und der Messdatenblock kann ohne Komplikationen übertragen werden. Liegt hingegen die Blockgröße addiert mit dem Startindex außerhalb der Messdaten, so werden alle Messdaten ab dem Startindex und bis zum Ende des Messdatenbereichs zurückgeschickt.

Die beiden Funktionen beschreiben die vollständige Messdatenauslese-Anwendung. Wie man sieht führt die Anwendung dabei nur Kommandos aus und verfügt über keine eigene Steuerlogik. Dadurch benötigte die Anwendung sehr wenig anwendungsspezifischen Speicherplatz.

4.4 Die Auswertesoftware

Die Auswertesoftware ist das letzte Glied des Hardware/Software Codesign Flows. Sie hat mehrere Aufgaben, wobei der Name schon vermuten lässt, dass ihre Hauptaufgabe die Analyse der Laufzeiten ist. Neben dieser Aufgabe hat sie aber auch noch ein paar Nebenaufgaben. Dazu gehören das Zusammenfügen mehrerer Messdatensätze, das Sortieren der Messdaten anhand der Messpunkte, das Speichern der bearbeiteten Messdaten in eine CSV-Datei sowie das Hinzufügen von Informationen über die Messdaten zu den einzelnen Messergebnissen. Die Informationen über die Messdaten könnte zum Beispiel eine an die Smartcard gesendete APDU oder ein verwendeter Schlüssel sein.

4.4.1 Anwendungsfeld der Auswertesoftware

Die Auswertesoftware kann die von der Messdatentransfer-Software übertragenen Messdaten aus der CSV-Datei einlesen. Dabei werden für die Laufzeitanalyse nur die beiden Spalten `ID` und `Laufzeit` verwendet. Die Spalte `Kommentar` kann zusätzliche Informationen zur Laufzeitmessung enthalten. Diese werden aber durch die Auswertesoftware nicht berücksichtigt. Sie dienen zum manuellen Überprüfen der Ergebnisse der Auswertesoftware, da sie einen Bezug zwischen den gemessenen Daten und dem Ergebnis der Laufzeitanalyse herstellen.

Wie bereits erwähnt, können die eingelesenen Messdaten auf unterschiedlichste Weise weiter bearbeitet werden. Dabei stellt die Analyse des Laufzeitverhaltens die üblichste Verwendung dar. Sie ordnet die einzelnen Laufzeiten den Messpunkten zu. Dies ermöglicht eine Laufzeitanalyse der einzelnen Messpunkte. Die Ergebnisse der Laufzeitanalyse der Messpunkte werden im Timing Bericht festgehalten. Dabei lässt sich der Grad des Informationsgehalts des Timing Berichts variieren. Angefangen von 'Bestanden' oder 'Nicht Bestanden' bis hin zur Ausgabe der einzelnen Laufzeiten inklusive der Kommentare.

Die weiteren Funktionen der Auswertesoftware dienen nicht der Analyse der Messdaten. Sie haben vielmehr die Aufgabe, den Komfort bei der Auswertung zu verbessern. Eine dieser Funktionen ist das Zusammenfügen von Messdatensätzen. Dadurch ist es möglich, eine komplexere Prüfung des Smartcard Systems durchzuführen, für die sonst der persistente Speicherplatz des Systems nicht ausgereicht hätte. Dafür muss nur die komplexe, gesamte Prüfung in mehrere kleine Teilprüfungen aufgeteilt werden. Diese können im Anschluss zu einem Gesamtprüfungsergebnis zusammengefasst werden.

Damit man die einzelnen Dateien nicht jedes Mal manuell selektieren muss, gibt es auch die Möglichkeit eine Liste der CSV-Dateien zu laden. Diese Liste wird ebenfalls in einer Listendatei gespeichert. Die Listendatei kann weiters auch noch Informationen über die einzelnen Messungen enthalten. Diese Informationen werden bei den Messdaten als `Kommentar` gespeichert. Dies erleichtert das manuelle Prüfen der Auswertungsergebnisse.

Eine weitere Funktion ist das Speichern der Messdaten. Damit können zum Beispiel zusammengefügte Messdatensätze in eine CSV-Datei gespeichert werden. Die letzte Funktion sortiert die Messdaten anhand der Messpunkte. Dabei entspricht jeder Messpunkt einer Gruppe, die mit den entsprechenden Laufzeitergebnissen befüllt wird. Die Gruppen lassen sich schneller verarbeiten, da nicht bei jedem Ergebnis geprüft werden muss, ob es der betrachteten Gruppe entspricht. Weiters erleichtert es auch die Bearbeitung in anderen

Mathematikanwendungen (zum Beispiel: Matlab[1], PSPP[2], Sarge[3], ...).

4.4.2 Ansatz zur Laufzeitanalyse der Auswertesoftware

Die Timing Attacke nutz mehrere mathematische Grundlagen für einen Angriff auf ein System. Diese wurden zum Teil im Unterkapitel 2.4.4 erklärt. Die Timing Attacke versucht dabei einen Zusammenhang zwischen den Laufzeitergebnissen zu finden. Wird ein Zusammenhang gefunden, so versucht die Timing Attacke die eigentliche Attacke auszuführen. Dafür werden ein Signifikanztest und ein Zufälligkeitstest eingesetzt. Häufig wird als Signifikanztest die Distanz von Mittelwerten der Laufzeiten verwendet. Dabei ist der Abstand zum Mittelwert das entscheidende Kriterium für die Auftrittswahrscheinlichkeit. Der Zufälligkeitstest prüft abschließend die Wahrscheinlichkeit, sodass die Auftrittswahrscheinlichkeit zufällig entstanden ist.

Der Unterschied zwischen der Timing Attacke und der Laufzeitanalyse der Auswertesoftware ist, dass die Auswertesoftware keine Attacke ausführt. Sie untersucht das Laufzeitverhalten eines beziehungsweise jedes Messpunktes. Für die Laufzeitanalyse wird, wie bei der Timing Attacke, die Distanz von Mittelwerten als Entscheidungskriterium herangezogen. Besteht dabei der Verdacht, dass es sich um eine Laufzeitabhängigkeit handelt, so wird der Messpunkt im Timing Bericht gekennzeichnet. Da die Timing Attacke auf Wahrscheinlichkeiten beruht und die Laufzeitanalyse Ansätze der Timing Attacke nutzt, kann es auch zu Falschmeldungen kommen. Diese hängen von dem gewählten Genauigkeitsgrad ab.

Laufzeitanalyse

Die Laufzeitanalyse versucht die Laufzeiten in Gruppen ähnlicher Werte einzuteilen. Gelingt dabei eine Unterteilung in mehrere Gruppen, so wird das Ergebnis als möglicherweise laufzeitabhängig eingestuft. Je klarer sich die Gruppen unterscheiden, desto größer ist die Wahrscheinlichkeit, dass die Funktion eine Laufzeitabhängigkeit besitzt.

Für die Laufzeitanalyse werden die mathematischen Funktionen Minimum, Maximum, Mittelwert und Standardabweichung benötigt. Das Minimum, das Maximum, die Anzahl der Elemente und die Standardabweichung dienen bei der Gruppierung als Abbruchsbedingung. Erreicht sie einen Wert, der keine weitere sinnvolle Unterteilung ermöglicht, so wird der Gruppierungsvorgang beenden. Dies ist zum Beispiel der Fall, wenn die Standardabweichung unter dem minimal messbaren Laufzeitunterschied liegt.

Die Gruppierung der Laufzeiten erfolgt über den Mittelwert und die Standardabweichung. Das Ziel der Gruppierung ist, dass die neuen Gruppen, eine geringere Standardabweichung von ihrem Mittelwert aufweisen, als die Gesamtgruppe zuvor. Weiters müssen sich die einzelnen, neu gebildeten Gruppen klar voneinander unterscheiden. Das heißt, dass sich die Mittelwerte der gebildeten Gruppen nicht im Entscheidungsbereich einer anderen Gruppe befinden dürfen.

Die Gruppierung erfolgt in drei Schritten und beruht auf Ansätzen des Signifikanztests sowie der Clusteranalyse [BPW10]. Der erste Schritt der Gruppierung ist das Berechnen

[1]MathWorks MATLAB http://www.mathworks.de/products/matlab/index.html
[2]GNU PSPP http://www.gnu.org/software/pspp/
[3]Sage http://www.sagemath.org/

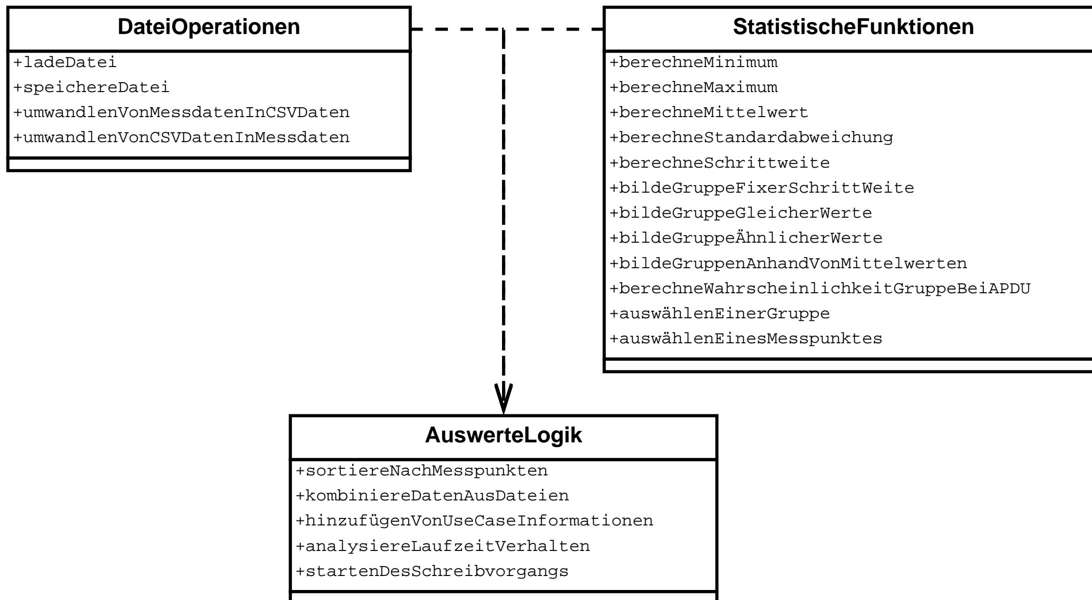

Abbildung 4.12: Klassenaufbau der Auswertesoftware

des Minimums und des Maximums sowie des Mittelwerts μ und der Standardabweichung s einer Gruppe. Der zweite Schritt ist das Bestimmen eines Elements $Element_\mu$, das die geringste Distanz zum Mittelwert aufweist. Dieses Element dient als neue Mitte, um das die Gruppe gebildet wird. Der dritte Schritt ist die Bildung der Gruppe, dabei werden alle Elemente in die Gruppe aufgenommen die die Bedingung

$$Element_\mu - s \leq Laufzeit \leq Element_\mu + s \tag{4.1}$$

erfüllen und natürlich keiner bisherigen Gruppe angehören.

Der Vorgang der Gruppierung wird iterativ auf die gebildeten Gruppen angewendet, bis die Abbruchsbedingung erreicht ist. Danach erfolgt eine Bewertung der Laufzeitabhängigkeit. Dafür werden die Distanzen der Mittelwerte betrachtet, die sich am nächsten sind. Nach der Berechnung der Distanzen zwischen den Mittelwerten, wird die maximale Distanz für die Bewertung herangezogen und durch die kleinst mögliche Messeinheit dividiert. Dadurch wird sichergestellt, dass die Bewertung von der benötigten Laufzeit unabhängig ist.

4.4.3 Implementation der Auswertesoftware

Die Auswertesoftware ist in Java implementiert, um eine plattformübergreifende Nutzung zu ermöglichen. Sie besteht aus drei Klassen, die in Abbildung 4.12 dargestellt sind. Der Datentype `DatenElement` ist das Grundelement, indem der Messpunkt, eine Laufzeit des Messpunkts und die Kommentare gespeichert werden. Dadurch stehen die gesamten Daten einer einzigen Messung während der Analyse zur Verfügung.

Die Klasse `DateiOperationen` implementiert die Funktionen zum Speichern und Laden von Datensätze, sowie zum Speichern des Auswerteberichts. Die Datensätze stehen, nachdem sie aus einer CSV-Datei geladen wurden, als Datentype `ArrayList` von `DatenElement`

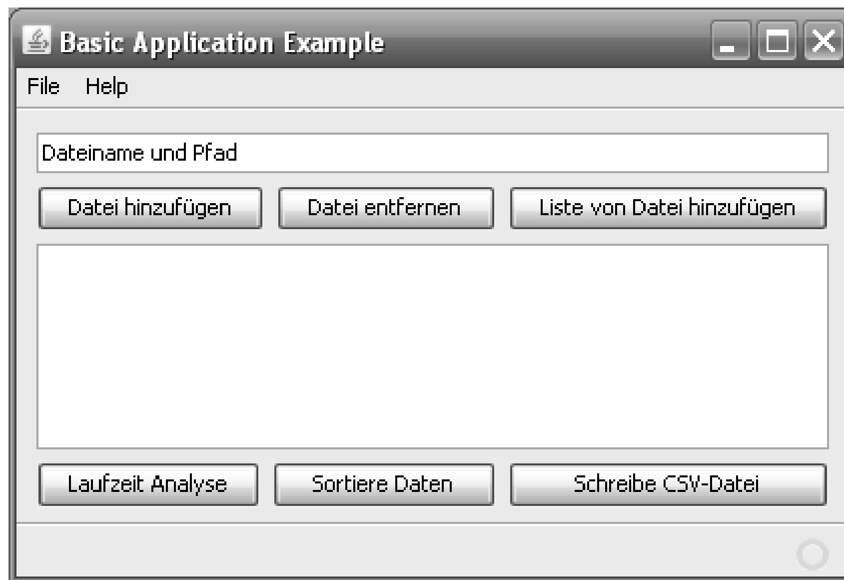

Abbildung 4.13: Grafische Benutzerschnittstelle für die Auswertesoftware

zur Verfügung. Der abstrakte Datentype `ArrayList` hat den Vorteil gegenüber einem herkömmlichen Array, dass er sich flexibel vergrößern und verkleinern lässt. Dieser Vorteil ist besonders beim Laden und Gruppieren der Messdaten von Bedeutung.

Die Klasse `StatistischeFunktionen` beinhalten die mathematischen Funktionen, die für die Analyse und Gruppierung der Messdaten benötigt werden. Alle Funktionen dieser Klasse arbeiten mit dem Datentype `ArrayList` von `DatenElement`. Einzelne Gruppen von Laufzeiten stehen, wie auch die aus der Datei geladenen Laufzeiten, als `ArrayList` von `DatenElement` zur Verfügung. Einzig die Funktion `auswahlEinerGruppe` verwendet als Parameter `ArrayList` von `ArrayList`. Dies erklärt sich durch die Implementation der Gruppierfunktionen, die als Rückgabewert `ArrayList` von `ArrayList` haben.

Die Klasse `AuswerteLogik` kombiniert die Funktionen der ersten beiden Klassen. Sie implementiert damit die einzelnen Funktionen der Auswertesoftware. Weiters prüft sie die Parameter für den Kommandozeilenbetrieb. Dieser ermöglicht eine automatisierte Auswertung der Laufzeitergebnisse. Alternativ zum Kommandozeilenbetrieb verfügt die Auswertesoftware auch über eine grafische Benutzeroberfläche. Diese ermöglicht eine intuitive Nutzung der Auswertesoftware. Die grafische Benutzeroberfläche wird in Abbildung 4.13 dargestellt und wurde unter Zuhilfenahme der Netbeans IDE[4] implementiert. Diese ermöglicht eine simple Umsetzung der Bedienungselemente, da sich die Oberfläche wie in einem Grafikprogramm zeichnen lässt. Sie ruft, wie im Kommandozeilenbetrieb, die einzelnen Funktionen der `AuswerteLogik` mit den entsprechenden Parametern auf. Daher unterscheiden sich die beiden Varianten der Auswertesoftware nur durch die Benutzerschnittstelle.

[4]Netbeans IDE http://netbeans.org/

Kapitel 5

Ergebnisse

In diesem Kapitel wird die Anwendung des Hardware/Software Codesign Flows zum Vorbeugen von Timing Attacken beschrieben. Es teilt sich in zwei Unterkapitel auf, wobei das erste der beiden Unterkapitel das Messverhalten der Zeitmesseinheit untersucht. Dafür wird die Messgenauigkeit bestimmt und das Verhalten bei wiederholter Messung einer konstanten Laufzeit untersucht.

Das zweite Unterkapitel beschreibt die Implementation einer Beispielanwendung. Diese wird anhand eines Smartcard Systems umgesetzt. Das Unterkapitel setzt sich dabei aus drei Teilen zusammen. Zuerst wird die Beispielanwendung spezifiziert, danach folgt die Implementation und zum Abschluss werden die Resultate der Laufzeitanalyse beschrieben.

5.1 Messverhalten der Zeitmesseinheit

Dieses Unterkapitel beschäftigt sich mit dem Messverhalten der Zeitmesseinheit. Dabei wird zum einen die maximale Genauigkeit der Zeitmessung bestimmt und zum anderen wird das Verhalten bei mehrfachem Vermessen einer konstanten Zeitdauer untersucht. Die sich daraus ergebenden Eigenschaften der Zeitmesseinheit sind für die Laufzeitanalyse wichtig, da sie die Frage klärt, wie genau die Laufzeitmessung ist. Weiters soll damit die Annahme bestätigt werden, dass bei wiederholter Messung einer konstanten Laufzeit das Ergebnis normalverteilt ist.

5.1.1 Bestimmung der Genauigkeit der Messergebnisse

Die Messgenauigkeit der Zeitmesseinheit wurde mit einem zyklengenauen Simulator und Emulator überprüft. Dafür wurde die Implementation der Zeitmesseinheit ohne das Smartcard Betriebssystem verwendet, um Einflüsse des Smartcard Betriebssystems auf den Messaufbau auszuschließen. Der kleinst mögliche Laufzeitunterschied, der mit der Zeitmesseinheit detektiert werden kann, wird durch den Hartwaretimer vorgegeben, der zur Generierung der Referenzintervalle dient. Er beträgt, bei Verwendung der bestmöglichen Einstellung des Hardwaretimers, 8 Mikrosekunden. Durch Änderungen der Hardwaretimereinstellungen kann das zu generierende Zeitintervall auf ein vielfaches von 8 Mikrosekunden vergrößert werden, wie zum Beispiel 32 Mikrosekunden.

Über den kleinsten messbaren Laufzeitunterschied wird die Empfindlichkeit E der Zeitmesseinheit berechnet, da der Simulator ein exakte Referenzintervalle erzeugt. Der Emula-

tor eignet sich nur begrenzt zur Bestimmung der Empfindlichkeit, da sich die Zeitdauer der Referenzintervalle von Emulator zu Emulator minimal ändert. Die Empfindlichkeit E berechnet man indem der minimal angezeigte Unterschied der Timerwerte $\Delta N_{Timerwert} = 1$ durch die Differenz der beiden Messgrößen (Stopp- und Startzeit) $\Delta x_{Messgröße}$ rechnet.

$$E = \frac{\Delta N_{Timerwert}}{\Delta x_{Messgröße}} = \frac{1Skalenteil}{8\mu s} = 0,125\frac{Skalenteil}{\mu s} = 125000\frac{Skalenteil}{s} \quad (5.1)$$

Für die Bestimmung der Genauigkeit wurde der Hardwaretimer auf die Generierung von 8 Mikrosekunden Zeitintervallen eingestellt. Das entspricht einer Abtastfrequenz von 125 Kilohertz. Auf Grund des Abtasttheorems von Shannon muss für die Bestimmung der Genauigkeit ein Messgerät eingesetzt werden, das zumindest die doppelte Abtastfrequenz aufweist.

Externe Zeitmessung am Emulator

Die externe Zeitmessung am Emulator erfolgt durch ein Oszilloskop. Dieses besitzt eine maximale Abtastfrequenz von 10 Megahertz, was um ein vielfaches höher ist, als die vorausgesetzte Mindestabtastfrequenz von 250 Kilohertz. Auf Grund der sehr viel höheren Abtastrate wurde die Messunsicherheit des Oszilloskops bei der Bestimmung der Genauigkeit nicht berücksichtigt. Für die Zeitmessung wurde ein Pin des Emulators, zu beginn des zu vermessenden Codes, auf High gesetzt und nach Abschluss des Codes wurde er wieder auf Low gesetzt. Die vergangene Zeit $t_{Oszilloskop}$ kann somit vom Oszilloskop abgelesen werden und entspricht der Zeit, die das Signal auf High war. Die Emulatorzeit $t_{Emulator}$ ist gleich der mit dem Oszilloskop bestimmten Zeit, nach Abzug des Overheads der Zeitmesseinheit $t_{ZeitmesseinheitOverhead}$.

$$t_{Emulator} = t_{Oszilloskop} - t_{ZeitmesseinheitOverhead} \quad (5.2)$$

Die Emulator-Oszilloskop Messung ist nur für kurze Laufzeiten möglich, da sie durch die Speichertiefe des Oszilloskop begrenzt ist. Dies stellt für die Bestimmung der Genauigkeit kein Problem da, denn der Anwendungsbereich der Zeitmesseinheit ist auf keinen Fall länger als die untersuchte Zeitdauer.

Externe Zeitmessung am Simulator

Die externe Zeitmessung am zyklengenauen Simulator erfolgt durch setzen von zwei Breakpoints. Diese wurde exakt an den Stellen gesetzt, die für das setzen und löschen des Pins verantwortlich sind. Beim Erreichen des ersten Breakpoints wird die Simulatorzeit auf Null gesetzt und der zu vermessene Code wird gestartet. Beim erreichen des zweiten Breakpoints wird die Simulatorzeit gestoppt und aufgenommen. Die aufgenommene Zeit $t_{Breakpoints}$ ist gleich der Zeit, die der Simulator $t_{Simulator}$ benötigt, wenn man die Zeit für den Overhead der Zeitmesseinheit abzieht.

$$t_{Simulator} = t_{Breakpoints} - t_{ZeitmesseinheitOverhead} \quad (5.3)$$

Zeitmessung durch die Zeitmesseinheit

Die Zeitmessung durch die Zeitmesseinheit erfolgte durch Setzen eines Messpunktes. Dieser beinhaltete das Setzen und Löschen des Pins, der für die Oszilloskop Messung benötigt wird. Die dafür benötigte Zeit ist bekannt und kann somit im Messergebnis korrigiert werden. Dies hat den entscheidenden Vorteil, dass der zu vermessende Code immer gleich ist. Dies ist auch der Grund warum die Zeitmesseinheit bei jeder Zeitmessung aktiv ist. Der Timerwert $N_{Timerwert}$ ist die Differenz zwischen Stopp- und Start-Zeitstempel. Die Umrechnung vom Timerwert $N_{Timerwert}$ zur durch die Zeitmesseinheit gemessenen Zeit $t_{Zeitmesseinheit}$ hängt von der Konfiguration des Hardwaretimers ab und wird durch die folgende Formel dargestellt.

$$t_{Zeitmesseinheit} = N_{Timerwert} * 8\mu s \tag{5.4}$$

Für die exakte Bestimmung der Laufzeit muss der zeitliche Overhead der Zeitmesseinheit $t_{ZeitmesseinheitOverhead}$ berücksichtigt werden. Dieser setzt sich aus mehreren Zeiten zusammen. Dies ist zum einen die Zeit $t_{InterruptOverhead}$, die für die Abarbeitung der Interrupts benötigt wird und zum andern die Zeit $t_{MesspunktOverhead}$, die die Zeitmesseinheit benötigt, um einen Messpunkt zu starten und zu stoppen.

$$t_{ZeitmesseinheitOverhead} = t_{MesspunktOverhead} + t_{InterruptOverhead} \tag{5.5}$$

Auf Grund der Implementierung der Zeitmesseinheit muss die Zeit, die für die Abarbeitung des Interrupts benötigt wird, weiter aufgeteilt werden. Die Ursache für diese Aufteilung liegt in der Tatsache, dass die Zeitmesseinheit eine 2 Byte Timererweiterung besitzt und die Berechnung der Timererweiterung von der Anzahl der veränderten Bytes abhängt.

$$t_{InterruptOverhead} = n_{Interrupt_8} * t_{Interrupt_8} + n_{Interrupt_{16}} * t_{Interrupt_{16}} \tag{5.6}$$

Der Vollständigkeit halber muss noch erwähnt werden, dass jeder gestartete und gestoppte Messpunkt ebenfalls die Zeit $t_{MesspunktOverhead}$ benötigt. Wird ein Messpunkt nur gestoppt oder gestartet, so muss die Zeit $t_{MesspunktOverhead}$ weiter unterteilt werden in die Zeit $t_{MesspunktStartOverhead}$ die zum Starten und in die Zeit $t_{MesspunktStopOverhead}$ die zum Stoppen des Messpunktes benötigt wird. Dies ist aber für die Bestimmung der Genauigkeit nicht von Bedeutung, da hierbei jeweils nur ein Messpunkt verwendet wird.

Programmcode des Messaufbaus

Der Programmcode des Messaufbaus ermöglicht die Zeitmessung auf der Smartcard. Er wird im Codeblock 5.1 dargestellt und zeigt, an welche Stelle im Messaufbau die Zeitmessung beginnt und endet, sowie wann der Prüfling beziehungsweise zu vermessende Programmcode abgearbeitet wird. Um die Zeitmessung mittels Oszilloskop zu vereinfachen, wird der Programmcode des Prüflings mehrfach abgearbeitet. Dadurch ist das am Oszilloskop dargestellte Messergebnis eine rechteckig geformte Schwingung. Um die rechteckige Form bei längeren Periodendauern leichter zu triggern, wird der Programmcode des Prüflings zweimal pro Schleifendurchlauf abgearbeitet. Der Logikpegel des Pins wird vor dem ersten Abarbeiten des Prüflings auf High gesetzt und nach dem Abarbeiten des

ersten Prüflings auf Low. Dadurch ergib sich für beide Halbwellen der Schwingung eine annähernd gleichlange Dauer. Für die Bestimmung der Genauigkeit wurde nur die Halbwelle vermessen, bei der der Pin auf dem High-Pegel ist. Der Grund dafür ist, dass die Halbwelle, bei der der Pin den Low-Pegel aufweist, den Overhead durch die for-Schleife beinhaltet, sowie auch den Start- und Stoppvorgang der Zeitmesseinheit.

```
void prüfeGenauigkeit(){
    int idx_mess=0, idx_außen=0, idx_innen=0, resultat=0;
    IO2DIR = 1; // set port IO2 to output mode
    IO2 = 0;
    starteZeitmesseinheit();
    for( idx_mess=0 ; idx_mess<10000 ; idx_mess++ ){

        starteZeitmesseinheit(ID1);
        IO2 = 1;
        Prüflingscode // zu vermessener Code, erster Prüfling
        IO2 = 0;
        stoppeZeitmesseinheit(ID1);

        Prüflingscode // zweiter Prüfling
    }
    beendeZeitmessung();
}
```

Codeblock 5.1: Programmcode für den Messaufbau für die Genauigkeitsbestimmung

Prüfling

Als zu vermessender Prüfling diente ein auf Schleifen basierender Programmcode, der im Codeblock 5.2 dargestellt wird. Er wurde in C implementiert und besteht aus zwei verschachtelten for-Schleifen, deren Indizes aufsummiert werden. Der Aufbau des Prüflings ermöglicht eine einstellbare Laufzeit. Dies ist durch eine Variation der beiden Variablen `DurchläufeAußen` und `DurchläufeInnen` möglich.

```
resultat=0;
for( idx_außen=0 ; idx_außen<DurchläufeAußen ; idx_außen++ ){
    for( idx_innen=0 ; idx_innen<DurchläufeInnen ; idx_innen++ ){
        resultat=resultat+idx_außen+idx_innen;
    }
}
```

Codeblock 5.2: Zu vermessender Programmcode(Prüfling)

Um ein möglichst einfach nachvollziehbares Messergebnis zu erhalten, wurde die Anzahl der inneren Schleifendurchläufe beziehungsweise die Variable `DurchläufeInnen` konstant gehalten. Dies hat eine konstante Laufzeit der inneren Schleife zur Folge. Die Laufzeit für die innere Schleife setzt sich aus dem Schleifen-Overhead und der Aufsummung der Indizes zusammen. Diese beiden Laufzeiten hängen direkt miteinander zusammen und werden mit $t_{Konstant}$ bezeichnet. Durch Variation der Anzahl der äußeren Schleifendurchläufe $n_{außen}$

wird die Laufzeit des Programmcodes verändert. Dies erfolgt über eine Änderung der Variable `DurchläufeAußen`. Die benötigte Laufzeit für einen äußeren Schleifendurchlauf ist $t_{Schleife} = t_{Konstant} + t_{Overhead}$, wobei $t_{Overhead}$ der äußere Schleifen-Overhead ist. Die gesamte Laufzeit für das Abarbeiten des Codes benötigt somit $t_{Prüfling} = n_{außen} * t_{Schleife}$.

Berechnung der Genauigkeit der Zeitmesseinheit

Für die Bestimmung der Genauigkeit wird der Prüfling mehrmals vermessen. Die Anzahl der inneren Schleifendurchläufe wird dabei auf 256 festgelegt (`DurchläufeInnen` = 256). Dadurch wird die zeitliche Basis für die Messung festgelegt. Die äußere Schleife wird in 256 Schritten erhöht und gibt an wie oft die innere Schleife abgearbeitet werden muss $Durchläufe_n = Durchläufe_{n-1} + 256$.

In der Tabelle 5.1 werden die Ergebnisse der Messung dargestellt. Die erste Spalte der Tabelle stellt die Anzahl der äußeren Schleifendurchläufe beziehungsweise den Zahlenwert der Variablen `DurchläufeAußen` dar. Die zweite bis vierte Spalte stellt die nicht korrigierten Messergebnisse der einzelnen Zeitmessgeräte dar. Dabei wird in der Tabelle 5.1 das Gerät angegeben, auf dem die Zeitmessung durchgeführt wird und nicht welches Messgerät diese Messung durchführt. Da die gemessenen Zeiten der Zeitmesseinheit am Simulator und Emulator gleich sind, werden sie nur einmal angegeben. Die Zeitmesseinheit wird aus Platzgründen in der Tabelle 5.1 mit ZME abgekürzt.

Durchläufe	**Gemessene Zeiten ohne Korrektur**			**Differenz**	
	Simulator	Emulator	ZME	ZME-Simu	ZME-Emu
	[s]	[s]	[s]	[μs]	[μs]
256	0,621333	0,621320	0,618300	2	-327,2
512	1,242664	1,242640	1,236341	5,5	-393,0
768	1,863996	1,863952	1,854383	1	-467,6
1024	2,485327	2,485272	2,471106	4,5	791,3
1280	3,106659	3,106592	3,089866	8	3,0
1536	3,728008	3,727904	3,708144	1	-295,0
1792	4,349339	4,349224	4,324478	4,5	1355,9
2048	4,970671	4,970544	4,941046	8	2770,5

Tabelle 5.1: Messergebnisse des Simulators, des Emulators und der Zeitmesseinheit (ZME)

Um die Abweichung der Zeitmesseinheit zu bestimmen, wurde nur die Differenz zwischen der von der Zeitmesseinheit gemessenen Zeiten $t_{Zeitmesseinheit}$ und den Zeiten die vom Simulator $t_{Simulator}$ und Emulator $t_{Emulator}$ gemessen wurden herangezogen. Diese werden in der fünften und sechsten Spalte der Tabelle 5.1 angegeben.

$$\Delta t_{Emulator-Zeitmesseinheit} = t_{Emulator} - t_{Zeitmesseinheit} \tag{5.7}$$

$$\Delta t_{Simulator-Zeitmesseinheit} = t_{Simulator} - t_{Zeitmesseinheit} \tag{5.8}$$

Die Zeitdifferenz zwischen Simulator und Emulator wurde ebenfalls berechnet, um festzustellen, wie sich die Hardware im Vergleich zur Simulation verhält. Diese wird allerdings in der Tabelle nicht aufgelistet, da sie nur eine Kontrollaufgabe hat und keine neue Information offenbart.

$$\Delta t_{Emulator-Simulator} = t_{Emulator} - t_{Simulator} \tag{5.9}$$

Die Zeitdifferenzen zwischen der am Simulator gemessenen Laufzeit und der Laufzeit, die durch die Zeitmesseinheit bestimmt wurde, weist nach der Korrektur des Overheads keine größere Abweichung als 8 Millisekunden auf. Dies entspricht dem erwarteten Grad der Genauigkeit der Zeitmesseinheit, da sie nicht genauer messen kann als das ihr zur Verfügung stehende Referenzintervall. Somit ergibt sich für Genauigkeit der Zeitmesseinheit am Simulator, bezogen auf die gesamte gemessene Zeit, eine maximale Abweichung von $\pm 1,6\, ppm$.

Beim Auswerten der Differenzen am Emulator wurde festgestellt, dass sich die durch das Oszilloskop gemessenen Zeiten von den durch die Zeitmesseinheit bestimmten Zeiten um mehr als 8 Millisekunden unterscheiden. Der Grund für die größeren Abweichungen könnte der Aufbau des Emulators sein. Die Genauigkeit der Zeitmesseinheit am Emulator, bezogen auf die gesamte gemessene Zeit, ergibt eine maximale Abweichung von $\pm 560,7\, ppm$.

5.1.2 Bestimmung des Verhaltens beim Messen einer konstanten Laufzeit

Allgemein wird für Messgeräte, bei denen eine konstante gehaltene Messgröße vermessen wird, eine Normalverteilung der Messergebnisse um den Wahrenwert angenommen [Sch98]. Dieses Verhalten ist auch für die Zeitmesseinheit erwünscht, da eine annäherte Normalverteilung der Messergebnisse eine Vorraussetzung für den Signifikanztest beziehungsweise die Timing Attacke ist.

Für die Überprüfung dieses Verhaltens wurde eine Funktion `berechnePrüfsumme`, die eine konstante Laufzeit besitzt, mehrmals mit der Zeitmesseinheit vermessen. Um möglichst konstante Bedingungen für die Abarbeitung der Funktion `berechnePrüfsumme` zu erzeugen, wurde sie mit konstant gehaltenen Parametern betrieben. Dies sollte dazu führen, dass das Ergebnis der Funktion, sowie die dafür benötigten Rechenoperationen immer gleich bleiben und somit eine konstante Laufzeit benötigen sollten. Um möglichst Nahe an das reale Anwendungsgebiet heranzukommen wird für die Zeitmessung die in das Smartcard Betriebssystem integrierte Zeitmesseinheit verwendet, da dies eine Vorraussetzung dieses Hardware/Software Codesign Flows ist. Beim Messen der Laufzeit stellt man aber fest, dass sich auf Grund geringer Unterschiede kleine Schwankungen der Messergebnisse ergeben. Diese lassen sich zum Beispiel durch den Startzustand des Smartcard Betriebssystems, die Kommunikation, die Erwärmung des Emulators, die Schwankungen der Betriebsspannung und weiteren Einflüssen erklären.

Für die Untersuchung der Annahme wurde eine Funktion `berechnePrüfsumme` gewählt, die eine Prüfsumme berechnet. Sie wird im Codeblock 5.3 dargestellt und berechnet die Prüfsumme, indem Zeichen für Zeichen aufsummiert wird. Als Ergebnis wird die Gesamtsumme in Form eines Integers zurückgegeben. Der Parameter `zeichenkette` stellt die Zeichenkette dar, über die die Prüfsumme berechnet wird.

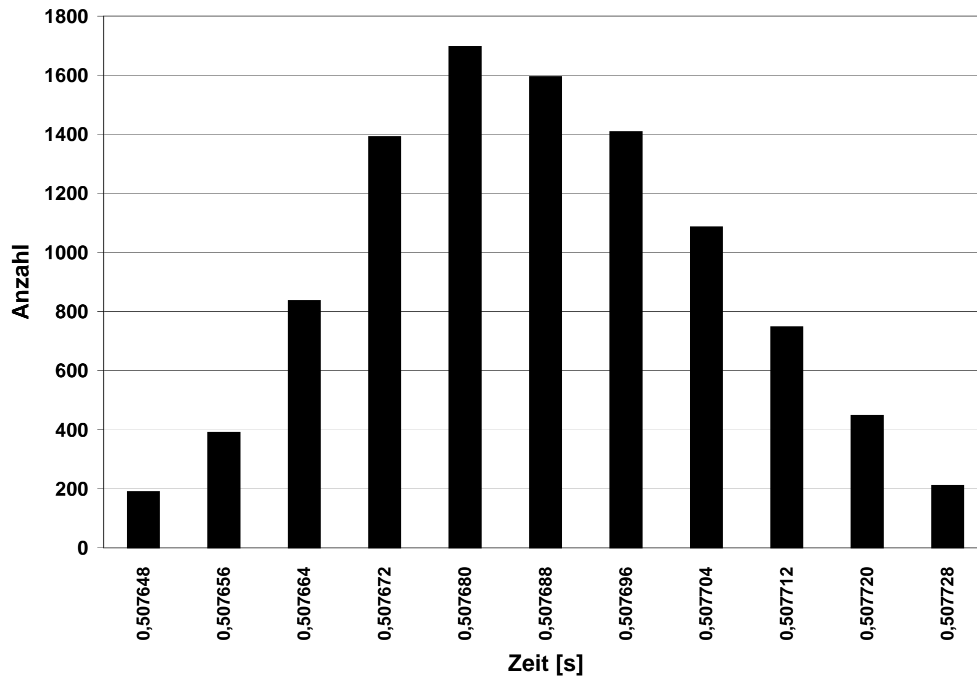

Abbildung 5.1: Verteilung der Messergebnisse bei einer konstanten Laufzeit von zirka 0,5 Sekunden

```
int berechnePrüfsumme(char *zeichenkette){
    int prüfsumme = 0;
    int zeichenkette_größe = strlen(zeichenkette);

    for(int idx = 0; idx < zeichenkette_größe ; idx++){
        prüfsumme = prüfsumme + zeichenkette[idx];
    }
    return(prüfsumme);
}
```

Codeblock 5.3: Prüfsummenberechnung

Wie man sieht, ist die Laufzeit nur von der Länge der Zeichenkette abhängig. Das Aufsummieren eines Zeichens, sowie der Schleifen-Overhead, benötigt dabei immer gleich viel Zeit. Da bei geringen Laufzeiten nur geringe Laufzeitunterschiede auftreten, musste eine entsprechend große Anzahl an Zeichen verglichen werden. Dies begründet sich dadurch, dass die Zeitmesseinheit nur eine Auflösung von 8 Mikrosekunden besitzt. Wird nun eine Laufzeit gemessen die zu klein ist, zum Beispiel 100 Mikrosekunden, so ergibt sich zwar auch eine Normalverteilung der Laufzeitergebnisse, allerdings sieht man hierbei die Schwankungen nur minimal, sodass der Eindruck entsteht, dass es sich um keine Normalverteilung handelt.

Es wurde angenommen, dass eine Laufzeit von zirka einer halben Sekunde ausreicht, damit die gemessenen Laufzeiten eine darstellbare Schwankung aufweisen. Dies entspricht

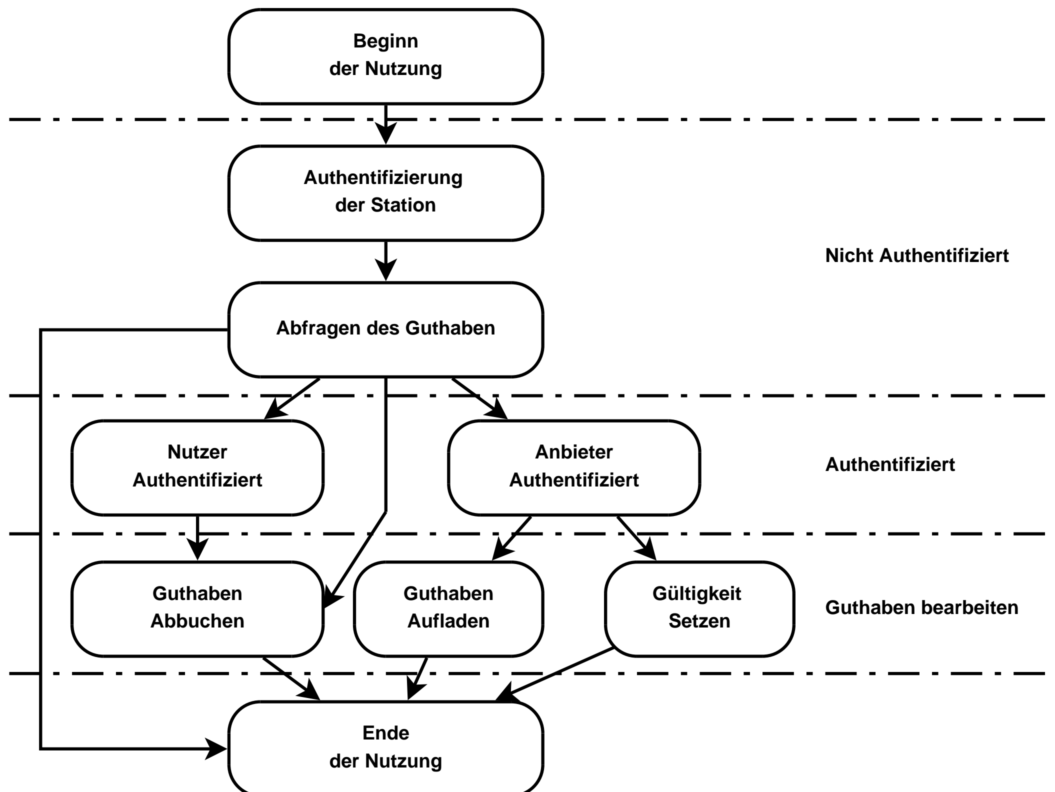

Abbildung 5.2: Programmabläufe der Prepaid-Kartenanwendung

zirka einer Anwendung der Funktion `berechnePrüfsumme` auf eine Zeichenkette mit 1000 Zeichen. Die Laufzeitmessung wurde 10000-mal durchgeführt, wodurch sich die in Abbildung 5.1 dargestellte Verteilung ergab. Wie man sieht, handelt es sich dabei um eine Normalverteilung. Dies wurde auch mathematisch unter zu Hilfenahme von SPSS[1] überprüft. Damit entspricht das Verhalten der Zeitmesseinheit bei einer Laufzeitmessung dem angenommenen Verhalten.

5.2 Beispiel

Dieses Unterkapitel zeigt die Anwendung des Hardware/Software Codesign Flows auf eine Beispielanwendung. Es unterteilt sich in drei Abschnitte. Der erste Abschnitt beschäftigt sich mit der Spezifikation der Beispielanwendung. Er beschreibt die Funktionalität und das Verhalten der Anwendung. Im zweiten Abschnitt wird die Implementation beschrieben. Dabei erfolgt eine Aufteilung in die einzelnen Funktionen, die Bestimmung der kritischen Bereiche und ihre Annotation. Der dritte Abschnitt beschreibt die Resultate der Laufzeitanalyse. Dafür wurden mehrere Implementationen der kritischen Bereiche vermessen und ihre Vorzüge erklärt.

[1]IBM SPSS Statistics http://www-01.ibm.com/software/de/analytics/spss/

5.2.1 Spezifikation der Beispielanwendung

Als Beispielanwendung wurde eine einfache Prepaid-Kartenanwendung gewählt. Diese entspricht im Großen und Ganzen einer Kreditkarte, die ähnlich wie eine Quickkarte zuvor mit einem Guthaben aufgeladen wird. Der Stand des Guthabens wird auf der Prepaid-Karte gespeichert. Dieses Guthaben kann danach verbraucht beziehungsweise abgebucht werden. Ist das Guthaben verbraucht, so muss die Karte wieder mit einem neuen Guthaben aufgeladen werden. Dieser Vorgang lässt sich praktisch beliebig wiederholen.

Soll Guthaben von der Prepaid-Karte abgebucht werden, so hängen die weiteren Aktivitäten von der Höhe des abzubuchenden Guthabens ab. Überschreitet das abzubuchende Guthaben dabei ein vorher festgelegtes Limit, so ist es notwendig, dass ein Sicherheitscode eingegeben wird. Anhand dieses Sicherheitscodes identifiziert sich der Nutzer. Schlägt die Authentifizierung fehl, so hat der Nutzer noch zwei weitere Versuche um den Sicherheitscode richtig einzugeben. Schlagen diese ebenfalls fehl, so wird die Karte als ungültig markiert und der Nutzer muss sich an den Anbieter wenden, um die Karte wieder zu aktivieren.

Die grundlegenden Ablaufpfade der Prepaid-Kartensoftware werden in der Abbildung 5.2 dargestellt. Jeder in der Abbildung dargestellte Block entspricht dabei einem Kommando, welches an die Prepaid-Karte gesendet werden muss. Die beiden Blöcke Beginn der Nutzung und Ende der Nutzung stellen keine Funktionen da. Sie definieren vielmehr einen einheitlichen Eintritts- und Austrittspunkt, an dem nicht systemspezifische Kommandos, wie zum Beispiel das ATR-Kommando, an die Prepaid-Karte gesendet werden können. Jeder Pfad innerhalb dieser beiden Blöcke muss vollständig durchlaufen werden, damit die korrekte Funktionalität der Karte sichergestellt ist.

5.2.2 Implementierung der Prepaid-Karten Anwendung

Bei der Implementierung der Prepaid-Kartenanwendung wurde darauf geachtet, dass sich ähnliche Funktionen zusammenfassen lassen. Dies hat den Vorteil, dass sich dadurch zum einen ein Fehler in einer Funktion schneller finden lässt, da die Funktion öfter verwendet wird und zum anderen, dass der Speicherplatz für identischen Programmcode eingespart wird. Wenn man sich die Prepaid-Kartenanwendung mit ihren sieben Kommandos ansieht, stellt man fest, dass die drei Authentifizierungskommandos eine ähnliche Funktionalität aufweisen. Sie können durch eine Funktion implementiert werden, bei der die auszuführende Funktion mit geprüften Parametern aufgerufen wird. Eine weitere Ähnlichkeit findet sich bei der Auflade- und Abbuchfunktion.

Sie unterscheiden sich nur darin, dass beim Aufladevorgang ein negatives Guthaben abgebucht wird beziehungsweise ein positives Guthaben aufgebucht wird. Wenn beim Ausführen des Abbuch-Kommandos überprüft wird, dass der verwendete Guthabenbetrag immer negativ ist und beim Aufladen immer überprüft wird, dass der Guthabenbetrag positiv ist, so könnte man eine `ändereGuthabenstand` Funktion einsetzen, die den Guthabenstand der Funktion um einen Wert erhöht oder verringert. Dadurch lässt sich im Grunde die Prepaid-Kartenanwendung durch die folgenden fünf Funktionen implementieren:

- `setzeGültigkeit`
- `prüfeGültigkeit`

- `prüfeAuthentifizierung`
- `ändereGuthabenstand`
- `abfragenGuthabenstand`

Die Überprüfung der Funktionsparameter erfolgt dabei vor dem Aufrufen der Funktion. Die Prepaid-Kartenanwendung benötigt zwei unterschiedliche Speichertypen. Zum einen wird der EEPROM Speicherplatz zum Speichern nicht flüchtiger Daten verwendet, wie zum Beispiel den Stand des Guthabens oder die Gültigkeit der Prepaid-Karte. Zum anderen wird der RAM Speicherplatz zum Speicher von flüchtigen Informationen verwendet. Er speichert zum Beispiel, ob sich der Anwender oder der Anbieter erfolgreich authentifiziert hat. Diese Information soll beim nächsten Start des Prepaid-Karten Systems natürlich verloren gehen.

Die Gültigkeit der Prepaid-Karten wird im Anwendungsgültigkeitsbyte gespeichert. Dieses befindet sich im EEPROM und kann nur über die Funktion `setzeGültigkeit` verändert werden. Will man überprüfen, ob eine Prepaid-Karte gültig ist, so muss die `prüfeGültigkeit` Funktion verwendet werden. Diese greift direkt auf das Anwendungsgültigkeitsbyte zu und fragt den Zustand ab. Die zurückgelieferte Antwort ist dabei ein boolescher Wert.

Die Funktion `prüfeAuthentifizierung` überprüft ob der im EEPROM gespeicherte Schlüssel identisch ist mit dem, welcher der Funktion als Parameter übergeben wurde. Die Unterscheidung zwischen dem Schlüssel des Anwenders, des Anbieters oder der Station erfolgt durch einen zweiten Parameter. Dieser gibt an, welcher Schlüssel zu verwenden ist und wie oft dieser falsch eingegeben werden darf. Danach erfolgt ein Vergleich der beiden Schlüssel. Sind sie identisch, so wird die entsprechende Authentifizierung in das Authentifiziertbyte geschrieben und die Anzahl der Fehlversuche wird gelöscht. Ist sie nicht identisch, so wird überprüft wie oft das Passwort falsch eingegeben wurde. Wird dabei die maximal zulässige Anzahl an Versuchen überschritten, so wird die Prepaid-Karte als nicht gültig markiert. Dies erfolgt über die `setzeGültigkeit` Funktion.

Die beiden Funktionen `ändereGuthabenstand` und `abfragenGuthabenstand` greifen auf die Guthabenbytes zu. Dabei dient die Funktion `abfragenGuthabenstand` nur zum Auslesen des aktuellen Guthabens. Die Funktion `ändereGuthabenstand` ändert das Guthaben, das auf der Prepaid-Karte gespeichert ist. Je nach dem, ob ein Guthaben von der Prepaid-Karte abgebucht oder aufgeladen werden soll, muss das Authentifiziertbyte entsprechend gesetzt sein. Dabei reicht eine Authentifizierung als Anwender nur zum Abbuchen, wohingegen der Anbieter nur Guthaben aufladen kann.

Die Prepaid-Kartenanwendung verfügt über sieben unterschiedliche Kommandos, die aus diesen fünf Funktionen aufgebaut sind. Die Kommandos sind im Grunde eine Schnittstelle, die die Prepaid-Kartenanwendung mit der Station verbindet. Man kann sich die Implementierung der Kommandos wie eine switch case Anweisung vorstellen, bei der die einzelnen Funktionen anhand der Kommandocodes aufgerufen werden. Ein Kommando kann dabei Parameter enthalten oder nicht. Ein Beispiel für ein Kommando ohne Parameter ist die Abfrage des Guthabens und ein Kommando mit Parameter wäre zum Beispiel eine Änderung des Guthabenstands, da hier ein Wert angegeben werden muss, um den sich das Guthaben ändert. Die Überprüfung der Parameter eines Kommandos erfolgt direkt nach dem Aufruf des Kommandos. Dabei findet als erstes eine Prüfung der Gültigkeit

der Prepaid-Karte statt. Schlägt diese Überprüfung fehl, so endet dass Kommando, danach werden die angegeben Parameter überprüft und an die im Kommando aufgerufene Funktion übergeben.

Zeitkritische Funktionen der Prepaid-Karten Anwendung

Die für die Implementierung notwendigen Funktionen lassen sich im Bezug auf ihre Laufzeitverhalten in zwei Gruppen einteilen, nämlich in die einen die ein problematisches Laufzeitverhalten aufweisen und in die die ein unbedenkliches Laufzeitverhalten aufweisen. Dabei wird die Entscheidung, in welche der beiden Gruppen eine Funktion fällt, durch ihre Aufgabe entschieden. Zum Beispiel ist eine Funktion, die ein eingegebenes Passwort überprüft, mit Sicherheit als problematisch einzustufen. Im Gegensatz dazu, ist eine Funktion die nur den Guthabenstand verändert und dafür das Authentifizierungsbyte überprüft als unbedenklich einzustufen.

Wie das Beispiel zeigt, kommt es hierbei weniger darauf an, welche Funktion eine Aufgabe umsetzt, als auf die Tatsache, auf welche Daten sie zugreifen muss, um die Aufgabe bewältigen zu können. Dabei unterscheidet man zwischen sensiblen und nicht sensiblen Daten. Sensible Daten sind personenbezogene und schutzwürdige Daten wie zum Beispiel, der Pin einer Bankomatkarte, ein Passwort, ein auf der Karte gespeicherter Authentifizierungscode. Erfolgt ein Zugriff auf sensible Daten, so empfiehlt es sich, diese Funktion für eine Laufzeitanalyse vorzumerken.

Ein weiteres Unterscheidungsmerkmal ist, wie auf die sensiblen Daten zugegriffen wird, damit ist gemeint, ob auf sensible Daten als ganzes oder nur teilweise zugegriffen wird. Weiters muss beachtet werden, warum auf diese Daten zugegriffen wird. Hierbei unterscheidet man zwischen einer Funktion die aktiv mit diesen Daten arbeitet und routinemäßigen Zugriffen, die zum Beispiel eine Prüfsumme über Speicherbereiche berechnet um sicher zu stellen, das keine Daten manipuliert wurden.

Wie man sieht, muss nur eine Funktion der Prepaid-Kartenanwendung, als problematisch eingestuft werden. Dies ist die Funktion `prüfeAuthentifizierung`, die den eingegeben Authentifizierungscode mit dem auf der Karte gespeicherten Authentifizierungscode vergleicht. Die anderen Funktionen `setzeGültigkeit`, `prüfeGültigkeit`, `ändereGuthabenstand`, `abfragenGuthabenstand` sind unbedenklich und müssen nicht explizit untersucht werden. Da diese Funktion mehrfach mit unterschiedlichen Parametern eingesetzt wird, fand die Annotierung in den Implementierungen der Kommandos statt. Dadurch wird zwar der Aufruf der Funktion ebenfalls mitgemessen, dies ist aber gänzlich unproblematisch, da der Aufruf einer Funktion in konstanter Zeit erfolgt und somit keine Veränderung des Laufzeitverhaltens zu befürchten ist.

5.2.3 Messergebnisse

Um zu zeigen wie sich unterschiedliche Implementierungen auf das Laufzeitverhalten der `prüfeAuthentifizierung` Funktion auswirken, wurden fünf Versionen implementiert. Dabei wird als erstes von einer naiven Implementierung ausgegangen die immer weiter verbessert wird, bis sie schlussendlich in konstanter Zeit arbeitet. Die in den folgenden fünf Abschnitten gezeigten Funktionen stellen die Funktionsweise der `prüfeAuthentifizierung` Funktion dar.

Dabei wird vorausgesetzt, dass der geheime Schlüssel (`geheime_schlüssel`) und der Eingabe Schlüssel (`eingabe_schlüssel`) immer existieren. Zudem wird angenommen dass sie dieselbe Länge aufweisen. Als Ergebnis der Funktion wird ein Wahrheitswert zurückgeliefert, der in der `prüfeAuthentifizierung` Funktion in einem Byte gespeichert wird. Dadurch ist eine simple Darstellung der Überprüfungsfunktion möglich.

Für die Bestimmung der Laufzeitergebnisse wird ein dezimaler, vierstelliger Authentifizierungscode verwendet. Dieser muss vollständig eingegeben werden. Er weist 10000 unterschiedliche Kombinationsmöglichkeiten auf, angefangen von '0000' bis hin zu '9999'. Das bedeutet, dass jede der fünf Authentifizierungsprüffunktionen maximal vier Zeichen vergleichen muss. Als gültiger Authentifizierungscode wurde die Kombination '1234' gewählt.

Der Grund für die Wahl eines kurzen Authentifizierungscodes ist eine übersichtliche Darstellung des Laufzeitverhaltens. Dieses zeigt jede Kombinationsmöglichkeit des Authentifizierungscodes in Verbindung mit der dazugehörigen Laufzeit. Die Länge des Authentifizierungscodes hat keinen Einfluss auf das darzustellende Laufzeitverhalten, da sich dieses auch schon bei einer geringen Anzahl an Stellen zeigt.

Die Laufzeitanalyse erfolgt aber nur auf einem Bruchteil der 10000 Kombinationsmöglichkeiten, da die Anwendungsfälle selten alle Kombinationsmöglichkeiten eines Systems oder einer Funktion abdecken. Es wurden 100 Anwendungsfälle ausgewählt, die unterschiedliche Authentifizierungscodes verwenden. Dabei wurde berücksichtigt, dass alle problematischen Stellen der Authentifizierungsprüffunktion mehrfach abgedeckt sind. Die problematischen Stellen beim Vergleichen von zwei Zeichenketten lassen sich in zwei Gruppen einteilen. Die eine Gruppe ist die **Anzahl der übereinstimmenden Zeichen** und die andere Gruppe ist die **Länge der zusammenhängenden, übereinstimmenden Zeichen**.

Es gibt je fünf unterschiedliche Längen, angefangen von null übereinstimmenden Stellen bis hin zu vier übereinstimmenden Stellen. Der Zählvorgang der Übereinstimmungen erfolgt dabei ab dem ersten Zeichen. Als Beispiel hierfür weist der eingegebene Authentifizierungscode '0234' null Übereinstimmungen mit dem gültigen Authentifizierungscode auf, da sich das erste Zeichen unterscheidet. Wird stattdessen der Authentifizierungscode '1274' eingegeben, so ergeben sich zwei Übereinstimmungen, da die ersten beiden Stellen übereinstimmen. Vier Übereinstimmungen bedeuten dabei, dass alle vier Stellen des Authentifizierungscodes richtig sind.

Dasselbe trifft auch auf die Anzahl der übereinstimmenden Zeichen zu. Sie unterscheiden sich nur dadurch von einander, dass bei der Anzahl an Übereinstimmungen, ein falsches Zeichen nicht zum Abbruch der Folge an Übereinstimmungen führt. Ein Beispiel hierfür ist der eingegebene Authentifizierungscode '0234'. Dieser weist bei der Länge der zusammenhängenden Zeichen null Übereinstimmungen auf, allerdings ist die Anzahl der Übereinstimmungen drei. Der Grund dafür ist, dass alle Zeichen, außer dem ersten Zeichen, übereinstimmen. Wie man sieht ist die Anzahl an Übereinstimmungen eine abgeschwächte Form der Länge der zusammenhängenden Zeichen. Die Tabelle 5.2 stellt mehrere unterschiedliche Kombinationen dar. Als gültiger Authentifizierungscode für die Tabelle 5.2 gilt ebenfalls die Kombination '1234'.

Insgesamt werden für jede Länge und Anzahl an Übereinstimmungen jeweils zehn Anwendungsfälle umgesetzt, was die vorhin erwähnten 100 Anwendungsfälle ergibt. Dadurch soll sichergestellt werden, dass beim Überprüfen des Laufzeitverhaltens die entsprechenden Stellen bei der Überprüfung enthalten sind. Wahlweise kann man noch zusätzlich zufällig

Eingegebener Authentifizierungscode	'1239'	'1204'	'0034'	'2341'	'1000'
Länge an Übereinstimmungen	3	2	0	0	1
Anzahl an Übereinstimmungen	3	3	2	0	1

Tabelle 5.2: Vergleich der Länge und Anzahl an übereinstimmenden Zeichen beim gültigen Authentifizierungscode '1234'

bestimmte Authentifizierungscodes überprüfen, um die Menge der Daten zu steigern. Dies wird für die Untersuchung des Laufzeitverhaltens aber nicht umgesetzt.

Naive Implementierung der prüfeAuthentifizierungs-Funktion

Die erste Implementierung stellt eine naive Überprüfung des eingegeben Schlüssels dar. Dabei wird der Authentifizierungscode direkt mit dem eingegeben Schlüssel verglichen. Der Vergleich erfolgt dabei Zeichen für Zeichen. Stimmen dabei zwei Zeichen nicht überein, so wird der `prüfer` auf den Wert `falsch` gesetzt und der Vergleich des eingegebenen Schlüssels mit dem Authentifizierungscode endet. Stimmen hingegen die beiden Zeichen überein, so werden die nächsten beiden Zeichen verglichen. Dieser Vorgang wird so lange wiederholt, bis alle Zeichen erfolgreich überprüft wurden oder zwei Zeichen nicht übereinstimmen.

```
bool prüfeAuthentifizierung1(char *eingabe_schlüssel){
    bool prüfer=true;//Der Prüfer wird auf richtig initalisiert
    int schlüssel_größe=strlen(eingabe_schlüssel);

    for(int idx=0;idx<schlüssel_größe;idx++){
        if(eingabe_schlüssel[idx]!=geheime_schlüssel[idx]){
            prüfer=false;
        }
        if(prüfer==false) break;
    }
    return(prüfer);//Rückgabe des Prüfers
}
```

Codeblock 5.4: Naive zeitsparende Überprüfung der Authentifizierung

Die Abbildung 5.3 zeigt das Laufzeitverhalten der Funktion, die im Codeblock 5.4 dargestellt ist. Sie zeigt die Laufzeiten aller möglichen Kombinationen des Authentifizierungscodes. Dabei sind in horizontaler Richtung alle möglichen 10000 Authentifizierungscodes aufgetragen, beginnend bei '0000' und endend bei '9999'. In vertikaler Richtung sind die dafür benötigten Laufzeiten in Sekunden aufgetragen. Wie man anhand der Stufenform klar erkennt, weist diese Funktion einen Zusammenhang zwischen dem eingegebenen Authentifizierungscode und der Laufzeit auf. Dabei benötigt die Funktion für jedes richtige Zeichen um 416 Mikrosekunden mehr.

Die 100 Anwendungsfälle wiesen dieses Verhalten ebenfalls auf, wie die Abbildung 5.4 zeigt. Die automatische Auswertung, durch die Auswertesoftware, funktionierte daher problemlos. Sie erkannte ein Laufzeitproblem und teilte die Laufzeitergebnisse in fünf

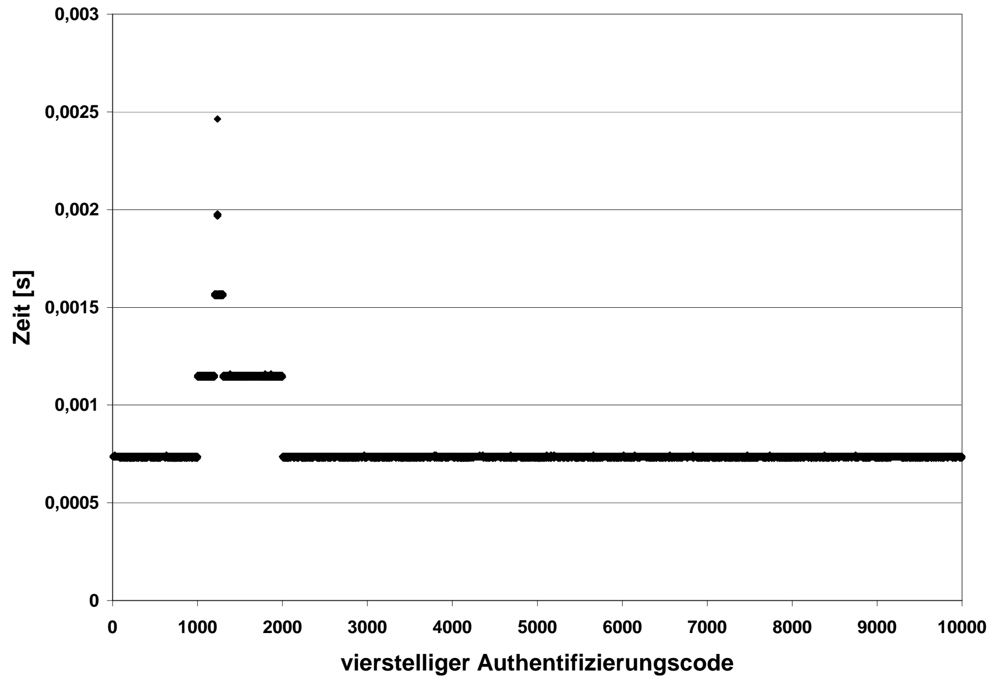

Abbildung 5.3: Laufzeitverhalten der naiven Implementierung

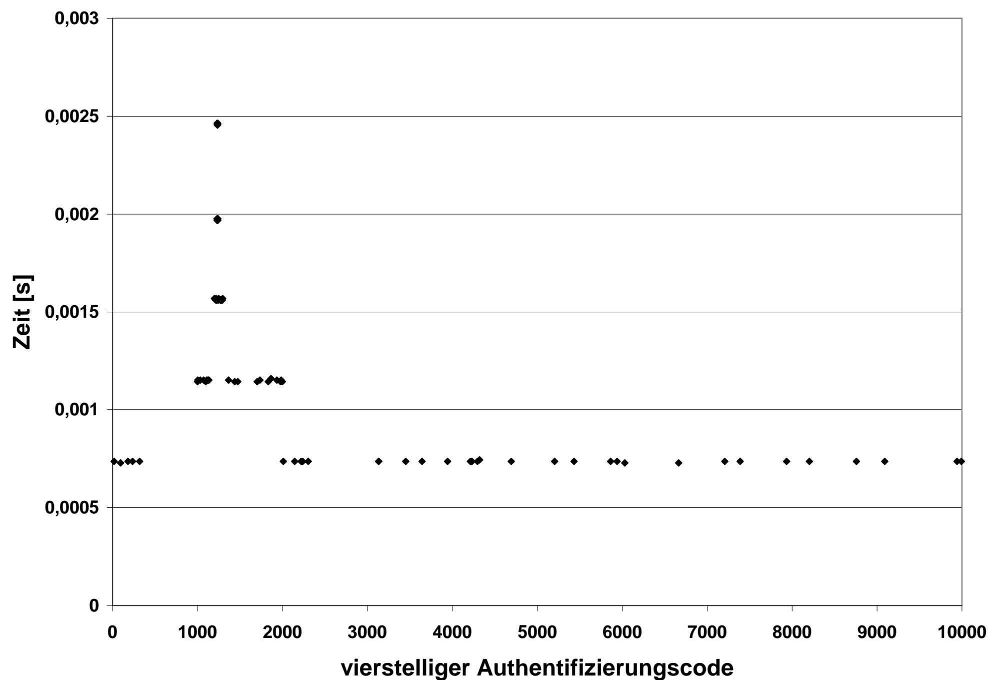

Abbildung 5.4: Laufzeitergebnisse der 100 Anwendungsfälle bei der naiven Implementierung

Gruppen ein. Diese fünf Gruppen entsprechen dabei der Länge der zusammenhängenden und übereinstimmenden Zeichen.

Das Ergebnis der Laufzeitanalyse wird in der Tabelle 5.3 dargestellt. Es beinhaltet den Mittelwert, die Standardabweichung sowie das Minimum und das Maximum. Dieser Werte werden für die Gruppen der 100 Anwendungsfälle angegeben. Weiters werden diese Werte auch für die fünf einzelnen Gruppierungen angegeben, die das Ergebnis der Laufzeitanalyse darstellen.

	Minimum	**Maximum**	**Mittelwert**	**Standardabweichung**
	[μs]	[μs]	[μs]	[μs]
Gesamt	728	2464	139,1	663,5
Gruppe 0	728	744	735,5	2,7
Gruppe 1	1144	1160	1149,1	4,8
Gruppe 2	1560	1568	1562,3	3,8
Gruppe 3	1968	1976	1973,1	4,0
Gruppe 4	2456	2464	2461,2	3,9

Tabelle 5.3: Ergebnisse der Laufzeitanalyse der naiven Implementierung anhand der Anwendungsfälle

Zustandsabhängige Implementierung der prüfeAuthentifizierungs-Funktion

Die zweite Implementierung der Überprüfungsfunktion unterscheidet sich im Wesentlichen nur durch ein Detail von der ersten Implementierung, die im Codeblock 5.4 dargestellt ist. Sie vergleicht den eingegebenen Schlüssel mit dem Authentifizierungscode zur Gänze. Dabei wird zuerst die Variable `prüfer` auf den Wert `WAHR` gesetzt. Diese speichert den Zustand des Vergleichs der beiden Zeichenketten. Die beiden Zeichenketten werden, wie bei der ersten Implementierung, Zeichen für Zeichen verglichen. Tritt bei einem Zeichen keine Übereinstimmung auf, so wird die Variable `prüfer` auf den Wert `FALSCH` gesetzt. Damit steht das Ergebnis des Vergleichs fest, da die Variable `prüfer` den Wert `WAHR` nicht mehr annehmen kann. Der Vergleich der beiden Zeichenketten endet damit allerdings nicht. Der Vergleich wird erst abgebrochen wenn die beiden Zeichenketten vollständig miteinander verglichen wurden.

```
bool prüfeAuthentifizierung2(char *eingabe_schlüssel){
    bool prüfer=true;
    int schlüssel_größe=strlen(eingabe_schlüssel);

    for(int idx=0;idx<schlüssel_größe;idx++){
        if(eingabe_schlüssel[idx]!=geheime_schlüssel[idx]){
            prüfer=false;
        }
    }
    return(prüfer);
}
```

Codeblock 5.5: Zustandsabhängige Überprüfung der Authentifizierung

In Abbildung 5.5 wird das Laufzeitverhalten der im Codeblock 5.5 dargestellten Funktion gezeigt. Hierbei ist ein Zusammenhang zwischen der benötigten Laufzeit und dem eingegebenen Schlüssel nicht mehr ganz so klar erkennbar. Betrachtet man den Bereich zwischen dem Authentifizierungscode '1000' und '2000', so erkennt man, dass dieser Bereich im Durchschnitt eine etwas kürzer Laufzeit aufweist, als die restlichen Authentifizierungscodes. Betrachtet man des Weiteren den Bereich zwischen '1200' und '1300' so erkennt man ebenfalls, dass die Laufzeit im Durchschnitt etwas kürzer ist als bei den restlichen Authentifizierungscodes. Dies lässt sich natürlich weiterführen bis man den gültigen Authentifizierungscode gefunden hat. Das Laufzeitverhalten der Funktion weist einen Zusammenhang zwischen der Anzahl an übereinstimmenden Stellen und der dafür benötigten Laufzeit auf. Betrachtet man den Codeblock 5.5 in Verbindung mit der Abbildung 5.5, so erkennt man, dass ein richtiges Zeichen des Authentifizierungscode die Laufzeit um zirka 32 Mikrosekunden verkürzt.

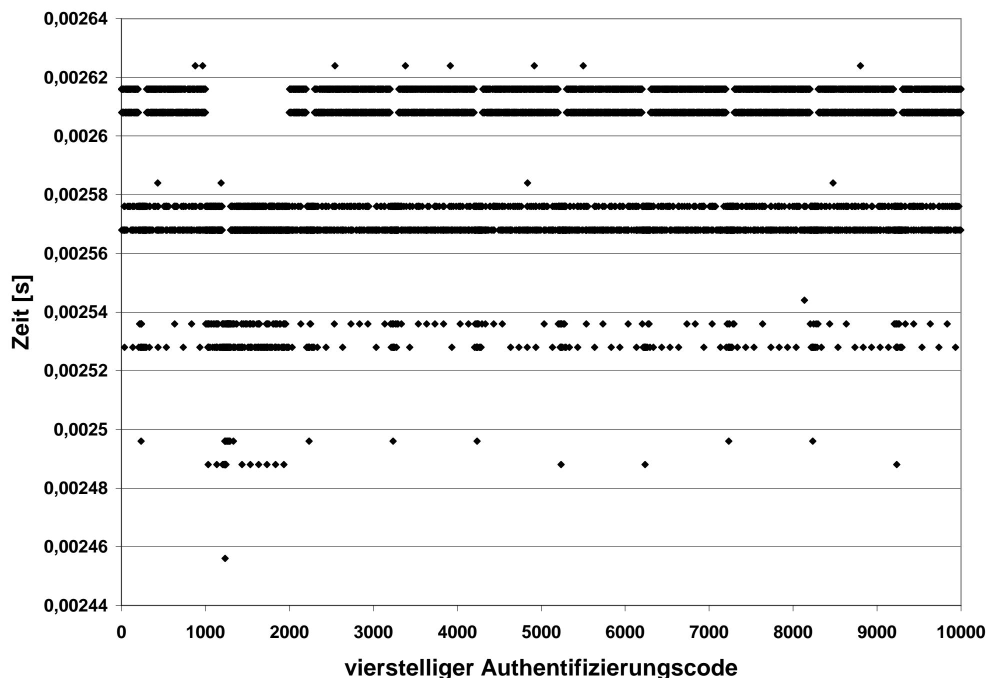

Abbildung 5.5: Laufzeitverhalten der zustandsabhängigen Implementierung

Die 100 Anwendungsfälle untersuchen wiederum nur einen kleinen Teil der 10000 möglichen Eingabekombinationen. Die Abbildung 5.6 zeigt die Laufzeitergebnisse der Anwendungsfälle. Wie man sieht verteilen sich die Anwendungsfälle auf den gesamten Bereich des Authentifizierungscodes. Der Bereich zwischen '1000' und '2000' wird dabei häufiger durch die Anwendungsfälle untersucht, da sich in diesem Bereich der gültige Authentifizierungscode befindet.

Wenn man die Abbildung 5.6 der Anwendungsfälle sieht, ohne zuvor die Abbildung 5.5 zu sehen, so ist das Erkennen der Laufzeitabhängigkeit nicht mehr so einfach. Die Auswertesoftware erkannte auch hierbei ein Problem mit dem Laufzeitverhalten. Es kategorisierte die Laufzeitergebnisse ebenfalls in fünf Gruppen, die diesmal von der Anzahl an

Übereinstimmungen abhängt. Die Tabelle 5.4 zeigt, die Ergebnisse der Auswertesoftware.

	Minimum	**Maximum**	**Mittelwert**	**Standardabweichung**
	[μs]	[μs]	[μs]	[μs]
Gesamt	2456	2616	2528,2	52,1
Gruppe 0	2456	2464	2458,8	3,9
Gruppe 1	2488	2496	2490,9	3,9
Gruppe 2	2528	2536	2530,9	3,9
Gruppe 3	2568	2576	2571,2	4,0
Gruppe 4	2608	2616	2608,5	2,0

Tabelle 5.4: Ergebnisse der Laufzeitanalyse der zustandsabhängigen Implementierung anhand der Anwendungsfälle

Bedachte Implementierung der prüfeAuthentifizierungs-Funktion

Die dritte Implementation ist sehr ähnlich wie die zweite Implementierung, die im Codeblock 5.5 dargestellt ist. Sie unterscheidet sich nur in zwei Details von der zweiten Implementierung. Zum einen wird der Else-Zweig der If-Abfrage verwendet und zum anderen wird der Zustand der Variable `prüfer` im Else-Zweig durch eine Oder-Verknüpfung verändert. Durch die Oder-Verknüpfung verändert sich der logische Zustand der Variable `prüfer` im Else-Zweig nicht, dafür findet nun aber im If- und Else-Zweig eine Zuweisung statt. Dadurch unterscheiden sich der If- und Else-Zweig nur noch durch eine einzige Oder-Verknüpfung die den Zustand der Variable nicht ändert.

```
bool prüfeAuthentifizierung3(char *eingabe_schlüssel){
    bool prüfer=true;
    int schlüssel_größe=strlen(eingabe_schlüssel);

    for(int idx=0;idx<schlüssel_größe;idx++){
        if(eingabe_schlüssel[idx]!=geheime_schlüssel[idx]){
            prüfer=false;
        }else{
            prüfer|=true;
        }
    }
    return(prüfer);
}
```

Codeblock 5.6: Bedachte Überprüfung der Authentifizierung

Die Abbildung 5.7 zeigt das Laufzeitverhalten der Funktion, die im Codeblock 5.6 dargestellt ist. Das Laufzeitverhalten dieser Funktion sieht wie das invertierte Laufzeitverhalten, der in Abbildung 5.5 gezeigte Funktion aus. Die Ähnlichkeit der beiden Abbildungen wird besonders deutlich, wenn man den Wertebereich zwischen '1000' und '2000' der Authentifizierungscodes betrachtet.

Es unterscheidet sich allerdings dadurch, dass die Funktion für die Überprüfung richtiger Zeichen um zirka 24 Mirkosekunden mehr Zeit benötigt. Dieser Unterschied entsteht

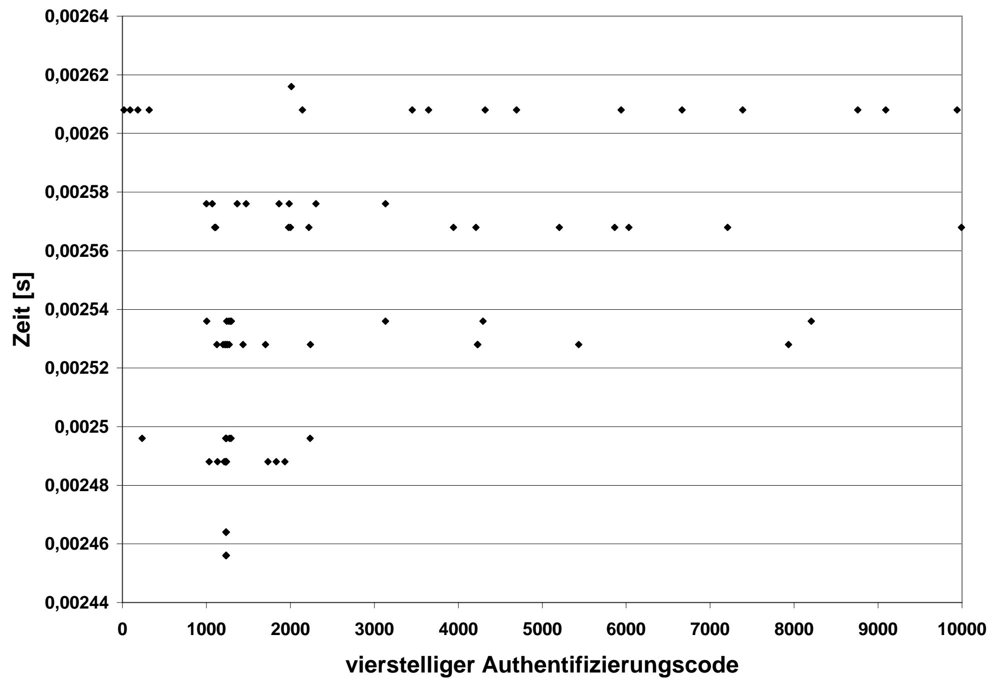

Abbildung 5.6: Laufzeitergebnisse der 100 Anwendungsfälle bei der zustandsabhängigen Implementierung

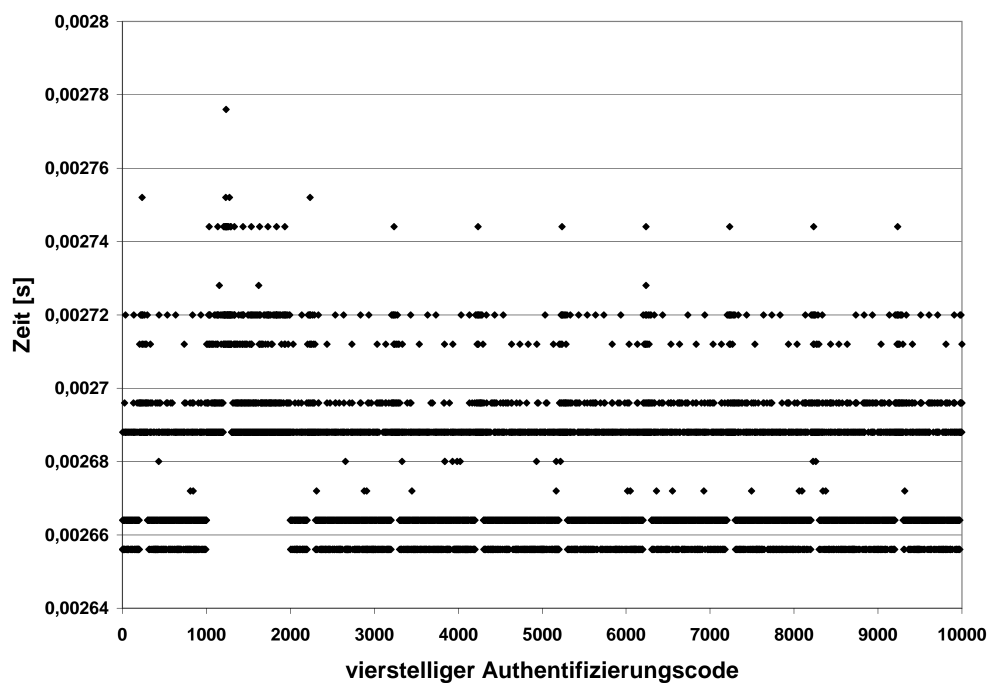

Abbildung 5.7: Laufzeitverhalten der bedachten Implementierung

durch die Verwendung des Else-Zweiges in Verbindung mit der Oder-Verknüpftung und einer Zuweisung. Würde hier in beiden Zweigen eine Verknüpfung in Verbindung mit Zuweisung erfolgen, so kann man nur noch den Unterschied zwischen If- und Else-Zweig feststellen. Die Abhängigkeit des Laufzeitverhaltenes bleibt weiterhin erkennbar.

Die 100 Anwendungsfälle zeigten eine schwache Abhängigkeit der Daten von der Laufzeit. Dies führt dazu, dass die Auswertesoftware eine Abhängigkeit erkennt und eine Zuordnung in fünf Gruppen durchführt. Die Ergebnisse werden in Tabelle 5.5 gezeigt. Bei der Zuordnung ist der Auswertesoftware allerdings ein Fehler unterlaufen. Dadurch wurde ein Laufzeitergebnis einer falschen Gruppe zugeordnet. Dies dürfte auf Grund der eng aneinander liegenden Laufzeitergebnisse aufgetreten sein. Diese befinden sich teilweise nur ein Timerinterintervall voneinander entfernt. Dadurch lässt sich bei der Gruppierung anhand der Mittelwerte nicht exakt sagen, in welche Gruppe ein Laufzeitergebnis fällt.

Dies führt zu einer Einschränkung bei der Nutzung der Auswertesoftware, da es möglich ist, dass zwei Gruppen nur ein Timerinterintervall voneinander entfernt sind. In diesem Bereich versagt die Auswertesoftware. Eine Lösung für dieses Problem stellt die Erhöhung der Genauigkeit der Zeitmesseinheit dar.

	Minimum	**Maximum**	**Mittelwert**	**Standardabweichung**
	[μs]	[μs]	[μs]	[μs]
Gesamt	2656	2784	2721,4	40,0
Gruppe 0	2656	2664	2661,5	3,8
Gruppe 1	2688	2696	2690,0	3,5
Gruppe 2	2712	2720	2717,0	3,8
Gruppe 3	2744	2760	2746,1	4,2
Gruppe 4	2776	2784	2779,2	3,9

Tabelle 5.5: Ergebnisse der Laufzeitanalyse der bedachten Implementierung anhand der Anwendungsfälle

Dual-Rail Implementierung der prüfeAuthentifizierungs-Funktion

Die vierte Implementierung nutzt das Dual Rail Prinzip, welches von der Power Attacke bekannt ist. Dabei werden jeweils alle logischen Zustände in normaler und komplementärer Form umgesetzt. Das bedeutet, dass bei einer If/Else-Anweisung einmal der If-Zweig und einmal der Else-Zweig durchlaufen wird.

Um dies in der Implementierung sicherzustellen, wurde der Vergleich beider Zeichen normal und komplementär ausgeführt. Das bedeutet, es wurde ein Mal verglichen ob die beiden Zeichen gleich sind und das andere Mal wurde verglichen ob die beiden Zeichen nicht gleich sind. Dadurch kann ausgeschlossen werden, dass ein Unterschied beim Vergleichen von zwei Zeichen auftritt, der auf Grund des Vergleiches entsteht.

Weiters wird unabhängig vom Zweig, jeweils eine Zuweisung und eine logische Verknüpfung abgearbeitet. Diese sind innerhalb einer If/Else-Anweisung immer gleich. Dadurch wird sicher gestellt, dass jeder Zweige dieselben Operationen verwendet. Der einzige Unterschied zwischen der normalen und komplementären Variante des Vergleichs ist, dass hierbei einmal eine Und-Verknüpfung und einmal eine Oder-Verknüpfung eingesetzt wird.

Dies stellt allerdings kein Problem dar, da bei jedem Durchlauf immer eine Und- sowie eine Oder-Verknüpfung abgearbeitet wird.

```
bool prüfeAuthentifizierung4(char *eingabe_schlüssel){
    bool richtigprüfer=true;
    bool falschprüfer=false;
    int schlüssel_größe=strlen(eingabe_schlüssel);

    for(int idx=0;idx<schlüssel_größe;idx++){
        if(eingabe_schlüssel[idx]!=geheime_schlüssel[idx]){
            richtigprüfer&=false;
        }else{
            richtigprüfer&=true;
        }
        if(eingabe_schlüssel[idx]==geheime_schlüssel[idx]){
            falschprüfer|=false;
        }else{
            falschprüfer|=true;
        }
    }
    return(richtigprüfer & (!falschprüfer));
}
```

Codeblock 5.7: Überprüfen der Authentifizierung mittels Dual Rail-Prinzip

Die Abbildung 5.8 zeigt das Laufzeitverhalten der Funktion, die im Codeblock 5.7 dargestellt ist. Hierbei sieht man, dass die Dual-Rail Implementierung eine annähernd konstante Laufzeit hervorruft. Es ist kein Zusammenhang zwischen den Laufzeitergebnissen und dem Authentifizierungscode ersichtlich. Weiters befinden sich die Messergebnisse in der Nähe der Empfindlichkeit der Zeitmesseinheit, wodurch eine genauere Untersuchung nicht möglich ist.

Die 100 Anwendungsfälle zeigen bei dieser Funktion keinen erkennbaren Zusammenhang. Deshalb kann die Auswertesoftware keine Gruppierung durchführen. Dies entspricht dem erwarteten Verhalten einer Funktion, die eine konstante Ausführzeit aufweisen soll. Die Tabelle 5.6 zeigt daher ausschließlich die Ergebnisse der gesamten Anwendungsfälle.

	Minimum	**Maximum**	**Mittelwert**	**Standardabweichung**
	[μs]	[μs]	[μs]	[μs]
Gesamt	2816	2832	2825,0	4,8

Tabelle 5.6: Ergebnisse der Laufzeitanalyse der Dual Rail Implementierung anhand der Anwendungsfälle

Zustandsunabhängige Implementierung der prüfeAuthentifizierungs-Funktion

Die fünfte Implementierung weist ebenfalls eine konstante Laufzeit für die Überprüfung von zwei Zeichenketten einer mit bestimmter Länge auf. Diese Implementierung ist keine

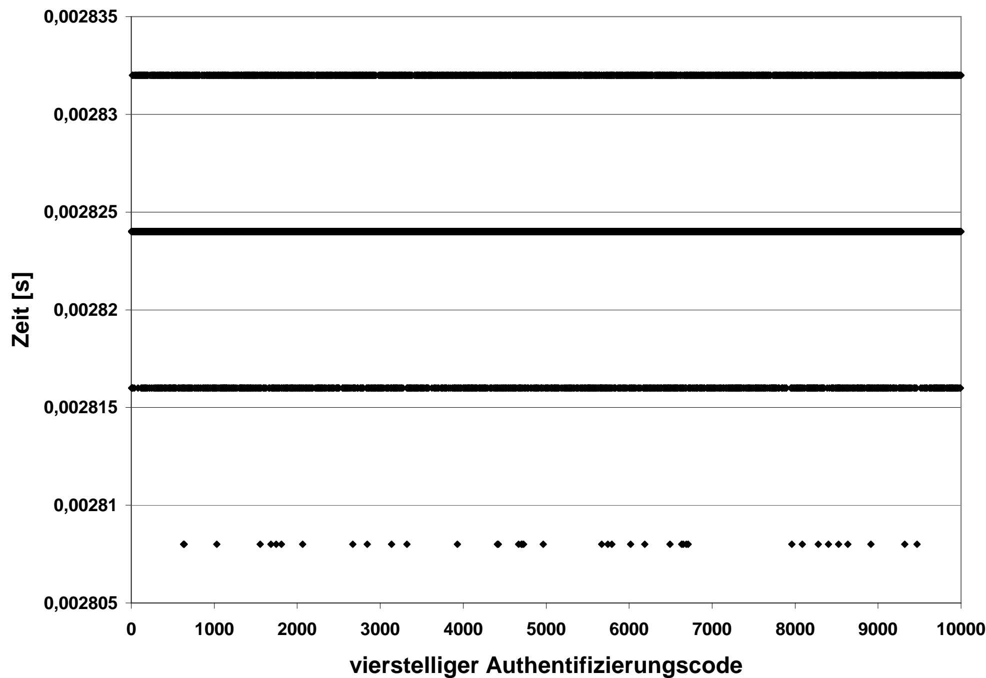

Abbildung 5.8: Laufzeitverhalten der Dual Rail Implementierung

direkte Weiterentwicklung der vorangegangen Implementierungen. Sie nützt viel mehr die Eigenschaften eines logischen Vergleichs aus, sodass das Ergebnis aus dem Vergleich direkt mit der Variable `prüfer` Verknüpft wird und somit keine Verzweigung im Programmcode entsteht. Diese Implementierung bietet auch noch den Vorteil, dass sie schneller ist als die vorangegangene Implementierung, da diese nur einen Vergleich und eine Zuweisung abgearbeiten muss.

```
bool prüfeAuthentifizierung5(char *eingabe_schlüssel){
    bool prüfer=true;
    int schlüssel_größe=strlen(eingabe_schlüssel);

    for(int idx=0;idx<schlüssel_größe;idx++){
        prüfer&=(eingabe_schlüssel[idx]==geheime_schlüssel[idx]);
    }
    return(prüfer);
}
```

Codeblock 5.8: Zustandsunabhängige Überprüfung der Authentifizierung

Die Abbildung 5.9 zeigt das Laufzeitverhalten der Funktion, die im Codeblock 5.8 dargestellt ist. Diese weist ebenfalls eine annähernd konstante Laufzeit auf, die in der Größenordnung der Empfindlichkeit der Zeitmesseinheit ist. Bei genauerer Betrachtung stellt man jedoch fest, dass sich die Laufzeitergebnisse im Bereich zwischen den Authentifizierungscodes '1000' und '2000' verkürzen. Dies fällt besonders bei den kürzesten und längsten Laufzeiten auf. Der Grund dafür dürfte der Compiler sein, der beim Übersetzen

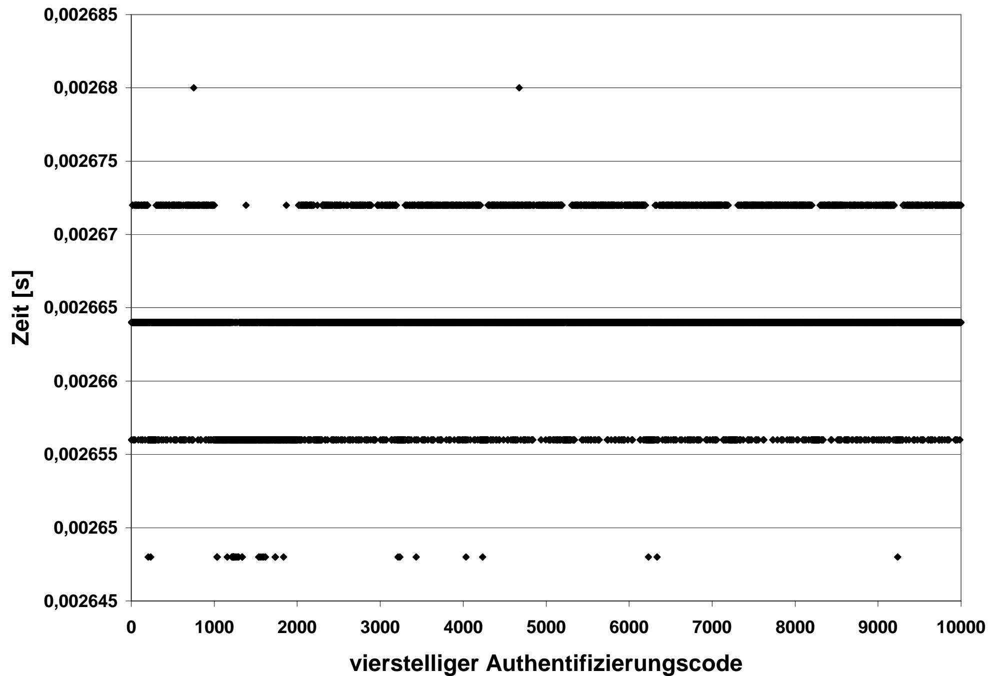

Abbildung 5.9: Messergebnisse der zustandsunabhängigen Implementierung

des Programmcodes auf die API Befehle des Smartcard Betriebssystems eine zusätzliche Sicherheitsüberprüfung einbaut.

Auch hierbei zeigt sich durch die 100 Anwendungsfälle kein Zusammenhang. Dies hat aber mehrere Gründe. Zum einen befindet sich das Ergebnis der Laufzeiten im Bereich der Empfindlichkeit der Zeitmesseinheit, wodurch exakte Aufteilung potentieller Gruppen erschwert wird und zum anderen ist der durch die Anwendungsfälle abgedeckte Bereich der Laufzeitergebnisse annähernd normalverteilt. Eine Gruppierung anhand der Mittelwerte ist nicht möglich, da der Analysemechanismus ein Verhalten erkennt, das wie die Streuung um den Mittelwert bei herkömmlichen Messergebnisse aussieht. Die Auswertesoftware kann wie erwartet keine Gruppierung durchführen, wodurch Tabelle 5.7 ausschließlich die Ergebnisse der gesamten Anwendungsfälle zeigt.

	Minimum	**Maximum**	**Mittelwert**	**Standardabweichung**
	[μs]	[μs]	[μs]	[μs]
Gesamt	2648	2672	2658,2	6,9

Tabelle 5.7: Ergebnisse der Laufzeitanalyse der zustandsunabhängigen Implementierung anhand der Anwendungsfälle

Kapitel 6

Zusammenfassung

Im Zuge dieser Masterarbeit wurde der Aufbau eines Hardware/Software Codesign Flows beschrieben, der in der Lage ist, Timing Attacken vorzubeugen. Er definiert zeitkritische Bereiche bereits in der Spezifikationsphase. Daraus werden im weiteren Verlauf des Hardware/Software Codesign Flows die Timingvorgaben gewonnen. Diese definieren welche Funktionen oder Codebereiche zu markieren sind, wodurch eine spätere Zeitmessung ermöglicht wird. In der Implementationsphase des Hardware/Software Codesign Flows werden die Annotationen im Programmcode durchgeführt. Diese erzeugen sobald die Anwendungs- oder Testfälle abgearbeitet sind Laufzeitmessergebnisse, die im Anschluss automatisch ausgewertet werden. Weist dabei ein als zeitkritisch definierter Bereich ein bedenkliches Laufzeitverhalten auf, so wird eine Warnung ausgegeben und im Timingbericht vermerkt.

Die Funktionsweise des Hardware/Software Codesign Flows wurde anhand eines Smartcard Systems gezeigt. Dafür wurde ein bestehendes Smartcard Betriebssystem verwendet, das um eine Zeitmesseinheit erweitert wurde. Das Smartcard Betriebssystem unterstützt mehrere Programmiersprachen, die auch durch die Zeitmesseinheit unterstützt werden. Weiters unterstützt sie auch, das Programmiersprachen übergreifende messen der Laufzeit. Die gemessenen Laufzeiten werden im EEPROM gespeichert. Weiters wurde eine Auswertesoftware implementiert, die die gemessenen Laufzeiten automatisch auswertet. Dafür werden die gespeicherten Laufzeitergebnisse von der Smartcard auf einen Computer übertragen. Im Anschluss wurden sie anhand von Mittelwertsunterschieden analysiert und aus den Ergebnissen der Analyse wird der Timingbericht generiert.

Abschließend wurde ein Smartcard System vermessen und die Messergebnisse ausgewertet. Dafür wurden fünf unterschiedliche Implementationen der Authentifikationsfunktion untersucht. Die Untersuchung erfolgte anhand von 100 Anwendungsfällen, die ein problematisches Laufzeitverhalten hervorrufen sollten. Vier der fünf Implementationen wiesen eine Laufzeitabhängigkeit auf. Die Auswertesoftware erkannte drei davon erfolgreich. Die vierte wurde nicht erkannt, da die Zeitmesseinheit eine zu geringe Auflösung hatte. Weiters versuchte die Auswertesoftware, die Laufzeitergebnisse der drei erkannten laufzeitabhängigen Implementation in ihre Abhängigkeitsgruppen zu unterteilen.

Dabei wurden die Laufzeitergebnisse von zwei Implementationen erfolgreich unterteilt. Die dritte Implementation wies einen minimalen Zuordnungsfehler auf. Dieser entstand auf Grund der eng aneinander liegenden Messergebnisse, die zu Überlappungen bei den Entscheidungskriterien führten.

6.1 Ausblick

In dieser Arbeit wurde ein bereits existierender Hardwaretimer verwendet, der ein Referenzintervall von 8 Mikrosekunden generiert. Würde man einen genaueren Timer für die Erzeugung der Referenzintervalle verwenden, so könnte man detaillierte Laufzeitergebnisse untersuchen, die vielleicht eine Abhängigkeit der Laufzeiten zeigen.

Weiters wäre es denkbar, eine alternative statistische Analyse der Laufzeitergebnisse durchzuführen. Hierbei könnte man das bestehende Konzept erweitern oder einen neuen Erkennungsalgorithmus zum Überprüfen von Laufzeitabhängigkeiten entwerfen. Hier wäre zum Beispiel der Einsatz eines künstlichen Intelligenz Systems denkbar.

Eine weitere Verbesserungsmöglichkeit wäre die Integration des Laufzeitberichts in eine Entwicklungsumgebung, wie zum Beispiel Eclipse. Dadurch würde ein Entwickler an der entsprechenden Stelle des Programmcodes sehen, ob eine Funktion eine potentielle Laufzeitgefährdung darstellt.

Abschließend stellt sich auch noch die Frage, wie man die Anwendungsfälle wählt um sicher zu stellen, dass ein potentielles Problem beim Laufzeitverhalten erkannt wird.

Abkürzungsverzeichnis

AES Advanced Encryption Standard

APDU Application Protocol Data Unit

API Application Programming Interface / Programmierschnittstelle

ATR Answer to Reset

CPU Central Processing Unit

CSV Comma-Separated Values

DES Data Encryption Standard

DPA Differential Power Analysis

ECC Elliptic Curve Cryptography

EEPROM Electrically Erasable Programmable Read Only Memory

FRS Functional Requirements Spezifikation

ID Identifikationskennzeichen

MDS-Code Maximum-Distance-Separable-Code

ppm parts per million / Teile von einer Million

RAM Random Access Memory

RC5 Rivest Cipher 5

ROM Read Only Memory

SPA Simple Power Analysis

SSL Secure Sockets Layer

TLS Transport Layer Security

TRS Timing Requirements Spezifikation

Literaturverzeichnis

[AAK09] H.R. Ahmadi and A. Afzali-Kusha. Very low-power flexible GF(p) elliptic-curve crypto-processor for non-time-critical applications. In *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*, pages 904 –907, 24-27 2009.

[AGM+09] M. Alam, S. Ghosh, M.J. Mohan, D. Mukhopadhyay, D.R. Chowdhury, and I.S. Gupta. Effect of glitches against masked AES S-box implementation and countermeasure. *Information Security, IET*, 3(1):34 –44, march 2009.

[BB03] D. Brumley and D. Boneh. Remote Timing Attacks are Practical. In *In Proceedings of the 12th USENIX Security Symposium*, pages 1–14, 2003.

[BEPM10] A. Bogdanov, T. Eisenbarth, C Paar, and Wienecke M. Differential Cache-Collision Timing Attacks on AES with Applications to Embedded CPUs. In *CT-RSA*, pages 235–251, 2010.

[Ber04] D. J. Bernstein. Cache-timing attacks on AES, 2004.

[BPW10] J. Bacher, A. Pöge, and K. Wenzig. *Clusteranalyse: Anwendungsorientierte Einführung in Klassifikationsverfahren*. Oldenbourg, auflage 3 edition, 2010.

[BR09] J. Bani and S.S. Rizvi. Minimizing Cache Timing Attack Using Dynamic Cache Flushing (DCF) Algorithm. *CoRR*, abs/0909.0573, 2009. informal publication.

[BZ09] M. Bühner and M. Ziegler. *Statistik für Psychologen und Sozialwissenschaftler*. Pearson Education Inc., 2009.

[CKNS01] J-S. Coron, P. Kocher, D. Naccache, and É. N. Supérieure. Statistics and Secret Leakage, 2001.

[CRW09] S. A. Crosby, R.H. Riedi, and D.S. Wallach. Opportunities and limits of remote timing attacks. Technical report, 2009.

[ESM+06] Y. Eslami, A. Sheikholeslami, Senior Member, P.G. Gulak, Senior Member, S. Masui, and K. Mukaida. K.: An area-efficient universal cryptography processor for smart cards. *IEEE Trans. VLSI Systems*, pages 43–56, 2006.

[GS02] A. V. Garcia and J-P. Seifert. On the implementation of the advanced encryption standard on a public-key crypto-coprocessor. In *CARDIS'02: Proceedings*

of the 5th conference on Smart Card Research and Advanced Application Conference, pages 14–14, Berkeley, CA, USA, 2002. USENIX Association.

[Hey98] H. Heys. A Timing Attack on RC5. In *In Workshop Record of SAC '98, Queen's*, pages 330–343, 1998.

[Jan08] P. Janke, M. und Laackmann. Sicherheit der Chipkarten-Plattform. Technical report, Infineon Technologies AG, 2008.

[Koc96] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. pages 104–113. Springer-Verlag, 1996.

[Lam83] B. W. Lampson. Hints for computer system design. *SIGOPS Oper. Syst. Rev.*, 17:33–48, October 1983.

[Lit05] T. Litte. Zufallsgeneratoren. Uni Goettingen, 2005.

[LSG10] L. Li, Y. Song, and M. Gao. A new genetic simulated annealing algorithm for hardware-software partitioning. In *Information Science and Engineering (ICISE), 2010 2nd International Conference on*, pages 1 –4, dec. 2010.

[Man10] S. Mangard, editor. *Side-Channel Attacks in the Presence of Countermeasures*, Leiden, The Netherlands, February 2010. Infineon Technologies, Munich, Germany, Chip Card, Security Innovation Groupy.

[MH10] C. Moreno and M.A. Hasan. An Adaptive Idle-Wait Countermeasure Against Timing Attacks on Public-Key Cryptosystems, 2010.

[Nac06] W. Nachtigall, C. und Wirtz. *Wahrscheinlichkeitsrechnung und Inferenzstatistik*, volume Teil 2. Juventa Verlag Weinheim und München, 2006.

[OST05] D.A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: the Case of AES. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006*, pages 1–20. Springer-Verlag, 2005.

[PGH+04] N. Pramstaller, F.K. Gurkaynak, S. Haene, H. Kaeslin, N. Felber, and W. Fichtner. Towards an AES crypto-chip resistant to differential power analysis. In *Solid-State Circuits Conference, 2004. ESSCIRC 2004. Proceeding of the 30th European*, pages 307 – 310, 21-23 2004.

[PTVF92] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C.* Cambridge University Press, Cambridge, UK, 2nd edition, 1992.

[Sch98] W. Schmusch. *Elektronik, Bd.6, Elektronische Meßtechnik.* Vogel Verlag Und Druck, 4., durchges. a. edition, 1998.

[Wik11] Intel MCS-51. http://de.wikipedia.org, März 2011. Wikipedia.

[WMMY06] R. E. Walpole, R. H. Myers, S. L. Myers, and Keying Ye. *Probability and Statistics: For Engineers and Scientists.* Pearson Education Inc., seventh edition, 2006.

[ZF05] Y. Zhou and D. Feng. Side-Channel Attacks: Ten Years After Its Publication and the Impacts on Cryptographic Module Security Testing," Cryptology ePrint Archive, Report 2005/388, 2005.