Studienarbeit 2

FIPS-140-2 Zertifizierung für strongSwan

Autoren:	Bruno Krieg	bruno_krieg@gmx.ch
	Daniel Wydler	daniel.wydler@gmx.ch
Betreuer:	Prof. Dr. Andreas Steffen	asteffen@hsr.ch
	Martin Willi	mwilli@hsr.ch
Industriepartner:	-	

Version:	1.0
Semester:	Sommer 2007 (02. April 2007 - 06. Juli 2007)

1 Abstract

Im Projekt "FIPS-140-2 Zertifizierung für strongSwan" ging es darum das Library-Modul auf eine mögliche FIPS-Zertifizierung vorzubereiten. Dabei wurde der Schwerpunkt auf die beiden Teilbereiche "Sicherstellung der Integrität der strongSwan-Library" und "Überprüfung der kryptographischen Funktionen mit Self-Tests auf Korrektheit" gelegt. Die realisierten Integritäts- und Self-Tests bieten den Anwendern von strongSwan die Möglichkeit, ihre strongSwan- Implementation zu überprüfen.

Der Integritätstest ermöglicht das Überprüfen der strongSwan-Library zur Programmausführung. Dazu wird eine Integritätsprüfung des Memory-Inhalts durchgeführt. Der gehashte Referenzwert dazu wird beim Erstellen der Applikation in strongSwan eingearbeitet, damit dieser nachträglich nicht verändert werden kann.
Der Integritätstest ist standardmässig nicht Bestandteil der strongSwan Applikation, kann jedoch bei Bedarf mit einer configure-Option aktiviert werden.

Es wurden Self-Tests für alle Crypto-Funktionen realisiert, die von FIPS verlangt werden. Zu diesen Crypto-Funktionen gehören Hashers, HMAC, Signers, PRF's, Crypters und RSA. Mit den realisierten Self-Tests werden die Crypto-Funktionen auf ihre Korrektheit überprüft. Manipulationen können somit festgestellt werden. Die Crypto-Funktionen können kritisch oder unkritisch eingestuft werden. Dies erlaubt dem Anwender verschiedene Szenarien auszuarbeiten, wie sich das Programm im Fehlerfall verhalten soll.
Die Selftests werden standardmässig mitcompiliert, können jedoch durch eine configure-Option deaktiviert werden.

2 Management Summary

Ausgangslage
Die FIPS-140-2 Zertifizierung erlaubt es, dass Programme auch in sehr kritischen Anwendungsgebieten eingesetzt werden können. Diese Zertifizierung ist jedoch sehr kostspielig und zeitaufwendig. Für die Zertifizierung sind nebst verschiedensten Dokumenten und der Überprüfung des Source-Codes, die korrekte Ausführung und Integrität der Cryptofunktionen ein zentraler Punkt. Mit den im Rahmen dieser Studienarbeit realisierten Integritäts- und Self-Tests soll genau dieser Punkt der Zertifizierung abgedeckt werden.

Integritätstest
Um die FIPS-140-2 Zertifizierung zu erreichen, muss die Integrität der Crypto-Funktionen gewährleistet werden können. Die Integrität wird durch ein Incorehashing über einen Bereich im Arbeitsspeicher sichergestellt. Bei einem Incorehashing wird ein HASH-Wert über dem Inhalt eines Bereichs im Arbeitsspeicher berechnet, der von einer Library oder einem anderen Modul belegt ist. Durch einen Vergleichswert, Signatur genannt, der während des Compiliervorgangs erstellt wurde, kann die Integrität der Library oder des Moduls jederzeit überprüft werden. Somit können Manipulationen oder Fehler der Library oder des Moduls erkannt werden.

Selftests
In der FIPS Dokumentation zum Standard 140-2 werden in einem Unterkapitel Self-Tests angesprochen. Self-Tests sind Methoden um Crypto-Funktionen auf ihre Korrektheit zu überprüfen. Durch Self-Tests können jegliche Arten von Manipulationen an der Implementierung der Crypto-Funktionen festgestellt werden. Um eine Crypto-Funktion zu prüfen wird ein Known-Answer-Test (KAT) durchgeführt. KAT sind Tests bei denen jeweils im voraus zu den Eingangsvektoren auch die korrekten Ergebnissvektoren bekannt sind. Somit kann der Eingangsvektor einer Crypto-Funktion übergeben werden, um den berechneten Istmit dem Soll-Ergnisvektor zu vergleichen. Falls die beiden Vektoren übereinstimmen gilt der Test als bestanden, andernfalls nicht. Weil die Crypto-Funktionen verschieden Merkmale aufweisen, können sie in die Gruppen Hashers, HMAC, Signers, PRF's, Crypters, und RSA eingeteilt werden.

Ausblick
Die Self-Tests für die bestehenden Crypto-Funktionen wurden vollständig und wunschgemäss implementiert. Das Self-Test Modul ist erweiterbar.

Da die Integritätstest sehr plattformabhängig sind, kann die korrekte Funktion des Integritätstests nicht auf allen Plattformen garantiert werden. In einem weiteren Schritt sollte die Implementierung auf verschiedensten Plattformen getestet und gegebenenfalls angepasst werden.

Inhaltsverzeichnis

3 Einleitung

Dieses Dokument führt sie durch die Dokumentation zur Studienarbeit FIPS-140-2 Zertifizierung für strongSwan. Das Dokument ist folgendermassen strukturiert.

- Analyse
- Integritätstest
- Self-Tests
- Schlussfolgerungen

3.1 Ausgangslage

strongSwan hat viele kryptographische Funktionen implementiert. Diese können vom Anwender einfach genutzt werden. Doch im Moment gibt es keine Garantie, dass diese auch ordnungsgemäss funktionieren. Manipulationen an der Implementierung der Crypto-Funktionen können nicht festgestellt werden. Dieses Sicherheitsrisiko soll mit diesem Projekt angegangen und minimiert werden.

3.2 Ziel und Zweck

In diesem Projekt geht es darum das Produkt strongSwan auf eine mögliche FIPS-Zertifizierung vorzubereiten. Diese Vorbereitsungsmassnahmen beinhalten die Sicherstellung der Integrität der Library "libstrongswan" sowie die Überprüfung der Crypto-Funktion mittels Self-Tests. Diese Massnahmen werden in der offiziellen FIPS-Publikationen für den Standard 140-2 gefordert.

3.3 Aufgabenstellung

Der bekannte U.S. FIPS-140-2 Standard zertifiziert den Sicherheitslevel von Crypto-Hardware und seit einiger Zeit auch von Crypto-Software. Die Linux strongSwan VPN Lösung verwendet starke Kryptographie und soll im Rahmen dieser Studienarbeit auf eine mögliche FIPS-140-2 Zertitfizierung vorbereitet werden. Die Grundvoraussetzung dafür ist die Kapselung aller Kryptofunktionen in ein geprüftes Bibliotheksmodul, sowie die Überprüfung einer Checksumme über dieses Modul während der Laufzeit, um sicher zu gehen, dass keine Veränderungen im Code vorgenommen wurden. Weiter müssen alle Kryptofunktionen beim Starten der Applikation mittels einer automatisierten Testsuite auf Korrektheit getestet werden.

- Studium des FIPS-140-2 zertifizierten OpenSSL Object Moduls und der Begleitdokumente "User Guide" und "Security Policy"
- Implementierung eines gekapselten strongSwan Crypto Object Moduls inklusive Self-Tests, ausgehend von den bestehenden Funktionen der libstrongswan Library (keine OpenSSL-Abhängigkeit).
- Optionales Konzept für die Einbindung des OpenSSL FIPS Object Moduls in die libstrongswan Library.

4 Anforderungsanalyse

4.1 Funktionale Anforderungen

4.1.1 Installationsmöglichkeiten

Es wird eine Standardinstallationsroutine mit sinnvollen Default-Einstellungen angeboten. Bei der Standardinstallation wird die strongSwan-Distribution in einem Standardverzeichnis installiert. Darüber hinaus kann die Installation für erfahrene Anwender über Installationsparameter gesteuert werden. Der Installationsprozess wird mittels Autoconf und Automake durchgeführt. Über ein Compile-Flag ist es möglich anzugeben, ob die Integritätstest und Self-Test Erweiterung installiert werden soll oder nicht.

4.1.2 Crypto-Modus

Es werden zwei Crypto-Modi angeboten. Der eine ist der standardmässige strongSwan-Modus und der andere ist der um die Self-Tests erweiterte Crypto Modus. Mit einem Aufruf der zugehörigen Funktion, wird mit einem Flag angegeben, in welchen Modus gewechselt werden soll. Beim Wechsel in den Self-Test Modus wird zuerst eine Reihe von Integritätstest und Self-Tests ausgeführt, bevor in diesen Modus gewechselt werden kann. Damit wird das Modul auf Integrität und Korrektheit der Crypto-Funktionen geprüft. Die einzelnen Crypto-Funktionen werden in kritische und unkritische Funktionen eingeteilt. Dabei müssen alle Integritäts- und Self-Tests für die kritischen Funktionen erfolgreich ablaufen, ansonsten wird strongSwan beendet. Ist der Selftest einer unkritischen Funktion fehlerhaft, kann strongSwan weiterhin benutzt werden, jedoch steht diese Funktion nicht mehr zur Verfügung.
Die Integritätsprüfung kann durch einen Config-Eintrag deaktiviert werden, um mögliche Probleme mit bestimmten Architekturen zu vermeiden. Die Self-Tests funktionieren dann aber weiterhin.
Sie können ebenfalls durch einen Config-Eintrag deaktiviert werden, damit in sehr zeitkritischen Anwendungen das starten von strongSwan nicht zu lange dauert.

4.1.3 Integrität Executable-Code

Beim Erstellen des Object-Moduls mit den Cryptofunktionen, wird über die Speicherbereiche der RODATA- (Read Only DATA) und TEXT-Sektion ein HASH ermittelt. Dieser Hash wird in einem weiteren Schritt in die Zielapplikation eingearbeitet. Beim Starten der Zielapplikation (hier strongSwan) wird das Object-Modul geladen. Dabei werden RODATA und TEXT in verschiedene Speicherbereiche gealden. Durch Start- und Endkennzeichnungen der einzelnen Sektionen kann aus dem Speicher wieder ein HASH gebildet werden. Dieser errechnete In-Core HASH wird anschliessend mit dem während der Build-Time erstellten HASH verglichen. Sind die beiden HASH-Werte nicht gleich, ist die Integrität des ausführbaren Codes im Arbeitsspeicher verletzt und das Programm

wird beendet.

4.1.4 Self-Tests

Um die Integrität der Crypto-Funktionen zu gewährleisten, können Self-Tests ausgeführt werden, um das korrekte Verhalten dieser Funktionen zu prüfen. Dabei muss zwischen kritischen und unkritischen Crypto-Funktionen unterschieden werden. Falls ein Self-Test einer kritischen Crypto-Funktion fehlschlägt, wird strongSwan beendet. Falls der Self-Test einer unkritischen Crypto-Funktion fehlschlägt, wird nur die betroffenen Crypto-Funktion deaktiviert. Der Selftest-Modus kann dann mit Einschränkungen verwendet werden. Die kritischen Crypto-Funktionen können in einer Liste definiert werden. Tests die während dem Initialisieren durchgeführt werden heissen Power-Up-Tests. Als Power-Up-Tests werden sogenannte Known-Answer-Tests und Conditional-Tests durchgeführt. Self-Tests können jederzeit zur Laufzeit ausgeführt werden. Auch im Standard strongSwan-Modus wird es möglich sein einzelne Self-Test auszuführen.

4.2 Nichtfunktionale Anforderungen

4.2.1 Bedienbarkeit

Die um die Self-Tests erweiterte strongSwan Library sollte intuitiv zu bedienen sein. Bei der Installation sollten keine speziellen Kenntnisse nötig sein, um den Installationsprozess durchzuführen. Um diese Anforderung zu erfüllen, sollte die Installation mit sinnvollen Standardeinstellungen automatisiert ausgeführt werden. Bei Problemen und Fehlern beim Ausführen von Operationen sollte dem User ein gutes Feedback anhand von Fehlermeldungen gegeben werden. Um Self-Tests auszuführen und Crypto-Funktionen anzuwenden, sollten keine vertieften kryptographischen Kenntnisse nötig sein.

4.2.2 Zuverlässigkeit

Die Zuverlässigkeit des erweiterten strongSwan Crypto-Modules kann, soweit sie im FIPS PUB 140-2 definiert wurden, garantiert werden. Dies gilt insbesondere für den Self-Test-Modus, in den nur gewechselt werden kann, wenn die Integritäts- und Self-Tests erfolgreich waren. In diesem Fall kann davon ausgegangen werden, dass das Modul nicht manipuliert wurde und damit ordnungsgemäss funktioniert.

4.2.3 Leistung

Die Installation des strongSwan Package sollte sich in Minutenbereich bewegen. Das Aufstarten des Self-Test-Modules sollte nicht mehr als zwei Sekunden auf einem aktuellen Computer in Anspruch nehmen.

4.2.4 Supportability

Mit Installationsparametern kann das Standardverhalten bei der Installation verändert werden. Es ist dann eine benutzerdefinierte Installation möglich. Zu den Installationsparametern gehören das Installationsverzeichnis. Zur Laufzeit können abgesehen vom Modus-Wechsel keine Einstellungen vorgenommen werden.

4.2.5 Schnittstellen

4.2.5.1 Benutzerschnittstellen

Eine grafische Benutzeroberfläche (User Interface) wird keine zur Verfügung gestellt.

4.2.5.2 Hardwareschnittstellen

Es wird keine Hardware-Schnittstelle benötigt.

4.2.5.3 Softwareschnittstelle

Bei der Installation vom strongSwan FIPS Package werden auf dem Betriebssystem die Crypto-Funktionen als Library installiert. Diese kann von anderen Applikationen eingebunden und genutzt werden.

4.2.6 Lizenzanforderungen

Es wird keine kostenpflichtige Lizenz benötigt. strongSwan wird unter GPL-2 (General Public License) Lizenz angeboten.

4.2.7 Verwendete Standards

Der Abschnitt 4.9 Self-Tests des FIPS PUB 140-2 Standards wird umgesetzt.

4.3 Use Cases

In diesem Abschnitt sind die Use Cases zu diesem Projekt aufgeführt. In der nachfolgenden Figur ist das Zusammenspiel zwischen Actors und Use Cases zu sehen.

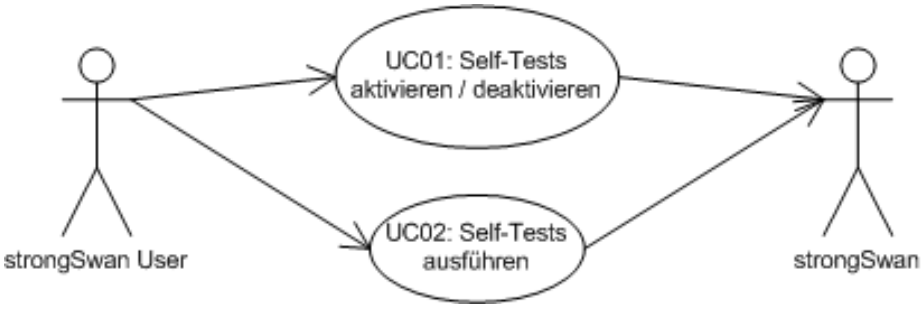

Abbildung 1: Use-Case Diagramm

Der strongSwan User stellt einen Benutzer von strongSwan dar und wird in diesem Bericht auch Benutzer genannt.
strongSwan stellt das strongSwan-Programm selber dar und wird in diesem Bericht auch System genannt.

Nachfolgend sind die einzelnen Use Cases als Fully Dressed Essential Use Cases aufgeführt.

4.3.1 Use Case 01: Self-Test Modus aktivieren(a) / deaktivieren(b)

Primary Actor:	strongSwan Benutzer
Interests:	strongSwan Benutzer möchte den Self-Test-Modus aktivieren / deaktivieren.
Precondition:	strongSwan ist installiert
Postcondition:	Self-Tests für strongSwan sind aktiviert / deaktiviert
Main Success Scenarion:	1a. Benutzer aktiviert die Self-Tests in den Config-Einstellungen 1b. Benutzer deaktiviert die Self-Tests in den Config-Einstellungen 2. Das System führt Integritätsprüfung durch 3a. Das System führt den Self-Test für jede FIPS Funktion durch 3b. Das System führt keine Self-Tests durch
Failure Scenario:	2. Integritätsprüfung schlägt fehl → System wird beendet 3a. Self-Tests einer kritischen Funktion schlägt fehl → System wird beendet 3a. Self-Tests einer unkritischen Funktion schlägt fehl → Funktion wird gesperrt

Tabelle 1: Use Case 01: Self-Tests Modus aktivieren / deaktivieren

4.3.2 Use Case 02: Self-Tests ausführen

Primary Actor:	strongSwan Benutzer
Interests:	strongSwan Benutzer möchte die Self-Tests ausführen.
Precondition:	System ist installiert
Postcondition:	Self-Tests wurden durchgeführt.
Main Success Scenarion:	1. Benutzer ruft Self-Tests für eine Funktion auf 2. Das System führt Self-Tests für diese Funktion durch
Failure Scenario:	2a. Self-Tests einer kritischen Funktion schlägt fehl → System wird beendet 2b. Self-Tests einer unkritischen Funktion schlägt fehl → Funktion wird gesperrt

Tabelle 2: Use Case 02: Self-Tests ausführen

5 Die FIPS-Zertifizierung

[FIPS PUB 140-2] [OpenSSL FIPS 140-2 Security Policy]

5.1 Beschreibung

Das National Institute of Standards and Technology (NIST) gab die FIPS-140 Standard Dokumente heraus, um die Anforderungen an Crypto Hardware Software zu koordinieren. Der U.S. FIPS-140-2 Standard zertifiziert den Sicherheitslevel von Crypto-Hardware und von Crypto-Software.
Nachfolgend sind die wichtigsten Kapitel für dieses Projekt beschrieben.

5.2 Security Levels

FIPS-140-2 ist in 4 Security Levels unterteilt. Mit höherem Level steigen auch die Anforderungen an die Kryptofunktionen.

5.2.1 Level 1

Der Securitiy Level 1 bietet das geringste Mass an Sicherheit. Es werden grundlegende Anforderungen an ein kryptographisches Modul gefordert (z.B. muss mindestens geprüfter Algorithmus oder Security-Function verwendet werden). Physikalische Sicherheitsmechanismen werden keine gefordert. Der Security Level 1 erlaubt Komponenten eines kryptographischen Moduls auszuführen, auch wenn das Betriebssystem des Computers nicht geprüft ist. Solche Implementierungen sind für low-level Sicherheits-Applikationen geeignet, auch wenn die Hardware keine physikalische Sicherheit aufweist. Solche kryptographischen Softwares sind kostengünstiger als korrelierende Hardware-Lösungen.

5.2.2 Level 2

Der Security Level 2 erweitert den Security Level 1 um physikalische Sicherheitsmechanismen.
Der Security Level 2 benötigt im Minimum eine rollenbasierte Authentifikation, in welcher ein kryptographisches Modul den Operator autorisiert, um zu prüfen welche Services ausgeführt werden dürfen.

5.2.3 Level 3

Zusätzlich zum Security Level 2 beschreibt es Sicherheitsmechanismen, um Eindringlingen den Zugriff zu verhindern. Die physiklischen Sicherheitsmechanismen, die für den Security Level 3 benötigt werden, sollten eine hohe Dedektierwahrscheinlichkeit bei pyhsiklischen Zugriffsversuchen oder beim Verändern des Crytpomodul aufweisen. Zusätzlich wird ein identitätbasierender Authentication-Mechanismus zusätzlich zu der rollenbasierten Athentifizierung gefordert. Ein kryptographisches Modul authentifiziert die Identität von einem Operator und prüft, ob er berechtigt ist die geforderten Services auszuführen.

5.2.4 Level 4

Der Security Level 4 bietet den höchsten Sicherheistlevel, der in diesem Standard definierten. Auf diesem Sicherheitslevel bieten die physikalischen Sicherheitsmechanismen eine komplette Schutzhülle um das kryptografische Modul.

5.3 Cryptographic Key Management

Die Sicherheitsanforderungen an das kryptographische Key-Management umfasst den ganzen Lebenszyklus von allen Schlüsselkomponenten die vom Modul benötigt werden. Key-Management beinhaltet random number und key generation, key establishment, key distribution, key entry/output, key storage, key zeroization.

5.3.1 Key-Storage

Kryptografische Schlüssel können entweder als Plain-Text oder in verschlüsselter From im Crypto-Modul gespeichert werden. Diese Schlüssel sollten ohne Authentifizierung von aussen nicht zugänglich sein.

5.3.2 Key-Zeroization

Das Crypto-Modul sollte Methoden zur Verfügung stellen um alle Plain-Text Geheimnisse Private Keys in einem Modul zu Nullen.

5.4 Self-Tests

Ein kryptographisches Modul muss Power-up, Self-Tests und Conditional Self-Tests durchführen, um sicherzustellen, dass die Crypto-Funktion ordnungsgemäss funktionieren. Power-up Self-Tests sollten ausgeführt werden, wenn das Crypto-Modul initialisiert wird. Conditional Self-Tests werden durchgeführt, wenn eine Security-Funktion zur Laufzeit aufgerufen wird. Wenn ein Cryptomodul nicht alle Power-up Self-Tests besteht, wird in einen Errorstatus gewechselt. Der FIPS-Modus der kryptographischen Funktionen kann nur verwendet werden, wenn alle Power-up Tests erfolgreich waren.

5.5 Power-Up Tests

Die Power-up Tests beinhalten kryptographische "known answer tests" und Integritäts-Tests. Power-up Tests werden automatisch beim Initialisieren des Moduls aufgerufen oder zu jeder anderen Zeit durch den Aufruf der entsprechenden Self-Test-Funktion.
Bei Known Answer Tests sind die Ausgangswerte von einer Funktion zu den dazugehörigen Eingangswerten bereits im Voraus bekannt. Möchte man zum Beispiel eine SHA-1 Funktion testen, so übergibt man einen Inputvektor an die Funktion und vergleicht die Ausgabe mit dem, was als ERgebnis herauskommen sollte. Falls diese zwei Ausgangsvektoren übereinstimmten, so kann davon

ausgegangen werden, dass die Crypto-Funktion ordnungsgemäss funktioniert.
Um Self-Tests auszuführen ist keine Authentifizierung nötig.
Die Integritäts-Tests werden mit HMAC Signaturen berechnet. Es wird eine HMAC-Signatur über den object code berechnet und gespeichert und anschliessend wird diese Signatur mit der aktuellen verglichen. Zusätzlich zu den Power-up Tests werden Conditional Tests durchgeführt. Dazu gehören paarweise Konsistenztests mit Public- / Private-Key. Dafür wird ein Plaintext mit dem Public-Key verschlüsselt und anschliessend wird der Cipher-Text wieder mit dem Private-Key entschlüsselt, was wieder den Ursprungs-Plaintext ergeben sollte. Stimmen diese beiden Plaintexte überein, so ist der Test erfolgreich.

5.6 Integrität des Source-Codes

Um die Integrität des Source Codes zu gewährleisten, wird eine HMAC-Signatur über die ganze Source-Code Distribution berechnet. Diese Signatur wird öffentlich publiziert und sollte vom Anwender des Source-Codes überprüft werden.

5.7 Integrität des Object-Codes

Aus den Source-Files wird ein Object-Modul generiert. Das Vorhandensein und die relative Reihenfolge des generierten Object-Codes müssen fix und unveränderlich sein. Normalerweise ist es irrelevant in welcher Reihenfolge die Object-Files gelinkt werden. Wenn jedoch das Objectmodul mit einer Signatur verglichen werden soll, ist es elementar, dass die Objectdateien immer in der gleichen Reihenfolge zusammengefasst werden. Um diese Vorgabe zu erfüllen, wird aus den Source-Files ein monolithisches Object-Modul generiert. Nun kann der Object-Code im Object-Modul nicht gelöscht, ausgetauscht oder erweitert werden.

5.8 Integrität des Executable-Codes

Das Design des Object-Moduls beinhaltet die Definition von Referenzpunkten im Object-Code, um den Code-Bereich zu definieren, welcher zur Läufzeit durch einen Integritätstest geschützt werden muss.
Zur Compilierungszeit wird ein HASH über den definierten Code-Bereich erstellt und in die Zielapplikation integriert. Zur Ausführungszeit der Zielapplikation, wird durch einen in-core Integritätstest die Integrität, zusammen mit dem errechneten Hash, sichergestellt.

6 Realisiertes System

6.1 Integritätstest

6.1.1 Einführung

Die Crypto-Funktionen von strongSwan befinden sich in der libstrongswan. Diese Funktionen sind von zentraler Bedeutung und deren Integrität muss gewährt sein. Genau dies wird mit dem nachfolgend beschriebenen Integritätstest bewerkstelligt.

6.1.2 Aktivierung des Integritätstests

Der Integritätstest ist standardmässig deaktiviert. Um ihn nutzen zu können, muss er daher zuerst aktiviert werden. Die Aktivierung erfolgt über ein Flag das dem ./configure mitgegeben wird.
Mittels:
./configure −−enable−integrity−test
wird der Integritätstest mitcompiliert und steht danach zur Verfügung.

6.1.3 Ausführung des Integritätstests

Der Integritätstest kann durch die Funktion run_integritytest() ausgeführt werden. Die Funktion ist im Header <fips/fips.h> definiert.

6.1.4 Design

6.1.4.1 In-Core Hashing

Unter In-Core Hashing ist das Bilden eines Hash-Werts über einen gewissen Speicherbereich eines laufenden Programms zu verstehen. Die Überprüfung dieses Hash-Werts gibt Aufschluss darüber, ob das Programm verändert wurde.

Damit dieser Vergleich durchgeführt werden kann, muss ein Vergleichs-Hash-Wert vorhanden sein. Dieser Referenzwert wird Signatur genannt.
Die Signatur muss während dem Erstellen des Programms erzeugt werden und ebenfalls in diesem Programm hinterlegt werden. Das Hinterlegen dieser Signatur in eine externe Datei wäre unsicher, weil missbräuchliche Änderungen einfacher vorgenommen werden könnten.
Nach [FIPS PUB 140-2] müssen alle Crypto-Funktionen in einem Modul zusammengefasst werden. Diese Modul muss durch Integritätstests geschützt werden. Das heisst, dass nur die Speicherbereiche die, durch dieses Modul beansprucht werden, durch In-Core Hashing geschützt werden müssen.

Objektmodul
Um ein In-Core Hashing eines Objektmoduls durchführen zu können, muss der Aufbau und das Laden in den Speicher bei, der Ausführung diese Moduls betrachtet werden.

Ein Objektmodul enthält Daten und ausführbaren Code, der aber noch nicht zu einer ausführbaren Applikation oder einer Library gelinkt wurde. Die Daten und der Code werden in unterschiedlichen Sektionen gespeichert.
Ein Objektmodul entsteht wenn ein Compiler oder Assembler, Quellcode verarbeitet. Das Modul enthält kompakten Code der auch Binärdaten genannt wird. Durch einen Linker wird das Objektmodul mit anderen Dateien zu einer ausführbaren Applikation oder einer Library gelinkt. Neben dem Programmcode können Objektmodule auch verschiedene Daten (u.a. Konstanten), Informationen zur Realokation, Programm Symbole (Namen von Variablen und Funktionen) für das Linking und Debuginformationen enthalten. Diese verschiedenen Arten von Code und Daten werden in verschiedenen Sektionen gehalten. Die nachfolgende Liste enthält einen Ausschnitt aus den wichtigsten Sektionen:

- **.TEXT**
  Die TEXT Sektion enthält ausführbare Computerinstruktionen. Die Sektion hat eine bestimmte Grösse und ist schreibgeschützt.
- **.DATA**
  Die DATA Sektion enthält globale Variablen die vom Programmierer bereits initialisiert wurden. Die Grösse ist bekannt, jedoch können die Werte dieser globalen Variablen geändert werden. Daher ist diese Sektion beschreibbar.
- **.RODATA**
  Die RODATA Sektion enthält wie die DATA Sektion globale Variablen, jedoch sind diese Variablen konstant. Die Grösse der Sektion ist ebenfalls fix und da die Variablen konstant, d.h. nicht veränderbar sind, ist die Sektion schreibgeschützt.
- **.BSS**
  Die BSS Sektion enthält alle globalen Variablen die vom Programmierer noch nicht initialisiert wurden. Beim Programmstart werden alle diese Variablen mit ihren Nullwerten initialisiert. Da die Variablen geändert werden können ist diese Sektion beschreibbar.
- **.COMMENT**
  Diese Sektion enthält Informationen zur Versionskontrolle.

Neben den oben aufgezählten Sektionen gibt es noch weitere, die jedoch für dieses Projekt irrelevant sind.

Damit ein HASH-Vergleich erfolgreich sein kann, müssen die Daten aus denen der Hashwert berechnet soll, immer konstant sein. Das heisst es dürfen

nur die Sektionen des Objektmoduls in den Hash aufgenommen werden, welche schreibgeschützt sind. Folgende Sektionen werden demnach in die Hashberechnung miteinbezogen:

- **.RODATA**
- **.TEXT**

Hashing der Sektionen
Damit ein Hash über die beiden Sektionen gemacht werden kann, muss die Start- und Endadresse der beiden Sektion bekannt sein. Um diese Adressen verfügbar zu machen, werden im Quellcode spezielle Symbole eingefügt. Da die Read-Only DATA Sektion eine andere Art von Daten enthält als die TEXT Sektion sind diese Symbole verschieden.

.RODATA
Wie oben beschrieben enthält diese Sektion konstante globale Variablen. In C sind diese mit *constant* bezeichnete Variablen die ausserhalb einer Funktion definiert werden. Die zur Ermittlung der Start- und Endadresse benötigten Symbole bzw. Variablen sind folgendermassen definiert:

- **const unsigned int** FIPS_rodata_start[]
  ={ 0x46495053, 0x5f726f64, 0x6174615f, 0x73746172 };
- **const unsigned int** FIPS_rodata_end[]
  ={ 0x46495053, 0x5f726f64, 0x6174615f, 0x73746172 };

Die Werte, die diesen Variablen zugewiesen werden, sind irrelevant, da es nur um die Adressen geht.
Der Zugriff auf diese Daten, aus einer anderen Source-Datei heraus, geschieht folgendermassen:

1. Bekanntgabe der Variable mit dem Schlüsselwort **extern**
   extern const unsigned char FIPS_rodata_start[];
   extern const unsigned char FIPS_rodata_end[];
2. Definition eines Zeigers der auf die Variable zeigt
   const unsigned char *p3 = FIPS_rodata_start;
   const unsigned char *p4 = FIPS_rodata_end;

Somit ist es möglich an die Start- und Endadresse der Read-Only DATA Sektion zu kommen.

.TEXT
Diese Sektion enthält ausführbare Computerinstruktionen. In C sind dies beliebige Zeilen Code, unter anderem Funktionen. Zur Ermittlung der Start- und Endadresse werden Funktionspointer definiert. Als Symbole dienen dann Variablen die auf diese Funktionspointer zeigen. Eine solche Definition sieht folgendermassen aus:

- Funktionsdefinition
  void ∗FIPS_ref_point(){ ... };
- Variable auf Funktionspointer
  FIPS_ref_point FIPS_text_start
  FIPS_ref_point FIPS_text_end

Der Zugriff auf diese Daten, aus einer anderen Source-Datei heraus, geschieht folgendermassen:

1. Bekanntgabe der Variable mit dem Schlüsselwort **extern**
   extern const void ∗FIPS_text_start();
   extern const void ∗FIPS_text_end();
2. Definition eines Zeigers der auf die Funktion zeigt
   const unsigned char ∗p1 = FIPS_text_start();
   const unsigned char ∗p2 = FIPS_text_end();

Somit ist es möglich an die Start- und Endadresse der TEXT Sektion zu kommen.

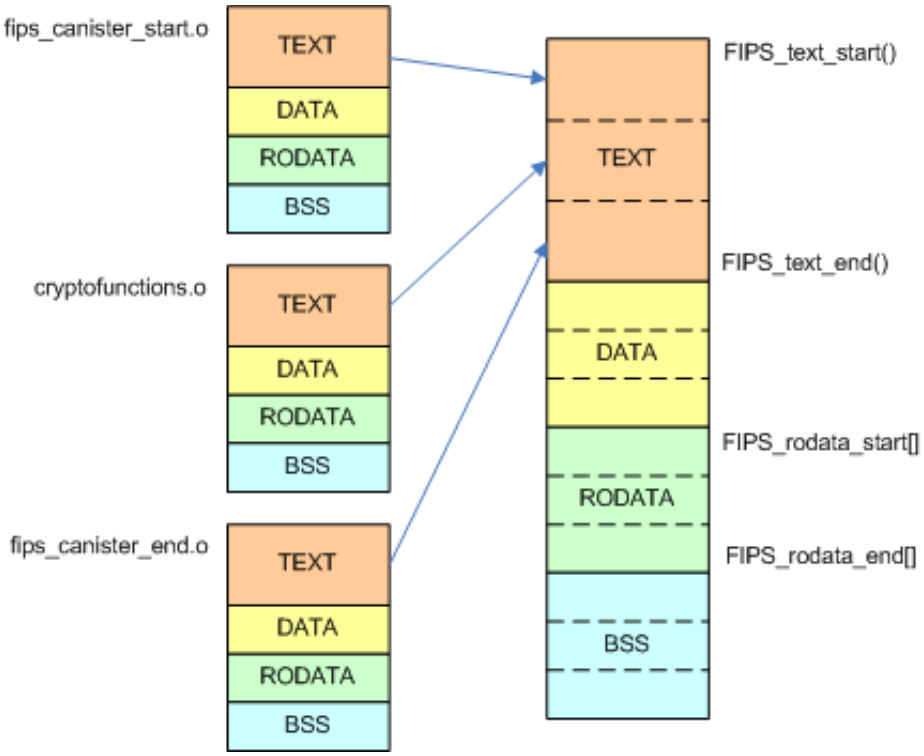

Abbildung 2: Zusammengelinktes Objektmodul aus mehreren einzelnen Modulen

Die Grafik in Abbildung 2 zeigt den Aufbau eines Objektmoduls. Die drei Module auf der linken Seite sind die Startmarkierung (oben), die Endmarkierung (unten) und das Modul mit den Cryptofunktionen (Mitte). Diese Module werden vom Compiler zu einem Objektmodul zusammengelinkt, wobei sich die Sektionen auf der linken Seite aufteilen, um gruppiert im resultierenden Objektmodul zu stehen. Die Symbole die neben dem resultierenden Objektmodul stehen, sollen verdeutlichen, wie die beiden Sektionen erkannt und eingegrenzt werden können.

6.1.4.2 Signatur erstellen

Alle Cryptofunktionen deren Integrität zur Laufzeit überprüft werden muss, befinden sich in der libstrongswan. Dies hat den Vorteil, dass nicht die Integrität der gesamten Applikation überprüft werden muss. Nur diese libstrongswan wird mit den oben beschriebenen Markierungssymbolen versehen.

Da die Sigantur in die Zielapplikation eingearbeitet werden muss, muss die Signatur vorgängig erzeugt werden. Dazu wird ein zusätzliches Programm generiert, welches ausschliesslich zur Erzeugung der Signatur dient. Dieses Programm heisst sigcreator. Es wird im gleichen Makefile.am wie die Zielapplikation definiert und bindet ebenfalls die libstrongswan ein.
Abbildung 4 zeigt welche Dateien sigcreater benötigt und wie es erzeugt wird.

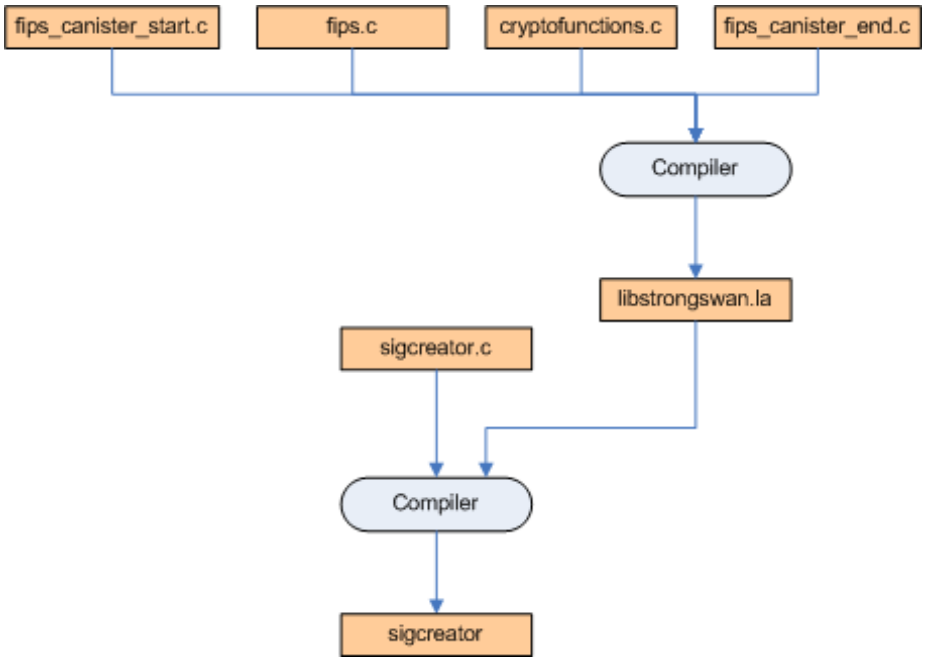

Abbildung 3: Erzeugen von sigcreator

Dieses zusätzliche Programm wird als Abhängigkeit zur eigentlichen Zielapplikation definiert, damit das sigcreator-Programm zuerst erstellt werden kann.
Dazu wird im Makefile.am unter BUILT_SOURCES ein Target namens makesig definiert. Alle Targets die in BUILT_SOURCES eingetragen sind, müssen als erstes erfüllt werden.
Das Target makesig ist folgendermassen definiert:

```
1 makesig: sigcreator
2         ./sigcreator
```

Als Abhängigkeit ist hier das Programm sigcreater eingetragen, welches so als erstes erstellt wird. Als eigentlicher Vorgang steht danach die Ausführung des gerade erzeugten Programms an. Durch Starten von sigcreator wird die Signatur in signature.h geschrieben.

Das Hashing der Sektionen wird wie in Abbildung 4 gezeigt durchgeführt. Die Inhalte der RODATA- und TEXT-Sektion werden hintereinander in einen Buffer kopiert. Über den Inhalt dieses Buffers wird mittels der SHA-1 Hash Funktion der libstrongswan ein SHA-1 Hash erzeugt. Dieser Hash ist die Signatur.

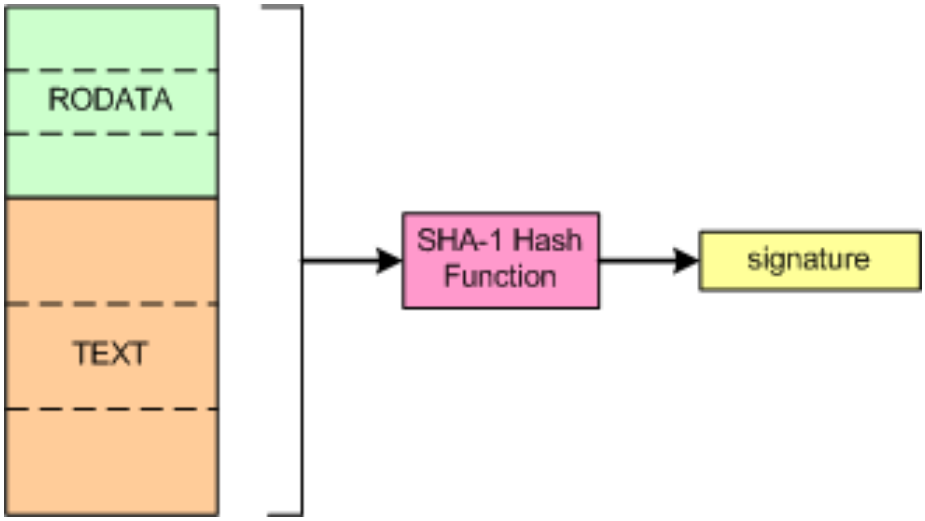

Abbildung 4: Hashing der Sektionen

6.1.4.3 Signatur in Zielapplikation einarbeiten

Da jetzt die Signatur in Form einer Konstante in signature.h vorliegt, kann die Zielapplikation erstellt werden. Die Header-Datei wird dazu in die SOURCE-Liste ins Makefile.am eingetragen.

Der Vergleich ob die Signatur mit dem Incore-Hash übereinstimmt, wird in fips.c, also in der libstrongswan durchgeführt. Dazu muss die Signatur in der libstrongswan bekannt gemacht werden. Zu diesem Zweck wurde in fips.c das Array MYFIPS_signature definiert. Beim Starten einer Applikation, die libstrongswan und deren Integritätstest benutzt, muss die erzeugte Signatur in dieses Array geladen werden. Um dies unabhängig von der Zielapplikation durchzuführen, wird eine zusätzliche premain-Funktion mitcompiliert.
Die premain-Funktion befindet sich in fips_premain.c und wird wie der Name schon sagt, vor der eingetlichen main-Funktion ausgeführt. Bei deren Ausführung wird die Signatur aus der Konstante von signature.h gelesen und ins Array MYFIPS_signature in fips.c kopiert. Somit steht der Integritätstestfunktion die Signatur zur Verfügung.
Abbildung 5 zeigt wie der Buildprozess der Zielapplikation aussieht. Als Beispiel einer Zielapplikation wurde dieses "testprogram" genannt.

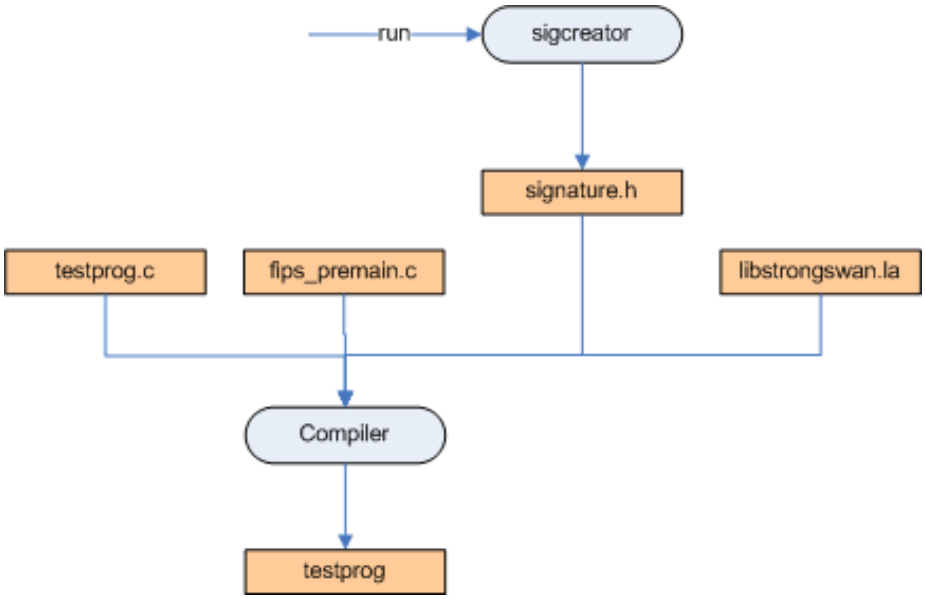

Abbildung 5: Zielapplikation erzeugen

6.1.4.4 Anpassungen an libstrongswan

Alle Dateien die zum Integritätstest gehören, befinden sich in der libstrongswan im Unterordner fips. Anpassungen müssen daher am Makefile.am der libstrongswan und dem configure.in der ganzen Applikation erfolgen.

configure.in

Die Änderungen in configure.in beschränken sich auf das Hinzufügen der ./configure Option um den Integritätstest zu aktivieren.

```
AC_ARG_ENABLE(
    [integrity-test],
    AS_HELP_STRING([--enable-integrity-test],[enable the
        integrity test of strongSwan (default is NO).]),
    [if test x$enableval = xyes; then
        integrity_test=true
        AC_DEFINE(INTEGRITY_TEST)
    fi]
)

AM_CONDITIONAL(USE_INTEGRITYTEST,
        test x$integrity_test = xtrue)
```

Der obere Teil der Konfiguration definiert bei aktiviertem Integritätstest das Compilerflag INTEGRITY_TEST, dass in beliebigen Codestellen verwendet werden kann. Das AM_CONDITIONAL im unteren Teil definiert das Flag USE_INTEGRITYTEST, welches von den Makefile.am-Dateien verwendet wird.

Makefile.am in libstrongswan

Wie oben beschrieben müssen zur Bestimmung des Memorybereichs der Crypto-Funktionen, bestimmte Markierungen eingearbeitet werden. Diese sind in Form von Objektmodulen. Aus diesem Grund muss am Anfang und am Ende eine zusätzliche Source-Datei in die SOURCE-Liste im Makefile.am eingetragen werden. Ausserdem gehören fips.c und fips.h mit den Integritätstestfunktionen auch noch dazu.

```
if USE_INTEGRITYTEST
libstrongswan_la_SOURCES = \
fips/fips_canister_start.c \
fips/fips.c fips/fips.h
else
libstrongswan_la_SOURCES =
endif
```

Das Flag USE_INTEGRITYTEST ist definiert, falls die Integritätstest im ./configure aktiviert wurden, d.h. dann wird der IF-Teil ausgewertet und nebst der Startmarkierung auch die Dateien mit den Integritätstestfuktionen mitcompiliert. Ansonsten werden diese Dateien weggelassen.

```
if USE_INTEGRITYTEST
```

```
2 libstrongswan_la_SOURCES += \
3 fips/fips_canister_end.c
4 endif
```

Auch hier wird nur bei aktiviertem Integritätstest die Datei, die die Endmarkierung kennzeichnet mitcompiliert.

6.2 Self-Tests

6.2.1 Einführung

In der libstrongswan wurden verschiedene Crypto-Funktionen (Hashers, HMAC's, Signers, Crypters, PRF's und RSA) implementiert. Dabei ist jedoch nicht sichergestellt, dass diese korrekt funktionieren. Bei Fehlverhalten könnte es sich um Implementierungsfehler oder böswilligen Veränderungen an der Implementierungen handeln. Es ist auch möglich, dass eine Crypto-Funktion inkonsistenten Output bei verschiedenen Inputlängen generiert. Um die korrekte Funktionsweise dieser Crypto-Funktionen sicherzustellen, müssen diese getestet werden. Dieser Prozess erfolgt mit sogenannten Self-Tests. Dabei ist im Voraus zu einem gegebenen Input-Wert, der zugehörige Output-Wert zu einer Crypto-Funktion bekannt. Diese Wertepaare werden Testvektoren genannt. Die Input-Werte werden an einer Crypto-Funktion übergeben und anschliessend wird der errechnete Output-Wert mit dem Testvektor Output-Wert verglichen. Falls die beiden Werte übereinstimmen, gilt der einzelne Test als bestanden. Bei den Crypters (AES, DES) wird zusätzlich ein Key und einen Initialisierungsvektor zu den Testvektoren gespeichert, da sie für die Verschlüsslung und Entschlüsslung nötig sind.

6.2.2 Ausführung der Self-Tests

Die Self-Tests werden beim Initialisieren der Library automatisch ausgeführt. Das stellt die korrekte Funktionsweise der Crypto-Algorithmen vor der Anwendung sicher. Zudem können die einzelnen Self-Tests unabhängig voneinander vom Anwender zur Laufzeit ausgeführt werden.

6.2.3 Realisierungsvarianten

6.2.3.1 Sichtbarkeit der Testdaten

Variante 1
Testdaten inklusive Testvektoren werden lokal in einer Funktion definiert.

Pro:

- Speicherplatz für die Testdaten wird nur kurzzeitig belegt.
- Testdaten sind in der Testfunktion gekapselt.

Kontra:

- Status der Self-Tests müsste zwischengespeichert werden, sofern er auch nach der Durchführung der Tests verfügbar sein müsste.
- Beim mehrmaligem Durchführen der Self-Tests müssen die Testdaten jedesmal neu angelegt werden.

Variante 2
Testdaten inklusive Testvektoren werden global mit dem Schlüsselwort static in einem Source-File definiert. Das bewirkt, dass die Testdaten innerhalb des Source-Files global verfügbar sind.

Pro:

- Testdaten und Testergebnisse sind immer innerhalb des gleichen Source-Files verfügbar.
- Sichbarkeit auf das jeweilige Source-File ist eingeschränkt.

Kontra:

- Grösserer Speicherplatzverbrauch, weil die Testdaten während der ganzen Laufzeit verfügbar sind.

Variante 3
Testdaten inklusive Testvektoren werden global definiert, d. h. sie sind von überall im Programmcode verfügbar.

Pro:

- Testdaten und Testergebniss sind immer verfügbar.

Kontra:

- Grösserer Speicherplatzverbrauch, weil die Testdaten während der ganzen Laufzeit verfügbar sind.
- Sichbarkeit ist uneingeschränkt. Der Zugriff ist von überall möglich.

6.2.3.2 Gliederung der Testdaten / Testmethoden

Variante 1
Die Testmethoden werden zusammen mit den Testdaten einer Crytpo-Funktionsgruppe in einem Source-File abgelegt.

Pro:

- Hohe Kohäsion, da die Testmethoden mit den Testdaten eine grosse Zusammengehörigkeit haben.

Variante 2
Alle Testmethoden werden in einem Source-File und die Testdaten in einem anderen File abgelegt.

Pro:

- Kombinierter Aufruf der Testmethoden ist einfacher.

Kontra:

- Erhöte Sichbarkeit als nötig.
- Schlechte Kohäsion

6.2.3.3 Entscheid
Die Testvektoren und die Testdaten werden Datei-intern global mit dem Schlüsselwort static definiert. Damit bleiben die Tetdaten und Testvektoren während der ganzen Laufzeit erhalten. Somit kann der Status der einzelnen Self-Tests jederzeit erfragt werden.
Sichtbarkeit Testdaten: Variante 2

Es wird für jede Art von Crypto-Funktionsgruppen (Hasher, Signer, Crypter, RSA /DSA) ein Source-File angelegt. Darin befinden sich die dazughörigen Testdaten / Testvektoren und nötigen Funktionen, um diese Algorithmen zu testen und deren Status zurückzugeben. Mit einem enum-Feld wird in den Testdaten angeben mit welchem Crypto-Algorithmus die Testvektoren geprüft werden sollen (z. B. MD5 oder SHA1).
Gliederung Testdaten / Testmethoden: Variante 1

Ein Manager ist für die Durchführung der Self-Tests der Crypto-Funktionsgruppen verantwortlich, d. h. er ruft deren Testmethode auf. Die Crypto-Funktionsgruppen führen anschliessend die eigentlichen Self-Test für die verschiedenen Alogrithmen innerhalb der Gruppe durch (z.B. MD5 oder SHA1... für die Hash-Gruppe).

6.2.4 Design

6.2.4.1 Gruppierung

Damit nicht für jeden einzelnen Crypto-Algorithmus eine eigene Testmethode geschrieben werden mussen, ist es sinnvoll die Crypto-Algorithmen in Gruppen einzuteilen. Dies erlaubt ähnliche Algorithmen gleich zu behandeln. Sprich es ist egal, ob es sich z. B. um einen MD-5 oder um einen SHA-1 Hahser handelt. Der Ablauf für die Generierung des Hashes bleibt immer derselbe. Dies lässt eine höhere Abstraktion der Implementierung zu.
Folgende Testgruppen wurden definiert:

- Hashers
- HMAC's
- Signers
- PRF's
- Crypters
- RSA

6.2.4.2 Klassendiagramme

Die Testdaten (x_testdata) bestehen aus Informationen zur Crypto-Gruppe (Hasher, HMAC, Crypter, Signer, PRF oder RSA) und kapseln die eigentlichen Testvektoren (x_testvector). Das gilt als Platzhalter für eine Crypto-Gruppe. Die Testvektoren sind ein char-Array mit mehreren Testwerten, bestehend aus Eingabewerten und den zugehörigen Ausgabewerten. Die Testdaten und Testvektoren werden als struct definiert. Sie sind innerhalb des Source-Files global sichtbar.

6.2.4.3 Hasher

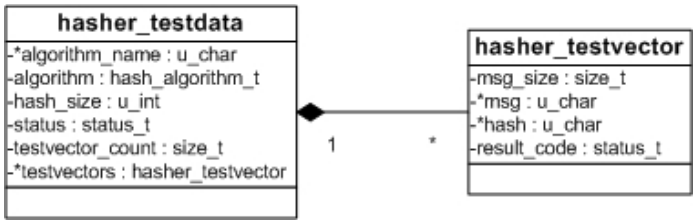

Abbildung 6: Hasher Klassendiagramm

algorithm_name	
Typ:	u_char*
Wertebereich:	Hasher_xxx
Beschreibung:	Name des Hash-Algorithmus.
algorithm	
Typ:	hash_algorithm_t
Wertebereich:	MD5, SHA1, SHA256, SHA384, SHA512
Beschreibung:	Definiert den verwendeten Hash-Algorithmus.
hash_size	
Typ:	u_int
Wertebereich:	16, 20, 32, 48, 64
Beschreibung:	Definiert die Länge des Hashes in Bytes.
status	
Typ:	status_t
Wertebereich:	SUCCESS (0), FAILED (1)
Beschreibung:	Gibt an, ob der Self-Test erfolgreich war oder nicht.
testvector_count	
Typ:	size_t
Wertebereich:	int
Beschreibung:	Anzahl Testvektoren.
testvectors	
Typ:	hasher_testvector*
Wertebereich:	-
Beschreibung:	Pointer auf ein Array mit Testvektoren vom Typ hasher_testvector.

Tabelle 3: Beschreibung zu Struct hasher_testdata

msg_size	
Typ:	size_t
Wertebereich:	-
Beschreibung:	Gibt die Länge der Message in Bytes an.
msg	
Typ:	u_char*
Wertebereich:	-
Beschreibung:	Die Message als C-String.
hash	
Typ:	u_char*
Wertebereich:	-
Beschreibung:	Der Hash-Wert als C-String.
result_code	
Typ:	status_t
Wertebereich:	SUCCESS (0), FAILED (1)
Beschreibung:	Gibt an, ob der zugehörige Test erfolgreich war oder nicht.

Tabelle 4: Beschreibung zu Struct hasher_testvector

6.2.4.4 HMAC's

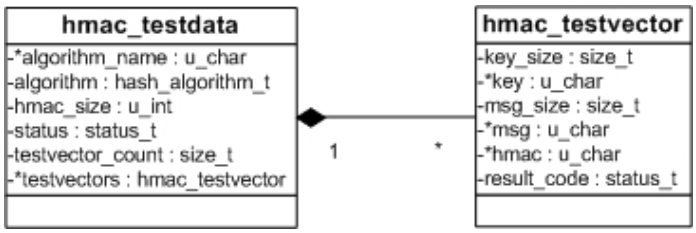

Abbildung 7: HMAC's Klassendiagramm

algorithm_name	
Typ:	u_char*
Wertebereich:	HMAC_xxx
Beschreibung:	Name des HMAC-Algorithmus.
algorithm	
Typ:	hash_algorithm_t
Wertebereich:	MD5, SHA1, SHA256, SHA384, SHA512
Beschreibung:	Definiert den verwendeten Hash-Algorithmus.
hmac_size	
Typ:	u_int
Wertebereich:	variabel
Beschreibung:	Definiert die Länge der HMAC-Signatur in Bytes.
status	
Typ:	status_t
Wertebereich:	SUCCESS (0), FAILED (1)
Beschreibung:	Gibt an, ob der Self-Test erfolgreich war oder nicht.
testvector_count	
Typ:	size_t
Wertebereich:	int
Beschreibung:	Anzahl Testvektoren.
testvectors	
Typ:	hmac_testvector*
Wertebereich:	-
Beschreibung:	Pointer auf ein Array mit Testvektoren vom Typ hmac_testvector.

Tabelle 5: Beschreibung zu Struct hmac_testdata

key_size	
Typ:	size_t
Wertebereich:	-
Beschreibung:	Gibt die Länge des Schlüssels in Bytes an.
key	
Typ:	u_char*
Wertebereich:	-
Beschreibung:	Der Schlüssel als C-String.
msg_size	
Typ:	size_t
Wertebereich:	-
Beschreibung:	Gibt die Länge der Message in Bytes an.
msg	
Typ:	u_char*
Wertebereich:	-
Beschreibung:	Die Message als C-String.
hmac	
Typ:	u_char*
Wertebereich:	-
Beschreibung:	Die HMAC-Signatur als C-String.
result_code	
Typ:	status_t
Wertebereich:	SUCCESS (0), FAILED (1)
Beschreibung:	Gibt an, ob der zugehörige Test erfolgreich war oder nicht.

Tabelle 6: Beschreibung zu Struct hmac_testvector

6.2.4.5 Signer

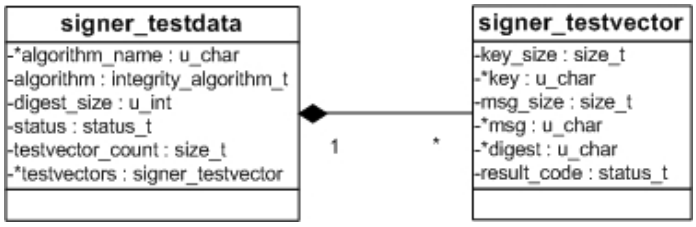

Abbildung 8: Signer Klassendiagramm

algorithm_name	
Typ:	u_char*
Wertebereich:	Signer_xxx
Beschreibung:	Name des Signer-Algorithmus.
algorithm	
Typ:	integrity_algorithm_t
Wertebereich:	MD5, SHA1, SHA256, SHA384, SHA512
Beschreibung:	Definiert den verwendeten Hash-Algorithmus.
digest_size	
Typ:	u_int
Wertebereich:	variabel
Beschreibung:	Definiert die Länge der Signatur in Bytes.
status	
Typ:	status_t
Wertebereich:	SUCCESS (0), FAILED (1)
Beschreibung:	Gibt an, ob der Self-Test erfolgreich war oder nicht.
testvector_count	
Typ:	size_t
Wertebereich:	int
Beschreibung:	Anzahl Testvektoren.
testvectors	
Typ:	signer_testvector*
Wertebereich:	-
Beschreibung:	Pointer auf ein Array mit Testvektoren vom Typ signer_testvector.

Tabelle 7: Beschreibung zu Struct signer_testdata

key_size	
Typ:	size_t
Wertebereich:	-
Beschreibung:	Gibt die Länge des Schlüssels in Bytes an.
key	
Typ:	u_char*
Wertebereich:	-
Beschreibung:	Der Schlüssel als C-String.
msg_size	
Typ:	size_t
Wertebereich:	-
Beschreibung:	Gibt die Länge der Message in Bytes an.
msg	
Typ:	u_char*
Wertebereich:	-
Beschreibung:	Die Message als C-String.
digest	
Typ:	u_char*
Wertebereich:	-
Beschreibung:	Die Signatur als C-String.
result_code	
Typ:	status_t
Wertebereich:	SUCCESS (0), FAILED (1)
Beschreibung:	Gibt an, ob der zugehörige Test erfolgreich war oder nicht.

Tabelle 8: Beschreibung zu Struct signer_testvector

6.2.4.6 Crypter

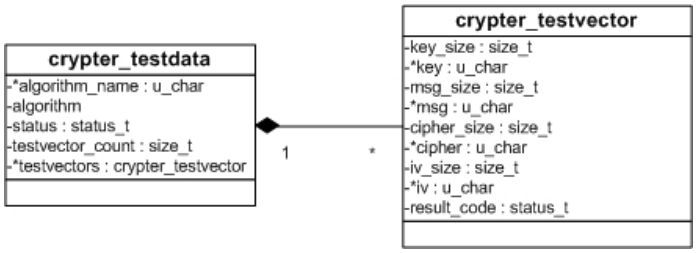

Abbildung 9: Crypter Klassendiagramm

algorithm_name	
Typ:	u_char*
Wertebereich:	Crypter_xxx
Beschreibung:	Name des Crypter-Algorithmus.
algorithm	
Typ:	encryption_algorithm_t
Wertebereich:	AES, DES, 3DES
Beschreibung:	Definiert den verwendeten Encryption-Algorithmus.
status	
Typ:	status_t
Wertebereich:	SUCCESS (0), FAILED (1)
Beschreibung:	Gibt an, ob der Self-Test erfolgreich war oder nicht.
testvector_count	
Typ:	size_t
Wertebereich:	int
Beschreibung:	Anzahl Testvektoren.
testvectors	
Typ:	crypter_testvector*
Wertebereich:	-
Beschreibung:	Pointer auf ein Array mit Testvektoren vom Typ crypter_testvector.

Tabelle 9: Beschreibung zu Struct crypter_testdata

key_size	
Typ:	size_t
Wertebereich:	-
Beschreibung:	Gibt die Länge des Schlüssels in Bytes an.
key	
Typ:	u_char*
Wertebereich:	-
Beschreibung:	Der Schlüssel als C-String.
msg_size	
Typ:	size_t
Wertebereich:	-
Beschreibung:	Gibt die Länge der Message in Bytes an.
msg	
Typ:	u_char*
Wertebereich:	-
Beschreibung:	Die Message als C-String.
cipher_size	
Typ:	size_t
Wertebereich:	-
Beschreibung:	Gibt die Länge des Ciphers in Bytes an.
cipher	
Typ:	u_char*
Wertebereich:	-
Beschreibung:	Der Cipher als C-String.
iv_size	
Typ:	size_t
Wertebereich:	-
Beschreibung:	Gibt die Länge des Initialisierungsvektors in Bytes an.
iv	
Typ:	u_char*
Wertebereich:	-
Beschreibung:	Den Initialisierungsvektor als C-String.
result_code	
Typ:	status_t
Wertebereich:	SUCCESS (0), FAILED (1)
Beschreibung:	Gibt an, ob der zugehörige Test erfolgreich war oder nicht.

Tabelle 10: Beschreibung zu Struct crypter_testvector

6.2.4.7 PRF's

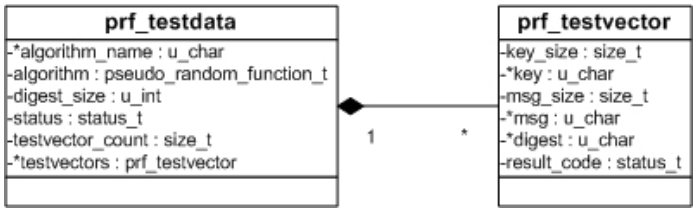

Abbildung 10: PRF's Klassendiagramm

algorithm_name	
Typ:	u_char*
Wertebereich:	PRF_xxx
Beschreibung:	Name des PRF-Algorithmus.
algorithm	
Typ:	pseudo_random_function_t
Wertebereich:	AES, MD5, SHA1, SHA256, SHA384, SHA512
Beschreibung:	Definiert den verwendeten PRF-Algorithmus.
digest_size	
Typ:	u_int
Wertebereich:	variabel
Beschreibung:	gibt die Länge des Digest's an.
status	
Typ:	status_t
Wertebereich:	SUCCESS (0), FAILED (1)
Beschreibung:	Gibt an, ob der Self-Test erfolgreich war oder nicht.
testvector_count	
Typ:	size_t
Wertebereich:	int
Beschreibung:	Anzahl Testvektoren.
testvectors	
Typ:	prf_testvector*
Wertebereich:	-
Beschreibung:	Pointer auf ein Array mit Testvektoren vom Typ prf_testvector.

Tabelle 11: Beschreibung zu Struct prf_testdata

key_size	
Typ:	size_t
Wertebereich:	-
Beschreibung:	Gibt die Länge des Schlüssels in Bytes an.
key	
Typ:	u_char*
Wertebereich:	-
Beschreibung:	Der Schlüssel als C-String.
msg_size	
Typ:	size_t
Wertebereich:	-
Beschreibung:	Gibt die Länge der Message in Bytes an.
msg	
Typ:	u_char*
Wertebereich:	-
Beschreibung:	Die Message als C-String.
digest	
Typ:	u_char*
Wertebereich:	-
Beschreibung:	Der Digest als C-String.
result_code	
Typ:	status_t
Wertebereich:	SUCCESS (0), FAILED (1)
Beschreibung:	Gibt an, ob der zugehörige Test erfolgreich war oder nicht.

Tabelle 12: Beschreibung zu Struct prf_testvector

6.2.4.8 RSA

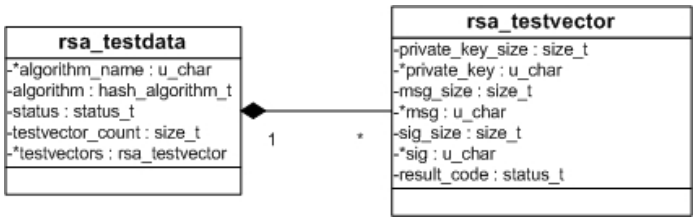

Abbildung 11: RSA Klassendiagramm

algorithm_name	
Typ:	u_char*
Wertebereich:	RSA_xxx
Beschreibung:	Name des RSA-Algorithmus.
algorithm	
Typ:	hash_algorithm_t
Wertebereich:	SHA1, SHA256, SHA384, SHA512
Beschreibung:	Definiert den verwendeten Hash-Algorithmus zum Signieren.
status	
Typ:	status_t
Wertebereich:	SUCCESS (0), FAILED (1)
Beschreibung:	Gibt an, ob der Self-Test erfolgreich war oder nicht.
testvector_count	
Typ:	size_t
Wertebereich:	int
Beschreibung:	Anzahl Testvektoren.
testvectors	
Typ:	rsa_testvector*
Wertebereich:	-
Beschreibung:	Pointer auf ein Array mit Testvektoren vom Typ rsa_testvector.

Tabelle 13: Beschreibung zu Struct rsa_testdata

private_key_size	
Typ:	size_t
Wertebereich:	-
Beschreibung:	Gibt die Länge des privaten Schlüssels in Bytes an.
private_key	
Typ:	u_char*
Wertebereich:	-
Beschreibung:	Der private Schlüssel im DER-Format.
msg_size	
Typ:	size_t
Wertebereich:	-
Beschreibung:	Gibt die Länge der Message in Bytes an.
msg	
Typ:	u_char*
Wertebereich:	-
Beschreibung:	Die Message als C-String.
sig_size	
Typ:	size_t
Wertebereich:	-
Beschreibung:	Gibt die Länge der Signatur in Bytes an.
sig	
Typ:	u_char*
Wertebereich:	-
Beschreibung:	Die RSA-Signatur.
result_code	
Typ:	status_t
Wertebereich:	SUCCESS (0), FAILED (1)
Beschreibung:	Gibt an, ob der zugehörige Test erfolgreich war oder nicht.

Tabelle 14: Beschreibung zu Struct rsa_testvector

6.2.5 Testablauf

Der Self-Test Manager stellt die Funktion run_selftests() zur Verfügung, um die Self-Tests der Crypto-Funktionen in folgender Reihenfolge auszuführen.

- hashers_selftest();
- hmac_selftest();
- signers_selftest();
- prfs_selftest();
- crypters_selftest();
- rsa_selftest();

Auf den folgenden zwei Seiten wird der Ablauf als Sequenzdiagramm dargestellt. Aus Platzgründen wurde es auf zwei Seiten aufgeteilt.

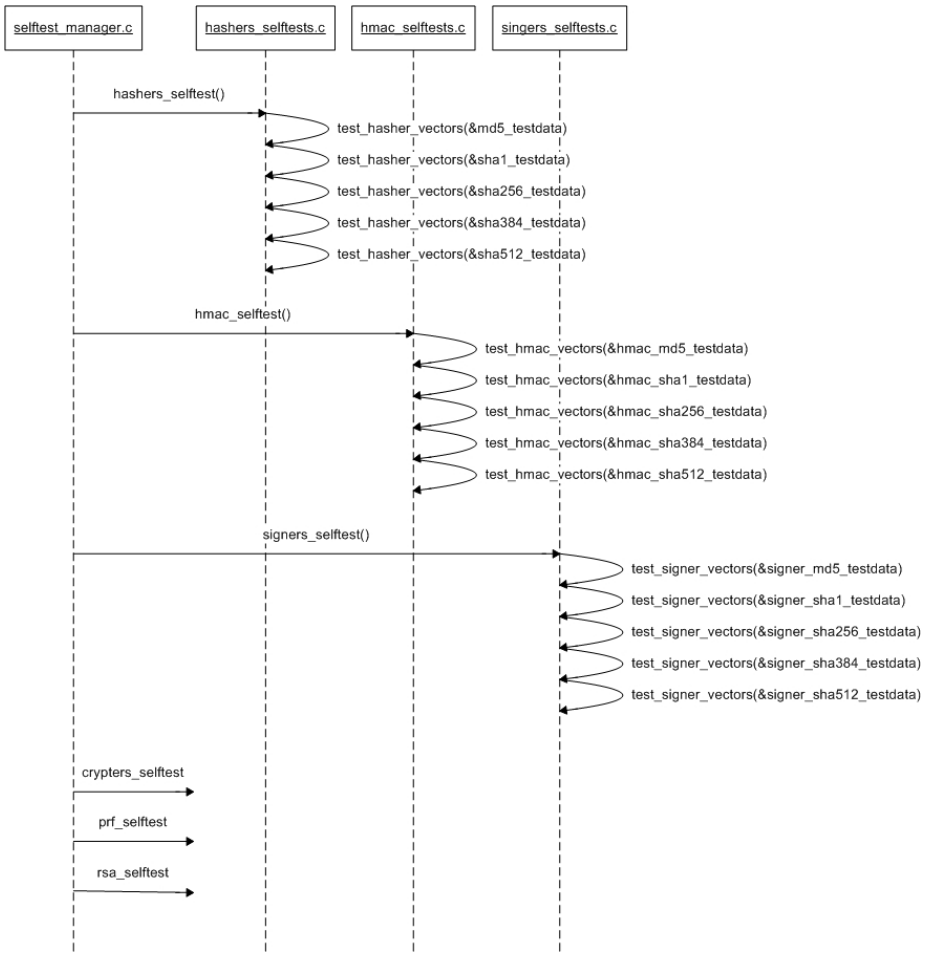

Abbildung 12: Sequenzdiagramm Self-Tests, Teil 1

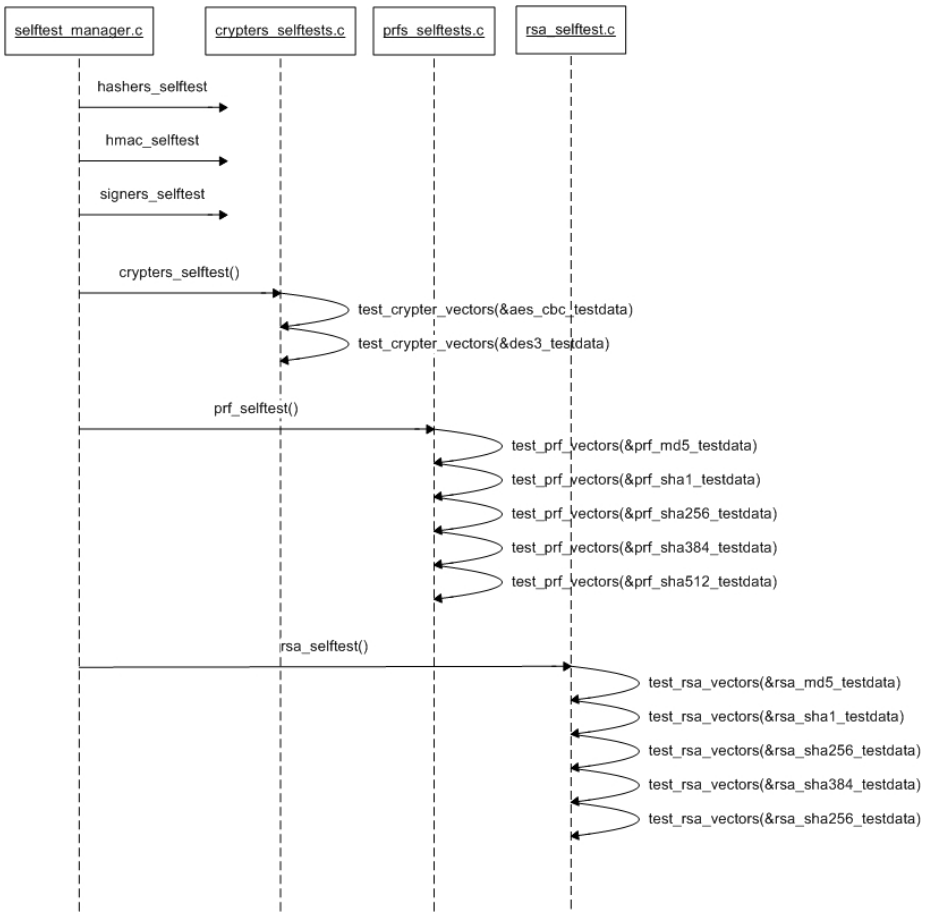

Abbildung 13: Sequenzdiagramm Self-Tests, Teil 2

6.2.6 Testverfahren

6.2.6.1 Hashers

Zu den Hashern gehören die Algorithmen MD-5, SHA-1, SHA-256, SHA-384 und SHA-512. Ein Testvektor umfasst die Nachricht (Input) und den dazugehörigen Hash (Ouput). Hasher werden getestet, indem die Nachricht der Hash-Funktion übergeben wird und der errechnete Hash-Wert anschliessend mit dem Hash-Wert im Testvektor verglichen wird. Falls die verglichenen Werte nicht übereinstimmen, wird der Status auf FAILED (1) gesetzt. Da Hasher nur One-Way Funktionen sind, muss nur die Hinrichtung getestet werden.

6.2.6.2 HMAC's

Der HMAC Algorithmus muss seperat getestet werden, weil zusäztlich zu den Hash-Algorithmen ein Key zum Signieren nötig ist. Ein Testvektor umfasst die Nachricht (Input), den Schlüssel und den dazugehörigen Keyed-Hash (Ouput). Zuerst wird der Schlüssel bei der HMAC-Funktion registriert. Anschliessend wird die Nachricht der HMAC-Funktion übergeben und der errechnete Keyed-Hash-Wert mit dem Keyed-Hash-Wert im Testvektor verglichen wird. Falls die verglichenen Werte nicht übereinstimmen, wird der Status auf FAILED (1) gesetzt. Da HMAC's nur One-Way Funktionen sind, muss nur die Hinrichtung getestet werden.

6.2.6.3 Signers

Die Gruppe der Signers setzen u. a. auf HMAC-Funktionen auf. Es können jedoch auch andere Crypto-Algorithmen verwendet werden um eine Signatur zu erstellen. Ein Testvektor umfasst die Nachricht (Input), den Schlüssel und den dazugehörigen Keyed-Hash (Ouput). Zuerst wird der Schlüssel bei der Signer-Funktion registriert. Nachfolgend wird die Nachricht der Signer-Funktion übergeben und der errechnete Keyed-Hash-Wert mit dem Keyed-Hash-Wert im Testvektor verglichen wird. Falls die verglichenen Werte nicht übereinstimmen, wird der Status auf FAILED (1) gesetzt. Da Signers nur One-Way Funktionen sind, muss nur die Hinrichtung getestet werden.

6.2.6.4 Crypters

Zu den Cryptern gehören die Algorithmen AES, DES, 3DES. Ein Testvektor umfasst die Nachricht (Input), den Schlüssel, den Initialisierungsvektor und den dazugehörigen Cipher (Ouput). Zuerst wird bei der Crypter-Funktion der Schlüssel registriert. Nachfolgend wird die Nachricht mit dem Initialisierungsvektor der Encrypter-Funktion übergeben und verschlüsselt. Anschliessend wird der errechnete Cipher-Wert mit dem Cipher-Wert im Testvektor verglichen. Das Entschlüsseln eines Ciphers wird folgendermassen getestet. Der Cipher-Wert vom Testvektor wird mit dem gleichen Schlüssel und Initialisierungsvektor der Decrypter-Funktion übergeben. Anschliessend wird die errechnete Nachricht mit der Ursprungsnachricht vom Testvektor verglichen. Falls die Vergleiche beim Verschlüsseln oder beim Entschlüsseln nicht übereinstimmten, wird der Status auf FAILED (1) gesetzt. Letzterer wird Conditional-Test genannt.

6.2.6.5 PRF's

PRF's bauen auf den grundlegenden Crypto-Algorithmen wie HMAC und AES auf. Um PRF's zu testen, können die gleichen Testvektoren wie für HMAC oder AES verwendet werden. Jedoch ist zu beachten, dass der errechnete PR-Wert meistens noch auf eine gewisse Länge verkürzt wird. Der Testvektor umfasst eine Nachricht (Input), einen Schlüssel, sowie einen dazugehörigen PR-Wert (Output). Zuerst wird der Schlüssel bei der PRF-Funktion registriert. Anschliessend wird die Nachricht der PRF-Funktion übergeben und der errechnete PRF-Wert mit dem PRF-Wert im Testvektor verglichen wird. Falls die verglichenen Werte nicht übereinstimmen, wird der Status auf FAILED (1) gesetzt.

6.2.6.6 RSA

RSA funktioniert asymmetrisch mit zwei verschiedenen Schlüsseln. Der Private-Key wird zum Erzeugen und der Public-Key zum Verifizieren einer Signatur benötigt. Zum Verschlüsseln einer Nachricht wird jedoch der Public-Key und zum entschlüsseln der der Private-Key genutzt. Ein Private-Key besteht aus (d, n), wobei d der Entschlüsselungsexponent und n der gemeinsame Generator ist. Ein Public-Key besteht aus (e, n), wobei e der Verschlüsselungsexponent ist. Der Generator n wird aus zwei völlig zufällig gewählten Primzahlen errechnet (n = p * q). Die Funktionalitäten in der libstrongswan beschränken sich auf das Erzeugen von Signaturen, somit muss nur dieses Verfahren getestet werden. Ein Testvektor umfasst den Private- sowie den Public-Key, die Nachricht (Input) und die Signatur (Output). Mit dem Private-Key wird wird aus der Nachricht eine Signatur errechnet und aschliessend wird diese mit der zu erwartenden Signatur im Testvektor verglichen. Falls diese Werte nicht überreinstimmen wird der Status auf FAILED (1) gesetzt. Danach wird die errechnete Signatur mit dem Public-Key überprüft. Falls dieser Test fehlschlägt, wird hier der Status im Testvektor ebenfalls auf FAILED (1) gesetzt.

6.2.7 Testvektoren

Die Testvektoren stammen aus den offiziellen RFC's und vom NIST. Die genaue Herkunft der Testvektoren ist im Source-Code vermerkt. Ausser für 3DES und RSA wurden keine passenden Test-Cases gefunden. Für 3DES und RSA wurden darum eigene Testvektoren definiert.

6.2.8 Debug-Ausgaben

Die detaillierten Self-Test Debug-Ausgaben können mithilfe der in den libstrongswan bereits vordefinierten Debug-Levels gesteuert werden. Standardmässig werden diese Debug-Informationen ausgegeben, wenn mind. das Debug-Level 2 definiert wurde. Die Debug-Informationen enthalten die folgenden wichtigen Angaben:

- Crypto-Algorithmus
- Anzahl Testvektoren pro Crypto-Algorithmus
- Input-Vektor (Message) Inhalt und Länge
- Output-Vektor (Digest) Länge und Ist- / Sollvergleich.
- Ausgabe des Testresultates jedes einzelnen Tests.

6.2.9 Self-Test Status

Bei den Self-Tests der Gruppen Hashers, HMAC, Signers, PRF's, Crypters, RSA wird das Ergebnis des Testes in ein status_t Flag geschrieben. Dieses Status-Flag ist über die gesamte Programmlauftzeit gültig. Bevor die Self-Tests das erste mal ausgeführt werden, ist das Flag mit SUCCESS (0) initialisiert. Beim Ablauf des Self-Tests wird es beim Fehlschlagen eines Tests auf FAILED (1) gesetzt oder bei einer erfolgreichen Prüfung auf SUCCESS (0) belassen. Dieses Status-Flag kann jederzeit über den Self-Test Manager für jede Untergruppe (z.B. Hasher -> MD5) abgefragt werden.

6.2.10 Kritische Crypto-Funktionen

Im Self-Test Manager können kritische Crypto-Funktionen definiert werden. Auf dieser Liste kann jeder Crypo-Algorithmus von libstrongswan definiert werden. Die Idee der kritischen Crypto-Funktionen ist, die eingetragenen Crypto-Funktionen beim Fehlschlagen des Self-Tests sperren zu können.

6.2.11 Tests

6.2.11.1 Performance-Tests

In den Anforderungsspezifikationen ist für die Ausführung aller Self-Tests die obere Schranke von zwei Sekunden festgelegt worden. Bei den folgenden Tests werden die Self-Tests 100 mal durchlaufen und nachher den Mittelwert der Laufzeiten ausgerechnet. Pro Self-Test Durchgang werden die Tests aller Crypto-Gruppen ausgeführt.

Gemessen wurden folgende Zeiten bei einem Intel Pentium 4:
ohne Debug-Ausgaben auf der Konsole: ca. 400ms / Self-Test Durchgang
mit Debug-Ausgaben auf der Konsole: ca. 500ms / Self-Test Durchgang

Aus diesen Messwerten ist ersichtlich, dass die Anforderungen eingehalten wurden.

6.2.11.2 Funktionstests

Es werden Funktionstests durchgeführt, um die Implementation auf das wünschenswerte Verhalten zu testen. Das folgende Vorgehen wird dafür angewandt...

- Testvektor-Message verändern
- Test ausführen und prüfen, ob ein Fehler erkannt wird.
- Status der Cryptofunktion abrufen.
- Falls FAILED (0) zurückgegeben wird ist dieser Test bestanden un der Fehler in der Message wurde erkannt.

6.2.12 Anpassungen an der Library "libstrongswan"

Damit die Funktionalität der RSA Self-Tests realisiert werden konnte, mussten Anpassungen an der bestehenden "libstrongswan" Library vorgenommen werden.

6.2.12.1 Private-Key Erzeugung

Mit den bestehenden RSA-Funktionen der "libstrongswan" gab es keine Möglichkeit einen Private-Key mithilfe von vorgegebenen Generatorwerten (p, q) und Ver- und Entschlüsslungsexponenten (d, e) zu generieren, welche jedoch für das Erzeugen eins Private-Key's mit Vektorwerten nötig ist. Die Datei rsa_private_key.c der "libstrongswan" wurde wegen diesen Gründen um die folgende Funktion ergänzt:

```
1 rsa_private_key_t *rsa_private_key_create_with_values(
2         size_t key_size, u_char* p_str, u_char* q_str,
3         u_char* d_str, u_char* e_str)
```

6.2.12.2 Public-Key Erzeugung

Damit eine RSA-Signatur verifiziert werden kann, benötigt man den Public-Key. In der "libstrongswan" war bereits einen leeren Funktionsrumpf für das Erzeugen des Public-Key's aus dem Private-Key vorhanden. Im Rahmen dieser Projektarbeit wurde diese Funktion implementiert, da sie für die RSA Self-Tests benötigt wird. Die Funktion hat folgende Signatur und befindet sich in rsa_private_key.c:

```
1 rsa_public_key_t *get_public_key(
2         rsa_private_key_t *private_key)
```

Die obengenannte Funktion delegiert den Aufruf an die folgende Funktion in der Datei rsa_public_key.c:

```
1 rsa_public_key_t *rsa_public_key_create_with_values(
2         size_t k, mpz_t n, mpz_t e)
```

7 Schlussfolgerungen

7.1 Resultate

Während dieser Arbeit wurde strongSwan um folgende Komponenten erweitert:

- Integritätstest
- Self-Tests

Der Integritätstest verifiziert den Inhalt der libstrongswan im Speicher. Dies wird dadurch erreicht, dass ein Incore-Hash über den betreffenden Speicherbereich errechnet wird und mit einer bei der Installation erzeugten Signatur verglichen wird.
Der Integritätstest kann wahlweise mit- oder auch nicht compiliert werden. Dies geschieht durch ein Flag beim configure-Aufruf. Standardmässig wird der Integritätstest nicht mitcompiliert.
Die Verifizierung muss manuell aus der Applikation aufgerufen werden, die die libstrongswan implementiert.

Die Self-Tests führen für alle Cryptofunktionen, der libstrongswan, Self-Tests mit bekannten Vektoren durch. Die verwendeten Testvektoren stammen zu einem grossen Teil aus den Standards der jeweiligen Crypto-Funktion.
Über einen Self-Test-Manager können alle Tests bequem und zentral ausgeführt werden. Des weiteren werden alle Ergebnisse des Test festgehalten, damit diese bei einem Fehlerfall genauer analysiert werden können.
Die Selftest-Funktion kann wahlweise mitcompiliert werden. Dies geschieht durch ein Flag bei configure. Standardmässig werden die Selftests mitcompiliert.
Die Self-Tests müssen manuell aus der Applikation aufgerufen werden, die die libstrongswan implementiert.

7.2 Ausblick

Da der Integritätstest über ein Incore-Hashing sehr komplex und plattformabhängig ist, wird diese Erweiterung standardmässig nicht mitcompiliert. Es kann nicht garantiert werden, dass der Integritätstest auf allen Plattformen korrekt funktioniert. Als weiteren Schritt ist daher das Austesten der Integritätsfunktion auf anderen Plattformen zum empfehlen.

Die Selftests für de bestehenden Crypto-Funktionen wurden wunschgemäss implementiert. Das Self-Test-Modul ist beliebig erweiterbar.

8 Glossar

SSL	[Wikipedia] Steht für Secure Sockets Layer, ein Netzwerkprotokoll zur sicheren Übertragung von Daten u. a. von Internetseiten.
OpenSSL	[Wikipedia] OpenSSL ist eine Open-Source-Implementierung des SSL-Protokolls und bietet darüber hinaus weitergehende Funktionen zur Zertifikat-Verwaltung und zu unterschiedlichen kryptographischen Funktionen.
strongSwan	[Wikipedia] strongSwan ist eine vollständige IPSec-Implementierung für Linux 2.4 und 2.6 Kernel.
libstrongswan	Eine Library von strongSwan welche die Cryptofunktionen enthält.
FIPS	[Wikipedia] Federal Information Processing Standard ist die Bezeichnung für technische Standards, die das Institute of Computer Science an Technology (ICST) in den USA herausgibt.
FIPS 140-2	[Wikipedia] Die FIPS 140-2 Publikation ist ein U.S. government computer security standard zu Verifizierung kryptografischer Module. Der Titel lautet *Security Requirements for Cryptographic Modules*.
RSA	[Wikipedia] RSA ist ein Asymmetrisches Kryptosystem, das sowohl zur Verschlüsselung als auch zur digitalen Signatur verwendet werden kann.
DES, 3DES	[Wikipedia] Der Data Encryption Standard ist ein weit verbreiteter symmetrischer Verschlüsselungsalgorithmus.
AES	[Wikipedia] Der Advanced Encryption Standard ist ein symmetrisches Kryptosystem welcher Nachfolger für DES bzw. 3DES ist.
PRF	[Wikipedia] In der Krypthographie versteht man unter Pseudo Random Functions ein Sammlung von effektiv berechenbaren Funktionen für Pseudo-Zufallszahlen.
KAT	Known Answer Tests (KATs) sind Tests, bei denen eine bestimmte Eingabe zu einer bestimmten Ausgabe führen muss, die man bereits kennt.

MD5	[Wikipedia] Message-Digest Algorithm 5 ist eine weit verbreitete kryptographische Hash-Funktion, die einen 128-Bit-Hashwert erzeugt.
SHA-1, SHA-xxx	[Wikipedia] Der Begriff Secure Hash Algorithm, bezeichnet eine Gruppe standardisierter kryptographischer Hash-Funktionen.
Hash-Funktion	[Wikipedia] Eine Hash-Funktion ist eine Funktion bzw. Abbildung, die zu einer Eingabe aus einer üblicherweise grossen Quellmenge eine Ausgabe aus einer im Allgemeinen kleineren Zielmenge erzeugt.
Conditional Tests	Das ist ein Testverfahren mit dem die Verschlüsselung und Entschlüsselung eines Crypto-Algorithmus testen kann. Es wird eine Nachricht verschlüsselt und dieser entstehende Output wieder entschlüsselt. Das Resultat nach dem Entschlüsseln wird mit der Anfangsnachricht verglichen. Falls die beiden Werte übereinstimmen wurde der Test erfolgreich durchgeführt.

9 Verzeichnisse

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[Bergo01] Felipe Bergo. *Using GNU Autoconf, Automake, Autoheader.* http://seul.org/docs/autotut/, zuletzt besucht: 16.04.2007

[FIPS PUB 140-2] NIST. *FIPS PUB 140-2.* http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf, zuletzt besucht: 19.04.2007

[OpenSSL FIPS 140-2 Security Policy] OpenSSL *FIPS 140-2 Security Policy.* http://www.openssl.org/docs/fips/SecurityPolicy-1.1.1.pdf, zuletzt besucht: 19.04.2007

[OpenSSL FIPS 140-2 User Guide] OpenSSL *FIPS 140-2 User Guide.* http://www.openssl.org/docs/fips/UserGuide-1.1.1.pdf, zuletzt besucht: 19.04.2007

[OpenSSL FIPS 140-2 Source Code] OpenSSL *FIPS 140-2 Source Code.* http://www.openssl.org/source/openssl-fips-1.1.1.tar.gz, zuletzt besucht: 25.04.2007

[Wikipedia] Wikipedia http://de.wikipedia.org, zuletzt besucht: 06.07.2007

Inhalt der CD

Datei	Beschreibung
\	
index.htm	Index-File
installationsanleitung.html	Installationsanleitung
$\backslash Dokumente\backslash$	
Bericht.pdf	Bericht der FIPS 140-2 Zertifizierung für strongSwan
kurzfassung_studienarbeit.doc	Abstract, das an die Schule verschickt wird.
$\backslash Dokumente\backslash Projektplanung\backslash$	
Projektplan.xls	Der Projekt-Zeitplan
$\backslash Dokumente\backslash OriginaleAufgabenstellung\backslash$	
Studienarbeit_2007_FIPS-140-2_strongSwan.doc	Die originale Aufgabenstellung
$\backslash Dokumente\backslash ReferenzierteDokumente\backslash FIPSPUB\backslash$	
fips1402.pdf	Offizielles FIPS PUB 140-2 Dokument.
$\backslash Dokumente\backslash ReferenzierteDokumente\backslash OpenSSLFIPS140-2\backslash$	
SecurityPolicy-1.1.1_733.pdf	OpenSSL FIPS 140-2 Seurity Policy Dokument.
UserGuide-1.1.1.pdf	OpenSSL FIPS 140-2 User Guide Dokument.
$\backslash Code\backslash strongSwanmitselftestsundintegritytest\backslash$	
*	Source von strongSwan mit Integritätstest und Self-Tests.
$\backslash Code\backslash BerichtalsEclipseLatexProjekt\backslash$	
*	Latex-Source des Berichts in Form eines Eclipse-Projekts.
$\backslash Visuals\backslash Designdiagramme\backslash$	
Integrity Diagrams.vsd	Diagramme zum Integritätstest
Self-Tests DCD.vsd	Diagramme zu den Self-Tests
$\backslash Visuals\backslash IncoreHashing\backslash$	
Build.vsd	Diagramme zum Buildvorgang
Hashing.vsd	Diagramme zum Incorehashing
$\backslash Visuals\backslash UseCasediagramme\backslash$	
Use-Case Diagramm.png	Use-Case Diagramm

Tabelle 15: Inhalt der CD

A Anhang

A.1 Persönliche Berichte

A.1.1 Bruno Krieg

Anfang April starteten Daniel Wydler und ich die zweite Studienarbeit "FIPS-140-2 Zertifizierung von strongSwan". Die ersten Wochen standen uns zum Einarbeiten zur Verfügung. Die offiziellen OpenSSL Dokumente mussten studiert werden, um anschliessend Überlegungen anzustellen, welche Teile in unserer strongSwan FIPS Erweiterung realisiert werden konnten.

Da ich sonst eigentlich nie mit Linux arbeite, musste ich mich zuerst mit der Umgebung vertraut machen. Darum dauerte es eine Weile bis ich wirklich startklar war. Doch am Betriebssystem fand ich sehr schnell gefallen.

Nach dem ersten Monat fingen wir mit dem Prototyp an. Das Projekt konnte sehr gut aufgeteilt werden, da es aus zwei elementaren, unabhängigen Teilen bestand. Die Integritätstests wurden von Daniel Wydler realisiert und ich widmete mich den Self-Tests. Es ging darum Testmethoden mit geeigneten Testdaten aufzubauen, um die Crypto-Algorithmen auf ihre korrekte Funktionsweise zu prüfen. Da die ganze strongSwan Libary in C geschrieben wurde, war die zu verwendende Technologie bereits festgelegt. Mit meinem Teil kam ich planmässig vorwärts. Der Prototyp beinhaltete die geforderten Funktionen. Für das Realease 1 musste ich noch die RSA Self-Test implementieren. Ich suchte nach Test-Cases für RSA und fand sie auch. Jedoch war nicht klar auf welchem Standard die strongSwan Library beruht. Schlussendlich brachten keine der gefundenen Test Cases den erwünschten Erflog und darum fiel der Beschluss, eigene Test Vektoren zu definieren. Einen Grossteil des RSA Test-Codes musste darum nochmals angepasst werden, wodurch sich leider einen Teil des bereits realisierten Codes erübrigte.

Meistens wusste ich nach den Sitzungen nicht genau, ob Herr Steffen mit unserer Arbeit zufrieden war oder nicht. Darum war es nicht immer eifach zu wissen, ob wir auf dem richtigen Weg waren.
Bei anfallenden Problemen mit der Verwendung mit der strongSwan Library versuchte uns Martin Willi, der Assistent von Herrn Steffen, zu helfen. Auf Fragen oder Problemstellungen bekamen wir von Herrn Steffen und Herrn Willi rasch eine nützliche Antwort.

Die Zusammenarbeit mit Daniel Wydler war gut. Da wir bei der ersten Studienarbeit zusammenarbeiteten, wussten wir sehr genau wo die einzelnen Stärken liegen. Diese wurden gezielt im Projekt eingesetzt. Schlussendlich blicke ich auf eine sehr lehrreiche Zeit zurück. Das Projekt brachte einige neue Technologien (Sprache C / Linux / Kryptographie) mit sich, mit denen ich vorher im produktiven Bereich noch nie gearbeitet hatte.

A.1.2 Daniel Wydler

Als die Ausschreibungen der zweiten Semesterarbeit veröffentlicht wurden, taten wir uns anfangs etwas schwer, eine passende Semesterarbeit zu finden. Bruno Krieg brachten diese Semesterarbeit als Vorschlag und nach kurzer Diskussion bewarben wir uns für diese Arbeit.
Ausschlaggebend waren für mich zwei Argumente, erweitern der Linux- und C-Kenntnisse.

Ich startete sehr motiviert und macht mich als erstes mit der Linux-Umgebung vertraut. Dazu gehören neben Linux selber auch die Autotools und der Gnu C Compiler, welche zu meinen alltäglichen Arbeitsutensilien gehören würden.
Auch die Analysephase verlief in meinen Augen sehr gut und auch da kamen wir flott voran.

Etwas gebremst wurde ich nach etwa einem Viertel der Projektdauer. Grund dafür waren die Sitzungen mit Herrn Steffen. Ich wusste nicht so recht was er von uns, besonders vom mir, bzw. meiner Arbeit hält. Ich konnte Ihn sehr schlecht einschätzen. Da ich diese Problem aber bei einer der nächsten Sitzungen ansprach, wurde dieses schnell geklärt und Herrn Steffen erklärte uns, dass wir auf einem guten Weg sind und er nur nicht wolle, dass wir plötzlich in Zeitnot geraten würden. Darum machte er etwas Druck.
Aus meinen Augen ist diese Reaktion zum Teil verständlich. Grund dafür waren die Problem, die ich mit dem Integritätstest hatte. Ich konnte mehrere Sitzungen hintereinander keine Lösungen, sondern nur immer neuen Fragen präsentieren. Da wir für dieses Problem aber genau aus diesem Grund genug Zeit eingeplant hatten, war ich gut im Zeitplan. Darum ist die Reaktion von Herrn Steffen mehr Druck zu machen, für mich verständlich Geworden.

Nach der Prototypenphase und den anfänglichen Problemen bereitete mir die Arbeit wieder mehr Freude. Ich bekam viele Probleme schnell in den Griff und dies gab mir die Möglichkeit, meine Arbeit frühzeitig zu Dokumentieren. Die Erfahrung der letzten Arbeit lehrte mich, dass man am Schluss der Arbeit fast keine Zeit mehr dazu hätte.

Während ich mich mit dem Integritätstest beschäftigte, realisierte Bruno Krieg die Self-Tests für die Crypto-Funktionen. Diese Aufgabe ist sehr programmierlastig und daher umfangreicher als mein Teil. Zusätzlich gab es auch Probleme mit Testvektoren für 3DES und RSA, die sehr viel Zeit kosteten. Aus diesen Gründen arbeitet ich zum Schluss der Arbeit zusammen mit Bruno an den Self-Tests.

Zusammenfassend ziehe ich eine positive Bilanz dieser zweiten Semesterarbeit. Die Vorgaben haben wir alle erfüllt und auch der Bericht gefällt mir.
Die Arbeit war sehr interessant (Linux, Programmiersprache C) und zum Teil musste ich ziemlich ins Detail (Autotools, Compiler, Objektmodul) gehen. Genau das habe ich erwartet und für zukünftige Projekte unter Linux mit der

Programmiersprache C, sollte ich nicht mehr so mühe haben mich darin zurecht zu finden.

A.2 Projektmanagement

A.2.1 Projektorganisation

Das Projektteam setzt sich aus Daniel Wydler und Bruno Krieg zusammen, welche für die Ausführung des Projektes verantwortlich sind.
Betreut wird das Projekt von Prof. Dr. Andreas Steffen.
Innerhalb des ausführenden Teams gibt es keine hierarchische Strukturierung. Die Arbeiten werden zwischen den einzelnen Teammitgliedern aufgeteilt.

A.2.2 Arbeitspensum

Wochentliches Arbeitspensum pro Teammitglied: 14h
Dauer des Projekts: 14 Wochen

Das ergibt einen Arbeitsaufwand von ungefähr 200 Stunden pro Teammitglied und einen Gesamtprojektaufwand von 400 Stunden.

A.2.3 Release-Definitionen

In diesem Abschnitt werden die Releases beschrieben. Dazu gehört der Inhalt und der Fertigstellungstermin.

A.2.3.1 Prototyp

Die Kernkomponente des Prototyps sind die Realisierung der Self-Tests und die Sicherstellung der Integrität des Executable-Codes. Auch die nötigen Build-Scripte sind zu erstellen.

Für folgende Crypto-Funktionen muss die korrekte Funktionsweise mit Self-Tests sichergestellt werden:

- Für folgende Crypto-Funktionen muss die korrekte Funktionsweise mit Self-Tests sichergestellt werden:
  - Hasher (MD-5, SHA-1, SHA-256, SHA-384, SHA-512)
  - HMAC (HMAC-MD-5, HMAC-SHA-1, HMAC-SHA-256, HMAC-SHA-384, HMAC-SHA-512)
  - Signers (Signer-MD-5, Signer-SHA-1, Signer-SHA-256, Signer-SHA-384, Signer-SHA-512)
  - PRFs (PRF-MD-5, PRF-SHA-1, PRF-SHA-256, PRF-SHA-384, PRF-SHA-512)
  - Crypters (AES, 3-DES)
- Der Status der einzelnen Tests wird abgespeichert und kann jederzeit über ein Status-Flag abgefragt werden.

- Im Standard-Modus werden nur die nötigsten Ausgaben auf die Konsole geschrieben.
- Im Debug-Modus werden alle Tests detailliert auf der Konsole ausgegeben.
- Die Integrität des ausführbaren Codes...
- Erstellung eines Hash-Fingerprints bei der Nutzung der Libary.
- Dieser Hash sollte auf der Konsole angezeigt werden.
- Die Integritätsüberprüfung sollte beim statischen, wie auch beim dynamischen Linken funktionieren.
- Der Hash-Fingerprint muss in die Zielapplikation gelinkt werden.
- Build-Scripte erstellen.

Fertigstellung: Woche 9, 3. Juni 2007

A.2.3.2 Release 1

Der Prototyp wird um die RSA Self-Tests erweitert und Fehler an bereits vorhandenen Code werden ausgebügelt. Zudem wird das Self-Test Handling um eine Liste mit kritischen Funktionen ergänzt.

- Kritische Crypto-Funktionen können in einer Liste definiert werden.
- RSA Self-Tests mit Testvektoren realisieren.
- Ganzer Integritäs-Überprüfungsteil kann mittels Compile-Flag weggelassen werden.
- Die Self-Test können bei der Kompilation ignoriert werden.
- Build-Scripte ergänzen.

Fertigstellung: Woche 13, 1. Juli 2007

A.2.4 Projektplan

A.2.4.1 Zeitplan
Der detaillierte Zeitplan befindet sich im File Projektplan.xls

A.2.4.2 Meilensteine
Im nachfolgenden sind die Meilensteine des Projekts aufgeführt.

Kürzel	**Beschreibung**	**Zeitpunkt der Fertigstellung**
A	Anforderungsanalyse fertig	Woche 5
P	Prototyp fertig	Woche 9
P	Bericht zu Prototyp fertig	Woche 10
R	StrongSwan Objektmodul (noch nicht vollständig)	Woche 12

Tabelle 16: Meilensteine

A.2.4.3 Iterationsplanung
In der nachfolgenden Tabelle ist der Zeitpunkt der Iterationen bestimmt.

RUP-Phase	**Zeitrahmen**	**Meilenstein**
Inception	02.04.07 - 15.04.07	
Elaboration	16.04.07 - 13.05.07	Anforderungsanalyse fertig
Construction	14.05.07 - 24.06.07	Prototyp fertig, Bericht zu Prototyp fertig, strongSwan Objektmodul
Transition	25.06.07 - 08.07.07	Abgabe

Tabelle 17: Iterationsplanung

A.2.4.4 Besprechung (Meetings)
Das Meeting mit dem Betreuer findet in regelmässigen Abständen (1 Woche) statt und dauert in der Regel ca. eine Stunde.

A.2.5 Arbeitspakete

In diesem Abschnitt werden die Arbeitspakete beschrieben und eine geschätzte Zeit für die Ausführung angegeben.
Die Reihenfolge der Arbeitspakete ist gleich wie in Projektplan.xls

Muster für die Arbeitspakete und der Bedeutung der einzelnen Felder.

Name des Pakets	Zuständigkeit	Zeit	Abhängigkeit
Beschreibung:	-		

A.2.5.1 Projekt Management

Projektplanung	KRB, WYL	20h	-
Beschreibung:	Das Erstellen des Projektzeitplans enthält die Tätigkeiten: Planung der Arbeitspakete, Setzen der Meilensteine, Festsetzen der Projektphasen und Soll-Arbeitszeiten festlegen. Neben dem Projektzeitplan werden die definierten Arbeitspakete beschrieben.		

Tabelle 18: Arbeitspaket Projektplanung

A.2.5.2 Anforderungsanalyse

Funktionale Anforderungen	KRB, WYL	8h	-
Beschreibung:	In diesem Schritt werden die funktionalen Anforderungen des Projektes beschrieben.		

Tabelle 19: Arbeitspaket Funktionale Anforderungen

Nicht-Funktionale Anforderungen	KRB, WYL	3h	-
Beschreibung:	In diesem Schritt werden die nicht-funktionalen Anforderungen des Projektes beschrieben.		

Tabelle 20: Arbeitspaket Nicht-Funktionale Anforderungen

Use Cases	KRB, WYL	3h	-
Beschreibung:	In diesem Schritt werden die Use Cases des Projektes beschrieben.		

Tabelle 21: Arbeitspaket Use Cases

A.2.5.3 FIPS-140-2 Zertifizierung

OpenSSL Dokumente	KRB, WYL	18.5h	-
Beschreibung:	Studium der OpenSSL Dokumente "User Guide" und "Security Policy".		

Tabelle 22: Arbeitspaket OpenSSL Dokumente

OpenSSL SourceCode	KRB, WYL	17h	-
Beschreibung:	Studium des Source-Codes, des als Vorlage dienenden OpenSSL FIPS-Modul.		

Tabelle 23: Arbeitspaket OpenSSL SourceCode

FIPS PUB 140-2	KRB	12h	-
Beschreibung:	Stuidum und Dokumentation des offiziellen Dokuments zur FIPS 140-2 Zertifizierung.		

Tabelle 24: Arbeitspaket FIPS PUB 140-2

A.2.5.4 Prototyp

Crypto-Funktionen Self-Tests	KRB	49h	-
Beschreibung:	Zu Begin der Benutzung einer FIPS-zertifizierten Crypto-Bibliothek müssen die Cryptofunktionen zuerst einen Self-Test durchlaufen. In diesem Arbeitspaket werden für die entsprechenden Funktionen zusammengestellt und im Prototyp implementiert.		

Tabelle 25: Arbeitspaket Prototyp Cryptofunktionen SelfTests

Scripte für Erstellung	KRB, WYL	6.5h	-
Beschreibung:	Schreiben der Buildscripte zur Erstellung des Prototypen.		

Tabelle 26: Arbeitspaket Prototyp Scripte für Erstellung

Erstellen von FIPS Object-Modul	WYL	20h	-
Beschreibung:	In diesem Arbeitspaket wird in Form eines Prototyps ein Object-Modul erzeugt, für welches zur Laufzeit ein Integritätstest vorgenommen werden kann.		

Tabelle 27: Arbeitspaket Prototyp Erstellen von FIPS Object-Modul

Integrität Executable Code	KRB, WYL	35h	-
Beschreibung:	Beim starten des FIPS-Moduls muss eine Integritätsprüfung stattfinden. Dies geschieht indem das geladene Object-Modul im Speicher mit dem über dem Object-Modul erstellten Hash verglichen wird.		

Tabelle 28: Arbeitspaket Prototyp Integrität Executable-Code

A.2.5.5 StrongSwan Objektmodul

Cryptofunktionen Self-Tests	KRB	24h	Prototyp
Beschreibung:	Die im Prototyp zusammengetragene und implementierten Self-Test-Funktionen werden hier in das StrongSwan-Projekt übernommen.		

Tabelle 29: Arbeitspaket Crypto-Funktionen Self-Tests

FIPS Crypto-Object-Modul in libstrongswan	WYL	5h	Prototyp
Beschreibung:	Das FIPS Object-Modul muss in die libstrongswan Library eingearbeitet werden.		

Tabelle 30: Arbeitspaket FIPS Crypto Object-Modul in libstrongswan

Integrität Executable Code	WYL	30h	Prototyp
Beschreibung:	Beim Aufstarten muss das FIPS-Modul im Arbeitsspeicher mit dem erstellen Objectmodul verglichen werden. Dieser Vergleich, bzw. diese Überprüfung wird hier realisiert.		

Tabelle 31: Arbeitspaket Integrität Executable-Code

Anpassung der Buildscripts	KRB, WYL	5h	-
Beschreibung:	Die Build-Scripts müssen für das strongSwan Projekt angepasst werden.		

Tabelle 32: Arbeitspaket Anpassung der Buildscripts

A.2.5.6 Dokumentation

Persönliche Berichte	KRB, WYL	2h	-
Beschreibung:	In diesem Arbeitspaket schreibt jedes Teammitglied am Ende des Projektes einen persönlichen Erfahrungsbericht.		

Tabelle 33: Arbeitspaket Persönliche Berichte

Abstract	KRB	2h	-
Beschreibung:	Schreiben des Abstracts.		

Tabelle 34: Arbeitspaket Abstract

Management Summary	KRB	6h	-
Beschreibung:	Schreiben des Management Summaries.		

Tabelle 35: Arbeitspaket Management Summary

Überarbeitung Bericht	KRB, WYL	20h	-
Beschreibung:	Überarbeiten des erstellten Berichts für die Schlussabgabe.		

Tabelle 36: Arbeitspaket Überarbeitung Bericht

A.2.5.7 Studium Technologien

Compiler, Autoconf und Automake	KRB, WYL	15h	-
Beschreibung:	Studium der GNU Autobuild-Tools und Compilern.		

Tabelle 37: Arbeitspaket Compiler, Autoconf und Automake

Crypto-Funktionen Self-Tests	KRB	10h	-
Beschreibung:	Finden und studieren von Selftest-Funktionen zu Crypto-Funktionen.		

Tabelle 38: Arbeitspaket Cryptofunktionen Selftests

Installationen (Linux)	KRB, WYL	15h	-
Beschreibung:	Installation der Linux-Entwicklungsumgebung. Studium zu verschiedenen Themen rund um Linux.		

Tabelle 39: Arbeitspaket Installationen (Linux)

A.2.5.8 Sitzungen

Sitzungen mit Betreuer	KRB, WYL	28h	-
Beschreibung:	Wöchentliche Sitzung mit dem Betreuer.		

Tabelle 40: Arbeitspaket Sitzungen mit Betreuer

Protokoll	KRB	7h	-
Beschreibung:	Protokolle zu den Sitzungen mit dem Betreuer.		

Tabelle 41: Arbeitspaket Protokoll

A.2.5.9 Qualitätssicherung

Code Walkthrought	KRB, WYL	8h	-
Beschreibung:	Gegenseitiges erklären und besprechen des realisierten Codes.		

Tabelle 42: Arbeitspaket Code Walkthrought

A.2.6 Auswertung

Die Auswertung umfasst eine Beschreibung der Realisierung der einzelnen Arbeitspakete und eine Zeitauswertung.

A.2.6.1 Arbeitspakete

In diesem Abschnitt wird die Realisierung der einzelnen Arbeitspakete beschrieben. Dabei wird das Vorgehen beschrieben, welche Probleme auftauchten und welche Punkte allenfalls noch offen sind.
Es werden dabei nur die wichtigsten Arbeitspakete aus A.2.5 beschrieben.

Analyse
Ziel dieses Arbeitspakets war das Analysieren der FIPS Zertifizierung und der Schritte die dazu notwendig waren.
In der Analyse wurden die funktionalen und nichtfunktionalen Anforderungen ausgearbeitet. Die funktionale Anforderung wurde mehrmals überarbeitet, bis alle Anforderungen wirklich klar waren.

Prototyp
Ziel dieses Arbeitspakets war das Design und die Implementierung der Self-Tests und des Integritätstests in Form eines Prototypen.
Die Anforderungen wurden komplett erfüllt und daraus resultierte ein lauffähiger Prototyp.

strongSwan Objektmodul
Ziel dieses Arbeitspakets war die Implementierung aller Self-Tests und das portieren der Self-Tests und Integritätstest in die libstrongswan.
Die Test wurden in die libstrongswan eingearbeitet. Integritätstest wie auch Selftests können mitcompiliert, oder auch weggelassen werden und während der Laufzeit aktiviert / deaktiviert werden.

A.2.6.2 Zeitplan

In diesem Abschnitt wird der Zeitplan bzw. die in das Projekt investierte Zeit dokummentiert.
Im ersten Unterabschnitt wird die totale Zeit aller Teammitglieder zusammen betrachtet.
Im zweiten Unterabschnitt werden die Teammitglieder miteinander verglichen.

A.2.6.2.1 Gemeinsam

Stunden pro Woche

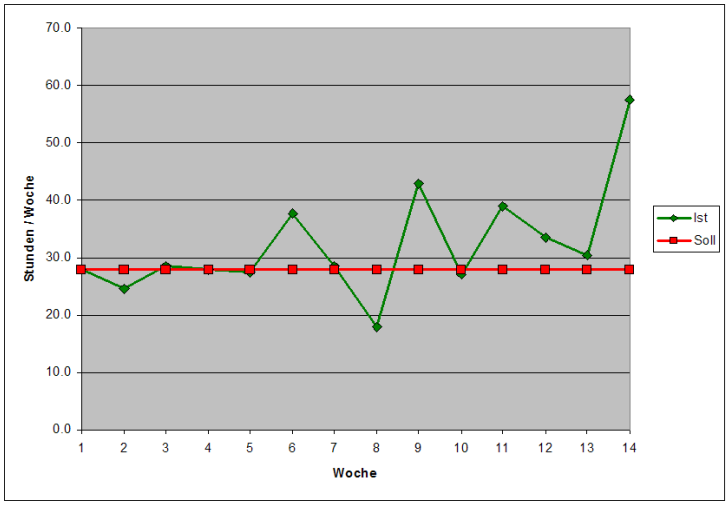

Abbildung 14: Gemeinsame Stunden pro Woche

Stunden aufsummiert

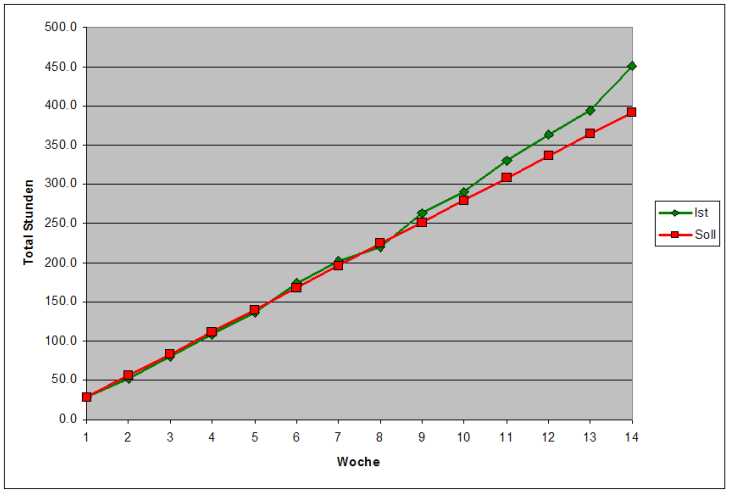

Abbildung 15: Gemeinsame Stunden pro Woche aufsummiert

A.2.6.2.2 Teammitgliedervergleich

Beim Vergleich der Teammitglieder wird ersichtlich, dass alle Teammitglieder in etwa gleich viel Zeit geleistet haben.

Stunden pro Woche

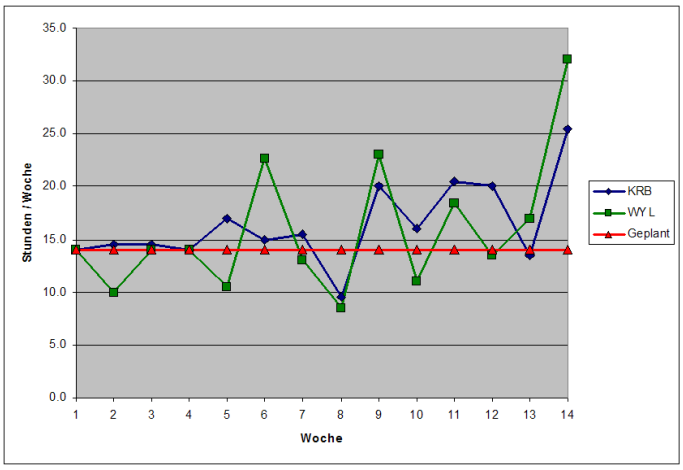

Abbildung 16: Stunden der Teammitglieder pro Woche

Stunden aufsummiert

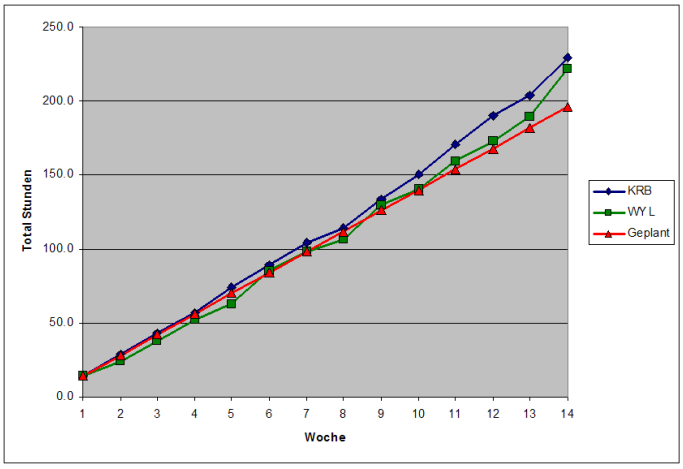

Abbildung 17: Stunden der Teammitglieder aufsummiert

A.3 Self-Test Zusatz

A.3.1 Dateibeschreibungen

A.3.1.1 common_functions.h
In dieser Datei werden ausschliesslich Funktionsdeklarationen für common_functions.c definiert.

A.3.1.2 critical_functions.h
In dieser Datei hat es eine Struktur, um kritische Crypto-Funktionen zu definieren. Diese Struktur wird im selftest_manager.c instantiert.

A.3.1.3 crypters_selftest.h
In dieser Datei werden die Testdaten- / Testvektor-Strukturen für die Crypter-Tests definiert.

A.3.1.4 crypto_group.h
In dieser Datei wird eine Enumeration für die Crypto-Gruppen definiert.

A.3.1.5 hashers_selftest.h
In dieser Datei werden die Testdaten- / Testvektor-Strukturen für die Hasher-Tests definiert.

A.3.1.6 hmac_selftest.h
In dieser Datei werden die Testdaten- / Testvektor-Strukturen für die HMAC-Tests definiert.

A.3.1.7 prfs_selftest.h
In dieser Datei werden die Testdaten- / Testvektor-Strukturen für die PRF-Tests definiert.

A.3.1.8 rsa_selftest.h
In dieser Datei werden die Testdaten- / Testvektor-Strukturen für die RSA-Tests definiert.

A.3.1.9 signers_selftest.h
In dieser Datei werden die Testdaten- / Testvektor-Strukturen für die Signer-Tests definiert.

A.3.1.10 selftest_manager.h
In dieser Datei werden ausschliesslich Funktionsdeklarationen für selftest_manager.c definiert.

A.3.1.11 common_functions.c

In dieser Datei werden allgemeine Funktionen zur Verfügung gestellt, die von allen Testgruppen genutzt werden können.

Folgende Funktionen stehen zur Verfügung.

print_message:
Gibt eine Nachricht auf der Konsole aus.

print_comparison:
Gibt der erwartete und der tatsächliche Vektorwert als Vergleich auf der Konsole aus.

print_single_test_result:
Eine Meldung wird auf der Konsole ausgegeben, ob der Einzeltest erfolgreich war oder nicht. Entschieden wird anhand des Parameters.

print_self_test_state:
Eine Meldung wird auf der Konsole ausgegeben, ob der Self-Test erfolgreich war oder nicht. Entschieden wird anhand des Parameters.

print_crypto_testdata:
Gibt allgemeine Informationen auf der Konsole aus, die von allen Testdaten definiert werden. Zu diesen Informationen gehören der Name des verwendeten Algorithmus und die Anzahl der Testvektoren.

print_hash_testdata_values:
Gibt Hash-spezifische Informationen zu den Testdaten auf der Konsole aus. Zuerst wird die Methode print_crypto_testdata aufgerufen und anschliessend wird die Hashgrösse in Byte ausgegeben.

print_crypter_vector_values:
Gibt Crypter-spezifische Informationen zu den Testdaten auf der Konsole aus. Es wird die Message, der Schlüssel, der Initialisierungsvektor und den Cipher mittels der Funktion print_vector_value ausgegeben.

print_rsa_vector_values:
Gibt RSA-spezifische Informationen zu den Testdaten auf der Konsole aus. Es wird die Message, den Cipher, p, q, d und e mittels der Funktion print_vector_value ausgegeben.

print_vector_value:
Gibt Informationen zu einem einzelnen Vektorwert wie die Länge und den eigentlichen Wert auf der Konsole aus. Das Ausgabe erfolgt als ASCII-String oder als HEX-String.

convert_hex_to_ascii:
Mit dieser Funktion kann ein mit Hex-Werten definiertes char-array in eine Hex-Representation gespeichert als ASCII-Werte konvertiert werden.

A.3.1.12 crypters_selftest.c

In dieser Datei wurden Testdaten für AES und 3DES definiert.
Die Datei umfasst folgende Funktionen.

crypters_selftest:
Die Funktion übergibt die AES- und die 3DES-Testdaten an die Funktion test_crypter_vectors.
test_crypter_vectors:
Die Funktion nimmt die Prüfung der Testdaten / Testvektoren vor. Beim Fehlschlagen der Tests werden die status-Flags angepasst.

get_crypter_status:
Gibt den Self-Test Status der Testdaten AES oder 3DES zurück.

A.3.1.13 hashers_selftest.c

In dieser Datei wurden Testdaten für MD-5, SHA-1, SHA-256, SHA-384 und SHA-512 definiert.

Die Datei umfasst folgende Funktionen.

hashers_selftest:
Die Funktion übergibt die MD5 und SHA-Testdaten an die Funktion test_hasher_vectors.
test_hasher_vectors:
Die Funktion nimmt die Prüfung der Testdaten / Testvektoren vor. Beim Fehlschlagen der Tests werden die status-Flags angepasst.

get_crypter_status:
Gibt den Self-Test Status der Testdaten MD-5, SHA-1, SHA-256, SHA-384 und SHA-512 zurück.

A.3.1.14 hmac_selftest.c

In dieser Datei wurden Testdaten für HMAC-MD-5, HMAC-SHA-1, HMAC-SHA-256, HMAC-SHA-384 und HMAC-SHA-512 definiert.
Die Datei umfasst folgende Funktionen.

hmac_selftest:
Die Funktion übergibt die HMAC-MD-5 und HMAC-SHA-Testdaten an die Funktion test_hmac_vectors.

test_hmac_vectors:

Die Funktion nimmt die Prüfung der Testdaten / Testvektoren vor. Beim Fehlschlagen der Tests werden die status-Flags angepasst.

get_hmac_status:
Gibt den Self-Test Status der Testdaten HMAC-MD-5, HMAC-SHA-1, HMAC-SHA-256, HMAC-SHA-384 und HMAC-SHA-512 zurück.

A.3.1.15 prfs_selftest.c :

In dieser Datei wurden Testdaten für PRF-MD-5, PRF-SHA-1, PRF-SHA-256, PRF-SHA-384 und PRF-SHA-512 definiert.

Die Datei umfasst folgende Funktionen.

prfs_selftest:
Die Funktion übergibt die PRF-MD-5 und PRF-SHA-Testdaten an die Funktion test_prfs_vectors.

test_prfs_vectors:
Die Funktion nimmt die Prüfung der Testdaten / Testvektoren vor. Beim Fehlschlagen der Tests werden die status-Flags angepasst.

get_prfs_status:
Gibt den Self-Test Status der Testdaten PRF-MD-5, PRF-SHA-1, PRF-SHA-256, PRF-SHA-384 und PRF-SHA-512 zurück.

A.3.1.16 rsa_selftest.c

In dieser Datei wurden Testdaten für RSA-MD-5, RSA-SHA-1, RSA-SHA-256, RSA-SHA-384 und RSA-SHA-512 definiert.

Die Datei umfasst folgende Funktionen.

rsa_selftest:
Die Funktion übergibt die RSA-MD-5 und RSA-SHA-Testdaten an die Funktion test_rsa_vectors.

test_rsa_vectors:
Die Funktion nimmt die Prüfung der Testdaten / Testvektoren vor. Beim Fehlschlagen der Tests werden die status-Flags angepasst.

get_rsa_status:
Gibt den Self-Test Status der Testdaten RSA-MD-5, RSA-SHA-1, RSA-SHA-256, RSA-SHA-384 und RSA-SHA-512 zurück.

A.3.1.17 signers_selftest.c

In dieser Datei wurden Testdaten für Signer-MD-5, Signer-SHA-1, Signer-SHA-

256, Signer-SHA-384 und Signer-SHA-512 definiert.

Die Datei umfasst folgende Funktionen.

singers_selftest:
Die Funktion übergibt die Signer-MD-5 und Signer-SHA-Testdaten an die Funktion test_signer_vectors.

test_signer_vectors:
Die Funktion nimmt die Prüfung der Testdaten / Testvektoren vor. Beim Fehlschlagen der Tests werden die status-Flags angepasst.

get_signer_status:
Gibt den Self-Test Status der Testdaten Signer-MD-5, Signer-SHA-1, Signer-SHA-256, Signer-SHA-384 und Signer-SHA-512 zurück.

A.3.1.18 selftest_manager.c

In dieser Datei werden die kritischen Crypto-Funktionen definiert. Zusätzlich werden vom Self-Test Manager alle Self-Tests aufgrufen. Der Satus der einzelnen Tests kann über den Self-Test Manager abgefragt werden.

Die Datei umfasst folgende Funktionen.

run_selftests:
Startet alle Self-Tests der Kryptogruppen nacheinander.

get_x_state:
Gibt den Status zurück.

A.4 Sitzungsprotokolle

A.4.1 Sitzungsprotokoll 1

Datum: 03.04.07, 08.30 - 09.30 Uhr

- Herr A. Steffen überreichte an Bruno Krieg und Daniel Wydler die Aufgabenstellung.
- Auf das nächste Treffen sollen wir uns in die Thematik einlesen, mittels den Dokumenten OpenSSL User Guide und Security Policy.
- Bekanntgabe unserer SSH-PublicKeys für den Zugriff auf den SVN-Branch an Martin Willi.

Nächste Sitzung: 10.04.07, 08.30 Uhr

A.4.2 Sitzungsprotokoll 2

Datum: 10.04.07, 08.30 - 09.30 Uhr

- SVN-Branch Zugang wurde von Martin Willi eingerichtet.
- Auf dem SVN-Branch sind die Crypto-Module von StrongSwan abgelegt.
- libstrongswan -> der Crypto-Teil ist in ein geprüftes Objektmodul auszulagern.
- Die Crypto-Funktionen in den Directories hashers, prfs, rsa, signers und zusätzlich DH und HMAC müssen im Objectmodul enhalten sein.
- Der FIPS-Mode soll nicht standardmässig aktiviert werden, sondern über ein Flag zum Ein- und Ausschalten gesteuert werden.
- Beim Fehlschlagen eines Self-Tests in der Crypto-Library, darf keine Crypto-Funktion angewendet werden. –> Ausgabe einer Fehlermeldung
- Über Compile-Options sollen die Power-Up-Überprüfungen aktiviert und deaktiviert werden können.
- Testfunktion soll generisch für alle Algorithmen realisiert werden.
- Für DH nach Testvektoren oder ähnliches im Internet forschen, um Alogrithmus sinnvoll testen zu können.

Weiteres Vorgehen...

- Studium der Tools autoconf und automake (beachte dazu configure.in / Makefile.am)
- Einarbeitung in die Problematik, um die Crypto-Funktionen zuverlässig testen zu können. Dazu werden Testvektoren verwendet.

Nächste Termine:
-Projektplan Woche 4 fertig

Nächste Sitzung: 17.04.07, 08.30 Uhr

A.4.3 Sitzungsprotokoll 3

Datum: 17.04.07, 08.30 - 09.30 Uhr

- Ein Object-Modul ist ein File mit der Endung .o (ar-Befehl).
- INCLUDES-Anweisungen im MAKEFILE.AM definieren die Header-Files.
- Im Notfall, wenn nicht anders möglich, sollte das Object-Modul mit den make-tools generiert werden. Als Namen sollte wenn möglich wegen Copyrightverletzung nicht fipscanister oder dergleichen verwendet werden.
- Statische Libraries werden immer gleich in den Speicher geladen.
- Dynamisch Libraries werden nur bei Bedarf in den Speicher geladen und es sind Anpassungen und Reorderings von Instruktionen möglich. Darum variert der Signaturwert. -> Problem dieses zu verifizieren.
- Jedoch sollten auch Dynamische Libraries von unserem Projekt unterstützt werden
- Um sich besser in die Funktionsweise von Autoconf / Automake einzuarbeiten, sollte ein kleines Testprojekt erstellt werden, das völlig entkoppelt vom restlichen OpenSSL-Code ist.

Auf die nächste Sitzung muss der Projektplan erstellt werden.

Nächste Sitzung: 24.04.07, 08.30 Uhr

A.4.4 Sitzungsprotokoll 4

Datum: 24.04.07, 08.30 - 09.30 Uhr

- Im Projektplan alle Arbeitspakte über die "Einbindung Open-SSL FIPS" vorläufig nicht einplanen (Stunden anrechnen).
- Plan kann angepasst werden, falls wir rasch vorwärts kommen würden.
- Folgende Meilensteine wurden definiert...
  - Woche 5: Anforderungsanalyse fertig
  - Woche 9: Prototyp fertig
  - Woche 10: Doku zum Prototyp fertig
  - Woche 12: strongSwan Objektmodul fertig
  - Woche 14: Bericht abgeschlossen
- Für Kommentare in C immer /* und nicht // verwenden.
- Die Instruktionen für fips_start und fip_send in fips_canister.c können 1:1 in das eigene Modul übernommen werden.
- Verweis im Code, dass Instruktionen von OpenSSL übernommen wurden.
- Das Integer-Array FIPS_rodata_start in fipscanister.c enthält FIPS_rodata_start ausgeschrieben mit ASCII-Codes.

Nächste Sitzung: 02.05.07, 10.00 Uhr

A.4.5 Sitzungsprotokoll 5

Datum: 02.05.07, 10.00 - 11.00 Uhr

- Beim Kompilieren soll mit einem Compile-Parameter definiert werden, ob das FIPS-Modul eingebunden werden soll.
- Self-Tests und Integritätstests werden mit dem FIPS-Modul ausgeliefert.
- Dementspreched macht es Sinn die Self-Tets auch für den standardmässigen strongSwan-Modus zu verwenden.
- Zur Runtime sollte die Möglichkeit geboten werden die Self-Tests auch im strong-Swan Modus auszuführen.
- Es muss zwischen kritischen und unkritischen Crypto-Funktionen unterschieden werden. Falls bei kritischen Crypto-Funktionen die Selbst-Tests fehlschlagen, sollte die Anwendung in den Error-Status wechseln, ansonsten sollten nur die fehlgeschlagenen Funktionen gesperrt werden.
  kritisch: SHA-1
- kritische Crypto-Funktionen sollten über eine Enumeration im Source-Code definiert werden.
- Der Integritätstest des Object-Moduls sollte ausschaltbar sein, wegen möglichen Problemen mit der Berechnung der Checksumme bei neueren 64-Bit Architekturen.
- Nicht-Funktionale Anforderungen...
- Leistung: Die Installation sollte sich im Minutenbereich bewegen und die Ausführung der Power-Up Tests sollte nicht mehr als 2s auf einem aktuellen System in Anpruch nehmen.
- Verwendete Standards: Realisierte FIPS 140-2 Abschnitte explizit erwähnen.
- Status des Self-Tests der einzelnen Funktionen bei der Erzeugung (in der Factory) überprüfen.

Nächste Sitzung: 08.05.07, 08.30 Uhr

A.4.6 Sitzungsprotokoll 6

Datum: 08.05.07, 08.30 - 09.30 Uhr

- Für FIPS-Modus ein anderer Name verwenden, da der Bezug zu dem OpenSSL FIPS-Modus zu gross ist.
- Beim Fehlschlagen kritischer Self-Test, sollte die Crypto-Funktion auch im normalen Modus verweigert werden.
- FIPS-Modus kann zur Kompilierzeit eingebunden werden oder nicht.
- Zur Runtime sollte die Prüfung der Integrität ein- und ausgeschaltet werden.
- Zahlen als Text ausschreiben: z. B. zwei Modi und nicht 2 Modi.
- Die Prüfung der Objektmodulintegrität muss der Linker durchführen. -> 2. Priorität.
- Wichtig ist die Integrität des Executable-Codes: Prio 1.
- Für strongSwan ist keine Lizenz nötig -> GPL 2
- UC03 (FIPS Cryptofunktionen anwenden) entfernen.
- Evtl. zusätzlicher Use-Case: FIPS-Modus zur Kompilierzeit einbinden.
- Generische Self-Tests, d. h. . .
- in Gruppen einteilen (Hasher, Crypter, Asymetrische Verfahren).
- Erweiterbarkeit
- Self-Tests separat auslagern.
- Der Object-Canister sollte 1:1 in die Library gelinkt werden (es wird kein Reordering durchgeführt) -> Linker-Optionen überprüfen
- Falls das nicht geht, können die Build-Scripts vom OpenSSL-Package als Grundlage verwendet werden.

Nächste Sitzung: 15.05.07, 08.30 Uhr

A.4.7 Sitzungsprotokoll 7

Datum: 15.05.07, 08.30 - 09.30 Uhr

- Anforderunsspezifikation: Orthografie und Rechtschreibung überprüfen.
- Objekt-Interität -> Speicherung des Hash in einem externen .sha-File
- Executable-Integrität: Checksumme wird im Code der Zielapplikation gespeichert.
- Checksumme sollte der Anwender ausdrucken und archivieren können.
- Im Debug-Modus sollte die Checksumme dem Anwender angezeigt werden.
- Checksumme wird mit dem SHA-1 Algorithmus generiert.
- Lizenzanforderungen...
- OpenSSL steht unter der BSD-Lizenz und muss darum ausdrücklich deklariert werden.
- strongSwan steht unter der GPL-2 Lizenz.
- Für Self-Test Strukturen genügt eine status_t Variable mit den Werten für Test bestanden (SUCCESS 0) oder Test nicht bestanden (FAILED 1).
- Variable rodata als Hex definieren, damit es keine Probleme mit big ung little-endian gibt.

Nächste Sitzung: 22.05.07, 08.30 Uhr

A.4.8 Sitzungsprotokoll 8

Datum: 22.05.07, 08.30 - 09.30 Uhr

- Checksummen-Signatur wird mittels des premain-Codes über der Canister-Library generiert, falls noch keine vorhanden ist.
- Anschliessend wird diese Signatur mittels Compile-Flag in die Zielapplikation und nicht in die Library integriert.
- Frage ob die libstrongswan statisch oder dynamisch gelkinkt werden muss? Das kann der Anwender bestimmen. Beide Varianten sollten möglich sein.
- Die Library sollt in den Stack im Bereich 0x80... geladen werden
- Compiler setzt realtive Adressen in absolute um.
- OpenSSL kann nicht dynamisch gelinkt werden. Diese Funktionalität ist noch nicht implementiert.
- Möglichkeit zu verhindern, dass die Library bei jedem Start in einen anderen Speicherbereich geladen wird: stack randomization ausschalten
- Es wäre ideal, wenn dieses Verhalten pro Prozess deaktiviert werden kann.
- Dynamisches Linken Ansatz austesten und wenn nicht möglich statisch Linken.
- ldd Befehl: ldd + ausführbare Testapplikation -> zeigt ob Library dynamisch oder statisch gelinkt wurde.
- Welche HMAC-Funktionen verwenden, die im Ordner crypto oder die im Ordner crypto/singner?
- Die hmac-Funktionen im Ordner crypto stellt die Grundfunktionalität für die HMAC-Generierung bereit. Die Funktionen im Ordner signer werden benutzt um Dokumente zu signieren. Dabei kann u. a. HMAC verwendet werden. Der Signer benutzt dann die HMAC Funktion im Ordner crypto. Darum müssen die Signer separat getestet werden.
- Anstatt debug-Variable können die bereits in der libstrongswan definierten Debug-Levels verwendet werden.

Nächste Sitzung: 29.05.07, 08.30 Uhr

A.4.9 Sitzungsprotokoll 9

Datum: 29.05.07

Keine Sitzung da A. Steffen und M. Will am Linuxtag waren.

Nächste Sitzung: 05.06.07, 08.30 Uhr

A.4.10 Sitzungsprotokoll 10

Datum: 05.06.07, 08.30 - 09.30 Uhr

- Der Entwurf zur Self-Test Dokumentation wurde durchgelesen und vom Inhalt her als gut befunden.
- Daniel Wydler erklärt anhand einer Grafik wie Hash-Signatur die Hash-Signatur generiert wird.
- Probleme mit dem libtool konnten behoben werden. Es ist egal, ob dynamisch oder statisch gelinkt wird.
- Falls die Signatur in einer Datei abgelegt wird, besteht der Vorteil, dass eine Abhängigkeit (dependency) auf dieses File erstellt werden könnte. Dadurch kann man sich die Compile-Optionen ersparen.
- Zudem muss mit einer Dependency sichergestellt werden, dass bei einer Änderung in der Library, auch das Signatur-File neu erstellt wird.
- Die Einbindung der Self-Tests kann über ifdef's gesteuert werden, jedoch müsste man, um sie einbinden zu können nochmals neu kompilieren.
- Besser Self-Tests standarmässig einbinden und ausführen, da sie sehr wenig Zeit in Anspruch nehmen. Falls die Ergebnisse nicht interessieren, können sie unbeachtet bleiben.
- Die Integritätsüberprüfungen des Executable-Codes müssen jedoch zur Laufzeit deaktiviert werden können, damit die Library bei einer nicht unterstützten Architektur trotzdem ohne Einschränkungen verwendet werden kann.
- Die Ausführung der Self-Tests und Integritätsüberprüfung kann über Config-Flags gesteuert werden.

Nächste Sitzung: 12.06.07, 08.30 Uhr

A.4.11 Sitzungsprotokoll 11

Datum: 12.06.07, 08.30 - 09.30 Uhr

- Der mittels 3-DES verschlüsselte Testinputvektor stimmt nicht mit dem Testoutputvektor überein.
- Mögliche Lösung: Key wurde nur einmal und nicht 3 mal in Serie verwendet.
- Die Testvektoren von www.rsa.com für RSA decken nur Bad-Cases ab.
- Herr Steffen sucht RSA Testvektoren für Good-Cases.
- RSA-Key Paar Erzeugung...
- RSA-Key Paare können mit Strongswan im DER-Format generiert werden.
- Jedoch werden diese mit völlig zufälligen Werten generiert, was für die Self-Tests nichts bringt.
- RSA-Key Paare können jedoch auch mit dem Scept-Client mit Input Werten erstellt werden und als chunk abgespeichert werden..
- Dies hat jedoch immer noch den Nachteil, dass im Nachhinein kein Rückschluss mehr auf die Input-Parameter (p, q, n, d, e) gemacht werden kann.
- Die beste Möglichkeit ist es die rohen Input-Parameter als Vektoren abzulegen und die Schlüsselpaare zur Laufzeit zu berechnen.
- Diese Variante kann nur mit Anpassungen am bestehenden RSA Strongswan-Code realisiert werden.

Nächste Sitzung: 19.06.07, 08.30 Uhr

A.4.12 Sitzungsprotokoll 12

Datum: 19.06.07, 08.30 - 09.30 Uhr

- 3DES: Auftretender Segmentation fault in set_key wurde behoben. Libstrongswan set_key() Funktion veränderte übergebenen Parameter. Diese wurde mit const definiert -> Segmentation Fault. Nun muss dieser Self-Test noch fertiggestellt werden.
- RSA: scheinbar gibt es verschiedene RAS-Implementationen. Nun sollen die Testvektoren von der PKCS1 Ver1.5 getestet werden.
- Bericht: Fassung etwas überarbeiten und A. Steffen schicken, damit diese bei der nächster Sitzung besprochen werden kann.

Nächste Sitzung: 26.06.07, 08.30 Uhr

A.4.13 Sitzungsprotokoll 13

Datum: 26.06.07, 08.30 - 09.30 Uhr

- 3DES: Die Testvektoren des Monte Carlo Tests von der NIST funktionieren nicht. Mit der libstrongswan Berechnung bekommt man nicht das gleiche Ergebnis.
- Herr Steffen wird diesen Monte Carlo Test genauer ansehen -> Ergebnis: Bei diesem Test werden mehrere hundert Iterationen durchgeführt.
- Bei genügend Zeit könnten wird auch mehrere Iterationen beim Self-Test implementieren.
- Alternativ können auch eigene Testvektoren definiert werden.
- RSA: Die Testvektoren im RSAVS (RSA Validation System) von NIST konnten nicht nachfolgzogen werden. Auch die Testvektoren für PKCS1 Ver1.5 von RSA Laboratories brachten keinen Erfolg.
- Es können eigene Testvektorn für RSA definiert werden.
- Bericht zum Projekt: Dieser wurde von Herrn Steffen als gut befunden.

Nächste Sitzung: 03.07.07, 08.30 Uhr

A.4.14 Sitzungsprotokoll 14

Datum: 03.07.07, 08.30 - 09.30 Uhr

- 3DES: Selbstdefinierte Testvektoren wurden erfolgreich realisiert.
- RSA: Bei der Verifikation einer RSA-Signatur mit dem dazugehörigen Public-Key gab es jeweils keine Übereinstimmung.
- Anderer Ansatz: Anstatt die RSA-Werte (p, q, d, e) einzeln abzuspeichern, könnten die Key-Paare direkt mit OpenSSL generiert und als Chunk abgespeichert werden.
- Dies hat nochmals eine Anpassung der Testdatenstruktur zur Folge.

A.5 GNU Autotools

[Bergo01] Zu den GNU Autotools gehören aclocal, autoheader, automake und autoconf. Diese Tools sind der quasi-Standard für den Build-Prozess von Linux Projekten.
Praktisch alle Linux Projekte sind mittels:

1. ./configure
2. make
3. make install

zu installieren.

Damit ein Programm auf möglichst vielen Systemen installiert werden kann, müssen für viele Systemkonfigurationen separate Makefiles geschrieben werden. Da dies nur in einem kleinen Rahmen möglich ist, wird vor der Installation eines neuen Programms zuerst ./configure aufgerufen.

Die Aufgabe von ./configure ist das Sammeln von Systeminformationen und das Generieren eines auf das System zugeschnittenes Makefiles. In diesem Makefile werden auch gleich die Abhängigkeiten richtig eingetragen, was in grossen Projekten ein grossen Vorteil ist.

Der zentrale Punkt des Build-Prozesses ist also ./configure und genau diese Datei wird mit den GNU Autotools erzeugt.

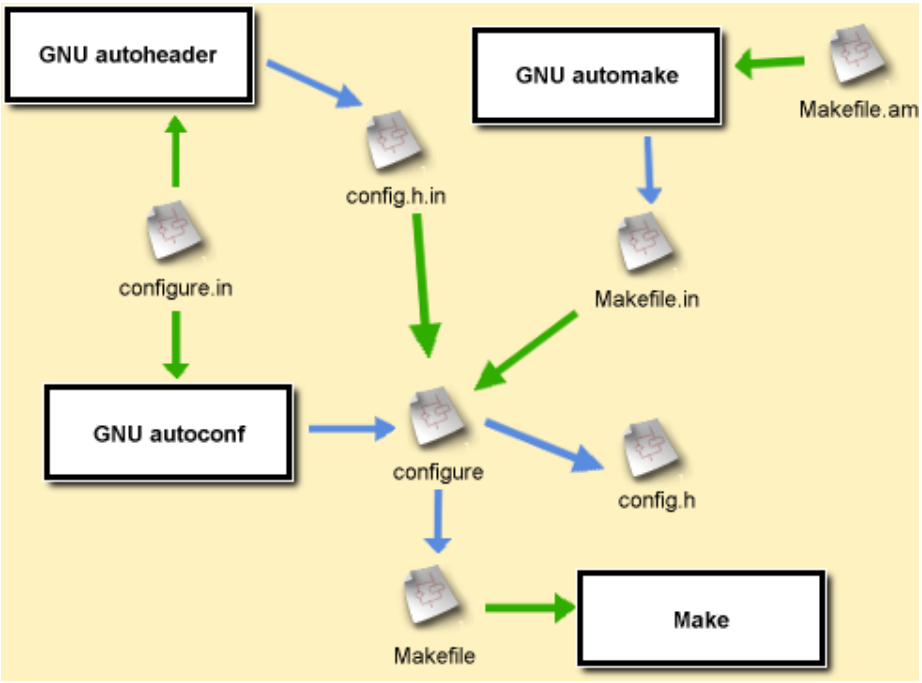

Abbildung 18: Autotools Input/Output Graph

aclocal
Erzeugt auf dem Inhalt von configure.in-Dateien basierend, entsprechende aclocal.m4 Dateien

autoheader
Dieses Werkzeug erzeugt C #define Anweisungen, die configure benutzen soll.

automake
Ein Werkzeug zur automatischen Erzeugung der Makefile.in Dateien. Durch scannen der configure.in Dateien findet das Programm automatisch alle benötigten Makefile.am Dateien und erzeugt daraus die entsprechende Makefile.in Datei.

autoconf
Generiert shell-Skripte, welche automatisch Quellcode-Pakete einrichten, um sich an unterschiedliche Unix-Systeme anzupassen.