INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Ein sicherer Startvorgang für OpenBSD

Markus Müller

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Ein sicherer Startvorgang für OpenBSD

Markus Müller

Aufgabensteller: Prof. Dr. Dieter Kranzlmüller

Betreuer: Dr. Vitalian Danciu
Dr. Nils gentschen Felde

Abgabetermin: 08. Juli 2014

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 08. Juli 2014

. . . . . .

(Unterschrift des Kandidaten)

Kurzfassung

Mit der Einführung des Unified Extensible Firmware Interface (UEFI) ist es für Angreifer deutlich leichter geworden, Manipulationen am Startvorgang eines Systems vorzunehmen. Klar definierte Schnittstellen für verschiedene Firmwares und Hardwareplattformen vereinfachen nicht nur die Arbeit der Hersteller von Treibern und Betriebssystemen, sondern auch die der Angreifer.

Schadsoftware wie Bootkits oder Firmware Rootkits, die den Bootloader verändern oder die Firmware manipulieren, müssen nicht mehr umständlich in Assembler programmiert, sondern können nun in Sprachen wie C erstellt werden und sich auf das Vorhandensein klar definierter Schnittstellen verlassen.

Mit der Standardisierung von UEFI Secure Boot sowie dem Erscheinen kompatibler Hardware kommt ein sicherer Startvorgang jedoch in greifbare Nähe. Ausgehend von einem sicheren, durch Hardware realisierten Perimeter werden hierbei alle geladenen Komponenten verifiziert und so deren Integrität und Authentizität sichergestellt.

Im Rahmen dieser Arbeit wird untersucht, welche Bedrohungen für den Startvorgang Unix-ähnlicher Systeme unter UEFI existieren und wie sich diese mit UEFI Secure Boot lösen lassen. Ausgehend von den Annahmen, dass die Hardware nicht kompromittiert ist und sich die UEFI-Firmware wie spezifiziert verhält, wird eine Architektur entwickelt, die einen sicheren Startvorgang für Unix-ähnliche Betriebssysteme ermöglicht. Eine prototypische Umsetzung für OpenBSD zeigt deren Machbarkeit und bestätigt einen praktischen Nutzen.

Synopsis

The standardization of the Unified Extensible Firmware Interface (UEFI) introduced many advantages to developers, but also created some pitfalls. Hardware manufacturers can easily create platform-independent drivers for the pre-boot environment and operating systems can unify their boot process. However, the same applies to attackers. As malware such as bootkits or firmware rootkits can be written in C instead of assembler and build on well-defined interfaces, attackers are increasingly able to manipulate the boot process.

With the standardization of UEFI Secure Boot and availability of compatible hardware a secure boot process is now made possible. Starting from a root of trust every component of the boot process can be verified so that integrity and authenticity are guaranteed.

This thesis focuses on possible threats to boot processes with UEFI for Unix-like operating systems and examines possible solutions based on UEFI Secure Boot. Based on the assumption that the hardware and software are valid and not compromised, the author introduces an architecture for a secure boot process for Unix-like operating systems. A software prototype for OpenBSD shows the feasibility and demonstrates a practical use.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Bisher benötigten Betriebssysteme für ihren Systemstart ein Basic Input/Output System (BIOS), eine Firmware, die grundlegende Schnittstellen zum Zugriff auf die Hardware zur Verfügung stellt. Die Struktur verschiedener BIOS-Firmwares ist historisch gewachsen, wenig einheitlich und bietet keine ausreichenden Möglichkeiten, die Sicherheit eines Systems beim Starten sicherzustellen.

Da das BIOS weder standardisiert noch beliebig erweiterbar war, wurden zunächst unter dem Namen Extensible Firmware Interface (EFI), später dann als Unified Extensible Firmware Interface (UEFI) Schnittstellen festgelegt, über die ein einheitlicher Zugriff auf Funktionen der Firmware ermöglich wurde.

Mittels dieser Schnittstellen ist es nun möglich, Komponenten wie Treiber für verschiedene Firmwares und Hardwareplattformen wiederzuverwenden. Dies stellt eine enorme Vereinfachung für die Hersteller von Treibern und Betriebssystemen dar, jedoch auch für Angreifer.

Schadsoftware wie Bootkits, die den Bootloader verändern oder Firmware Rootkits, die die Firmware oder Teile dieser manipulieren, müssen nicht mehr umständlich in Assembler programmiert werden, sondern können nun in komplexeren Sprachen wie C erstellt werden und sich auf das Vorhandensein klar definierter Schnittstellen verlassen.

1.1. Problemstellung und Motivation

Mit der Einführung des Unified Extensible Firmware Interface (UEFI) ist es für Angreifer deutlich leichter geworden, Manipulationen an der Firmware eines Systems vorzunehmen. Um dieser Entwicklung entgegenzuwirken, wurden in der Vergangenheit verschiedene Ansätze entwickelt.

Mit der Spezifikation *UEFI Secure Boot* sowie dem Erscheinen kompatibler Hardware kommt ein sicherer Startvorgang in greifbare Nähe. Secure Boot stellt einen Ansatz dar, der ausgehend von einem vertrauenswürdigen Startpunkt alle geladenen Komponenten verifiziert und so die Sicherheit dieser garantiert. Die praktische Umsetzbarkeit dieses Ansatzes hat Microsoft mit Windows 8 bereits gezeigt [Micr 12], ebenso existieren Ansätze für Linux-basierte Betriebssysteme, die beispielsweise durch die Distributionen Fedora [BFJ+ 13] und SUSE Linux Enterprise Server [SUSE 14] umgesetzt wurden.

Bislang existiert jedoch keine umfassende Betrachtung der Sicherheit eines modernen, UEFI-basierten Startvorgangs für Unix-ähnliche Betriebssysteme. Ob bestehende Ansätze wie Secure Boot zur Absicherung moderner Startvorgänge auch mit Unix-ähnlichen Betriebssystemen möglich sind, wurde noch nicht geklärt. Insbesondere ist dabei unklar, ob die Integrität und Authentizität aller für den Startvorgang benötigten Komponenten dauerhaft sichergestellt werden kann.

1.2. Zielsetzung und Ergebnisse

In dieser Arbeit wird ein Konzept entwickelt und prototypisch umgesetzt, das die Frage, ob ein sicherer Startvorgang für klassische, Unix-ähnliche Betriebssysteme auf handelsüblicher PC-Hardware ohne Anpassungen an dieser möglich ist, positiv beantwortet.

Ausgehend von den Annahmen, dass die Hardware nicht kompromittiert ist und sich die UEFI-Firmware wie spezifiziert verhält, wird eine Architektur entwickelt, die einen sicheren Startvorgang basierend auf Secure Boot für Unix-ähnliche Betriebssysteme ermöglicht. Eine prototypische Umsetzung für OpenBSD zeigt sodann deren Machbarkeit und bestätigt einen praktischen Nutzen.

1.3. Abgrenzung des Themas

Die vorliegende Arbeit betrachtet sichere Startvorgänge mit Secure Boot und setzt damit ein zu UEFI kompatibles System voraus. Eine ausführliche Beschreibung von UEFI und UEFI Secure Boot folgt in Abschnitt 2.2.

Im Rahmen dieser Arbeit wird auf sogenanntes *Measured Boot* der Trusted Computing Group (TCG) in Abschnitt 2.3 eingegangen. Da dieser Ansatz jedoch spezielle Hardware voraussetzt, wird er nicht weiter verfolgt und nicht für die Lösung verwendet.

Es wird zudem davon ausgegangen, dass der Startvorgang lokal von einem Datenträger und damit nicht über ein Netz stattfindet. Kompatibilitätslösungen wie das UEFI Compatibility Support Module (CSM), die ein klassisches BIOS nachbilden, werden ebenso nicht verwendet.

1.4. Überblick über den Aufbau der Arbeit

Das Vorgehen dieser Arbeit ergibt sich aus der Fragestellung und gliedert sich wie in Abbildung 1.1 dargestellt.

In Kapitel 2 werden zunächst die für diese Arbeit benötigten Grundlagen erklärt und Begriffe festgelegt. Eine Bedrohungsanalyse zum Finden von Anforderungen an einen sicheren Startvorgang folgt in Kapitel 3. Darauf aufbauend werden in Kapitel 4 bestehende Systeme betrachtet und auf ihre Defizite hin untersucht. Kapitel 5 folgt mit der Vorstellung einer Architektur eines sicheren Startvorgangs sowie dessen prototypischen Umsetzung für OpenBSD, die mit den Anforderungen aus Kapitel 3 abgeglichen wird. Eine Zusammenfassung der Ergebnisse dieser Arbeit sowie einen Überblick über verbleibende Fragestellungen folgt in Kapitel 6.

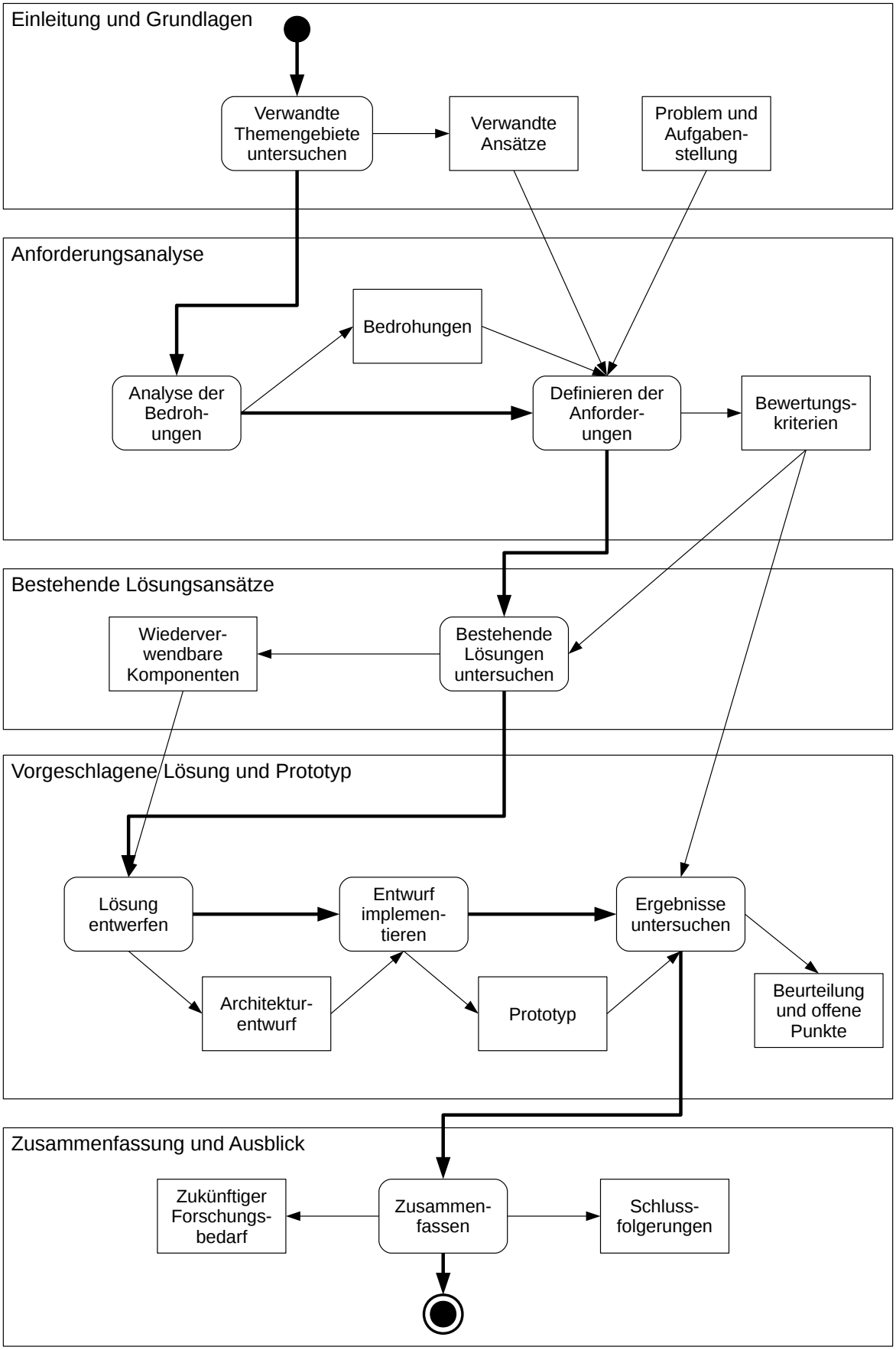

Abbildung 1.1.: Überblick über die Vorgehensweise und Bestandteile diese Arbeit.

2. Grundlagen

Ein sicherer Startvorgang im Rahmen dieser Arbeit beinhaltet den Zeitraum vom Einschalten eines Computers bis zur Ausführung des Kernels eines Betriebssystems. In diesem Kapitel werden die für das Verständnis dieses Ablaufs notwendigen Grundlagen klassischer und moderner Startvorgänge erläutert.

In Abschnitt 2.1 wird zunächst der Startvorgang mit dem Basic Input/Output System (BIOS) erläutert, der als Vorläufer moderner Startvorgänge diese maßgeblich beeinflusst hat. Abschnitt 2.2 geht dann zunächst allgemein auf die Komponenten des Unified Extensible Firmware Interface (UEFI) ein, um darauf basierend einen modernen, sicheren Startvorgang mit UEFI Secure Boot zu erläutern. In Abschnitt 2.3 wird zuletzt ein weiterer Ansatz zur Absicherung von Startvorgängen vorgestellt, der der inhaltlichen Abgrenzung dieser Arbeit dient.

Die in diesem Kapitel vorgestellten Grundlagen werden sodann in Kapitel 3 zur Bestimmung von Anforderungen an sichere Startvorgänge genutzt, die in Abschnitt 3.4 zu einer Liste von Bewertungskriterien zusammengefasst werden.

2.1. Klassischer Startvorgang mit dem Basic Input/Output System (BIOS)

Für eine Diskussion sicherer Startvorgänge ist eine Kenntnis herkömmlicher Startvorgänge unabdingbar. Dieser Abschnitt geht daher zunächst auf den Ablauf beim Starten eines herkömmlichen Systems ein, der durch die in den folgenden Abschnitten vorgestellten Ansätze modernisiert und abgesichert wurde.

Die folgende Betrachtung des Startvorgangs mit dem Basic Input/Output System (BIOS) erfolgt anhand von Abbildung 2.1. Sie gibt einen Überblick über den Startvorgang und beschreibt kurz die Vorgänge der einzelnen Phasen.

Das BIOS ist eine Firmware-Implementierung für die Intel-Architektur, die aus dem ursprünglichen IBM PC Design entstanden ist. Es ist eine konkrete Implementierung einer Firmware, die die Plattform initialisiert und ein Betriebssystem lädt. Historisch bedingt ist das BIOS typischerweise in 16-Bit x86-Assembler geschrieben und hat sich im Laufe der Jahre nur unwesentlich verändert.

In der ersten Phase nach dem Anschalten eines Systems bezieht der Prozessor seine Anweisungen aus dem Boot Block, einem im System verbauten Speicher, der das BIOS enthält. Diese führen einen Systemcheck (Power-On Self-Test, POST) durch und initialisieren die Hardware mittels Option-ROMs. Ein Option-ROM ist eine Firmware, die vom BIOS geladen und ausgeführt wird. Sie dient beispielsweise der Initialisierung und Ansteuerung einer Grafik- oder Netzwerkkarte.

Im Anschluss hieran wird ein Bootloader aus dem Master-Boot-Record (MBR) eines Speichermediums gestartet, das für gewöhnlich ein Hintergrundspeicher wie eine Festplatte ist. Dieser nun gestartete Bootloader ist in seiner Größe und damit seiner Funktionalität beschränkt. Er kann jedoch mit Dateisystemen (UFS bei OpenBSD) umgehen und damit einen weiteren Bootloader bzw. Bootmanager von einem Dateisystem starten.

Der vom Dateisystem geladene und ausgeführte Bootloader ist nun in der Lage, eine Konfigurationsdatei auszulesen, die Informationen zum Systemstart beinhaltet. Er bietet darüber hinaus die Möglichkeit, sofern nötig, mit dem Nutzer zu interagieren. Über die Konfigurationsdatei und die Nutzerinteraktion können beispielsweise der zu startenden Kernel und ggf. zusätzliche Parameter für diesen, z.B. die zu verwendende Root-Partition oder Ausgabekonsole, festgelegt werden.

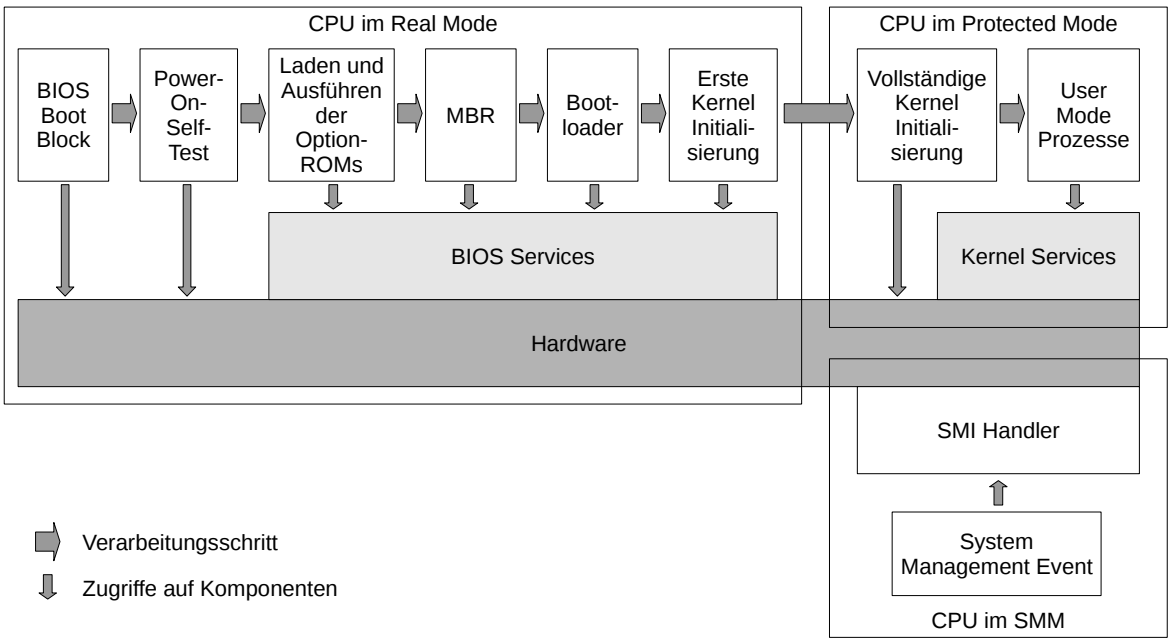

Abbildung 2.1.: Überblick über einen klassischen Startvorgang mit dem Basic Input/Output System (BIOS) nach [CPRS 11].

Sobald die Auswahl über den zu startenden Kernel getrofen wurde (Standardwert, durch einen Eintrag in der Konfigurationsdatei oder durch die Eingabe des Benutzers), wird dieser gestartet. Dem Kernel werden hierbei Informationen über den Speicher (memory map), dem Speichermedium, zum Advanced Configuration and Power Interface (ACPI) sowie optional weitere Informationen übergeben.

Der Kernel übernimmt sodann die Kontrolle über die Hardware und fährt mit seiner Ausführung fort. [Open 14c] [Wiet 12]

2.2. Moderner Startvorgang mit PI und UEFI

Dieses Kapitel beginnt zunächst mit einem Überblick über die Komponenten und den Aufbau des Unified Extensible Firmware Interface (UEFI) in Abschnitt 2.2.1, der für ein Verständnis eines UEFI-basierten Startvorgangs notwendig ist. Nach einer Betrachtung eines Startvorgangs mit UEFI in Abschnitt 2.2.2 folgt in Abschnitt 2.2.3 UEFI Secure Boot, in dem Sicherheitsmechanismen, die der Absicherung des Startvorgangs dienen, erläutert werden.

2.2.1. Aufbau des UEFI

Während das BIOS eine Firmware-Implementierung für handelsübliche PC-Hardware darstellt, ist das Unified Extensible Firmware Interface (UEFI) ein Ansatz verschiedener PC-Hersteller, den Bootvorgang von Systemen zu modernisieren. Die UEFI-Spezifikationen beschreiben Schnittstellen zwischen dem Betriebssystem und der Firmware, die von UEFI-Implementierungen in der Firmware umgesetzt werden. UEFI wird durch die Platform Initialization (PI) Spezifikationen dahingehend ergänzt, dass diese die Komponenten und Abläufe, die für die Bereitstellung der UEFI-Umgebung benötigt werden, definieren. UEFI-Implementierungen führen zum Starten eines Systems ähnliche Schritte wie das BIOS durch, nämlich das Initialisieren der Plattform und das Laden eines Betriebssystems.

GUID Partition Table

Die GUID Partition Table (GPT) stellt einen Nachfolger für auf dem Master Boot Record (MBR) basie-

rende Partitionstabellen dar. Sie ist im UEFI-Standard spezifiziert und bietet gegenüber MBR-basierten Partitionstabellen verschiedene Vorteile, wie beispielsweise die Adressierbarkeit größerer Speichermengen und die Sicherstellung der Datenintegrität durch Prüfsummen.

Boot und Runtime Services

Die UEFI-Schnittstellen bestehen aus Datenstrukturen mit Informationen zum System sowie Systemfunktionen (service calls), die sowohl während des Startens (boot services) als auch während der Laufzeit des Systems (runtime services) zur Verfügung stehen.

Durch den Zugriff auf UEFI Boot und Runtime Services kann die Firmware eines Systems mit UEFI-Treibern und -Anwendungen erweitert werden. Boot Services stehen ausgeführten Anwendungen solange zur Verfügung, bis diese die Dienste beenden und selbst die Kontrolle über das System übernehmen. Runtime Services stellen auch nach dem Beenden der Boot Services dem laufenden System Funktionen und Daten zur Verfügung. [BaCr 12]

UEFI-Images

UEFI-Images können Anwendungen (UEFI Applications), Treiber für den Startvorgang (UEFI Boot Service Drivers) sowie Treiber für die Laufzeit (UEFI Runtime Drivers) sein. [Unif 13b] Sie enthalten ausführbaren Programmcode und können von der UEFI-Plattform geladen werden. Hierfür müssen sie in einem angepassten PE/COFF-Format vorliegen, bei dem der Header modifiziert wurde und 64-Bit-Relocation unterstützt wird. [Micr 13a]

UEFI-Shell

Die UEFI-Plattform enthält eine Shell, in der Befehle und UEFI-Anwendungen ausgeführt werden können. Diese wird für gewöhnlich dann gestartet, wenn in der Firmware nicht konfiguriert wurde, welche UEFI-Anwendung beim Starten geladen werden soll. [Unif 13b]

UEFI Boot Manager

Zudem enthält die UEFI-Plattform einen Boot Manager, der das Laden von UEFI-Images ermöglicht. Um eine Anwendung zu starten, alloziert die UEFI-kompatible Firmware ausreichend Speicher, in den die Anwendung kopiert wird. Es werden Anpassungen für die Relocation durchgeführt, die Speicherstelle wird entsprechend markiert (code oder data) und die Kontrolle wird an die Anwendung übergeben. Wenn sich eine Anwendung beendet hat, wird der von ihr verwendete Speicher freigegeben und die UEFI-Shell oder die UEFI-Anwendung, die die Anwendung geladen hat, erhält die Kontrolle zurück. [Unif 13b]

2.2.2. Startvorgang mit UEFI

Dieser Abschnitt betrachtet einen Startvorgang entsprechend der UEFI Platform Initialization Specification. [Unif 13a] Einen Überblick über den Ablauf liefert Abbildung 2.2.

Zunächst wird die Hardware nach dem Anschalten vorbereitet, sodass diese eine UEFI-Umgebung bereitstellen kann. Sobald diese zur Verfügung steht, kann das UEFI in Zusammenarbeit mit der Firmware das Betriebssystem oder einen Bootloader (d.h. eine UEFI-Anwendung) starten.

Initialisierung der Firmware: Security (SEC)

Nach dem Einschalten (Power On) führt das System den Reset Vector aus, der die Komponenten der Early Platform Initialization Phase überprüft, in den Speicher lädt und anschließend ausführt.

Nach dem Einschalten befindet sich das System in der Security (SEC) Phase. Diese lädt das Boot Firmware Volume (BFV) aus einem nicht-beschreibbaren Speicherbereich (ROM) in einen temporären Speicherbereich der CPU und führt es aus. Die SEC-Phase wird nur nach einem initialen Power-On ausgeführt; nach einem Neustart wird sofort in die PEI-Phase übergegangen.

Das BFV enthält die Komponenten, die für die Pre-EFI Initialization Phase (PEI) benötigt werden. Die SEC-Phase stellt zudem eine Schnittstelle bereit, mit der die PEI-Phase ihre Komponenten, die geladen werden, verifiziert.

Initialisieren der Low-Level-Hardware: Pre-EFI Initialization (PEI)

Die Pre EFI Initialization Phase (PEI) initialisiert das System, sodass die Driver Execution Environment

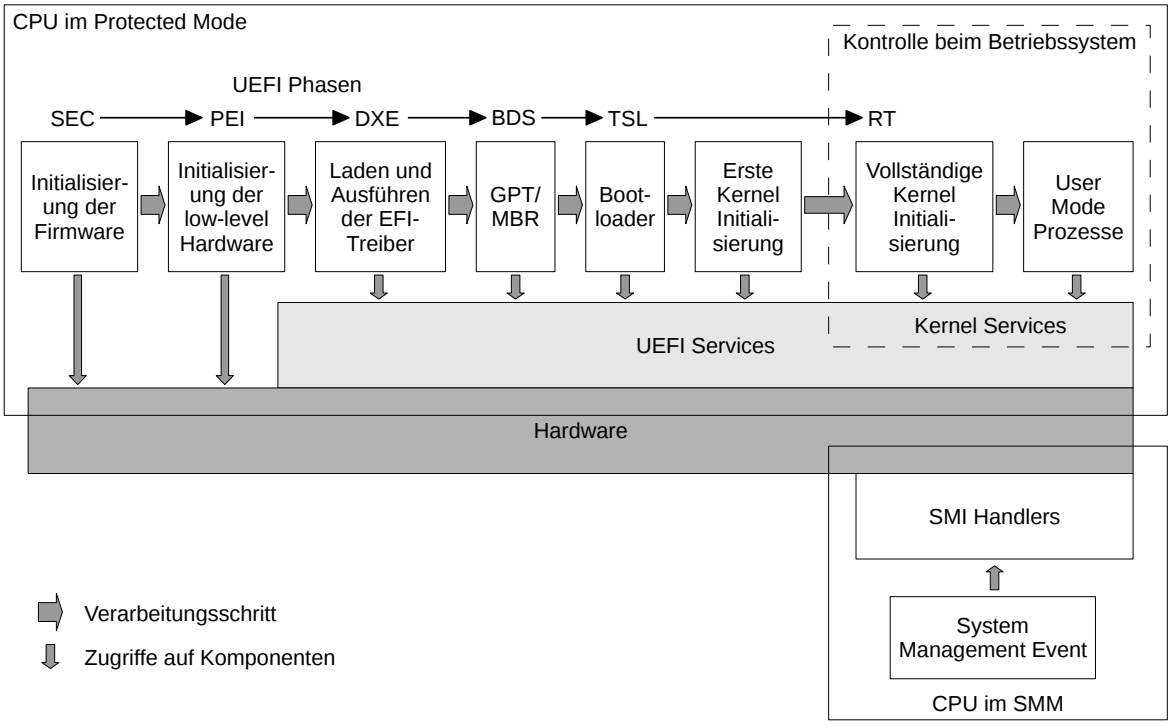

Abbildung 2.2.: Überblick über den Startvorgang mit dem UEFI nach [CPRS 11].

(DXE) Phase übernehmen kann. Sie besteht aus der PEI-Foundation, die mittels Pre-EFI Initialization Modules (PEIMs) Hardware initialisieren kann.

Die PEI-Phase stellt einen ausreichend großen zusammenhängenden Speicherbereich bereit, in den die Komponenten der DXE-Phase geladen und ausgeführt werden können. Hierfür lädt die PEI-Foundation die PEIMs und verifiziert diese mit der aus der SEC-Phase bereitgestellten Schnittstelle. Die PEIMs initialisieren den Teil der Hardware, der für die Ausführung des DXE benötigt wird, also CPU, Chipsatz, Mainboard und den Arbeitsspeicher.

Der Zustand des Systems am Ende der PEI-Phase wird der DXE-Phase über positionsunabhängige Datenstrukturen, den Hand-Off Blocks (HOB), übergeben.

Starten der UEFI-Umgebung: Driver Execution Environment (DXE)

In der Driver Execution Environment (DXE) Phase wird der Großteil der Systeminitialisierung durchgeführt. Sie stellt zudem die Boot und Runtime Services zur Verfügung, die von UEFI-Anwendungen verwenden werden können.

Die DXE-Phase besteht aus dem DXE-Kern, dem DXE-Dispatcher und den DXE-Treibern. Der DXE-Kern stellt die Boot-, Runtime- und DXE-Services bereit. Der DXE-Dispatcher sorgt dafür, dass DXE-Treiber auf den verfügbaren Speichermedien gefunden und in der korrekten Reihenfolge ausgeführt werden. Die DXE-Treiber sind dafür zuständig, die Hardware zu initialisieren und Schnittstellen für Ein-/Ausgabegeräte sowie Boot-Geräte zur Verfügung zu stellen. Bevor ein DXE-Treiber ausgeführt wird, wird dessen Authentizität vom DXE-Dispatcher überprüft.

Zuletzt wird der DXE Initial Program Load (IPL) ausgeführt und damit an die BDS-Phase übergeben.

Auswahl des Boot-Geräts: Boot Device Select (BDS)

Zusammen mit der Boot Device Selection (BDS) Phase richtet die DXE-Phase Systemkonsolen ein und startet weitere UEFI-Anwendungen oder ein Betriebssystem. Die DXE-Phase ist beendet, sobald eine Anwendung bzw. ein Betriebssystem geladen wurde, das die Boot-Services beendet hat.

Für den Start eines Betriebssystems ist es möglich, dem System über UEFI-Variablen mitzuteilen, welche Anwendung gestartet werden soll. Diese Variablen können zusätzlich Informationen für die zu star-

tende Anwendung enthalten, die dieser beim Start übergeben werden. Hierdurch ist die Firmware in der Lage, ein Boot Menü anzuzeigen, das dem Anwender alle verfügbaren Anwendungen und Betriebssysteme, die ausgeführt werden können, anzeigt. Da UEFI lediglich die Schnittstellen für den Zugriff auf UEFI-Variablen spezifiziert, hängt die Umsetzung vom Hersteller der Firmware ab.

Starten eines Betriebssystems oder einer UEFI-Anwendung: Transistent System Load (TSL)

In dieser Phase wird die zuvor festgelegte UEFI-Anwendung ausgeführt. Dies ist meistens ein Bootloader, der anschließend das Betriebssystem lädt. Dieser ist eine spezielle UEFI-Anwendung, die die Kontrolle über das gesamte System von der Firmware übernimmt. Sie verhält sich nach dem Laden wie gewöhnliche UEFI-Anwendungen, d.h. um Hardware anzusprechen, kann sie zunächst nur UEFI-Dienste und -Protokolle verwenden, die von der Firmware angeboten werden. Sobald der Bootloader ein Betriebssystem bzw. Kernel erfolgreich geladen hat, kann dieses die Kontrolle über das System übernehmen, indem es die Boot Services beendet. Ab diesem Zeitpunkt stehen nur noch die UEFI Runtime Services zur Verfügung.

Ausführung eines Betriebssystems: Runtime (RT)

Mit der Übergabe der Kontrolle über die Hardware vom Bootloader an das Betriebssystem ist der Startvorgang im Rahmen dieser Arbeit abgeschlossen. Das Betriebssystem verwaltet die Hardware eigenständig. Ihm stehen nur noch die UEFI Runtime Services zur Verfügung, über die u.a. auch ein Zugriff auf UEFI-Variablen möglich ist.

Während ein Startvorgang mit dem BIOS durch das wiederholte Laden und Ausführen verschiedener jeweils größerer Bootloader gekennzeichnet ist, geschieht dies mit UEFI zunächst nur einmal. Die geladene UEFI-Firmware initialisiert die Hardware und lädt bei Bedarf weitere Treiber nach, sodass sie zuletzt ein Betriebssystem starten kann. Ein Wechsel der Bitbreite von 16- auf 32- und ggf. auf 64-Bit entfällt dabei bei UEFI-Firmwares.

2.2.3. UEFI Secure Boot

Bei einem klassischen Startvorgang ohne UEFI Secure Boot sucht die Firmware des Systems einen Bootloader und führt diesen ohne Überprüfung aus. Sie kann hierbei zwischen vertrauenswürdiger Software und Schadsoftware nicht unterscheiden.

Das Ziel von UEFI Secure Boot ist es daher, die Ausführung nicht vertrauenswürdigen Programmcodes vor Ausführung des Betriebssystems zu verhindern. Hierfür wird mit einer kleinen Codebasis begonnen, die von einem vertrauenswürdigen Update-Prozess geschützt ist und damit einen sicheren Ausgangspunkt (Root of Trust) darstellt. Basierend auf diesem sicheren Ausgangspunkt wird jeweils sichergestellt, dass der nachfolgend ausgeführte Programmcode vertrauenswürdig ist und nicht verändert wurde. Ist ein auszuführendes UEFI-Image nicht vertrauenswürdig, wird dieses nicht ausgeführt und der Bootvorgang anderweitig fortgesetzt oder der Nutzer benachrichtigt. [WiRi 13]

Mittels asymmetrischer Kryptografie kann ein Softwarehersteller seine UEFI-Images signieren. Die Firmware kann die Authenzität und Integrität der Images mittels digitale Signaturen vor der Ausführung überprüfen. Damit diese als gültig angesehen werden, müssen sie von einer vertrauenswürdigen Stelle, d.h. von einem vertrauenswürdigen Zertifikat (trusted certificate), signiert worden sein. Das vertrauenswürdigen Zertifikat wird hierfür in Firmware gespeichert und die Signaturen in die UEFI-Images integriert.

Ablauf eines sicheren Startvorgangs mit Secure Boot

Der Startvorgang beginnt von einer initialen geschützten Firmware aus, die im weiteren Verlauf des Startvorgangs zusätzliche UEFI-Images, also Anwendungen und Treiber, lädt und ausführt. Jedes Image, das geladen wird, wird dabei von der Firmware überprüft und seine Signatur wird mit den im System hinterlegten Daten verglichen. [WiRi 13]

In einer Signatur-Datenbank ist verzeichnet, welche UEFI-Images autorisiert sind und ausgeführt werden dürfen. Da es aufgrund von Fehlern möglich ist, dass Schadsoftware signiert werden kann, existiert zusätzlich

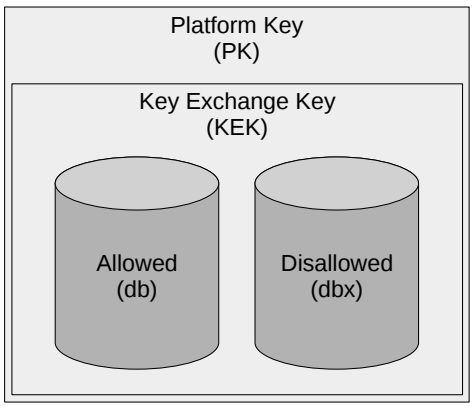

Abbildung 2.3.: Vertrauensbeziehung zwischen den Komponenten des Secure Boot angelehnt an [Sino 11].

eine Signatur-Datenbank für nicht vertrauenswürdige UEFI-Images.

Nach der Initialisierung der CPU und des Chipsatzes wird zunächst die Signatur der Firmware überprüft. Ist diese Prüfung erfolgreich, wird die Firmware geladen und ausgeführt. Diese erste Phase stellt damit den *Root of Trust* des Systems dar.

Beim Starten von UEFI-Images vergleicht die Firmware sodann die Signaturen der Images mit denen aus den Datenbanken. Wenn es für ein Image keine Signatur in der Signatur-Datenbank gibt oder die Signatur in der Datenbank der nicht vertrauenswürdigen Images steht, wird dieses nicht gestartet. Wenn es sich beim zu startenden Image um einen Bootloader handelt und für diesen keine Signatur vorliegt oder dieser nicht vertraut wird, wird stattdessen die nächste Boot-Option ausgeführt.

Da es eine große Anzahl an UEFI-Anwendungen und -Treibern sowie Betriebssystemen und Bootloadern gibt, ist es nahezu unmöglich, dass eine Firmware alle Signaturen hierfür beinhaltet. Um nicht von jedem vertrauenswürdigen Unternehmen ein Zertifikat speichern zu müssen, ist es möglich, Secure Boot mit zentralen Certificate Authorities (CA) zu verwenden. Die Signaturen der UEFI-Images werden dann in den Datei-Headern oder im System gespeichert. [Unif 13b]

Vertrauensbeziehungen zwischen den Komponenten

Zum Überprüfen der Signaturen werden verschiedene Signaturen, Schlüssel und Speicherorte verwendet. In Abbildung 2.3 wird deren Beziehung zueinander dargestellt.

Platform Key (PK)

Der Platform Key (PK) stellt eine Vertrauensbeziehung zwischen dem Hersteller der Plattform und der Firmware der Plattform her. Er wird für gewöhnlich vom Hersteller der Firmware beim Bau der Hardware gesetzt, der hierfür den öffentlichen Teil seines Schlüsselpaars in der Firmware speichert. Die Aufgabe des PK ist es, den Key Exchange Key (KEK) vor Veränderungen zu schützen, indem nur vom PK signierte Updates zugelassen werden. Der Hersteller kann so mit dem privaten Teil seines Schlüsselpaars den KEK anpassen.

Ist kein PK eingerichtet, befindet sich die Firmware im Setup-Modus. In diesem verlangt die Firmware keine Authentifizierung, sodass PK und die KEKs verändert werden können. Nachdem ein Platform Key eingerichtet wurde, befindet sich die Firmware im User-Modus und bleibt in diesem, bis der Key entfernt wird. [WiRi 13] [Unif 13b]

Key Exchange Key (KEK)

Key Exchange Keys (KEK) stellen eine Vertrauensbeziehung zwischen der Firmware und ausgeführten Anwendungen her. Im System können mehrere KEKs vorhanden sein. Ein KEK ist ein Schlüsselpaar, deren öffentlicher Teil vom PK signiert wurde und dazu dient, Updates der Signatur-Datenbanken zu signieren und damit vor unberechtigten Veränderungen zu schützen. Ohne Zugriff auf den privaten

Schlüssel des Schlüsselpaars können daher keine Veränderungen an der Signatur-Datenbank vorgenommen werden. [WiRi 13]

Signature Database (db) und Forbidden Signature Database (dbx)

Der Besitzer eines gültigen KEK kann Signaturen zu den Signatur-Datenbanken hinzufügen und entfernen. Diese Datenbanken verwalten zwei unterschiedliche Arten von Signaturen: Signaturen für Programmcode, der ausgeführt werden darf (db) sowie Signaturen für Programmcode, dessen Ausführung verboten ist (dbx).

Eine Signatur-Datenbank (signature database) besteht aus einer beliebigen Anzahl an Signaturlisten (signature lists). Jede Signaturliste besteht dabei aus einer Liste von Signaturen eines bestimmten Typs. Die Typen können hierbei beispielsweise SHA-256-Hashes, RSA-2048-Schlüssel oder RSA-2048-Signaturen sein.

Beim Ausführen eines UEFI-Images wird dessen Signatur vom UEFI-Boot-Manager überprüft. Ein Image wird nur dann ausgeführt, wenn das Image nicht signiert ist und dessen Hash in der Signature Database vorhanden ist, das Image signiert ist und die Signatur in der Signature Database vorhanden ist, oder wenn das Image signiert, der Signing Key in der Signature Database vorhanden und die Signatur valide ist. Ist der Hash oder die Signatur des Images in der Forbidden Signature Database, wird das Image nicht ausgeführt. [Unif 13b]

UEFI Authenticated Variables

Bei UEFI Authenticated Variables handelt es sich um sichere Speicherstellen, in denen die Firmware und Anwendungen sicherheitsrelevante Informationen hinterlegen können. Genutzt werden sie insbesondere als Speicherort für die Signaturen der Signatur-Datenbanken, die jeweils in einer UEFI-Variable gespeichert werden. Für die Anpassung der Authenticated Variables (und damit der beiden Datenbanken) stellt die Firmware einen Mechanismus bereit, mit dem die enthaltenen Hashes und Signaturen angepasst werden können. [Unif 13b]

2.3. TCG Measured Boot

Während es das Ziel von UEFI Secure Boot ist, die Ausführung von nicht vertrauenswürdigen Programmcodes vor Ausführung des Betriebssystems zu verhindern, versucht Trusted Computing Group (TCG) Measured Boot, Systeme, die manipuliert wurden, zu erkennen. [Trus 06]

Wenn ein System mit einem Rootkit infiziert wurde, kann dessen Existenz vom System selbst nicht festgestellt werden, da dieses seine Existenz vor dem System verstecken kann. Infizierte Systeme melden daher valide zu sein und können sich so beispielsweise weiterhin mit einem Unternehmensnetz verbinden.

Um dies zu verhindern, werden mittels Measured Boot während des Startens alle sicherheitsrelevanten Teile wie Firmware, Bootloader oder Kernel mit einer Hash-Funktion *gemessen*. Die Ergebnisse dieser Messung (measurement) werden in einem sicheren Speicherbereich, dem Platform Configuration Register (PCR), abgelegt.

Damit die Ergebnisse der Messungen von Dritten überprüft und nachträglich nicht verändert werden können, benötigt ein System hierfür ein sogenanntes Trusted Platform Module (TPM). Dieses ist ein Mikrocontroller mit eingebautem Speicherbereich, der im System verbaut und angebunden ist. [ZRM 10] [Trus 11]

In Abbildung 2.4 wird dargestellt, welche Komponenten beim Starten mittels Measured Boot bei einem UEFI-kompatiblen System gemessen werden. Die Messungen werden in unterschiedliche PCRs abgelegt: Messungen der EFI Plattform Firmware mit EFI Boot und Runtime Services werden beispielsweise in PCR 0 gespeichert, eine Messung der Partitionstabelle in PCR 5 und die Messung des Bootloaders in PCR 4.

Hervorzuheben ist zudem die Messung des Betriebssystems, in der Abbildung als PCR8+ gekennzeichnet: Messungen von Komponenten des Betriebssystems werden in PCR8 gespeichert und mit weiteren Messungen des Betriebssystems erweitert. Hierfür wird ein Hash über die neue Messung zusammen mit dem bestehenden Wert von PCR 8 gebildet und in PCR 8 abgelegt. [Trus 06]

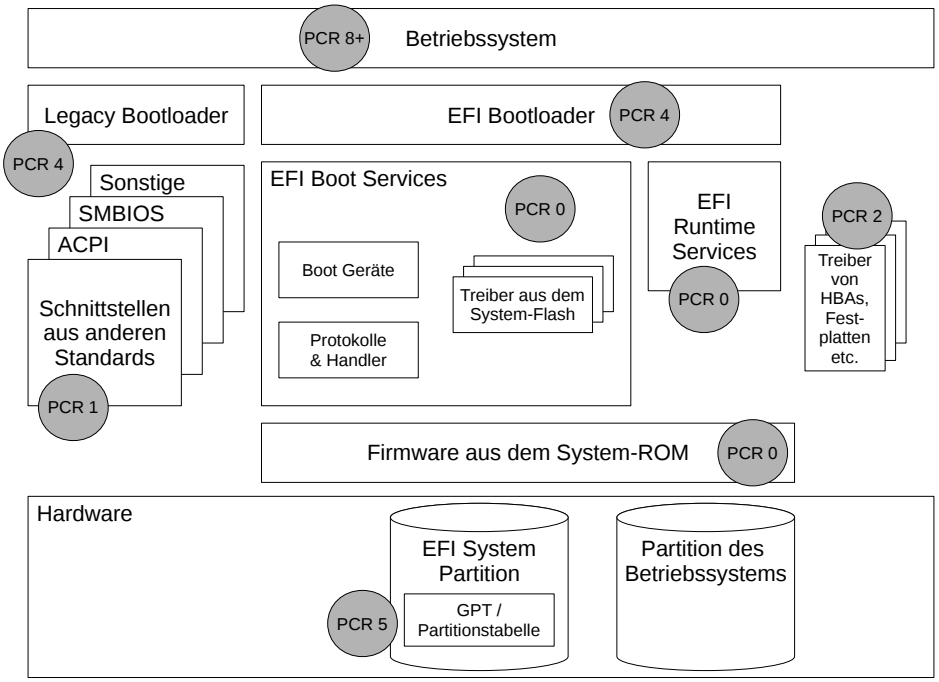

Abbildung 2.4.: Übersicht über UEFI-Komponenten, deren Prüfsummen beim Trusted Boot gespeichert werden entsprechend [Trus 06].

Im weiteren Verlauf der Arbeit wird zur Absicherung des Startvorgangs auf UEFI Secure Boot zurückgegriffen. Eine Betrachtung von Measured Boot entfällt dabei. Auf Basis der zuvor geklärten Begriffe und Grundlagen erfolgt im nächsten Abschnitt eine Analyse der Anforderungen an einen sicheren Startvorgang mit UEFI Secure Boot, der aufbauend auf einer Bedrohungsanalyse Annahmen und weitere Anforderungen definiert und so zur Erstellung von Bewertungskriterien führt.

3. Anforderungsanalyse

Ziel der vorliegenden Arbeit ist es, einen sicheren Startvorgang für OpenBSD zu ermöglichen. Dieser soll auf gängiger PC-Hardware möglich sein und auf UEFI aufbauen. Der Entwurf sicherer Systeme beginnt nach [Bund 92] mit Feststellung der Schutzbedürftigkeit, auf die eine Bedrohungs- sowie eine Risikoanalyse folgen.

Die Schutzbedürftigkeit des Startvorgangs von OpenBSD ergibt sich aus der Aufgabenstellung dieser Arbeit, sodass in Abschnitt 3.1 daher direkt die Bedrohungen analysiert werden, die die beim Starten verwendeten Komponenten betreffen. Eine Risikoanalyse erfolgt implizit durch die Definition von Annahmen in Abschnitt 3.2, die mögliche Bedrohungen auf Szenarien, die in dieser Arbeit behandelt werden können, begrenzen. Zusätzliche Anforderungen, die sich aus der Aufgabenstellung dieser Arbeit ergeben, werden in Abschnitt 3.3 besprochen. Schlussendlich wird in Abschnitt 3.3 eine Liste von Bewertungskriterien festgelegt, nach denen bestehende und zu konzeptionierende Lösungen bewertet werden sollen.

3.1. Analyse der Bedrohungen

Zu Beginn des Entwurfs eines sicheren Systems steht die Analyse der Bedrohungen, wie auch in [Ecke 13] ausgeführt. Hierfür werden die einzelnen beim Starten durchlaufenden Phasen hinsichtlich ihrer Sicherheit untersucht und auf die Rolle der betroffenen Komponenten der jeweiligen Phase eingegangen.

Eine Betrachtung der Bedrohungen entlang des zeitlichen Verlaufs des Startvorgangs ist naheliegend, da dies die übliche Nutzung des Systems nachbildet. Diese und die in der jeweiligen Phase verwendeten Komponenten werden in Abbildung 3.1 noch einmal zusammengefasst dargestellt. Eine ausführliche Beschreibung der Abläufe der jeweiligen Phase findet sich in Abschnitt 2.2.2.

Initialisierung der Firmware: Security (SEC)

In der SEC-Phase wird das System eingeschaltet und die Firmware initialisiert.

Verfügbarkeit

Mittels verschiedener technischer Ansätze (z.B. Fault Injection) ist es möglich, Hardware zu einer fehlerhaften Ausführung zu bewegen. Hierdurch kann beispielsweise die Verarbeitung kryptografischer Funktionen manipuliert werden. Für Näheres hierzu siehe [BECN+ 06]. Ebenso ist die Verfügbarkeit der SEC-Phase durch fehlerhafte Hardware wie dem Speicher beeinträchtigt. Generell kann die Verfügbarkeit der Komponenten dieser Phase nur durch die verwendete Hardware beeinträchtigt werden, die nicht Bestandteil dieser Arbeit ist.

Integrität

Die Funktionen der SEC-Phase liegen in einem nicht-beschreibbaren Speicherbereich (ROM) und können entsprechend nicht verändert werden. Die Integrität der in dieser Phase verwendeten Komponenten ist damit sichergestellt.

Authentizität

Da die Komponenten der SEC-Phase in einem nicht-beschreibbaren Speicherbereich liegen, ist deren Authentizität sichergestellt.

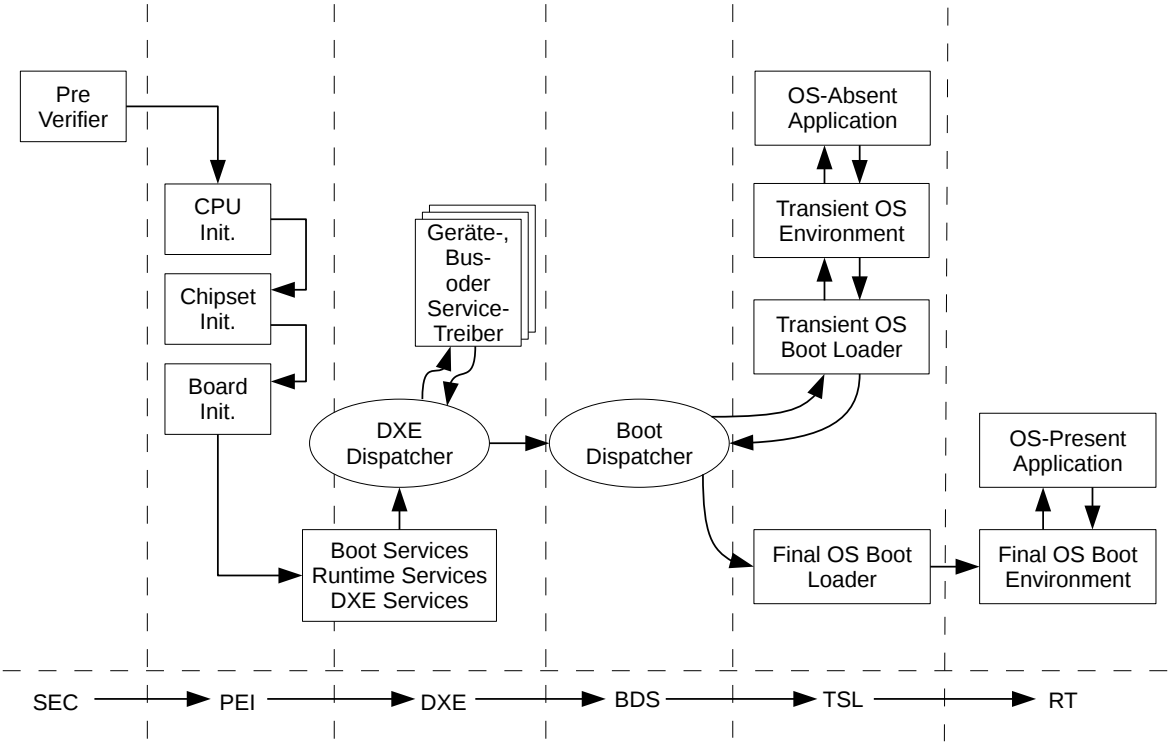

Abbildung 3.1.: Vereinfachte Darstellung der Abläufe in den Phasen des Startvorgangs nach [ZRM 10].

Initialisieren der Low-Level-Hardware: Pre EFI Initialization (PEI)

Nachdem die Firmware initialisiert wurde, wird sie ausgeführt. Sie lädt PEIMs nach und initialisiert erste Teile der Hardware, in die die DXE-Komponenten geladen werden. Der Übergang zur DXE-Phase erfolgt mittels HOBs.

Verfügbarkeit
Die Verfügbarkeit der Komponenten dieser Phase kann nur durch die verwendete Hardware beeinträchtigt werden, die nicht Bestandteil dieser Arbeit ist.

Integrität
Die Funktionen der PEI-Phase stammen aus dem BFV, das aus dem ROM geladen wurde. Aufgrund dessen konnten diese nicht verändert werden. Die Integrität der in dieser Phase verwendeten Komponenten ist damit sichergestellt.

Authentizität
Durch die Nutzung eines ROMs für das BFV ist auch dessen Authentizität sichergestellt.

Starten der UEFI-Umgebung: Driver Execution Environment (DXE)

In der DXE-Phase wird die UEFI-Umgebung gestartet. Hierzu werden Treiber für die Hardware geladen sowie Boot und Runtime Services angeboten. Mit dem Start des IPL wird an die BDS-Phase übergeben.

Verfügbarkeit
Durch fehlerhaft implementierte UEFI-Treiber kann die Verfügbarkeit des Systems beeinträchtigt werden.

Integrität
Dadurch, dass alle Treiber der DXE-Phase vor ihrer Ausführung durch den DXE-Dispatcher überprüft werden müssen, ist deren Integrität sichergestellt.

Authentizität

Dadurch, dass alle Treiber der DXE-Phase vor ihrer Ausführung durch den DXE-Dispatcher authentifiziert werden müssen, ist deren Authentizität sichergestellt.

Auswahl des Boot-Geräts: Boot Device Select (BDS)

Die zu startende UEFI-Anwendung erfährt die Firmware aus einer UEFI-Variable. Sie stellt sodann entweder ein Boot Menü an oder führt die konfigurierte Anwendung direkt aus. Ist keine zu startende Anwendung festgelegt oder verfügbar, wechselt die Firmware entweder in einen Setup-Modus oder startet eine UEFI-Shell.

Verfügbarkeit

Wie verfahren wird, wenn die BDS-Phase fehlschlägt, hängt vom Hersteller der Firmware ab. Zumeist wird entweder eine UEFI-Shell gestartet oder in den Setup-Modus gewechselt. Der Setup-Modus ist, wie erwähnt, abhängig vom Hersteller, bietet zumeist jedoch die Möglichkeit, das System neuzustarten oder auszuschalten. Wird eine UEFI-Shell ausgeführt, kann ein Angreifer mit den in der Shell vorhandenen Funktionen nahezu beliebige Veränderungen am System vornehmen und damit die Verfügbarkeit direkt beeinflussen.

Integrität

Secure Boot stellt sicher, dass lediglich Anwendungen ausgeführt werden können, deren Integrität und Authentizität sichergestellt ist; die Integrität der Komponenten dieser Phase ist damit sichergestellt.

Authentizität

Wie zuvor erwähnt, ist durch Secure Boot die Integrität und Authentizität der ausgeführten Anwendungen sichergestellt.

Starten eines Betriebssystems oder einer UEFI-Anwendung: Transistent System Load (TSL)

In dieser Phase wird die zuvor festgelegte UEFI-Anwendung ausgeführt. Für gewöhnlich handelt es sich hierbei um einen Bootloader, der anschließend das Betriebssystem lädt.

Verfügbarkeit

Durch Programmierfehler im Bootloader kann ein Angreifer diesen zum Abstürzen bringen oder eine abweichende Ausführung erwirken.

Integrität

Der Bootloader ist dafür verantwortlich, die Integrität des zu startenden Systems bzw. Kernels sicherzustellen. Ist dies nicht der Fall, kann ein Angreifer den Kernel und andere zu startende Komponenten durch eine manipulierte Dateien ersetzen oder den Start eines abweichenden Kernels bewirken. Beim Starten von OpenBSD wird die Integrität des Kernels aktuell nicht sichergestellt.

Authentizität

Wie bei der Integrität ist der Bootloader für die Sicherstellung der Authentizität des zu startenden Kernels verantwortlich. Beim Starten von OpenBSD wird dies aktuell nicht sichergestellt.

Ausführung eines Betriebssystems: Runtime (RT)

Mit der Übergabe der Kontrolle über die Hardware vom Bootloader zum Betriebssystem ist der Startvorgang im Rahmen dieser Arbeit abgeschlossen. Da ein Betriebssystem jedoch an den Phasen des Startvorgangs Veränderungen vornehmen kann, werden Bedrohungen, die sich durch die Ausführung des Betriebssystems ergeben, weiterhin betrachtet. Mit der Übergabe der Kontrolle über das System an das Betriebssystem stehen diesem nur noch die UEFI Runtime Services zur Verfügung, über die u.a. auch ein Zugriff auf UEFI-Variablen möglich ist.

Verfügbarkeit

Durch Fehler in den UEFI Runtime Services kann die Verfügbarkeit des Systems beeinträchtigt werden. In einer UEFI-Implementierung von Samsung konnte ein Nutzer mit Zugriff auf UEFI-Variablen einen

Phase/Schutzziel	SEC	PEI	DXE	BDS	TSL	RT
Verfügbarkeit	+	+	+	–	–	+
Integrität	+	+	+	+	✓	+
Authentizität	+	+	+	+	✓	+

Tabelle 3.1.: Abgleich von Bedrohungen mit den Schutzzielen der Phasen.

Fehler in der Implementierung dieser ausnutzen und sein Gerät damit unbrauchbar machen. [Garr 13b] Ebenso führt die Verwendung des Bootloaders *GRUB* zusammen mit UEFI bei neueren Lenovo Thinkpads zu Problemen, die die Hardware unbrauchbar macht. [Leem 14] Da das Betriebssystem Zugriff auf UEFI-Variablen hat, kann es diesen dafür nutzen, die verfügbaren Boot-Einträge zu verändern oder entfernen.

Integrität

Die Sicherstellung der Integrität der ausgeführten Programme obliegt allein dem Betriebssystem. Es kann hierfür jedoch auf die UEFI-Variablen zugreifen, in denen sicherheitsrelevante Informationen (Zertifikate, Signaturen) liegen und diese für die Überprüfung der Integrität nutzen. Mit dem Zugriff auf die UEFI-Variablen kann das Betriebssystem auch Anpassungen an den sicherheitsrelevanten Informationen in diesen vornehmen, also insbesondere Einträge in den Signatur-Datenbanken verändern. Dem Betriebssystem ist es zudem möglich, durch Update-Mechanismen die Firmware zu aktualisieren. Dieses sollte durch entsprechende Signaturen abgesichert sein, jedoch ist es beispielsweise bei ASUS durch einen Fehler in der UEFI-Implementierung möglich, eine nicht-signiertes Update der Firmware vorzunehmen, sodass die Sicherheitsmechanismen von Secure Boot umgangen werden können. Eine Sicherstellung von Integrität und Authentizität ist damit nicht mehr möglich. [BFB 13]

Authentizität

Gleiches wie für die Integrität gilt auch für die Authentizität der RT-Phase.

Es ist zu erkennen, dass die Verfügbarkeit nur bis zum Beginn der BDS-Phase sichergestellt ist, da die verwendeten Komponenten bis dahin keinen Zugriff von Außen ermöglichen. Die Integrität und Authentizität aller am Systemstart beteiligten Komponenten ist hingegen bis zum Start des Bootloaders durch die Verifikation der geladenen Dateien sichergestellt.

Aufgrund der Tatsache, dass die Verfügbarkeit des Systems maßgeblich von seiner Hardware abhängt, soll im weiteren Verlauf nur noch die Integrität und Authentizität der am Startvorgangs beteiligten Komponenten betrachtet werden. Hierbei ist zu erkennen, dass der Übergang zwischen der UEFI-kompatiblen Firmware (BDS) zum Kernel des Betriebssystems (RT) eine Lücke besteht, in der weder die Integrität noch die Authentizität der Komponenten gewährleistet sind.

Im Folgenden soll daher versucht werden, die Integrität und Authentizität der am Startvorgangs beteiligten Komponenten bis zur Ausführung des Kernels sicherzustellen. Das hierfür notwendige Element ist der Bootloader, der in der Phase TSL den Kernel lädt und ausführt.

Tabelle 3.1 fasst die genannten Erkenntnisse zusammen. Ein + stellt hierbei eine erfülltes Schutzziel der jeweiligen Phase dar, ein – die Abwesenheit ebendieser. Schutzziele einer Phase, die mit einem ✓markiert sind, werden aktuell noch nicht erfüllt, sollen aber im Rahmen dieser Arbeit erfüllt werden.

3.2. Annahmen

Ein sicherer Startvorgang im Rahmen dieser Arbeit soll allgemein die Integrität und Authentizität seiner Komponenten und des zu startenden Betriebssystems sicherstellen. Ist dies sichergestellt, kann das Betriebssystem selbst die Integrität und Authentizität der für den weiteren Betrieb notwendigen Komponenten feststellen.

Generell gibt es zwei Typen von Angreifern: Zum einen Personen oder Organisationen, die generell nicht autorisiert sind, das System zu benutzen, sowie zum anderen Personen oder Organisationen, die zwar rechtmäßig Zugriff auf das System haben, diesen aber missbrauchen. Dies kann durch das Vortäuschen einer anderen oder

falschen Identität, die nicht-autorisierte Benutzung von Funktionen oder den Missbrauch bestehender Befugnisse, um eine andere oder falsche Identität vorzutäuschen oder erweiterte Rechte zu erlangen, geschehen.

Die Bedrohung durch die Angreifer ist abhängig von ihrem Kenntnisstand bzw. Wissen, ihren Beweggründen sowie der verfügbaren Ressourcen. In den folgenden Betrachtungen werden keine Einschränkungen hinsichtlich der zuvor genannten Kriterien eines Angreifers gemacht.

Basierend hierauf ergeben sich Annahmen, von denen bei der Entwicklung eines sicheren Startvorgangs ausgegangen wird und die im Folgenden nach Hard- und Software getrennt näher erläutert werden.

3.2.1. Hardware

Die Hardware umfasst alle Komponenten des Systems, die einen Einfluss auf den Startvorgang haben können.

Integrität der Hardware
Die verwendete Hardware verhält sich wie spezifiziert, enthält keine Hintertüren und ist nicht bösartig. [KKS+ 08] Dies bedeutet insbesondere, dass beispielsweise das im System verbaute TPM-Modul nicht kompromittiert werden kann und etwaige kryptografisch relevante Funktionen der CPU nicht manipuliert sind. Ebenso wird angenommen, dass Software, die sich in einem nicht-schreibbaren Speicherbereich (ROM) befindet, nicht verändert werden kann.

Veränderungen an der Hardware
Veränderungen an (sicherheitsrelevanten) Teilen der Hardware (TPM, Festplatte, Tastatur etc.) können nicht unbemerkt stattfinden. [KKS+ 08]

3.2.2. Software

Die Software des Systems umfasst alle Anwendungen und Firmware, die während der Ausführung des Systems mit diesem interagieren, insbesondere auch die Firmware von Hardwarekomponenten.

Integrität der UEFI-Implementierung und Treiber/Firmware
Die Implementierung des UEFI-Frameworks enthält keine Fehler und verhält sich wie spezifiziert. Gleiches gilt für UEFI-Treiber und die Firmware der Hardware.

Sicheres Betriebssystem
Da die Betrachtung der Sicherheit von Betriebssystemen nicht den Fokus dieser Arbeit darstellt, wird dessen Sicherheit zur Laufzeit angenommen.

Vertrauenswürdige Signaturen
Ausgestellte Signaturen sind als vertrauenswürdig anzusehen. Das bedeutet insbesondere, dass die privaten Schlüssel sicher sind und ein Angreifer keine zusätzlichen Signaturen erzeugen kann.

Kompatibilität mit Windows 8.1
Damit Hersteller angeben dürfen, zu Microsoft Windows 8.1 kompatibel zu sein, müssen deren Systeme bestimmte Kriterien erfüllen. Diese Kriterien beinhalten Angaben zu UEFI Secure Boot, die von den meisten Hardwareherstellern umgesetzt wurden und im Rahmen dieser Arbeit als sinnvoll angesehen werden.

Hervorzuheben sind die folgende Punkte: [Micr 14]

- Das Compatibility Support Module (CSM) muss deaktiviert sein, wenn Secure Boot aktiviert ist.
- Im System sind PK und KEKs eingerichtet und die beiden UEFI-Signaturdatenbanken sind vorhanden und abgesichert.
- Der Zugriff auf die UEFI-Variablen, die die Signaturen und den Secure Boot Status enthalten, muss über die spezifizierte Schnittstelle möglich sein.
- Der Core Root of Trust für Secure Boot muss von einem Public Key ausgehen, der entweder durch einen sicheren Update-Mechanismus geschützt ist oder im ROM liegt.

- UEFI-Firmware und -Treiber müssen Eingaben korrekt validieren um Buffer Overflows etc. zu vermeiden.
- Die UEFI-Firmware überprüft die Signaturen aller beim Startvorgang geladenen Komponenten.
- Die UEFI-Shell und vergleichbare Anwendungen (z.B. für Wartung oder Setup) dürfen nicht signiert sein, d.h. sie können bei aktiviertem Secure Boot standardmäßig nicht gestartet werden.
- Secure Boot darf nur im Firmware-Setup deaktiviert werden, nicht während der Ausführung der UEFI-Umgebung oder danach.

3.3. Zusätzliche Anforderungen

Auf Basis der Rahmenbedingungen dieser Arbeit ergeben sich folgende, weitere Anforderungen, die für eine mögliche Lösung berücksichtigt werden müssen.

Unterstützung von UFS-Dateisystemen und des ELF-Dateiformats

Gegenstand dieser Arbeit ist ein sicherer Startvorgang für Unix-ähnliche Betriebssysteme am Beispiel OpenBSD. Es muss daher möglich sein, einen OpenBSD-Kernel sicher zu laden und auszuführen. Unter den meisten Unix-ähnlichen Systemen wie Oracle Solaris, Red Hat Enterprise Linux oder HP-UX ist das Dateiformat *Executable and Linkable Format* (ELF) gebräuchlich, ebenso unterstützen viele Unix-ähnlichen Betriebssysteme das *Unix File System* (UFS) als Dateisystem. [Orac 12a] [EaDo 14] [Orac 12b] [CoMo 14]

Der im Rahmen dieser Arbeit betrachtete OpenBSD-Kernel liegt im ELF-Dateiformat auf einer auf einer UFS-Partition vor, sodass eine zu implementierende Lösung diese unterstützen muss. [Open 14d] [Open 14b]

Freie Verfügbarkeit

Für eine spätere Integration in das OpenBSD-Projekt muss die Lösung unter einer liberalen Open-Source-Lizenz (BSD, MIT o.ä.) stehen. Das OpenBSD vermeidet es, Programmcode, der unter restriktiveren Lizenzen wie etwa der GNU General Public License (GPL) steht, aufzunehmen.

Unterstützung des UEFI

Bestandteil der Aufgabenstellung ist die Kompatibilität zu gängiger Hardware, weshalb eine zu implementierende Lösung auf dieser lauffähig sein muss. Aufgrund der Tatsache, dass die Betriebssysteme von Microsoft auf Desktop-PCs am weitestens verbreitet sind [NetM 14] und in Abschnitt 3.2.2 eine Kompatibilität zu Windows 8.1. angenommen wird, muss die zu entwickelnde Lösung Windows-kompatible Hardware und damit UEFI unterstützen.

3.4. Bewertungskriterien

Wie in der Bedrohungsanalyse gezeigt wurde, können die Phasen SEC bis BDS nur durch Fehler in den verwendeten Komponenten oder durch falsche Signaturen manipuliert werden. Beide Szenarien werden mit den Annahmen aus Abschnitt 3.2 von den weiteren Betrachtungen ausgenommen. Die Sicherheit des Betriebssystems ist nicht Bestandteil dieser Arbeit und wurde ebenfalls ausgeschlossen. Im Folgenden werden daher lediglich die Komponenten der TSL-Phase betrachtet, also dem Bootloader.

Aus der Analyse der Bedrohungen, dem Abgleich mit den Annahmen und den zusätzlichen Anforderungen ergibt sich deshalb folgender Kriterienkatalog, nach dem mögliche Lösungen bewertet werden sollen.

Integrität

Die entwickelte Lösung kann die Integrität des Systems feststellen und erkennt Manipulationen am System oder dessen sicherheitsrelevanten Konfiguration. [KKS+ 08]

Authentizität

Die entwickelte Lösung kann ihre Authentizität feststellen.

Sicherer Zustand

Die entwickelte Lösung kann während der Ausführung einen sicheren Zustand nicht mehr verlassen. [Nati 07]

Unterstützung von UFS-Dateisystemen

Gegenstand dieser Arbeit ist ein sicherer Startvorgang für Unix-ähnliche Betriebssysteme am Beispiel OpenBSD. Es muss daher möglich sein, einen OpenBSD-Kernel sicher zu laden und auszuführen. Der OpenBSD-Kernel liegt dabei auf einer auf einer UFS-Partition vor, sodass eine zu implementierende Lösung hiermit umgehen können muss.

Unterstützung des ELF-Dateiformats

Wie zuvor erwähnt, befasst sich diese Arbeit mit sicheren Startvorgang für Unix-ähnliche Betriebssysteme. Da diese überwiegend das ELF-Dateiformat unterstützen, in dem auch der OpenBSD-Kernel vorliegt, muss eine zu implementierende Lösung ebenfalls dieses Format unterstützen.

Freie Verfügbarkeit

Für eine spätere Integration in OpenBSD muss die Lösung unter einer liberalen Open-Source-Lizenz wie der BSD-Lizenz stehen.

Unterstützung des UEFI

Eine zu implementierende Lösung muss auf gängiger Hardware (amd64) und damit auf einer UEFI-kompatiblen Firmware ohne CSM lauffähig sein.

Mit den zuvor genannten Bewertungskriterien ist es möglich, bestehende Ansätze zu vergleichen. Im Folgenden werden daher zunächst bereits existierende Implementierungen betrachtet, um dann ihre Wiederverwendbarkeit hinsichtlich der zuvor genannten Kriterien zu prüfen.

4. Bestehende Lösungsansätze

Als eine der ersten verfügbaren Implementierungen von UEFI Secure Boot stellt der Startvorgang von Windows 8 eine gewisse Referenz dar und wird in Abschnitt 4.1 entsprechend zunächst betrachtet.

Anschließend werden zwei Open-Source-Ansätze betrachtet: Eine Kombination aus *shim* und *GRUB* am Beispiel Fedora Linux in Abschnitt 4.2, sowie eine Alternative zu *shim* in Abschnitt 4.3. Beide wurden von unterschiedlichen Entwicklern zum Absichern des Startvorgangs von Linux-Distributionen erstellt. Durch ihre offene Lizenzierung ist eine tiefergehende Analyse des jeweiligen Ansatzes möglich und bietet zudem teilweise die Möglichkeit, Quellcode in die zu erstellende Lösung zu übernehmen.

Basierend auf einer Betrachtung des aktuellen Stands der Entwicklung in OpenBSD in Abschnitt 4.4, wird schlussendlich in Abschnitt 4.5 eine kritische Betrachtung der bestehenden Ansätze vorgenommen, sodass ersichtlich wird, welche Komponenten wiederverwendet werden können und welche für die Absicherung des OpenBSD-Startvorgangs neu erstellt werden müssen.

4.1. Windows 8

Wenn ein System mit UEFI und aktiviertem Secure Boot startet, wird der Bootloader des Systems nur gestartet, wenn er von einer vertrauenswürdigen Stelle signiert ist, deren Schlüssel in den Signaturdatenbanken des UEFI (vgl. Abschnitt 2.2.3) gespeichert ist oder wenn seine Signatur vom Nutzer explizit in diese übernommen wurde.

In Abbildung 4.1 wird der Startvorgang mit Secure Boot unter Windows beschrieben, bei dem der erste Bootmanager von Microsoft signiert wurde und der Schlüssel von Microsoft im UEFI hinterlegt ist. [Micr 12] [Micr 13b]

1. Zunächst werden von der UEFI-Firmware der zu startende Bootmanager (BootMgr) und alle UEFI-Komponenten überprüft und, sofern diese vertrauenswürdig sind, ausgeführt.
2. Im Anschluss hieran überprüft der *BootMgr* den Bootloader (WinLoad). Ist dieser vertrauenswürdig, wird er ausgeführt; ist dies nicht der Fall, wird er durch eine Sicherungskopie ersetzt. Ist auch diese nicht vertrauenswürdig, zeigt die UEFI-Firmware eine Fehlermeldung an und wechselt in einen vertrauenswürdigen Zustand, der abhängig vom Hersteller der Firmware ist. Diese Überprüfung wird vom folgenden Bootloader weitergeführt, sodass auch der Kernel überprüft wird.
3. Wurde der Bootloader *WinLoad* geladen, überprüft er neben dem Kernel alle zu ladenden Treiber und Komponenten. War auch diese Überprüfung erfolgreich, wird die Kontrolle über das System an den Windows-Kernel übergeben.

4.2. Shim und GRUB am Beispiel Fedora Linux

Als eine der ersten Linux-Distributionen hatte Fedora Linux einen mittels Secure Boot abgesicherten Startvorgang. Fedora setzt hierfür auf die Verwendung von *shim* mit *GRUB*. Dieser ist ein Bootloader, der verschiedene Betriebssysteme laden kann und bei Fedora standardmäßig zum Laden des Linux-Kernels verwendet wird. In Zusammenarbeit mit einem weiteren Bootloader, der die Anforderungen von UEFI Secure Boot umsetzt, kann dieser Betriebssysteme laden und dessen Integrität und Authentizität sicherstellen. Dieser Teil wird hierbei von *shim* übernommen.

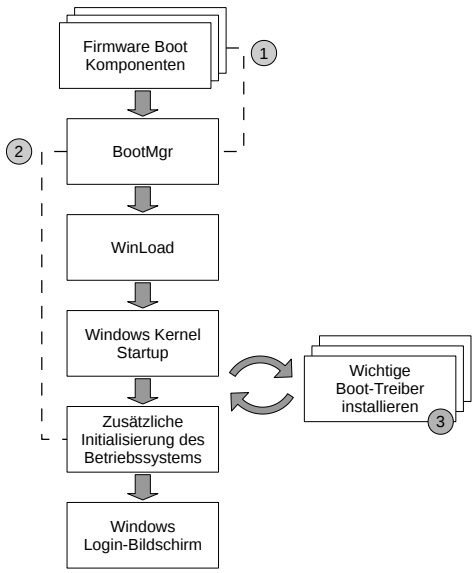

Abbildung 4.1.: Sicheres Starten mit Windows 8 nach [Nieh 13].

Einen Überblick über Secure Boot unter Fedora Linux gibt [BFJ+ 13]. Hier wird beschrieben, dass es im Wesentlichen zwei Möglichkeiten gibt, Fedora mittels Secure Boot zu starten. Die erste Methode verwendet einen sogenannten first-stage Bootloader (*shim*), der von Microsoft signiert wurde und einen weiteren Bootloader laden kann. Die zweite Methode ist eine allgemeinere Variante der ersten, die anstelle der Microsoft-Signatur vom Benutzer erstellte Schlüssel und Signaturen verwendet. Diese werden im System einrichtet und zum Signieren und Verifizieren von auszuführenden Komponenten des Startvorgangs benutzt.

In beiden Fällen werden die Informationen zu den Zertifikaten in UEFI-Variablen gespeichert, im ersten Fall (Microsoft-Schlüssel) sind diese jedoch meistens schon im System vorhanden. Microsoft verlangt dies für die Bestätigung der Kompatibilität eines Systems mit Windows 8, sodass die Schlüssel von Microsoft mit nahezu jeder im Handel verfügbaren Hardware ausgeliefert werden, was dazu führt, dass ein von Microsoft signierter Bootloader problemlos auf dieser laufen sollte.

Der erste Bootloader, *shim*, startet einen weiteren Bootmanager, *GRUB*, sobald er diesen verifiziert hat. Dieser wiederum lädt, verifiziert und startet dann den Kernel. *Shim* enthält ein selbst-signiertes Zertifikat (entweder von Fedora oder dem Benutzer erstellt), mit dem der *GRUB*-Bootloader verifiziert wird. Bevor ein Kernel gestartet wird, greift *GRUB* auf Funktionen von *shim* zu und überprüft hierüber die Signatur des Kernels. Damit können alle Komponenten mit Ausnahme von *shim* selbst von der eigenen Certificate Authority (CA) zertifiziert und überprüft werden.

Wenn das System mit Secure Boot gestartet wurde (User Mode), erkennen *shim*, *GRUB* und der Kernel dies und validieren die jeweils nächste Stufe vor der Ausführung.

GRUB ist von der Fedora Certificate Authority signiert. Sobald dessen Signatur von *shim* überprüft wurde, wird er ausgeführt. *GRUB* selbst enthält keine Schlüssel, weshalb er für die Überprüfung des Kernels auf Verifizierungs-Funktionen zurückgreift, die von *shim* bereitgestellt werden.

Wenn der Kernel erkennt, dass er mit aktiviertem Secure Boot gestartet wurde, akzeptiert er nur bestimmte Boot-Parameter, überprüft alle zu ladenden Kernel-Module und unterbindet den direkten Zugriff auf den Arbeitsspeicher (Direct Memory Access, DMA) durch ausgeführte Anwendungen.

4.3. Linux Foundation Bootloader und GRUB

Ein weiterer Ansatz zur Absicherung des Linux-Startvorgangs ist der Secure Boot Bootloader der Linux Foundation. Dieser überprüft und lädt, ähnlich wie *shim*, einen weiteren Bootloader, der dann wiederum das Be-

	Windows 8	Fedora Linux (shim & GRUB)	LF-Loader & GRUB	OpenBSD bisher
Integrität	+	+	+	–
Authentizität	+	+	+	–
Sicherer Zustand	+	+	+	–
Unterstützung von UFS-Dateisystemen	–	+	+	+
Unterstützung des ELF-Dateiformats	–	+	+	+
Freie Verfügbarkeit	–	–	–	+
Unterstützung des UEFI	+	+	+	–

Tabelle 4.1.: Vergleich bestehender Ansätze anhand der Bewertungskriterien aus Abschnitt 3.4.

triebssystem lädt. Er unterscheidet sich von *shim* dahingehend, dass dieser kryptografische Hashes verwendet; *shim* hingegen basiert auf Signaturen. Ein Nutzer des Linux Foundation Bootloaders muss daher die Hashes aller zu startenden Anwendungen selbst im System konfigurieren. Ein wesentlicher Vorteil hiervon ist, dass ein Nutzer oder eine Linux-Distribution weder Signatur-Infrastruktur bereitstellen, noch seinen Build-Prozess anpassen muss. [Garr 13a]

4.4. OpenBSD

Die aktuelle Version des OpenBSD-Betriebssystems unterstützt weder die Überprüfung der Integrität, noch der Authentizität des Kernels. Ebensowenig kann ein OpenBSD-System mit seinen Bootloadern ohne das UEFI Compatibility Support Module (CSM), das in dieser Arbeit nicht betrachtet wird, von einer UEFI-kompatiblen Firmware gestartet werden. [Open 14c]

Da OpenBSD auf der Berkeley Software Distribution (BSD) basiert, unterstützt es bereits das UFS-Dateisystem und steht überwiegend unter der freien BSD-Lizenz. Eine Unterstützung des ELF-Dateieiformats bietet OpenBSD bereits seit einigen Jahren. [Open 14b] [Open 14a] [Open 14d]

4.5. Kritische Betrachtung der bestehenden Ansätze

Tabelle 4.1 gibt einen Überblick über die verschiedenen, schon bestehenden Ansätze. Hierbei werden den betrachteten Ansätzen die Bewertungskriterien aus Abschnitt 3.4 zugeordnet, sodass leichter zu erkennen ist, ob ein Ansatz eine Anforderung erfüllt oder nicht.

Wie sich erkennen lässt, erfüllen Windows sowie die *GRUB*-basierten Startvorgänge die Anforderungen, die Secure Boot an einen sicheren Startvorgang stellt, nicht jedoch OpenBSD. Mit Ausnahme des Windows-Bootloaders sind die Ansätze quelloffen, jedoch stehen nur *shim* und OpenBSD selbst unter einer freien Lizenz, die eine Integration in OpenBSD ermöglicht.

In Zusammenarbeit mit *GRUB* sind sowohl der Bootloader der Linux Foundation als auch *shim* in der Lage, einen OpenBSD-Kernel im ELF-Dateiformat von einem UFS-Dateisystemen zu laden und auszuführen.

Für eine zu entwickelnde Lösung kommt aufgrund der Lizenzierung nur die Verwendung von *shim* infrage. Zusätzlich muss OpenBSD muss um die Sicherstellung der Integrität und Authentizität erweitert werden und eine Unterstützung des UEFI erhalten.

Im Folgenden wird daher ein Ansatz entwickelt, der mittels einer zweigeteilten Architektur auf Basis von *shim* und einem angepassten OpenBSD-Bootloader die Sicherheit des Startvorgangs gewährleistet und den Bewertungskriterien genügt.

5. Lösung und Prototyp

Basierend auf den in Kapitel 3 festgelegten Anforderungen wird im Folgenden ein Bootloader entworfen, der einen sicheren Startvorgang mit OpenBSD ermöglicht.

In Abschnitt 5.1 wird zunächst ein Konzept aufgezeigt, das den Anforderungen entspricht. Die konkrete Implementierung des Prototypen wird anschließend in Abschnitt 5.2 vorgestellt und erläutert.

5.1. Architektur

Wie in Abbildung 5.1 gezeigt wird, basiert die Architektur der Lösung grundsätzlich auf zwei Bootloadern, die nacheinander geladen werden. Der erste Bootloader ist im Funktionsumfang begrenzt und dient der Kompatibilität mit gängiger Hardware. Er lädt einen zweiten Bootloader, der mit dem Nutzer interagieren kann und ausreichend Funktionen bereitstellt, um einen OpenBSD-Kernel laden, verifizieren und ausführen zu können.

Allgemein

Da kryptografische Funktionen häufig unsicher implementiert werden, greift die Lösung auf bestehende Implementierungen zurück. [Schm 13]

Ebenso hat sich gezeigt, dass Software nie fehlerfrei ist. Zugleich steigt die Zahl der Fehler mit der Größe der Software, sodass die Lösung möglichst klein gehalten wird. [McCo 04]

Mit der Dauer der Verwendung und der Zahl der Nutzer steigt die Wahrscheinlichkeit, dass Fehler in der Software gefunden werden und behoben werden können, was wiederum für die Verwendung bestehender Implementierungen spricht.

Shim Bootloader

Da die KEKs von Microsoft in jeder Windows-kompatiblen Hardware konfiguriert sind, sollte ein Bootloader von Microsoft signiert sein. Dieser Prozess muss dabei mit jeder Änderung am Bootloader wiederholt werden. Um möglichst schnell neue Versionen signieren zu können und dennoch kompatibel zur meisten Hardware zu sein, ist eine zweistufige Lösung, wie sie auch unter Linux (Abschnitte 4.2 und 4.3) verwendet wird, zu bevorzugen.

Wegen seiner Größe und seinem klar begrenzten Funktionsumfang bietet sich dabei die Nutzung von *shim* an. In diesem muss im Programmcode ein öffentlicher Schlüssel einprogrammiert werden, der dann bei der Ausführung anderer UEFI-Anwendungen zum Überprüfen von Signaturen dient.

Somit ist es möglich, dass *shim* einmalig von Microsoft signiert wird und der eigene, neu erstellte Bootloader, dann nur noch eine Signatur aufweisen muss, die mit dem eigenen Private Key erzeugt wurde. Der vergleichsweise aufwendige Vorgang, einen Bootloader bei Microsoft signieren zu lassen, findet somit nur noch einmal statt. Werden Änderungen am funktional umfangreicheren eigenen Bootloader vorgenommen, können dessen neue Versionen unabhängig von Microsoft selbst signiert werden.

Neuer Bootloader

Da der erste Bootloader in seinem Funktionsumfang begrenzt ist und lediglich der Kompatibilität mit gängiger Hardware dient, wird ein weiterer Bootloader benötigt, der den OpenBSD-Kernel laden und ausführen kann. Dieser muss neu erstellt werden und kann dabei auf bestehende Softwarekomponenten von OpenBSD zurückgreifen.

Shim lädt die zu startende UEFI-Anwendung (den Bootloader) über eine UEFI-Schnittstelle direkt von einer speziellen Partition mit einem FAT-Dateisystem, der *EFI-Systempartition*. Da der UEFI-Standard

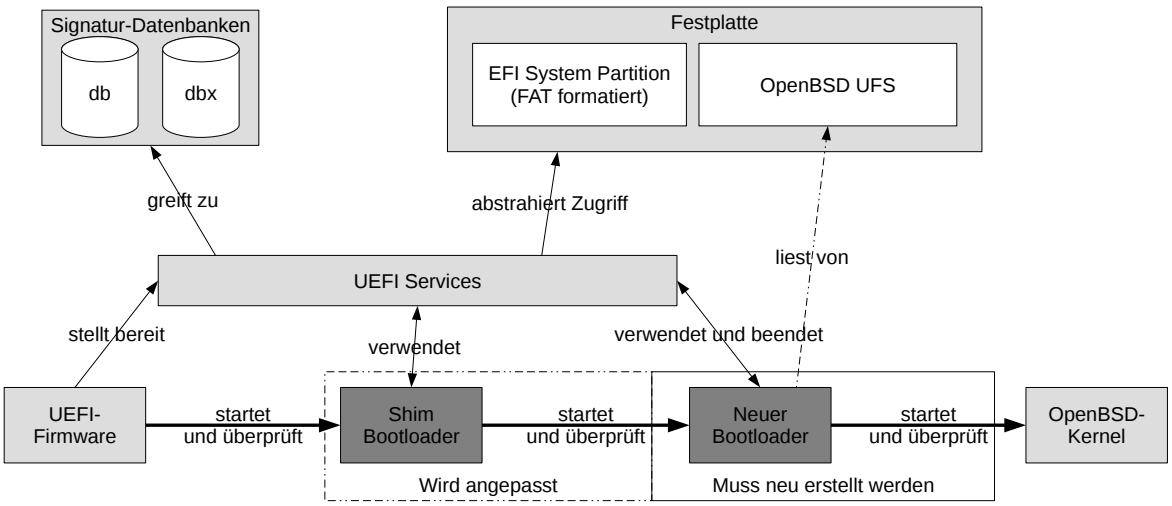

Abbildung 5.1.: Überblick über die Architektur der entwickelten Lösung.

lediglich die Unterstützung FAT-formatierter Partitionen verpflichtend vorschreibt, muss der neue Bootloader zum Zugriff auf UFS-Partitionen, die OpenBSD verwendet, direkt auf die Festplatte zugreifen. Hierfür kann der Bootloader über eine weitere UEFI-Schnittstelle blockweise auf die Festplatte zugreifen und die Partitionstabelle verarbeiten. Sobald er eine OpenBSD-Partition gefunden hat, kann er von dieser das OpenBSD-Disklabel lesen und auswerten, um dann den OpenBSD-Kernel von der UFS-Partition zu lesen.

Der neue Bootloader muss hierbei die Integrität und Authentizität des Kernels sicherstellen, wofür er dessen Hash mit denen aus den Signaturdatenbanken vergleicht. Im Erfolgsfall wird der Kernel ausgeführt, im Fehlerfall wird dem Nutzer die Möglichkeit gegeben, einen anderen Kernel auswählen.

5.2. Prototypische Umsetzung

Im Folgenden wird der Prototyp vorgestellt, der im Rahmen dieser Arbeit entwickelt wurde. Eine Beschreibung der Übersetzungs- und Testvorgänge findet sich in Anhang A und B. Die Implementierung ist, wie eingangs erwähnt, auf die amd64- bzw. x86_64-Plattform beschränkt.

Für die Absicherung des Startvorgangs greift die Umsetzung auf *shim* zurück, der als erstes gestartet wird und den zweiten Bootloader überprüft und ausführt. An *shim* wurden geringfügige Änderungen vorgenommen, die in Abschnitt 5.2.1 erläutert werden.

Der zweite Bootloader wurde im Rahmen dieser Arbeit neu erstellt und verwendet große Teile der bestehenden OpenBSD-Boot-Infrastruktur. Seine Umsetzung und damit verbundene Anpassungen am OpenBSD-Kernel werden in Abschnitt 5.2.2 näher betrachtet.

5.2.1. Shim

An *shim*, dem ersten Bootloader, der von der Firmware direkt gestartet wird, wurden keine funktionalen Änderungen vorgenommen.

Verändert wurden nur der Name des Bootloaders, der von *shim* automatisch aufgerufen wird (von “grubx64.efi” auf “loader.efi”). Ebenso wurde der einprogrammierte öffentliche Schlüssel, mit dem *shim* auszuführende Bootloader überprüft, durch einen selbst erzeugten Schlüssel ersetzt, sodass die Bootloader vom Autor der Arbeit selbst signiert werden können.

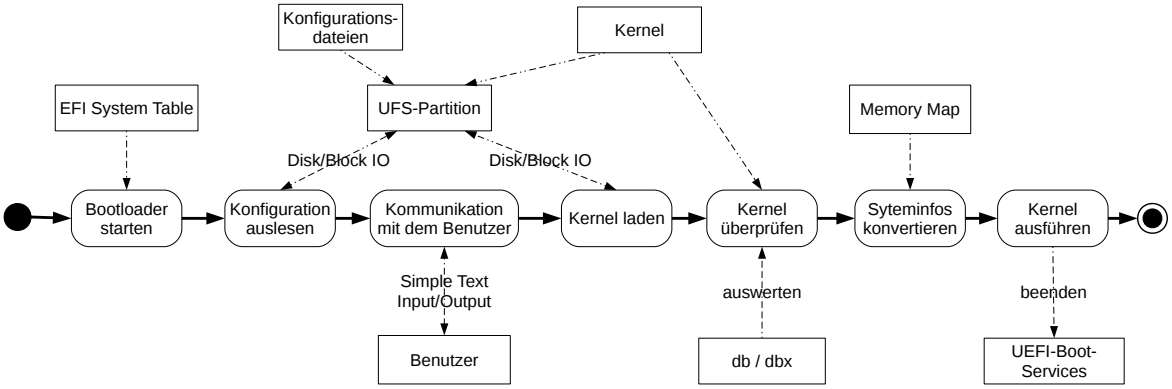

Abbildung 5.2.: Ablauf des neuen OpenBSD-Bootloaders.

5.2.2. Neuer OpenBSD-Bootloader

Der neu erstellte Bootloader bietet hinsichtlich seiner Bedienung und Konfigurierbarkeit einen ähnlichen Umfang wie der OpenBSD-Bootloader für BIOS-basierte Systeme.

Während die für den Nutzer sichtbaren Änderungen gering sind, ist der technische Unterbau des Bootloaders deutlich umfangreicher: Der Bootloader liest den OpenBSD-Kernel von einer GPT-Partition statt einer MBR-Partition und interagiert mit den UEFI-Services statt mit dem BIOS. Neu erstellt wurde zudem die Möglichkeit, geladene Kernel zu verifizieren, sodass die Chain of Trust des Secure Boot aufrecht erhalten wird.

Um möglichst wenig Programmcode zu verändern und damit eine Übernahme in das OpenBSD-Projekt zu vereinfachen, wurde auf bestehende Komponenten zurückgegriffen. Wie andere OpenBSD-Bootloader auch, verwendet der neue Bootloader daher große Teile der *libsa*, einer OpenBSD-spezifischen, von der Hardwarearchitektur unabhängigen Library mit allgemeinen Funktionen wie dem Zugriff auf UFS-Dateisysteme, die zum Starten benötigt werden. Zudem verwendet er Teile der Hardware-spezifischen *libsa* für amd64-Systeme, die Funktionen zum Umgang mit Intel-Architektur-spezifischen Eigenschaften enthält. Insbesondere der Umgang mit UFS-Dateisystemen, die Kommandozeile und das Laden ausführbarer Dateien stammen aus den beiden Libraries.

Build-Umgebung

Für eine Integration in OpenBSD muss ein UEFI-kompatibler Bootloader unter OpenBSD übersetzt werden können. Hierfür wird neben dem Compiler und Linker ein Programm benötigt, das Dateien aus dem ELF- ins PE/COFF-Format umwandeln kann. Diese Umwandlung des Bootloaders erfolgt mittels *objcopy*. Die im OpenBSD vorhandene Version unterstützt jedoch das PE/COFF-Format auf der amd64-Plattform nicht, sodass zunächst eine aktuellere Version des Programms portiert wurde.

Für die Unterstützung von Secure Boot müssen die Bootloader signiert und ein Hash des Kernels in der Signaturdatenbank hinterlegt sein. Unter OpenBSD existierte für die Signierung von UEFI-Anwendungen bisher keine Anwendung, ebensowenig für die Verwaltung der Signaturdatenbanken. Aus diesem Grund wurden zusätzlich Teile der Softwarepakete *sbsigntool* und *efitools* portiert.

UEFI

Für die Nutzung von UEFI Secure Boot wurde die Unterstützung von UEFI im OpenBSD-Bootloader ergänzt. Einen Überblick über die Umsetzung gibt Abbildung 5.2.

Der Zugriff auf UEFI-Funktionen erfolgt über spezifizierte Schnittstellen. Eine Beschreibung dieser Schnittstellen wurde aus den EFI-Header-Dateien von FreeBSD übernommen. Von der Verwendung der *gnu-efi*-Library wurde abgesehen, da diese zwar einige Hilfsfunktionen mitbringt, aufgrund ihrer Größe aber hauptsächlich zu zusätzlicher Komplexität führt. Für den eigentlichen Zugriff auf die Funktionen werden Informationen von der Firmware benötigt. Diese erhält der Bootloader aus der *EFI System Table*, einer Datenstruktur, die einer UEFI-Anwendung bei der Ausführung übergeben wird.

Die Informationen, die der Kernel benötigt, werden entweder aus einer Konfigurationsdatei entnom-

men, vom Nutzer erfragt oder von der UEFI-Firmware bereitgestellt. Dabei wandelt der Bootloader die Informationen, die er von der UEFI-Firmware erhält, in ein Format um, das der Kernel versteht. Hierfür wird beispielsweise die Memory Map, einer Beschreibung des Arbeitsspeichers, in der angegeben ist, welche Speicherbereiche reserviert und welche vom Kernel frei verwendbar sind, in eine klassische BIOS-Memory-Map konvertiert, sodass keine weitere Anpassung am Kernel notwendig ist.

Für den Zugriff auf die Festplatte werden das *Block-* und das *Disk-I/O*-Protokoll verwendet. Diese ermöglichen einen Zugriff auf die Festplatte unter Angabe das jeweils gewünschten Blocks oder Bytes.

Die Interaktion mit dem Nutzer erfolgt über die Protokolle *Simple-Text-Input* sowie *Simple-Text-Output*, die Ein- und Ausgaben des Benutzers abstrahieren.

GPT

OpenBSD bietet keine Unterstützung für UEFI und GPT (für GPT vgl. Abschnitt 2.2.1). Obwohl es möglich ist, mit UEFI eine UEFI-Anwendung wie einen Bootloader von einem Datenträger mit einer MBR-Partitionstabelle zu laden, stellt dies lediglich eine Abwärtskompatibilität dar. Im UEFI-Standard wird die GPT definiert, die für die Verwendung mit UEFI-Firmwares vorgesehen ist. Die Unterstützung für GPT entsprechend dieses Standards wurde im Prototypen berücksichtigt und implementiert.

Die Unterstützung von GPT ist dabei zweigeteilt: Während der GPT-kompatible Bootloader über die UEFI-Services von der Festplatte den Kernel liest und ausführt, muss der Kernel selbst auf die Festplatte zugreifen und die GPT-Partitionstabelle verarbeiten.

Zum OpenBSD-Kernel wurde eine Funktion *readgptlabel()* hinzugefügt, die es ermöglicht, das OpenBSD-Disklabel von einer GPT-Partition auszulesen und damit auf die Daten des Systems zugreifen zu können.

Da der Bootloader keinen direkten Zugriff auf die Hardware hat, liest er die GPT-Partitionstabelle mittels UEFI-Funktionen von der Festplatte und wertet diese aus. Er durchsucht die Partitionen dabei nach OpenBSD-Disklabels und lädt, sobald er diese ausgewertet hat, Konfigurationsdateien und den Kernel von einer OpenBSD-UFS-Partition.

Um den Prototypen zu testen, wurde das Userland-Programm *gdisk* für OpenBSD portiert, mit dem GPT-Partitionen verwaltet werden können.

Kryptografie

Der OpenBSD-Bootloader wurde um Funktionen zur Verifikation von Hashes erweitert, sodass er die Integrität und Authentizität des OpenBSD-Kernels sicherstellen kann. Die kryptografischen Funktionen, die für das Erstellen von Hashes benötigt werden, stammen aus OpenBSD, sodass davon ausgegangen wird, dass diese aufgrund ihrer Verbreitung als sicher und korrekt implementiert anzusehen sind.

Der Bootloader bildet zum Überprüfen ein SHA256-Hash des Kernels und vergleicht diese mit den Hashes, die er mittels UEFI-Funktionen aus den Signaturdatenbanken erhält. Wie in Abschnitt 5.1 festgelegt, wird der Kernel nur im Erfolgsfall ausgeführt; im Fehlerfall kann der Nutzer einen anderen Kernel auswählen.

Für eine Validierung des implementierten Prototypen sind verschiedene Schritte notwendig, die im Anhang genauer beschrieben werden. Zunächst müssen die OpenBSD-Komponenten wie in Anhang A dargestellt erzeugt werden, sodass diese wie in Anhang B beschrieben getestet werden können.

Hierfür wird zunächst ein OpenBSD-System installiert und eingerichtet, auf dem dann die für den Startvorgang benötigten Bootloader übersetzt werden können. Darüber hinaus werden Schlüsselpaare und Zertifikate erstellt, mit denen die Bootloader signiert wird und die für den Test benötigt werden.

Für den Test wird zunächst die UEFI-Implementierung *OVMF* erstellt und mit der Virtualisierungssoftware *qemu* ausgeführt. Anschließend werden die erzeugten Zertifikate in der Firmware eingerichtet und der OpenBSD-Kernel zur Liste der vertrauenswürdigen Dateien hinzugefügt. Zuletzt erfolgt in Anhang B.3 eine Beschreibung verschiedener Startabläufe, die die Funktionsfähigkeit des Prototypen darlegen.

	Windows 8	shim & GRUB	LF-Loader & GRUB	OpenBSD bisher	Erweitertes OpenBSD
Integrität	+	+	+	–	✓
Authentizität	+	+	+	–	✓
Sicherer Zustand	+	+	+	–	✓
Unterstützung von UFS-Dateisystemen	–	+	+	+	✓
Unterstützung des ELF-Dateiformats	–	+	+	+	✓
Freie Verfügbarkeit	–	–	–	+	✓
Unterstützung des UEFI	+	+	+	–	✓

Tabelle 5.1.: Vergleich bestehender Ansätze aus Tabelle 4.1 erweitert um die implementierte Lösung.

5.3. Bewertung des Systems

In Abschnitt 4.5 wurde festgestellt, dass Lösungen existieren, die die Anforderungen, die Secure Boot an einen sicheren Startvorgang stellt, erfüllen, jedoch nicht für die Integration in OpenBSD geeignet sind. Es wurde weiterhin erkannt, dass aus Lizenzgründen für eine zu entwickelnde Lösung nur die Verwendung von *shim* infrage kommt und dass OpenBSD um die Sicherstellung der Integrität und Authentizität seiner am Startvorgang beteiligten Komponenten erweitert werden und eine Unterstützung für das UEFI erhalten muss.

Die Komponenten des bestehenden OpenBSD-Bootloaders sind bereits in der Lage, mit UFS-Dateisystemen und dem ELF-Dateiformat umzugehen, sodass kein weiterer Anpassungsaufwand nötig war. Für die Sicherstellung von Integrität und Authentizität des Kernels wurde der Bootloader erweitert, sodass er nun den Kernel vor dem Starten verifiziert. Die Integrität und Authentizität des Bootloaders sowie die Kompatibilität zu Windows-kompatibler Hardware werden durch die Verwendung von *shim* sichergestellt, dessen Sicherheit wiederum sich aus der Verwendung von UEFI Secure Boot ergibt.

Während *shim* für UEFI-kompatible Firmware entworfen wurde und keine weiteren Anpassungen bedurfte, musste die UEFI-Unterstützung für den OpenBSD-Bootloader neu erstellt werden. Durch die Verwendung von *shim*, der eine zu OpenBSD kompatible Lizenz besitzt sowie die Anpassung bestehender Komponenten des OpenBSD-Bootloaders und die Freigabe der entwickelten Komponenten des Autors unter der BSD-Lizenz, ist eine Integration in OpenBSD problemlos möglich.

Zum besseren Überblick wird der Vergleich bestehender Ansätze mit den Bewertungskriterien aus Abschnitt 3.4 in Tabelle 5.1 um die im Rahmen dieser Arbeit erstellte Lösung erweitert. Es lässt sich erkennen, dass mit der Umsetzung der Architektur aus Abschnitt 5.1 ein sicherer Startvorgang für OpenBSD ermöglicht wurde. Die mittels *shim* und eigenen Anpassungen am OpenBSD-Bootloader erstellte Implementierung erfüllt dabei alle festgelegten Bewertungskriterien und bestätigt die entworfene Architektur. Der Prototyp wird deshalb im Anschluss an diese Arbeit überarbeitet und weiter verbessert, sodass er vom OpenBSD-Projekt übernommen und integriert werden kann.

6. Zusammenfassung und Ausblick

In dieser Arbeit wurde untersucht, welche Bedrohungen für den Startvorgang Unix-ähnlicher Systeme unter UEFI existieren und wie sich diese mit UEFI Secure Boot lösen lassen. Ausgehend von den Annahmen, dass die Hardware nicht kompromittiert ist und sich die UEFI-Firmware wie spezifiziert verhält, wurde eine Architektur entwickelt, die einen sicheren Startvorgang für Unix-ähnliche Betriebssysteme ermöglicht. Dessen prototypische Umsetzung für OpenBSD zeigte die Machbarkeit und bestätigt einen praktischen Nutzen.

Aufbauend auf eine Begriffsdefinition wurde im Rahmen einer Bedrohungsanalyse gezeigt, dass die Sicherheit eines Unix-Startvorgangs hinsichtlich Integrität und Authentizität unvollständig ist. Die Phase, in der der Bootloader ausgeführt wird, konnte dabei als unsicher bezüglich der zuvor genannten Kriterien ausgemacht werden. Basierend auf den Bedrohungen wurden Anforderungen und Annahmen definiert, die zur Aufstellung von Bewertungskriterien führten. Im Vergleich bestehender Ansätze hinsichtlich dieser Bewertungskriterien wurde festgestellt, dass zwar Ansätze existieren, die ein sicheres Laden von OpenBSD ermöglichen, jedoch nicht zur Integration in OpenBSD geeignet sind.

Mit dem Entwurf einer Architektur, die den Bedrohungen, Anforderungen, Annahmen und damit den Bewertungskriterien genügt, konnte anschließend gezeigt werden, dass ein sicheres Starten von OpenBSD möglich ist. Mit der konkreten Umsetzung bzw. Erstellung zweier Bootloader wurde deutlich, dass diese Architektur auch in der Praxis Bestand hat. Eine besondere Relevanz dieser Arbeit wird durch die Verwendung der Erkenntnisse durch unterschiedliche Parteien deutlich.

Der Autor der Arbeit hat im Rahmen des Google Summer of Code [1] die Möglichkeit bekommen, die Unterstützung von GPT und UEFI in OpenBSD zu integrieren [2], um damit die Grundlage für einen sicheren Startvorgang zu erstellen. Es ist geplant, auf dieser Basis einen mit Secure Boot kompatiblen Startvorgang für OpenBSD zu ermöglichen, der auf den Ergebnissen dieser Arbeit aufbaut.

Zudem werden die Ergebnisse dieser Arbeit bei der Entwicklung neuer Produkte wie dem *vs-top* der Firma Genua berücksichtigt. Genua ist ein Unternehmen, das Netzkomponenten wie Firewalls unter besonderer Berücksichtigung der IT-Sicherheit herstellt. Genua erstellt zudem eine mobile Workstation, die einen sicheren Startvorgang benötigt, der den Anforderungen des Bundesamts für Sicherheit in der Informationstechnik (BSI) genügt. Ausgehend von den Ergebnissen dieser Arbeit wird daher von Genua in Zusammenarbeit mit dem Autor ein sicherer Startvorgang basierend auf UEFI Secure Boot entworfen und für die neuen Produkte implementiert.

Die Arbeiten auf dem Gebiet der Sicherheit von Startvorgängen sind jedoch längst nicht abgeschlossen. Weiterer Forschungsbedarf besteht hinsichtlich Measured Boot, welches in dieser Arbeit nicht berücksichtigt wurde. Es sollte untersucht werden, ob es auch unter Unix-ähnlichen Systemen wie OpenBSD mittels Measured Boot möglich ist, Änderungen bzw. Manipulationen an der Konfiguration oder Software des Systems festzustellen und zu protokollieren.

Ebenso gilt es zu evaluieren, inwiefern sich die Konzepte des Secure Boot auch auf den Kernel übertragen lassen. Hierfür sollte geklärt werden, ob eine Signierung von Treibern, Kernel-Modulen und Userland-Programmen unter OpenBSD und verwandten Systemen möglich ist und ob dies einen Sicherheitsgewinn gegenüber anderen Konzepten darstellt.

[1] https://www.google-melange.com/gsoc/homepage/google/gsoc2014
[2] https://www.google-melange.com/gsoc/project/details/google/gsoc2014/mmu/5639274879778816

A. Erstellung der Komponenten

Im Rahmen dieser Arbeit wurden verschiedene Komponenten erstellt und angepasst. Die Quelltexte hierzu finden sich auf der beigelegten CD. Die nachfolgenden Abschnitte beschreiben, wie diese übersetzt und eingerichtet werden können.

Für die Erstellung der Komponenten und einem späteren Test des Prototypen wird ein OpenBSD-System benötigt. Dieses wird zunächst installiert und alle für den sicheren Startvorgang benötigten Komponenten darin erzeugt. Die Installation erfolgt dabei in einer virtuellen Maschine unter Ubuntu 13.10.

A.1. OpenBSD installieren

Auf dem Linux-System wird zunächst eine virtuelle Maschine erzeugt, in der dann das OpenBSD-System installiert werden kann.

Listing A.1: Virtuelle Maschine erstellen und starten

```
$ qemu-img create openbsd.img 10G
$ qemu-system-x86_64 -hda openbsd.img -cdrom install55.iso -m 512 -boot d
```

Die Installation kann mit den Standardwerten durchgeführt werden. Im Anschluss an die Installation muss die dieser Arbeit beiliegende CD im OpenBSD zur Verfügung gestellt werden. Hierfür muss die virtuelle Maschine mit der Option `-cdrom /dev/cdrom` neu gestartet werden.

Listing A.2: CD bereitstellen

```
$ qemu-system-x86_64 -hda openbsd.img -cdrom /dev/cdrom -m 512
```

Sobald das OpenBSD gestartet ist, kann die CD mit folgendem Befehl verwendet werden.

Listing A.3: CD einbinden

```
# mount_cd9660 /dev/cd0a /mnt
```

A.2. Build-Umgebung vorbereiten

Zum Erzeugen von UEFI-Anwendungen für amd64-Systeme unter OpenBSD wird eine aktuelle Version von *objcopy* benötigt, die zunächst installiert werden muss. Zusätzlich müssen weitere Pakete, die Programme zum Signieren von UEFI-Anwendungen und zum Verwalten der Signaturdatenbanken enthalten, installiert werden.

Listing A.4: Hilfsprogramme aus einem Paket installieren

```
# pkg_add -i /mnt/packages/amd64/objcopy-2.24.tgz
# pkg_add -i /mnt/packages/amd64/efitools-1.4.2.tgz
# pkg_add -i /mnt/packages/amd64/sbsigntool-0.6.tgz
```

Die Anwendungen können alternativ auch als Port aus ihrem Quelltext übersetzt werden. Im Folgenden wird beispielhaft die Erstellung des *objcopy*-Ports gezeigt.

Listing A.5: Objcopy aus dem Quelltext erstellen und installieren

```
# tar xzf /mnt/src/objcopy-port.tar.gz -C /usr/ports/devel
# cd /usr/ports/devel/objcopy
# make && make install
```

A.3. Bootloader erstellen

Für die Erstellung des Bootloaders wird der OpenBSD-Quelltext benötigt, der dafür zunächst entpackt werden muss. Anschließend kann der OpenBSD-UEFI-Bootloader erstellt werden.

Listing A.6: OpenBSD-Quelltext entpacken und den Bootloader erstellen

```
# tar xzf /mnt/src/src.tar.gz -C /usr
# cd /usr/src/sys/arch/amd64/stand/efiboot
# make obj
# make
```

A.4. GPT-Support erstellen

Die im Rahmen dieser Arbeit erstellte Unterstützung für GPT wird für die Testdurchläufe nicht benötigt. Um sie dennoch zu benutzen, muss zunächst der OpenBSD-Kernel neu übersetzt werden.

Listing A.7: Kernel erstellen

```
# cd /usr/src/sys/arch/amd64/conf
# config UEFI
# cd ../compile/UEFI
# make clean && make
# make install
# reboot
```

Anschließend muss das Userland neu gebaut werden, damit die zur Verwaltung von GPT-Partitionen notwendigen Anwendungen erstellt werden.

Listing A.8: Userland erstellen

```
# rm -rf /usr/obj/*
# cd /usr/src
# make obj
# cd /usr/src/etc && env DESTDIR=/ make distrib-dirs
# cd /usr/src
# make build
```

A.5. Shim erzeugen

Shim liegt als OpenBSD-Port vor, der entpackt, erzeugt und installiert werden muss. Bei der Erstellung des Ports wird automatisch ein Zertifikat erzeugt, das in *shim* eingebaut ist und mit dem der zweite Bootloader signiert wird.

Listing A.9: Shim erstellen und installieren

```
# tar xzf /mnt/src/shim-port.tar.gz -C /usr/ports/sysutils
# cd /usr/ports/sysutils/shim
# make && make install
```

A.6. Schlüssel- und Zertifikatserstellung

Für die Secure-Boot-Signaturen werden verschiedene Schlüsselpaare benötigt. Hierfür wird zunächst ein 2048-Bit RSA-Schlüssel für den Platform Key sowie ein selbst-signiertes Zertifikat für diesen erstellt. Anschließend wird analog dazu ein eigener KEK mit zugehörigem Zertifikat erstellt.

Listing A.10: PK erstellen

```
$ openssl genrsa -out pk-key.rsa 2048
$ openssl req -new -x509 -sha256 -subj '/CN=testpk' \
  -key pk-key.rsa -out pk-cert.pem
$ openssl x509 -in pk-cert.pem -inform PEM -out pk-cert.der -outform DER
$ openssl genrsa -out kek-key.rsa 2048
$ openssl req -new -x509 -sha256 -subj '/CN=testkek' \
  -key kek-key.rsa -out kek-cert.pem
$ openssl x509 -in kek-cert.pem -inform PEM -out kek-cert.der -outform DER
```

A.7. Signieren der Bootloader

Damit *shim* unter Secure Boot ausgeführt werden kann, muss er mit dem KEK signiert sein.

Listing A.11: Shim signieren

```
$ mkdir /tmp/efi
$ sbsign --key kek-key.rsa --cert kek-cert.pem --output /tmp/efi/shim.efi \
  /usr/local/share/efi/shim.efi
```

Der zweite Bootloader, *loader.efi*, wird mit dem in *shim* gespeicherten Zertifikat überprüft. Hierfür wird der Bootloader also mit dem *shim*-Schlüssel wie folgt signiert.

Listing A.12: loader.efi signieren

```
$ sbsign --key /usr/local/share/efi/secureboot/shim.key \
  --cert /usr/local/share/efi/secureboot/shim.crt \
  --output /tmp/efi/loader.efi /usr/obj/sys/arch/amd64/stand/efiboot/loader.efi
```

A.8. Anwendungen und Zertifikate für den Test bereitstellen

Für die Testdurchführung werden neben den Zertifikaten und Bootloadern weitere Hilfsprogramme benötigt, die in einem temporären Verzeichnis gesammelt und anschließend auf das Linux-Hostsystem kopiert werden.

Listing A.13: Sammeln und Kopieren der Zertifikate und Anwendungen

```
$ mkdir -p /tmp/efi/certs
$ cp pk-cert.der kek-cert.der /tmp/efi/certs/
$ cp /usr/local/share/efi/secureboot/shim.der /tmp/efi/certs/
$ cp /usr/local/share/efi/{HashTool.efi,KeyTool.efi} /tmp/efi/
$ cp /usr/local/share/efi/shim.efi /tmp/efi/shim-unsigned.efi
$ cp /usr/obj/sys/arch/amd64/stand/efiboot/loader.efi /tmp/efi/loader-unsigned.efi
$ cp /bsd /tmp/efi/
$ scp -r /tmp/efi user@linux:disk
```

Das OpenBSD-System muss anschließend wieder ausgeschaltet werden, damit es für einen Test mit UEFI genutzt werden kann.

B. Test des Prototypen in einer Virtualisierungsumgebung

Der Test des Prototypen kann generell auf zwei verschieden Arten durchgeführt werden: Zum einen auf Hardware mit einer UEFI-kompatiblen Firmware und zum anderen in einer virtualisierten Umgebung. Im Folgenden wird von einem Host-System auf Basis von Ubuntu 13.10 ausgegangen, unter dem ein virtualisertes System für die Tests ausgeführt wird.

Zum Testen der erstellten Komponenten werden *Qemu* sowie die UEFI-Firmware *Open Virtual Machine Firmware* (OVMF) benötigt. Da Änderungen an UEFI-Variablen mit älteren *Qemu*-Versionen bei jedem Neustart verworfen werden und ein Test damit deutlich aufwendiger ist, wird eine Version 1.7 oder neuer vorausgesetzt. Aufgrund der Tatsache, dass die Binärpakete von *OVMF* ohne Unterstützung für Secure Boot angeboten werden, muss die Firmware aus dem Quelltext mit veränderten Einstellungen neu erzeugt werden.

B.1. UEFI-Firmware erstellen

Für den Test des Prototypen wird eine aktuelle Firmware benötigt, die UEFI unterstützt. Die Erstellung dieser Firmware mit Unterstützung für Secure Boot wird im Folgenden beschrieben.

Zu Beginn müssen die zum Bauen der Firmware benötigten Abhängigkeiten installiert werden.

Listing B.1: Abhängigkeiten installieren

```
$ sudo apt-get install build-essential git uuid-dev iasl
```

In einem passenden Verzeichnis wird anschließend der Quellcode heruntergeladen und entpackt.

Listing B.2: EDK2 Quellcode herunterladen

```
$ mkdir ~/src
$ cd ~/src
$ git clone https://github.com/tianocore/edk2.git
$ cd edk2
$ git checkout -b 9afa7499df3139379b5b6553fd89d797a7ba3ded
```

Im nächsten Schritt müssen die für das Übersetzen des Quellcodes benötigten Programme erstellt werden.

Listing B.3: Build-Dependencies erzeugen

```
$ make -C BaseTools
```

Darauf aufbauend wird die Übersetzungs-Umgebung eingerichtet.

Listing B.4: Environment einrichten

```
$ . edksetup.sh
```

Anschließend muss das sogenannte build target angepasst werden, damit die Firmware erzeugt werden kann. In der Datei `Conf/target.txt` müssen dafür die Werte einiger Variablen wie folgt angepasst werden.

Ursprüngliche Werte:

Listing B.5: Ursprüngliche Werte

```
ACTIVE_PLATFORM       = Nt32Pkg/Nt32Pkg.dsc
TOOL_CHAIN_TAG        = MYTOOLS
TARGET_ARCH           = IA32
```

Neue Werte:

Listing B.6: Neue Werte

```
ACTIVE_PLATFORM       = OvmfPkg/OvmfPkgX64.dsc
TOOL_CHAIN_TAG        = GCC48
TARGET_ARCH           = X64
```

Für Secure Boot benötigt die Firmware das Softwarepaket *OpenSSL*, das wie folgt hinzugefügt wird.

Listing B.7: OpenSSL hinzufügen

```
$ cd CryptoPkg/Library/OpensslLib/
$ wget http://www.openssl.org/source/openssl-0.9.8za.tar.gz
$ tar xzf openssl-0.9.8za.tar.gz
$ cd openssl-0.9.8za
$ patch -p0 -i ../EDKII_openssl-0.9.8za.patch
$ cd ..
$ ./Install.sh
$ cd ../../..
```

Im letzten Schritt kann nun die Firmware erzeugt werden.

Listing B.8: Firmware erzeugen

```
$ build -DSECURE_BOOT_ENABLE=TRUE
```

B.2. UEFI-Firmware einrichten

Um Secure Boot mit der Firmware testen zu können, muss diese ausgeführt und eingerichtet werden. Hierfür wird zunächst die Virtualisierungsumgebung *Qemu* vorbereitet und ausgeführt. Dabei wird davon ausgegangen, dass sich die Komponenten aus Anhang A im Verzeichnis `~/disk` befinden.

Listing B.9: Qemu vorbereiten und starten

```
$ cd ~
$ cp ~/src/edk2/Build/OvmfX64/DEBUG_GCC48/FV/OVMF.fd ovmf-secureboot.fd
$ qemu-system-x86_64 -pflash ovmf-secureboot.fd -hda fat:disk/ -net none \
  -hdb openbsd.img -serial stdio -m 512 -display none
```

Nach dem Starten der Firmware befindet sich standardmäßig im Setup-Modus und eine UEFI-Shell wird geladen.

Aufgrund dessen, dass der OpenBSD-Kernel noch nicht auf UEFI-Variablen zugreifen kann und so unter OpenBSD keine Veränderungen an den Signaturdatenbanken vorgenommen werden können, muss derzeit auf einen Workaround zurückgegriffen werden: Der OpenBSD-Kernel wird temporär auf die EFI-Systempartition kopiert, sodass dann die Anwendung `HashTool.efi` wie folgt den Hash des Kernels bilden und in die Signaturdatenbank eintragen kann.

Listing B.10: In der UEFI-Shell die Anwendung `HashTool.efi` starten

```
Shell> fs0:\HashTool.efi
```

In der Anwendung müssen folgende Schritte durchgeführt werden:

1. Im Menü den Eintrag Enroll Hash auswählen.
2. Die Datei bsd auswählen und den Hash mit Yes bestätigen.
3. Die Anwendung mit Exit verlassen.

Damit Firmware und Bootloader Signaturen überprüfen können, müssen die Signaturen zunächst in der Firmware eingerichtet werden. Die Firmware wird dafür in den Setup-Modus überführt, in dem der PK und der KEK festgelegt werden können und somit Secure Boot aktiviert wird.

Nach dem Ausführen des Workarounds wird weiterhin die UEFI-Shell ausgeführt. Diese muss beendet werden, damit das Einstellungsmenü der Firmware erscheint.

Listing B.11: UEFI-Shell beenden

```
Shell> exit
```

Im Einstellungsmenü müssen dann folgende Schritte durchgeführt werden:

1. Zunächst den Eintrag Device Manager, dann Secure Boot Configuration auswählen.
2. Dort den Secure Boot Modus auf Custom Mode ändern.
3. Den Eintrag Custom Secure Boot Options auswählen.
4. Danach die Einträge PK Options, Enroll PK und anschließend Enroll PK Using File auswählen.
5. Zum Verzeichnis certs navigieren, dort die Datei pk-cert.der auswählen und mit Commit Changes and Exit bestätigen.
6. Erneut den Eintrag Secure Boot Configuration, danach Custom Secure Boot Options, KEK Options, Enroll KEK und Enroll KEK using File auswählen.
7. Zum Verzeichnis certs navigieren, dort die Datei kek-cert.der auswählen und mit Commit Changes and Exit bestätigen.
8. Wieder die Einträge Secure Boot Configuration und Custom Secure Boot Options auswählen.
9. Sodann DB Options, Enroll Signature und Enroll Signature Using File auswählen.
10. Erneut zum Verzeichnis certs navigieren, dort die Datei kek-cert.der auswählen und mit Commit Changes and Exit bestätigen.
11. Zuletzt mit der Escape-Taste das Menü verlassen und mit Continue den Bootvorgang fortsetzen.

B.3. Testdurchläufe

Die Testdurchläufe werden der Einfachheit halber über die UEFI-Shell ausgeführt. Für eine Verwendung auf realen Systemen sollte die Shell deaktiviert sein und der Name der auszuführenden UEFI-Anwendung (*shim*) in der Firmware gespeichert sein.

Im Folgenden wird davon ausgegangen, dass sich auf fs0 ein FAT-formatiertes Dateisystem befindet, das die UEFI-Anwendungen enthält.

B.3.1. Erfolgreicher Startvorgang

Für einen erfolgreichen Startvorgang wird *shim* gestartet, der dann den zweiten Bootloader überprüft und startet. Dieser wiederum überprüft und startet anschließend den Kernel.

Listing B.12: Shim ausführen

```
Shell> fs0:\shim.efi
```

B.3.2. Nicht vertrauenswürdiger Kernel

Um zu überprüfen, ob der Bootloader den auszuführenden Kernel verifiziert, wird statt des erlaubten Kernels ein anderer Kernel ausgeführt, sodass der Bootloader eine Fehlermeldung anzeigen kann.

Listing B.13: Starten eines nicht vertrauenswürdigen Kernels

```
boot> bsd.mp
```

B.3.3. Nicht vertrauenswürdige UEFI-Anwendung

Um sicherzustellen, dass die Plattform nur vertrauenswürdige UEFI-Anwendungen wie *shim* ausführt, kann eine beliebige, nicht signierte UEFI-Anwendung ausgeführt werden, die zur Ausgabe einer Fehlermeldung führt.

Listing B.14: Qemu-Befehl

```
Shell> fs0:\shim-unsigned.efi
```

B.3.4. Nicht vertrauenswürdiger Bootloader

Shim lädt automatisch einen zweiten Bootloader (*loader.efi*). Um zu testen, ob *shim* dessen Signatur überprüft, wird der signierte Bootloader durch den unsignierten ersetzt, sodass *shim* eine Fehlermeldung anzeigen kann.

Listing B.15: Bootloader ersetzen

```
$ mv loader-unsigned.efi loader.efi
```

Listing B.16: Shim ausführen

```
Shell> fs0:\shim.efi
```

Glossar

ACPI Das Advanced Configuration and Power Interface (ACPI) beschreibt Schnittstellen zur Energieverwaltung.

amd64 Mit amd64 wird eine 64-Bit-Architektur bezeichnet, die auf der Intel-Architektur aufbaut und deren Befehlssatz erweitert. Sie kann mit 64-Bit breiten Worten umgehen und ist abwärtskompatibel zu i386.

BIOS Das Basic Input/Output System ist die Firmware eines herkömmlichen Computers. Es wird zunehmend durch das UEFI abgelöst.

Boot Firmware Volume (BFV) Das BFV ist die Firmware des Systems, die aus dem ROM geladen wird und alle Komponenten, die für die Bereitstellung des UEFI benötigt werden, beinhaltet.

Boot Service Boot Services sind UEFI-Schnittstellen bestehend aus Datenstrukturen mit Informationen zum System sowie Systemfunktionen, die während während des Startens des Systems zur Verfügung stehen.

Bootkit Ein Bootkit ist eine Programmsammlung, die nach einem Einbruch in ein System den Zugriff auf dieses durch Änderungen an dessen Startvorgang sicherstellt.

Bootloader Der Bootloader ist eine Software, die von der Firmware eines Systems geladen wird und das Betriebssystem startet.

CA Eine Certificate Authority (CA) ist eine Organisation, die digitale Zertifikate signiert, die dazu dienen, öffentliche Schlüssel Personen oder Organisatioenn zuzuordnen.

CSM Das UEFI Compatibility Support Module (CSM) bildet die Funktionalität eines BIOS unter UEFI nach, sodass Anwendungen, die mit UEFI nicht interagieren können, weiterhin ausgeführt werden können.

Disklabel Das Disklabel ist eine OpenBSD-spezifische Datenstruktur, die die Festplatte in Partitionen aufgeteilt.

Driver Execution Environment (DXE) In der Driver Execution Environment (DXE) Phase wird der Großteil der Systeminitialisierung durchgeführt. Sie stellt zudem die Boot und Runtime Services zur Verfügung, die von UEFI-Anwendungen verwenden werden können.

ELF Das Executable and Linkable Format (ELF) ist das Dateiformat für Anwendungen unter OpenBSD.

Firmware Als Firmware wird Software bezeichnet, die in elektronischen Geräten verbaut wurde und zumeist den Zugriff auf die Hardware abstrahiert.

FreeBSD FreeBSD ist ein BSD-basiertes Betriebssystem ähnlich zu OpenBSD.

GPT GUID Partition Table (GPT) ist als Teil des UEFI-Standards ein Standard für Partitionstabellen auf Datenträgern.

Hash Eine Hashfunktion ist eine nicht umkehrbare Funktion, die eine Zeichenfolge beliebiger Länge auf eine Zeichenfolge mit fester Länge abbildet.

Header-Datei Eine Header-Datei ist eine Textdatei, die Funktionssignaturen, Datenstrukturen, Deklarationen und andere Bestandteile des Quelltextes enthält.

i386 Mit i386 wird eine 32-Bit-Prozessorarchitektur mit entsprechenden Befehlen bezeichnet, die unter andem von Intel und AMD entwickelt wird.

Kernel Der Kernel ist der zentrale Bestandteil eines Betriebssystems. Er stellt Schnittstellen für den Zugriff auf die Hardware zur Verfügung und verwaltet Prozesse und Daten.

Key Exchange Key (KEK) Key Exchange Keys stellen eine Vertrauensbeziehung zwischen der Firmware und ausgeführten Anwendungen her.

MBR Der Master Boot Record ist der erste Sektor eines in Partitionen aufteilbaren Speichermediums wie beispielsweise einer Festplatte. Der MBR enthält eine Partitionstabelle, die die Aufteilung des Datenträgers beschreibt sowie einen eingeschränkten Bootloader.

Measured Boot Measured Boot ist ein Sicherheitsmechanismus, der versucht, Systeme, die manipuliert wurden, zu erkennen.

OpenBSD Das OpenBSD-Projekt erstellt ein freies, auf dem 4.4-BSD-Kernel basierende Unix-ähnliches Betriebssystem. Sein Fokus ist Portabilität, Standardkonformität, Korrektheit, Sicherheit und Kryptographie.

Option-ROM Ein Option-ROM ist Firmware, die vom BIOS aufgerufen wird und der Anbindung weiterer Hardware ans System dient.

PCR Messungen beim Measured Boot werden in einem sicheren Speicherbereich, dem Platform Configuration Register (PCR), abgelegt.

PE/COFF Portable Executable (PE) und Common Object File Format (COFF) bezeichnet ein Dateiformat, das unter anderem für UEFI-Anwendungen verwendet wird.

PEI-Phase Die Pre EFI Initialization Phase (PEI) initialisiert das System, sodass die Driver Execution Environment (DXE) Phase übernehmen kann. Sie besteht aus einem Kern, der PEI-Foundation, die mittels Pre-EFI Initialization Modules (PEIMs) Hardware initialisieren kann.

PI UEFI wird durch die Platform Initialization (PI) Spezifikationen dahingehend ergänzt, dass diese die Komponenten und Abläufe, die für die Bereitstellung der UEFI-Umgebung benötigt werden, definieren.

Platform Key (PK) Der Platform Key ist der öffentliche Teil eines asymmetrischen Schlüsselpaars, der eine Vertrauensbeziehung zwischen dem Hersteller der Plattform und der Firmware der Plattform herstellt.

POST Der Power-On Self-Test (POST) ist eine Systemüberprüfung, die vom BIOS beim Starten eines Systems durchgeführt wird.

Power-On Unter dem Power-On wird das erstmalige Einschalten eines Systems verstanden.

Pre-EFI Initialization Modules (PEIMs) PEIMs sind Softwarekomponenten, die in der PEI-Phase zur Initialisierung der Hardware genutzt werden.

Reset Vector Der Reset Vector ist die Standard-Speicherstelle, aus der die CPU ihre erste Anweisung nach einem Reset, also dem Zurücksetzen auf den Ursprungszustand, bezieht.

ROM Read-Only Memory ist ein nicht-schreibbarer Speicherbereich.

Root of Trust Der Root of Trust stellt den Ausgangspunkt eines sicheren Bootprozesses dar.

Rootkit Ein Rootkit ist Programmsammlung, die nach einem Einbruch in ein System den Zugriff auf dieses durch Änderungen am Betriebssystem oder den Systemprogrammen sicherstellt.

Runtime Service Runtime Services sind UEFI-Schnittstellen bestehend aus Datenstrukturen mit Informationen zum System sowie Systemfunktionen, die während der Laufzeit des Systems zur Verfügung stehen.

SEC-Phase Nach dem Einschalten befindet sich das System in der Security (SEC) Phase. Diese lädt das Boot Firmware Volume (BFV) aus einem nicht-beschreibbaren Speicherbereich (ROM) in einen temporären Speicherbereich der CPU und führt es aus. Die SEC-Phase wird nur nach einem initialen Power-On ausgeführt; nach einem Neustart wird sofort in die PEI-Phase übergegangen.

shim Shim ist ein UEFI Secure Boot kompatibler Bootloader, der andere UEFI-Anwendungen überprüfen und ausführen kann.

Signatur-Datenbank Die UEFI-Signatur-Datenbanken verwalten zwei unterschiedliche Arten von Signaturen: Signaturen für Programmcode, der ausgeführt werden darf (db) sowie Signaturen für Programmcode, dessen Ausführung verboten ist (dbx).

TPM Ein Trusted Platform Module (TPM) ist ein Chip, der Sicherheitsfunktionen bereitstellt.

Trusted Computing Group Die Trusted Computing Group (TCG) ist eine Organisation, die Standards zum Absichern von Systemen festlegt. Sie hat hierfür Funktionen von TPM-Chips sowie Mechanismen zum Absichern und Überprüfen von Startvorgängen definiert.

UEFI Das Unified Extensible Firmware Interface ist eine standardisierte Beschreibung von Schnittstellen für eine Firmware. Sie vereinheitlicht und abstrahiert den Zugriff auf die Hardware für Komponenten, die vor dem Starten eines Betriebssystems ausgeführt werden.

UEFI Authenticated Variables Bei UEFI Authenticated Variables handelt es sich um UEFI-Variablen, die besonders abgesichert sind und in denen die Firmware und Anwendungen sicherheitsrelevante Informationen hinterlegen können.

UEFI Boot Manager Der UEFI-Boot-Manager ist eine UEFI-Anwendung, die in der UEFI-Plattform enthalten ist und das Laden von UEFI-Images ermöglicht.

UEFI Secure Boot Secure Boot ist ein Sicherheitsmechanismus, der beim Starten eines Systems ausgehend von einem Root of Trust die Integrität und Authentizität der jeweils nachfolgend ausgeführten Komponente sicherstellt.

UEFI-Image UEFI-Images sind UEFI-Treiber und -Anwendungen, die ausführbaren Programmcode enthalten und von der UEFI-Plattform geladen werden können.

UEFI-Shell Die UEFI-Shell ist eine UEFI-Anwendung, die interaktiv eingebaute Befehle und weitere UEFI-Anwendungen zur Laufzeit der UEFI-Firmware ausführen kann.

UEFI-Variable Eine UEFI-Variable ist ein Speicherbereich der UEFI-Firmware, in dem persistent Daten zum System abgelegt werden können.

UFS Das Unix file system (UFS), auch als Fast File System (FFS) bekannt, ist das Standard-Dateisystem unter OpenBSD.

Unix Unix ist ursprünglich ein Mehrbenutzer-Betriebssystem. Es steht heute stellvertretend für Betriebssysteme, die ihren Ursprung im Unixsystem von AT&T haben oder dessen Konzepte implementieren.

x86 vgl. i386

x86_64 vgl. amd64

Literaturverzeichnis

[BaCr 12] Barry, Peter und Patrick Crowley: *Modern Embedded Computing - Designing Connected, Pervasive, Media-Rich Systems*. Morgan Kaufmann Publishers, 2012.

[BECN+ 06] Bar-El, Hagai, Hamid Choukri, David Naccache, Michael Tunstall und Claire Whelan: *The Sorcerer's Apprentice Guide to Fault Attacks*. Proceedings of the IEEE, 94(2):370 – 382, Februar 2006.

[BFB 13] Bulygin, Yuriy, Andrew Furtak und Oleksandr Bazhaniuk: *A Tale of One Software Bypass of Windows 8 Secure Boot*, Juli 2013, https://media.blackhat.com/us-13/us-13-Bulygin-A-Tale-of-One-Software-Bypass-of-Windows-8-Secure-Boot-Slides.pdf .

[BFJ+ 13] Boyer, Josh, Kevin Fenzi, Peter Jones, John Bressers und Florian Weimer: *UEFI Secure Boot Guide*, 2013, https://docs.fedoraproject.org/en-US/Fedora/18/html-single/UEFI_Secure_Boot_Guide/index.html .

[Bund 92] BUNDESAMT FÜR SICHERHEIT IN DER INFORMATIONSTECHNIK (BSI): *IT-Sicherheitshandbuch - Handbuch für die sichere Anwendung der Informationstechnik*. Version 1.0. Bundesdruckerei, März 1992, https://web.archive.org/web/*/http://www.bsi.bund.de/literat/sichhandbuch/sichhandbuch.zip .

[CoMo 14] Cooper, Chris und Chris G. Moore: *HP-UX 11i Internals*. Prentice Hall Professional, 2014.

[CPRS 11] Cooper, David, William Polk, Andrew Regenscheid und Murugiah Souppaya: *BIOS Protection Guidelines*, April 2011, http://csrc.nist.gov/publications/nistpubs/800-147/NIST-SP800-147-April2011.pdf .

[EaDo 14] East, Jacquelynn und Don Domingo: *Red Hat Enterprise Linux Developer Guide: Debugging*, Juni 2014, https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Developer_Guide/debugging.html .

[Ecke 13] Eckert, Claudia: *IT-Sicherheit: Konzepte - Verfahren - Protokolle*. Berlin, Boston: Oldenbourg Wissenschaftsverlag, 978-3486721386, 8. Auflage, 2013, http://www.degruyter.com/view/product/226493 .

[Garr 13a] Garrett, Matthew: *Linux Foundation Secure Boot support released - what does it mean?*, Februar 2013, http://mjg59.dreamwidth.org/23113.html .

[Garr 13b] Garrett, Matthew: *Using pstore to debug awkward kernel crashes*, März 2013, http://mjg59.dreamwidth.org/23554.html .

[KKS+ 08] Kurth, H., G. Krummeck, C. Stüble, M. Weber und M Winandy: *HASK-PP: Protection Profile for a High Assurance Security Kernel*, Juni 2008, https://web.archive.org/web/*/www.sirrix.com/media/downloads/54500.pdf .

[Leem 14] Leemhuis, Thorsten: *UEFI: Linux kann aktuelle Thinkpads beschädigen*, Februar 2014, http://www.heise.de/open/artikel/UEFI-Linux-kann-aktuelle-Thinkpads-beschaedigen-2105920.html .

[McCo 04] McConnell, Steve: *Code Complete*. Microsoft Press, 978-0735619678, 2. Auflage, Juli 2004.

[Micr 12] MICROSOFT, INC.: *Secured Boot and Measured Boot: Hardening Early Boot Components Against Malware*, September 2012, http://msdn.microsoft.com/en-us/library/windows/hardware/br259097.aspx .

[Micr 13a] MICROSOFT, INC.: *Microsoft Portable Executable and Common Object File Format Specification*, Februar 2013, http://msdn.microsoft.com/en-us/library/windows/hardware/gg463119.aspx .

[Micr 13b] MICROSOFT, INC.: *Secure Boot*, November 2013, http://technet.microsoft.com/en-us/library/dn486875.aspx .

[Micr 14] MICROSOFT, INC.: *Windows Hardware Certification Requirements for Client and Server Systems*, Januar 2014, http://msdn.microsoft.com/en-us/library/windows/hardware/jj128256 .

[Nati 07] NATIONAL SECURITY AGENCY (NSA): *U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness*, Juni 2007, http://www.niap-ccevs.org/pp/pp_skpp_hr_v1.03.pdf .

[NetM 14] NetMarketShare: *Desktop Operating System Market Share*, Mai 2014, http://www.netmarketshare.com/operating-system-market-share.aspx?qprid=8&qpcustomd=0 .

[Nieh 13] Niehus, Oliver: *Windows 8: Trusted Boot: Secure Boot – Measured Boot*, Januar 2013, http://blogs.msdn.com/b/olivnie/archive/2013/01/09/windows-8-trusted-boot-secure-boot-measured-boot.aspx .

[Open 14a] OPENBSD PROJEKT: *OpenBSD Copyright Policy*, Mai 2014, http://www.openbsd.org/policy.html .

[Open 14b] OPENBSD PROJEKT: *OpenBSD Frequently Asked Questions: Disk Setup*, Mai 2014, http://www.openbsd.org/faq/faq14.html#LargeDrive .

[Open 14c] OPENBSD PROJEKT: *OpenBSD Manual Page: boot_amd64 - amd64 system bootstrapping procedures*, Mai 2014, http://www.openbsd.org/cgi-bin/man.cgi?query=boot_amd64&sektion=8&arch=amd64&manpath=OpenBSD+5.5 .

[Open 14d] OPENBSD PROJEKT: *OpenBSD Manual Page: elf - format of ELF executable binary files*, Mai 2014, http://www.openbsd.org/cgi-bin/man.cgi?query=elf&sektion=5&manpath=OpenBSD+5.5&arch=amd64 .

[Orac 12a] ORACLE: *Oracle Solaris 11.1 Linkers and Libraries Guide: ELF Application Binary Interface*, Oktober 2012, http://docs.oracle.com/cd/E26502_01/html/E26507/glcfv.html .

[Orac 12b] ORACLE: *Transitioning From Oracle Solaris 10 to Oracle Solaris 11: Oracle Solaris 11 File System Changes*, März 2012, http://docs.oracle.com/cd/E23824_01/html/E24456/filesystem-10.html .

[RLZH 10] Rothman, M., T. Lewis, V. Zimmer und R. Hale: *Harnessing the UEFI Shell: Moving the Platform Beyond DOS*. Intel Press, 1. Auflage, 2010, https://noggin.intel.com/intelpress/categories/books/harnessing-uefi-shell .

[Schm 13] Schmeh, Klaus: *Kryptografie: Verfahren, Protokolle, Infrastrukturen*. dpunkt.verlag, 978-3864900150, 3. Auflage, Februar 2013.

[Sino 11] Sinofsky, Steven: *Protecting the pre-OS environment with UEFI*, September 2011, http://blogs.msdn.com/b/b8/archive/2011/09/22/protecting-the-pre-os-environment-with-uefi.aspx .

[SUSE 14] SUSE LINUX GMBH: *Administration Guide: Secure Boot*, April 2014, https://www.suse.com/documentation/sles11/book_sle_admin/data/sec_uefi_secboot.html .

[Trus 06] TRUSTED COMPUTING GROUP (TCG): *TCG EFI Platform Specification*, Juni 2006, https://www.trustedcomputinggroup.org/files/temp/644EE853-1D09-3519-ADB133C09A5F2BDF/TCG EFI Platform Specification.pdf .

[Trus 11] TRUSTED COMPUTING GROUP (TCG): *TPM Main Part 1 Design Principles*, März 2011, https://www.trustedcomputinggroup.org/files/static_page_files/72C26AB5-1A4B-B294-D002BC0B8C062FF6/TPM Main-Part 1 Design Principles_v1.2_rev116_01032011.pdf .

[Unif 13a] UNIFIED EXTENSIBLE FIRMWARE INTERFACE (UEFI) FORUM: *Platform Initialization Specification*, März 2013, http://www.uefi.org/specs/access .

[Unif 13b] UNIFIED EXTENSIBLE FIRMWARE INTERFACE (UEFI) FORUM: *Unified Extensible Firmware Interface Specification*, Juni 2013, http://www.uefi.org/specs/access .

[Wiet 12] Wietzke, Joachim: *Embedded Technologies: Vom Treiber bis zur Grafik-Anbindung*. Springer, 2012, http://books.google.de/books?id=r8qipwAACAAJ .

[WiRi 13] Wilkins, Richard und Brian Richardson: *UEFI Secure Boot in Modern Computer Security Solutions 2013*, 2013, http://www.uefi.org/sites/default/files/resources/UEFI_Secure_Boot_in_Modern_Computer_Security_Solutions_2013.pdf .

[ZRM 10] Zimmer, Vincent, Michael Rothman und Suresh Marisetty: *Beyond BIOS: Developing with the Unified Extensible Firmware Interface*. Intel Press, 978-1-934053-29-4, 2. Auflage, 2010, http://noggin.intel.com/intelpress/categories/books/beyond-bios-2nd-edition .