TU WIEN Informatics

TPM 2.0 als Sicherheitsmaßnahme gegen Rootkits auf Linux-basierten Desktop-Systemen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Jasmin Marmsoler, BSc
Matrikelnummer 01426211

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Thomas Grechenig
Mitwirkung: Florian Fankhauser

Wien, 1. März 2021

Unterschrift Verfasserin

Unterschrift Betreuung

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

TPM 2.0 as a Security Measure against Rootkits on Linux Based Desktop Systems

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Software Engineering and Internet Computing

by

Jasmin Marmsoler, BSc
Registration Number 01426211

to the Faculty of Informatics

at the TU Wien

Advisor: Thomas Grechenig
Assistance: Florian Fankhauser

Vienna, 1st March, 2021

Signature Author

Signature Advisor

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TPM 2.0 als Sicherheitsmaßnahme gegen Rootkits auf Linux-basierten Desktop-Systemen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Jasmin Marmsoler, BSc
Matrikelnummer 01426211

ausgeführt am
Institut für Information Systems Engineering
Forschungsbereich Business Informatics
Forschungsgruppe Industrielle Software
der Fakultät für Informatik der Technischen Universität Wien

Betreuung: Thomas Grechenig

Wien, 1. März 2021

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Jasmin Marmsoler, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. März 2021

Jasmin Marmsoler

Kurzfassung

Rootkits sind Schadsoftware, die Komponenten befallen, welche tiefer als das Betriebssystem liegen, wie etwa den Kernel, den Bootloader oder das BIOS. Sie agieren mit erhöhten Rechten und sind daher schwer von Betriebssystemebene aus zu erkennen. Für ein vertrauenswürdiges System reicht es nicht aus, die letzte Komponente des Bootprozesses vor Kompromittierungen zu schützen. Es muss eine sichere Basis geben, von welcher eine Vertrauenskette bis zum Betriebssystem aufgebaut wird. Dabei wird von jeder Komponente des Bootprozesses vor ihrer Ausführung der Hashwert berechnet und sicher gespeichert. Diesen Prozess nennt man auch Measured Boot. Wenn nur valide Komponenten ausgeführt werden, spricht man von Secure Boot.
Das Trusted Platform Module (TPM) ist ein kryptographischer Mikrocontroller, welcher die Messwerte des Bootprozesses sicher speichert und ihre Integrität beglaubigen kann. Dadurch können Anwendende aber auch Drittsysteme den Systemzustand überprüfen und sicherstellen, dass der Bootprozess nicht kompromittiert wurde. Das TPM ist ein passives Modul, welches von anderen Komponenten und Programmen aufgerufen wird.
Diese Arbeit beschreibt ein Konzept, um die Ausführung von Rootkits, durch einen sicheren Bootprozess mit TPM 2.0, auf Linux basierten Desktop-Systemen zu verhindern. Das Konzept ist eine Hybridlösung zwischen Secure Boot und Measured Boot. Dabei soll das gewünschte Aktualisieren von Bootkomponenten die Vertrauenswürdigkeit des Systems nicht brechen.

Keywords: *TPM 2.0, Rootkits, Secure Boot, Measured Boot, Linux Bootprozess*

Abstract

Rootkits are a kind of malware that compromise components lower than the operating system for example the kernel, bootloader and BIOS. Rootkits pose a major threat to computer security as they operate with elevated privileges and are often hard to detect from the operating system level. A computer system requires a secure basis and a chain of trust on all levels up to the operating system to increase safety. To achieve this every component of the boot process is measured before being loaded and executed, this method is also known as measured boot. Secure Boot on the other side executes only components with a valid signature or a valid hash.
The Trusted Platform Module (TPM) is a cryptographic microcontroller located on the computer's motherboard. It securely stores the measurements of the boot process and can attest to the component's integrity. This also means that not only users but also remote entities can check the system state. The TPM is a passive module which is called by other components and software.
This thesis describes a concept to prevent the execution of rootkits on Linux-based desktop systems through a boot process with TPM 2.0. The concept is a combination of a secure and measured boot in which updating of components should not break the attestation or the trustworthiness of the system.

Keywords: *TPM 2.0, Rootkits, Secure Boot, Measured Boot, Linux boot process*

Inhaltsverzeichnis

KAPITEL 1

Einleitung

Die vorliegende Diplomarbeit befasst sich mit Rootkits, Trusted Platform Module (TPM) 2.0 und sicherem Booten. In diesem Kapitel wird die Problemstellung, die Motivation, das Ziel und der Aufbau der Arbeit beschrieben.

1.1 Problemstellung

In den vergangenen zehn Jahren ist die Anzahl von Malware (Malicious Software) stark angestiegen. Laut dem Institut AV-TEST werden in Summe jeden Tag mehr als 350.000 neue Malware und potenziell unerwünschte Anwendungen (PUAs) registriert [68]. Rootkits wurden von AV-TEST zwar registriert, waren jedoch aufgrund der geringen Anzahl statistisch nicht relevant [67]. Laut Kuzminykh u. a. [82] gab es in den vergangenen Jahren immer mehr Angriffe mit komplexeren und fortgeschritteneren Rootkits.
Während Rootkits laut Hoglund u. a. [59] sich Systemadministratorrechte erarbeiten, ist Pelaez [106] der Meinung, dass sie die Rechte lediglich aufrechterhalten beziehungsweise wiedererlangen. Beide sind sich jedoch einig, dass Rootkits Angreifenden dazu dienen, unentdeckt zu bleiben, um das System für längere Zeit zu infiltrieren. Dabei werden mehrere kleine Programme genutzt, um zu verstecken, dass das System kompromittiert wurde und um eventuell eine andere Schadsoftware einzuschleusen.
Anfangs haben Rootkits, so Kruegel u. a. [80], nur Systemüberwachungsprogramme manipuliert, um unerkannt zu bleiben. Doch heutzutage befallen sie auch den Kernel und das BIOS/UEFI (Bootkits). Sobald der Kernel infiziert ist, ist es sehr schwer ohne Hardwareerweiterungen die Kompromittierung zu erkennen. Um einem Computersystem vertrauen zu können, ist laut Garrett [47] eine vertrauenswürdige Basis und ein sicherer Bootprozess essenziell. Dabei reicht es nicht aus, dass der Kernel seine Integrität sichern kann, aber der Bootprozess manipulierbar ist.
Ein TPM bildet häufig die hardwarebasierte Wurzel (Root of Trust) einer Vertrauenskette (Chain of Trust) und kann helfen, das System abzusichern. TPMs sind von der Trusted

Computing Group (TCG) [122] spezifizierte kryptographische Mikrocontroller, vergleichbar mit einer Smart Card. Sie befinden sich am Mainboard und sind in den meisten Computern bereits eingebaut. Ein TPM ist, so Arthur u. a. [8], eine passive Komponente, die von anderen aufgerufen wird. Sie hat viele Funktionen, unter anderem das Generieren und Speichern von kryptographischen Schlüsseln, aber auch die Korrektheit des Systems zu beglaubigen. Um zu kontrollieren, ob das System kompromittiert wurde, werden beim Bootprozess Hashwerte der einzelnen Komponenten berechnet und in den Platform Configuration Register (PCR) gespeichert. Diese Hashwerte werden dann, entweder von einem externen System oder von einer lokalen Software, mit bestehenden vertrauenswürdigen Werten verglichen. Bei Unstimmigkeiten können verschiedenste Maßnahmen getroffen werden, zum Beispiel kann die Entschlüsselung der Festplatte oder der Zugang zu einem bestimmten Dienst verweigert werden.
TPM 2.0 bietet zahlreiche Verbesserungen gegenüber seinem Vorgänger, zum Beispiel schnelleres Laden von Schlüsseln und Flexibilität bei der Wahl des kryptographischen Algorithmus [8]. Doch TPM 2.0 ist, so Król [79], nicht kompatibel mit dem Vorgänger TPM 1.2. Dadurch kann TPM-Software für 1.2 nicht mit TPM 2.0 verwendet werden. Die Entwicklung der neuen TPM2-Software ist sehr schnell, sodass Linux Distributionen nicht immer auf dem aktuellsten Stand sind. Seit 2016 setzt Microsoft [94] jedoch TPM 2.0 als Hardwarekomponente für Windows 10 Desktop-Systeme voraus.
Sowohl Sicherheitsupdates als auch Systemintegrität verfolgen, so Lee u. a. [85] dasselbe Ziel, die Sicherheit des Systems zu gewährleisten. Jedoch stehen sie im Widerspruch zueinander. Das Updaten einer Komponente würde ihre Integrität brechen. Dieses Problem sprechen sowohl Garrett, in seiner Präsentation [47] beim Linux Security Summit, als auch TCG in ihrer TPM 2.0 Spezifikation [122] an. Wenn Komponenten wie etwa das BIOS aktualisiert werden, verändern sich die entsprechenden PCR-Werte. Daten, die an diese Register gebunden sind, sind nicht mehr zugänglich und ein Wiederherstellungsprozess ist erforderlich [122]. Eine genaue Beschreibung von PCR-Werten und das Binden von Zugriffsrechten sind in den Abschnitten 5.2, 5.3 zu finden.
Bestehende Lösungen wie UEFI Secure Boot sind nicht auf das TPM angewiesen, können aber mit diesem kombiniert werden. Ohne TPM lässt UEFI Secure Boot Nutzenden die Freiheit sowohl Betriebssystem als auch Firmwareupdates durchzuführen. Jedoch ist der Root of Trust, so Matrosov u. a. [91], nicht in der Hardware, sondern in einer Firmwarekomponente verankert. Diese kann manipuliert werden und den ganzen Bootprozess kompromittieren. Um sicherzustellen, dass die Unified Extensible Firmware Interface Spezification (UEFI) Firmware nicht manipuliert wurde, muss die Chain of Trust auf Hardwareebene erweitert oder dort verankert werden (siehe Abschnitt 6.4). Intel Trusted Execution Technology (Intel TXT) [72] ist eine Intel spezifische Technologie den Bootprozess abzusichern. Dabei wird die Root of Trust in der CPU verankert. Intel TXT unterstützt zwar TPM 2.0, wird jedoch nur von bestimmten Intel CPUs unterstützt. Da diese Technologie nicht für alle Systeme anwendbar ist, fokussiert sich diese Arbeit auf eine Alternative, um den Bootprozess abzusichern und eine Chain of Trust aufzubauen.

1.2 Motivation

Rootkits sind leise Parasiten, die sich in ein System einnisten und versuchen so lange wie möglich unerkannt zu bleiben und mit möglichst hohen Rechten zu arbeiten. Sie agieren auf den verschiedensten Ebenen. Vom BIOS bis zum Betriebssystem sind sie zu finden. Umso tiefer sie sich einnisten, umso schwieriger ist es sie zu erkennen und zu entfernen. Daher ist der beste Weg sie zu bekämpfen, eine Infektion beziehungsweise die erste Ausführung zu verhindern. Um die Ausführung von Rootkits zu verhindern, muss der Bootprozess abgesichert werden und eine Chain of Trust von der Hardware bis zum Betriebssystem aufgebaut werden [47]. Dabei misst jede Komponente die Nächste, bevor diese ausgeführt wird.

1.3 Zielsetzung

Um den Bootprozess abzusichern, müssen mehrere Komponenten des Bootprozesses miteinander kombiniert werden. Es gibt bereits zahlreiche Komponenten die helfen, den Linux Bootprozess abzusichern. Diverse Lösungsansätze werden in Abschnitt 6 behandelt. Einige davon sind bereits kompatibel mit TPM 2.0. Ziel dieser Arbeit ist es, ein Konzept für einen sicheren Linux Bootprozess für Desktop-Systeme unter Verwendung von TPM 2.0 zu entwickeln und dieses durch ein praktisches Experiment zu überprüfen. Dabei soll das gezielte Updaten von Bootkomponenten nicht die Vertrauenswürdigkeit des Systems brechen. Zu Beginn müssen verschiedene Linux Bootkomponenten, welche TPM 2.0 unterstützen, ermittelt werden. Anschließend müssen einige davon miteinander kombiniert werden, um den kompletten Bootprozess abzusichern.
Das TPM ist eine passive Hardwarekomponente, welche die Messwerte lediglich sicher speichert und wiedergibt. Eine andere Komponente oder Software muss diese auswerten und bei Inkonsistenzen im Bootprozess eingreifen [73]. Die Werte können entweder lokal überprüft oder an einen Server (Remote Attestation) geschickt werden. Da es in dieser Arbeit um Desktop-Systeme geht, wird der Fokus auf eine lokale Überprüfung gelegt.
Diese Arbeit soll folgende Forschungsfragen klären: Wie kann ein sicherer Bootprozess bei Endnutzenden umgesetzt werden, sodass immer noch Updates installiert werden können? Welchen Mehraufwand zieht das für Anwendende mit sich? Gibt es unter Linux ausreichend Software-Unterstützung, um TPM 2.0 effizient einzusetzen?

1.4 Aufbau der Arbeit

In Kapitel 3 werden allgemeine Grundlagen, für ein besseres Verständnis der Arbeit, erörtert. Insbesondere wird dabei auf den Linux Bootprozess und Trusted Computing eingegangen. Die möglichen Angriffspunkte und Rootkitsarten werden in Abschnitt 4 behandelt. Dabei werden zunächst Rootkits und Bootkits genau definiert. Anschließend wird auf die einzelnen kompromittierbaren Komponenten anhand theoretischer Beschreibungen und Beispielfällen eingegangen.
Das Kapitel 5 beinhaltet die TPM 2.0 relevanten Grundlagen. Es wird der allgemei-

ne Aufbau und die Funktionsweise sowie die wichtigsten Konzepte erklärt. Einige der bekanntesten Schutzmechanismen sind in Abschnitt 6 zu finden. Diese bestehenden Lösungsansätze werden genau studiert und gegebenenfalls versucht in das Konzept zu integrieren.
Das erarbeitete Konzept und die prototypische Umsetzung mit TPM 2.0 befinden sich in Abschnitt 7. Dabei werden Empfehlungen gemacht, den Bootprozess größtmöglich abzusichern und gleichzeitig den Aufwand für Anwendende so gering wie möglich zu halten. Das regelmäßige Aktualisieren von Komponenten sollte dabei weiterhin möglich sein.
Kapitel 8 analysiert das Konzept und die prototypische Umsetzung und zeigt Limitierungen dieser auf. Abschließend werden in Kapitel 9 die wichtigsten Punkte der Arbeit nochmals zusammengefasst und ein Ausblick für zukünftige Arbeiten gegeben.

KAPITEL 2

Related Work

Es gibt verschiedene Ansätze Rootkits zu erkennen und die Sicherheit des Systems umzusetzen. Die Basis dafür bildet ein sicherer Bootprozess. Sailer u. a. [113] entwickelten 2004 die Open Source Integrity Measurement Architecture (IMA) für Linux. Dabei wird jede Komponente vor ihrer Ausführung gemessen und mit den Werten im TPM verglichen. Unter dem Begriff „Messen“ versteht man die Berechnung eines kryptographischen Hashwertes. Sailer u. a. waren die Ersten, die das TCG Trust-Measurement-Konzept auf die Applikations-Ebene erweitern. Dadurch ermöglicht IMA es zur Ladezeit, die Integrität des Systems und der laufenden Programme zu überprüfen und anderen gegenüber zu beglaubigen (Remote Attestation). Wang u. a. [141] erweiterten IMA, um TPM 2.0 zu unterstützen.
Laut Jaeger u. a. [74] weist IMA jedoch zwei Probleme auf. Erstens reicht es nicht aus, den Code lediglich beim Laden zu überprüfen, da es das Laufzeitverhalten nicht genau widerspiegelt. Zweitens ist es ineffektiv alle Programme zu messen und mit einem vertrauenswürdigen Wert zu vergleichen, wenn diese keine Auswirkungen auf das System haben. Jaeger u. a. [74] fokussieren sich in ihrer Arbeit „Policy-Reduced Integrity Measurement Architecture (PRIMA)“ auf die Informationsfluss-Integrität. Durch die Kopplung von IMA mit der SELinux Policy konnten sie die Anzahl der zu messenden Komponenten reduzieren.
Um eine Messung vertrauenswürdig zu machen, muss der Integrity Measurment Mechanism (IMM) geschützt werden. Wei u. a. [144] erweitern die beiden Ausführungsmodi Usermode und Kernelmode (siehe Kapitel 3.3.1) um einen dritten Modus, den Trustmode, welcher über dem Kernelmode steht. In diesem Modus wird der Core Measurement Mechanism (CMM) ausgeführt. Dieser misst den IMM. Auch wenn der Kernel kompromittiert wurde, kann der CMM richtig arbeiten. Auch Jia u. a. [75] gehen in ihrer Arbeit auf die Manipulation beziehungsweise Umgehung des Messprozesses ein. Dabei unterteilen sie das System in zwei Welten: einmal in die „normal computation world“ und einmal in die „isolated security world“. In der Sicherheitswelt befinden sich alle sicherheitsrelevanten

Aufgaben. Auf die Daten der isolierten Ebene kann die normale Ausführungsebene nicht zugreifen.
Die Überwachung vom Kernel zu isolieren ist auch der Ansatz von Seshadri u. a. [115]. SecVisor ist ein Hypervisor, der Kernel-Code-Integrität gewährleistet. SecVisor läuft mit Virtual Machine Monitor (VMM) Rechten und ist somit höher als Kernel-Rechte. Der Prototyp Nixer von Grimm u. a. [52] läuft ebenfalls auf Hypervisor Level außerhalb der Reichweite von Rootkits. In ihrer Arbeit gehen sie auf die Verhinderung von Kernel Rootkits in Cloud-Umgebungen ein. Dafür legen sie einen Referenzwert für jede Komponente fest, die sich nicht mehr ändert, wie zum Beispiel Kernel Code oder System-Call-Tables, und vergleichen diese kontinuierlich mit dem aktuellen Stand.
Tian u. a. [131] verwenden Virtualisierung und Machine Learning, um Kernel Rootkits zu erkennen. Dabei führen sie die Kernel-Module in einer emulierten Umgebung aus und untersuchen das Laufzeitverhalten. Um die Kernel-Modul Interaktionen untersuchen zu können, wenden sie eine hardwareunterstützte Virtualisierungstechnologie an. Der Hypervisor schreibt alle Operationen, die auf Speicher- oder Hardwareregister zugreifen in die Logfiles. Diese werden dann regelmäßig analysiert.
Systemupdates stellen bei Remote Attestation eine große Herausforderung dar. Trust Update on Linux Booting (Tux) von Lee u. a. [85] erweitert das Integritätsmanagement System von Intel OpenCIT, um einen transparenteren Updateprozess gegenüber dem Server zu gewährleisten. Des Weiteren entwickelten sie TS-Boot, einen robusteren Bootprozess. TS-Boot ist eine Kombination von UEFI Secure Boot, Shim- und TrustedGRUB Bootloader (Genauere Beschreibungen dieser Komponenten befinden sich in den Abschnitten 6.3, 3.2.1, 6.2). Der Ansatz von Lee und Yoo erlaubt zwar häufige Updates ohne die Remote Attestation zu brechen, jedoch ist dieser Ansatz aufgrund der Remote Attestation nicht für einzelne Desktop-Systeme geeignet.
Eine beliebte Lösung für Server, welche auch für bestimmte Desktop-Systeme eingesetzt werden kann ist Intel TXT. Intel TXT ist eine Erweiterung für Intel Chips und verwendet eine Kombination aus Static Root of Trust for Measurement (S-RTM) und Dynamic Root of Trust for Measurement (D-RTM). Der statische Teil misst die Plattformkonfigurationen und der dynamische Teil die Software inklusive Konfigurationen und Policys. Die Ergebnisse werden in den PCRs des TPM gespeichert [40]. tboot ist ein Open Source Pre-Kernel Modul für Linux, welches basierend auf Intel TXT einen Dynamic Chain of Trust aufbaut. Diese kann im Gegensatz zur statischen Chain of Trust auch zu Systemlaufzeiten neu gemessen und aufgebaut werden, wenn das Betriebssystem in einen Vertrauensmodus wechseln möchte. Die Werte können entweder an eine dritte Stelle zur Beglaubigung geschickt werden, oder auch lokal mittels Policys überprüft werden. Die Launch Control Policy (LCP) beinhaltet eine Liste vertrauenswürdiger Werte. Anfangs ist die LCP mit der Hersteller-Policy ausgestattet, diese kann aber von Anwendenden mit einer eigenen Policy erweitert werden [72]. Dadurch können Anwendende zum Beispiel: einen beliebigen Kernel installieren, solange die LCP erweitert wird. Die PCR-Werte können aber auch zur Überprüfung an einen remoten Server, wie zum Beispiel Intel OpenCIT, geschickt werden. Da tboot als Bootloader Modul fungiert ist es, so Sharkey [118], anfällig für Rootkit-Angriffe. In den vergangenen Jahren gab es einige Angriffe auf tboot und Intel

TXT [150, 147, 148, 57].
Die gesamte Chain of Trust baut auf den ersten ausgeführten Code der Boot-Firmware auf. Dieser misst die erste Komponente und wird Static Core Root of Trust for Measurement (S-CRTM) genannt. Einige Plattformen bieten einen unveränderbaren, hardwaregeschützten S-CRTM, welcher das Basic Input/Output System (BIOS) vor seiner Ausführung überprüft. Intel Boot Gurad [100] und HP Sure Start [61] sind Beispiele dafür. Diese Technologien sind jedoch nur in den neusten Intel und HP-Plattformen integriert. Chevalier u. a. [23] entwickelten mit BootKeeper einen Ansatz den Messprozess der Boot-Firmware zu überprüfen ohne die Notwendigkeit eines hardwaregeschützten S-CRTMs.
Auch für TPM 2.0 wurden bereits Sicherheitslücken gefunden. Han u. a. [57] konnten aufgrund einer inkorrekten Spezifikation die PCR-Werte manipulieren und so einen vertrauenswürdigen Systemzustand vortäuschen. Moghimi u. a. [96] wenden eine Seitenkanalattacke an, um die privaten Schlüssel zu ermitteln, welche für Signaturen mittels elliptischer Kurven verwendet werden. Beide Schwachstellen und Angriffsszenarien sind genauer in Abschnitt 5.9 beschrieben.
Hardware TPMs haben, so Kim u. a. [76], eine sehr niedrige Leistung und können dadurch kryptographische Operationen nur langsam verarbeiten. In ihrer Arbeit präsentieren sie eine Hybridlösung aus Hardware TPM und Software TPM. Dabei können Anwendende den Modus an ihre Bedürfnisse anpassen. Auch Yang u. a. [151] beachten in ihrer Arbeit die beschränkte Leistungsfähigkeit von TPMs. Sie optimieren den Direct Anonymous Attestation (DAA)-Signaturprozess (siehe Abschnitt 5.4).
Wie relevant die Thematik eines sicheren Bootprozesses ist, zeigt auch die 2021 erschienene Untersuchung von Hagl u. a. [55]. Diese betrachtet die Handhabung von UEFI Secure Boot in Kombination mit einer vollständigen Festplattenverschlüsselung für fünf Linux-Distributionen. Dadurch bieten sie einen aktuellen Überblick, welche Funktionalitäten in den verschiedenen Distributionen zur Verfügung stehen. Des Weiteren liefern sie einen kurzen Einblick in die praktische Umsetzung ihres Konzeptes. Hagl u.a. verwenden für die Verschlüsselung der Festplatte dieselbe Software (LUKS) wie in dieser Arbeit eingesetzt wird. Während sie in ihrer Untersuchung den Fokus auf den Vergleich der Distributionen legen und auf die praktische Umsetzung nur kurz eingehen, behandelt diese Arbeit eine Distribution im Detail 7.1. Des Weiteren wird in dieser Arbeit TPM 2.0 als zusätzliche Sicherheitsmaßnahme eingesetzt, um einen Hardware-basierte Root of Trust (RoT) zu schaffen.
Eine wichtige Funktionalität von TPM ist die Beglaubigung des Systemzustandes und der Identität. Wagner u. a. [140] verwenden die „Remote Attestation“ Funktionalität von TPM 2.0, um einen sicheren Kommunikationskanal zwischen zwei Parteien aufzubauen. Sie zeigen Angriffsszenarien auf bestehende Protokolle auf und demonstrieren, wie ihre Implementierung diesen standhält. Ihr Protokoll ist einfach in bestehende Technologien integrierbar und unabhängig vom verwendeten Netzwerk-Stack. Die geheimen Teile ihres Diffie-Hellman Schlüsselaustausches werden im TPM generiert und gespeichert. Dies schützt ihre Implementierung gegen Seitenkanalattacken auf die CPU. In der Attestation Phase tauschen beide Kommunikationspartner signierte PCR-Werte aus und verifizieren diese mit dem entsprechenden AIK Zertifikat (siehe 5.4, 5.5).

Das TPM kann nicht nur zur Absicherung des Bootprozesses oder für Remote Attestation eingesetzt werden. Wadkar [139] verwenden TPM 2.0, um die Internet-Browser-Umgebung abzusichern. Dabei wird der Fokus auf die Browser-Konfigurationsdatei gelegt. Diese beinhaltet nicht nur private Daten wie etwa das Suchverhalten, sondern hilft auch Angreifenden Zugriff auf das System zu erlangen und dort maliziösen Code auszuführen. Der Zugriff auf die Datei wird mittel PCR-Policy beschränkt. Für die Policy werden die Messwerte des BIOS, des Bootloaders, des Betriebssystems und der Ausführbaren-Datei des Browsers verwendet.
Auch für mobile Geräte hat TCG [133] eine TPM 2.0 basierte Architektur spezifiziert. TPM Mobile (MTM) wird als „Trusted Application“ innerhalb einer geschützten Umgebung ausgeführt. Die genaue Umsetzung ihrer Spezifikation überlässt TCG den Herstellerfirmen.
Der Prozessorhersteller ARM bietet mit TrustZone ebenfalls eine Lösung für ARM basierte Systeme. ARM CPUs werden häufig für mobile Geräte und Internet of Things eingesetzt. TrustZone bietet, so Pinto u. a. [107], eine System-on-Chip Hardware-basierte Sicherheitslösung für bestimmte ARM Prozessoren. Diese Erweiterung beinhaltet signifikante architektonische Veränderungen zum Beispiel die Unterteilung in zwei Ausführungsebenen. Dabei wird auf Prozessorebene zwischen einer „Secure World“ und einer „Normal World“ unterschieden. Beide Ausführungsebenen sind auf Hardwareebene voneinander isoliert. Dabei kann unautorisierte Software nicht auf Ressourcen der sicheren Welt zugreifen. ARM TrustZone kann verwendet werden, um einzelne Software-Libraries aber auch ein separates Betriebssystem abzusichern. Der Prozessor kann zur selben Zeit in nur einer der beiden Ebenen agieren. Ein weiterer Prozessormodus namens „Secure Monitor“ wurde eingeführt, um zwischen den Welten wechseln zu können.
Ashraf u. a. [9] analysieren in ihrer Arbeit Hardware-basierte Sicherheitslösungen und Standards für mobile Geräte. Dabei gehen sie sowohl auf die MTM Spezifikation von TCG als auch auf ARM TrustZone ein. Aufgrund von physischen Limitierungen wurde, so Ashraf u. a., MTM nicht großflächig umgesetzt. Sie präsentieren in ihrer Arbeit einen weitere TPM basierte Lösung (mTPM)für mobile Geräte. Ihr Ansatz ist kompatible mit den bestehenden Spezifikationen von TCG und NIST. mTPM ist ein Sicherheitsprozessor integriert in die CPU des Gerätes. Dadurch wird kein zusätzlicher Hardware-Chip benötigt. Gegenüber ARM TrustZone bietet ihre Lösung die Möglichkeit eines sicheren Speichers. Einer der wichtigsten Faktoren von TPM, ist die sichere Speicherung von Daten, wie etwa Schlüssel, und die Zugriffsbeschränkung auf diese.
Auch Chakraborty u. a. [20] gehen in ihrer Arbeit auf Limitierungen bezüglich Platzes, Kosten und Leistung von mobilen Geräten und ihre Auswirkungen auf Sicherheitslösungen ein. Um ein Hardware-basierte TPM ohne zusätzliche Chips zu ermöglichen integrieren sie TPM 2.0 Funktionalität in die SIM-Karte des Gerätes. Eine große Herausforderung stellt dabei sowohl die Ressourcenbeschränkung der Karte als auch ihre mangelnde Bindung zum Gerät dar. Ein TPM ist normalerweise strikt an ein Gerät gebunden und sollte nicht so einfach wie eine SIM-Karte übertragen werden können. Wenn ein TPM von einem Gerät auf ein anderes übertragen werden kann stellt dies ein großes Sicherheitsproblem dar. Das TPM könnte während des Bootprozesses in ein anderes Gerät integriert wer-

den, welches es mit validen Messwerten befüllt. Das TPM könnte anschließend in das anzugreifende System wieder integriert werden, um so einen falschen Systemzustand vorzutäuschen und dadurch Sicherheitsmaßnahmen wie Policys zu umgehen. Sie setzen Trusted Execution Environment als Proxy ein, um eine Bindung zwischen SIM-Karte und Gerät zu gewährleisten und dieses Problem zu lösen. Sie kombinieren simTPM mit der ARM Trusted Firmware (ATF) um den Bootprozess zusätzlich abzusichern. Performance-Analysen ergaben, dass ihre SIM-Karten basierte Lösung vergleichbar mit existierenden Hardware TPMs ist.

KAPITEL 3

Grundlagen

Dieses Kapitel beinhaltet relevante technische Konzepte und Begriffe, die für ein besseres Verständnis dieser Arbeit relevant sind. Da in der Fachliteratur Begriffe oft als Synonyme verwendet werden, werden in diesem Kapitel auch Fachbegriffe in ihrem Verwendungszweck definiert. Durch das detaillierte Beschreiben des Bootprozesses und der wichtigsten Bootkomponenten möchte die Autorin diese Arbeit auch für Lesende ohne Expertise zugänglich machen. Ein grundlegendes technisches Verständnis in den Bereichen Kryptographie und Computersicherheit wird jedoch vorausgesetzt.

3.1 Linux-basierte Desktop-Systeme

Linux ist, so Kofler [77], ein Open-Source, Unix-ähnliches Betriebssystem. Im Gegensatz zu den meisten Unix-Systemen ist bei Linux der ganze Source Code frei verfügbar. Es gibt viele verschiedene Distributionen. Eine Distribution ist ein Paket aus Betriebssystem (Kernel) und den Zusatzprogrammen.
Negus [99] definiert Desktop-Systeme als Personal Computer (PC), welche über eine graphische Benutzeroberfläche bedient werden können. In den vergangenen Jahren wurden die graphischen Benutzeroberflächen von Linux immer leichter zu bedienen, ähnlich wie bei macOS oder Windows. Bei bestimmten Linux-Distributionen, wie zum Beispiel Fedora, openSUSE oder Ubuntu, kann bei der Installation die Desktop-Umgebung aus einer Auswahl frei gewählt werden. Im Falle von Debian GNU/Linux kann eine beliebige Umgebung installiert werden. Es gibt viele verschiedene Desktop-Umgebungen, die verbreitetsten sind K desktop environment (KDE) (www.kde.org), GNOME(www.gnome.org) und X zusammen mit einem Window-Manager(X.org, XFree86.org + Window Manager). Wobei es davon wiederum viele Ableitungen, wie MATE oder Cinamon, gibt.

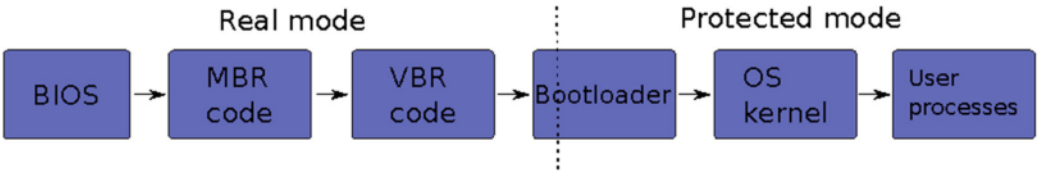

Abbildung 3.1: Bootprozess mit BIOS (Basierend auf Grill u. a. [51])

3.2 Bootprozess

Beim Starten eines Computers wird zu Beginn Code ausgeführt, der die Hardware initialisiert. Dieser Code befindet sich in einem nicht flüchtigen Speicher und wird als „boot firmware“ bezeichnet. Die Hauptfirmware zum Starten des Systems ist das Basic Input/Output System (BIOS). Es gibt verschiedene Arten von BIOS Firmware. Einige beruhen auf der Unified Extensible Firmware Interface Spezification und werden UEFI BIOS oder nur UEFI genannt. Die Hauptaufgabe des BIOS ist das Überprüfen und Initialisieren der Hardware und anschließend das Laden des restlichen Systems. Dieser Vorgang wird Bootprozess genannt. Das BIOS wird normalerweise von Herstellerfirmen (Original Equipment Manufacturers (OEMs)) oder unabhängigen BIOS-Lieferanten entwickelt (für weitere Informationen siehe NIST Publikation [26]). Eine detailliertere Darstellung für einen möglichen Linux-Bootprozess mit BIOS ist in Abbildung 3.1 zu finden. Im Allgemeinen führt ein BIOS-Bootprozess folgende Schritte aus [26]:

1. **Ausführen des Core Root of Trust:** Das BIOS verfügt über einen kleinen Codeblock, welcher als Erster ausgeführt wird und in manchen Fällen die restliche Firmware überprüft. Dieser wird BIOS Boot Block genannt. Bei Trusted Computing Systemen kann er auch den Core Root of Trust for Measurement (CRTM) beinhalten (siehe Abschnitt 3.4.2).

2. **Initialisierung Low-Level Hardware:** Am Anfang des Bootprozesses testet und initialisiert das BIOS zudem Hardwarekomponenten wie etwa Motherboard, Speicher und CPU.

3. **Ausführen anderer Firmware-Module:** Das BIOS lädt und führt zusätzliche Firmware-Module aus. Diese können entweder die Fähigkeiten des BIOS erweitern oder zusätzliche Hardware initialisieren.

4. **Auswahl Boot Gerät:** Im nächsten Schritt sucht das BIOS nach Geräten, die als bootbar identifiziert wurden. Normalerweise bootet das BIOS das erste Bootgerät, das einen gültigen Master Boot Record (MBR) hat [26]. Der MBR befindet sich, so Grill u. a. [51] im ersten Sektor (512 Bytes) der Festplatte. Dieser gibt die Kontrolle an den Volume Boot Record (VBR) weiter, welcher zusätzliche Bootinformationen beinhaltet. Anschließend wird der Bootloader geladen. Die CPU wechselt dabei vom Real Mode in den Protected Mode (siehe Abschnitt 3.3)

5. **Betriebssystem laden:** Der Bootloader lädt und initialisiert dann den Kernel. Sobald der Kernel funktionsfähig ist, startet er das restliche Betriebssystem und die Kontrolle geht vom BIOS an das Betriebssystem über [26].

Der UEFI Bootprozess führt ähnliche Schritte aus (siehe Abbildung 3.2). Zu Beginn wird ebenfalls ein kleiner Codeblock ausgeführt, welcher lediglich den nächsten auszuführenden Code überprüft (ähnlich wie BIOS Boot Block). Diese Etappe wird Security (SEC)-Phase genannt und dient als Core Root of Trust des Systems. Der nächste Schritt ist die Pre-EFI Initialization (PEI)-Phase, wo die Hauptkomponenten, wie Prozessor und Motherboard initialisiert werden. Die PEI-Phase bereitet das System für die Driver Execution Environment (DXE)-Phase vor. In der DXE-Phase werden die meisten Komponenten initialisiert, passende Treiber gesucht und ausgeführt. Die Boot Device Selection (BDS)-Phase bildet den letzten Schritt zum Booten des Betriebssystems. Der Bootloader (BL) wird entweder vom MBR oder der GUID Partition Table (GPT) geladen. Genauere Informationen zu MBR und GPT in den Abschnitten 4.2.2 und 4.2.3. Dieser lädt wiederum den Kernel, welcher dann in die Runtime (RT)-Phase wechselt und das Betriebssystem startet. Während des Bootprozesses werden System Management Interrupt (SMI)-Handler, auch bekannt als System Management Mode (SMM) Code, geladen [26]. Bei SMM handelt es sich, wie in Abschnitt 3.3 beschrieben, um einen bestimmten Ausführungsmodus der CPU.
UEFI als Nachfolger von BIOS wurde für neuere Hardware konzipiert. Das Compatibility Support Module (CSM) kann verwendet werden, um Legacy- BIOS-Kompatibilität zu erreichen und dadurch ältere Hardware initialisieren zu können. Dafür muss im Bootmenü der UEFI-Modus auf Legacy BIOS gesetzt werden (siehe Intel Spezifikation [71]).

3.2.1 Shim

Vor dem eigentlichen Bootloader, beschrieben in Abschnitt 3.2.2, kann ein Pre oder First-Stage Bootloader eingesetzt werden. Dieser ist minimal gehalten und hat lediglich die Aufgabe den Second-Stage Bootloader zu laden und gegebenenfalls vorher zu überprüfen. Shim ist ein First-Stage UEFI Bootloader, welcher von einer Gruppe von Linux-Entwickelnden, unter anderem Garrett, implementiert wurde, um UEFI Secure Boot in Linux-Systemen zu unterstützen [46]. UEFI Secure Boot ist eine Methode den Bootprozess abzusichern, indem nur Komponenten von der Firmware geladen werden, welche eine gültige Signatur oder einen validen Hashwert haben (mehr Details zu UEFI Secure Boot in Abschnitt 6.3). Diese Signaturen werden standardmäßig, so Babar [11], von bestimmten Certificate Authoritys (CAs) ausgestellt. Diese erzeugen ein Schlüsselpaar, signieren mit dem privaten Schlüssel die Komponente, wie etwa den Bootloader, und hinterlegen den öffentlichen Schlüssel in der UEFI-Firmware. Microsoft ist ein Beispiel für eine CA. Es signiert jedoch nicht nur eigene Komponenten, sondern auch jene im Auftrag von anderen Unternehmen, so auch den Linux Bootloader. Um nicht jeden Linux Bootloader, bei jeder Änderung signieren zu lassen, wurde ein kleinerer Bootloader namens Shim entwickelt. Dieser ändert sich kaum und hat lediglich die Aufgabe den eigentlichen Bootloader zu überprüfen und zu laden. Shim wurde, so Garrett [42], entwickelt, um die

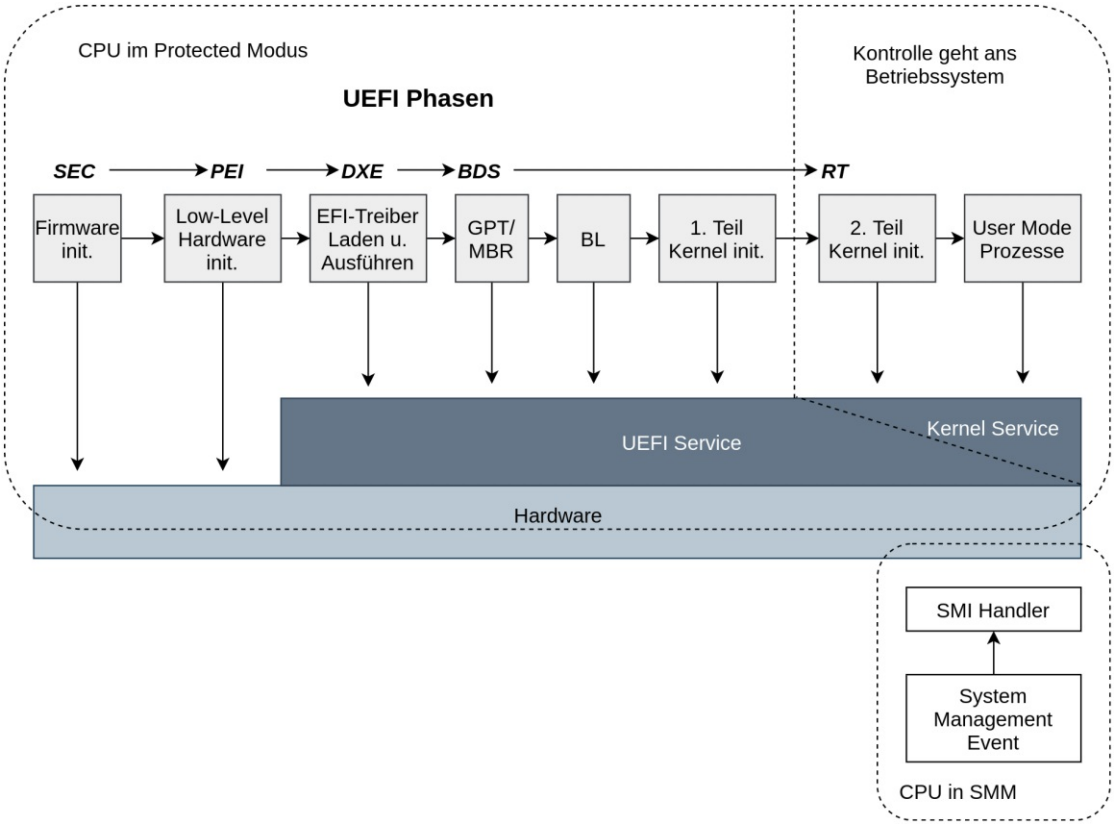

Abbildung 3.2: Bootprozess mit UEFI [26]

Lücke zwischen dem Microsoft Root of Trust (RoT) und dem eigenen RoT zu schließen. Laut Kunst u. a. [81] haben die Distributionen Ubuntu, Red Hat, openSUSE, und Debian jeweils eine eigene Version von Shim mit einem distributions-spezifischen CA Schlüssel. Dieser kann verwendet werden, um den mitgelieferten Second-Stage Bootloader, wie zum Beispiel GRUB2 zu überprüfen. Da es für kleinere Distributionen, so Pavlík [105], nicht praktikabel ist, ihr eigenes Zertifikat in das Shim Paket zu integrieren und dieses von Microsoft signieren zu lassen, wurde die Möglichkeit einer eigenen Schlüsseldatenbank eingebaut. Diese ist nicht Teil des signierten Paketes und kann von den Anwendenden mit eigenen Schlüsseln, den sogenannten Machine Owner Keys (MOKs), erweitert werden. Diese Schlüssel können verwendet werden, um beliebige Bootloader und Kernel zu signieren. In Abbildung 3.3 ist sowohl der Ablauf eines UEFI Secure Bootprozesses mit Standard-openSUSE als auch mit beliebigen Komponenten dargestellt. Anstelle von openSUSE kann jegliche Distribution mit eigenem CA-Zertifikat verwendet werden. Die UEFI Schlüssel und Datenbanken werden in Abschnitt 6.3 genauer erklärt.

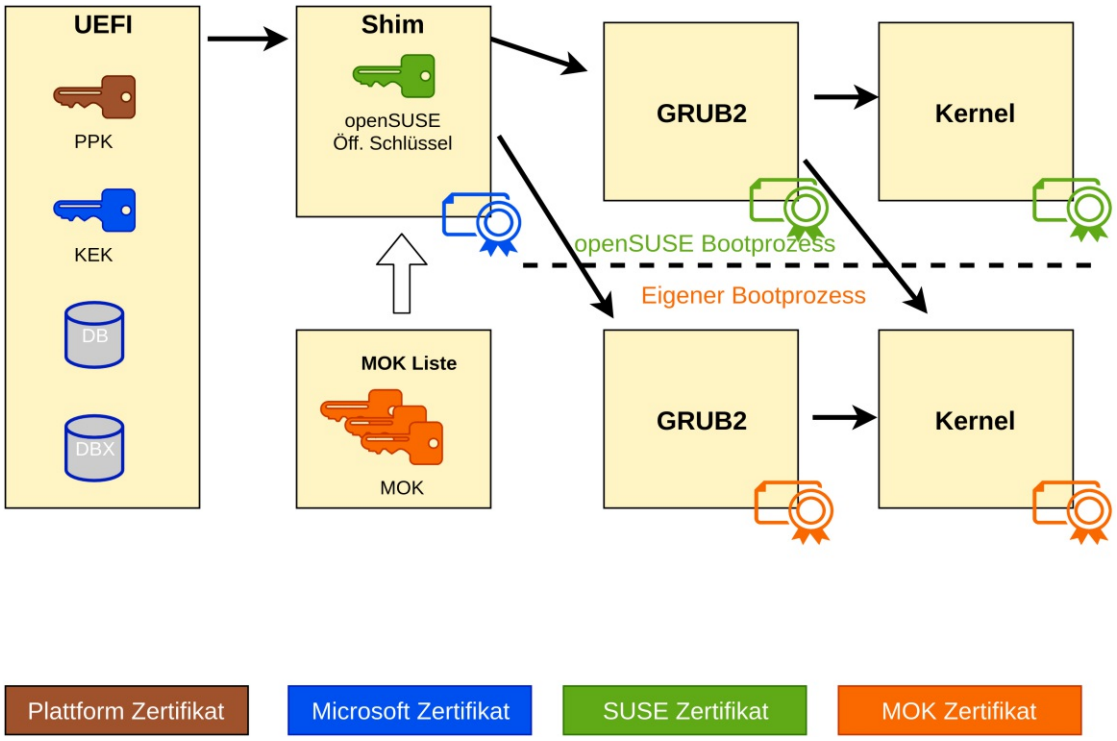

Abbildung 3.3: UEFI Secure Boot mit Shim (Basierend auf [153], [81])

3.2.2 GRUB

Ein Bootloader ist, so Matzigkeit u. a. [92], die erste Software, die beim Starten eines Computers ausgeführt wird. Seine Aufgabe ist es den Kernel zu laden und die Kontrolle an diesen weiter zu geben. Sobald das erfolgt ist, wird der Bootloader nicht mehr benötigt. GNU GRUB oder GRand Unified Bootloader ist ein weit verbreiteter Multiboot Bootloader. Er kann eine Vielzahl an freien und proprietären Betriebssystemen laden. Normalerweise werden bei der Installation eines neuen Betriebssystems die gesamten Bootmechanismen installiert. Mittels Multiboot können mehrere verschiedene Betriebssysteme auf einem Gerät installiert sein. Ursprünglich wurde GRUB von Erich Stefan Boleyn entwickelt und implementiert. Später wurde es im Sinne des GNU GRUB Projekts erweitert. Mittlerweile wurde GRUB von GRUB2 ersetzt. Das ursprüngliche GRUB wird nicht mehr weiterentwickelt und wird GRUB Legacy genannt. GRUB2 ist auf der einen Seite für normale Endnutzende einfach in der Handhabung, kann jedoch von Erfahreneren flexibel angepasst werden [92].
Die GRUB2-Konfigurationsdatei heißt, so Both [15], `grub.cfg` und wird bei der Linux Installation im `/boot/grub2` Verzeichnis erzeugt. Sie beinhaltet Bash ähnlichen Code und eine Liste der installierten Kernel. Im Ordner `/etc/grub.d` befinden sich die Haupt-Konfigurationsdateien, welche dann zum `grub.cfg` zusammengefasst werden. Die Dateien sind nummeriert, sodass die resultierende Datei die richtige Sequenz beinhaltet.

Die Inhalte jeder Datei werden im `grub.cfg` mit einem Start- und End-Kommentar (wie in Listing 3.1) gekennzeichnet. Diese Dateien sollten jedoch nicht ohne Erfahrung und einem Backup verändert werden. Einfache Konfigurationen können auch mittels `/etc/default/grub` Datei gemacht werden. Diese beinhaltet simple Schlüssel/Wert-Paare und erlaubt eine einfache Konfiguration.

Listing 3.1: Beispiel für einen grub.cfg Eintrag

```
### BEGIN /etc/grub.d/10_linux ###
...
### END /etc/grub.d/10_linux ###
```

3.2.3 Kernel

Der Kernel ist, so Mauerer [93], im Allgemeinen ein Vermittler zwischen Software und Hardware. Seine Aufgabe ist es, Anfragen von Applikationen entgegenzunehmen und an die Hardware weiterzugeben. Wenn eine Applikation, zum Beispiel eine Datei lesen möchte, muss der Kernel wissen, wo diese Datei liegt und welche Befehle notwendig sind, um diese zu lesen. Für die Anwendung ist dies alles irrelevant, da die Details vom Kernel abstrahiert werden. Anwendungen kommunizieren mit dem Kernel mittels System-Calls. Der Kernel agiert auch als Verwalter für Systemressourcen.
Jedes Programm und jede Applikation, die in einem UNIX System läuft, werden als Prozess angesehen. Jedem Prozess wird ein Adressraum im virtuellen Speicher der CPU zugewiesen. Die Prozesse agieren unabhängig und teilweise komplett isoliert voneinander (für weitere Informationen siehe Abschnitt 3.3.3). Wenn mehrere Prozesse parallel laufen, muss der Kernel die vorhanden Ressourcen wie CPU, Speicher, Netzwerkverbindungen etc. einteilen [93].

3.2.4 initrd/initramfs

Das Ziel des Bootprozesses ist, so Babar [10], Anwendenden Dateien und Dienste bereitzustellen, welche sich im Dateisystem befinden. Der Kernel muss nun dieses Wurzelverzeichnis finden, mounten und Anwenden zur Verfügung stellen. Dafür muss jedoch `systemd` ausgeführt werden, welches sich wiederum im Wurzelverzeichnis befindet. `systemd` ist ein Programm welches zum Starten, Beenden und Überwachen von Prozessen verwendet wird. Um dieses Henne-Ei-Problem zu lösen wird ein temporäres Dateisystem eingesetzt. Früher wurde Initial-RAMdisk (initrd) verwendet, während heutzutage immer häufiger initramfs eingesetzt wird.
Das initiale RAM-basierte Dateisystem (initramfs) ist ein cpio-Archiv, welches in den Arbeitsspeicher geladen wird und eine kleine Linux-Umgebung bereitstellt. Dies ermöglicht das Ausführen von kleinen Programmen. Sowohl initrd als auch initramfs beinhaltet immer ein Programm namens „init“. Dieses Programm lädt Treiber und hängt das eigentliche Root-Dateisystem ein [10].
Das Root-Dateisystem bildet das Wurzelverzeichnis des gesamten Dateisystems. Darin

befinden sich Programme und Dateien, welche nach dem Starten automatisch ausgeführt werden (für weitere Informationen siehe openSUSE Dokumentation [14]).

3.3 Ausführungsmodus und Virtualisierung

Intel Prozessoren unterscheiden zwischen drei Ausführungsmodi: Real Address Mode, Protected Mode und System Management Mode (SMM). Der Ausführungsmodus bestimmt welche Befehle und architektonischen Funktionen verwendet werden können (für weitere Informationen siehe Intel Spezifikation [70]). Real Address Mode wird, so Duflot u. a. [32], meist beim Starten oder Zurücksetzen verwendet. SMM wird für systemweite Operationen wie Energiemanagement, Hardwarekontrolle oder proprietären Original Equipment Manufacturer (OEM) designten Code eingesetzt. Protected Mode ist der normale oder Standard-Modus des Prozessors und wird nochmals in vier Stufen (Ringe) unterteilt.

3.3.1 CPU-Ringe

Laut Lee u. a. [84] werden die Rechte eines Prozesses bereits auf Hardwareebene eingestuft. Die Privilegierungsstufen oder Ringe der CPU bestimmen auf welche Hardware ein Prozess zugreifen, beziehungsweise, welche Befehle er ausführen darf. x86 Prozessoren unterscheiden die Stufen Ring 0 bis 3, wobei Ring 0 (Kernelmode) am meisten Rechte hat und Ring 3 (Usermode) am wenigsten. Moderne, auf x86 Architektur basierende, Betriebssysteme implementieren davon lediglich Usermode für Applikationen und Kernelmode für Kernel, Betriebssystem-Services und Treiber. Später wurden, so Wojtczuk u. a. [149] und Domas [31] weitere Rechtehierarchien hinzugefügt, um Prozesse im Kernelmode einzuschränken. Diese werden oft auch Ringe genannt. Ring -1 (Hypervisor) kann Ring 0 Code isolieren und Ring -2 (SMM) kann Ring -1 Prozesse einschränken. Rootkits können in verschiedenen Privilegierungsstufen agieren. Man unterscheidet laut Tereshkin u. a. [130] zwischen:

- Ring 3 Usermode Rootkits
- Ring 0 Kernelmode Rootkits
- Ring -1 Hypervisor Rootkits
- Ring -2 SMM Rootkits

3.3.2 Hypervisor

Popek u. a. [108] definierten Hypervisor (oder VMM) als eine Firm- oder Software, welche virtuelle Maschinen erstellt, ausführt und verwaltet. Ein Hypervisor erlaubt es, wie in Abbildung 3.4 zu sehen, mehrere virtuelle Maschinen (Gast Systeme) auf einem Computer (Host System) auszuführen. In diesem Gast System läuft ein eigenes Betriebssystem und

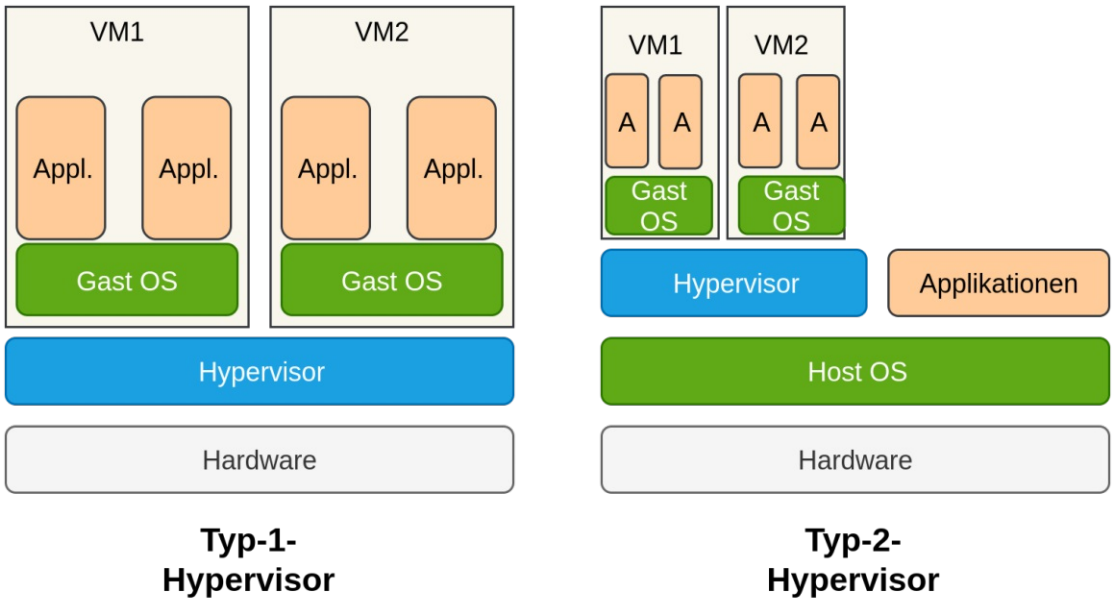

Abbildung 3.4: Typ 1 und 2 Hypervisor (Basierend auf [13])

beliebige Applikationen. In ihrer Arbeit definieren sie drei essenzielle Aspekte, damit eine Software als Hypervisor gilt: als erstes muss der Hypervisor eine Ausführungsumgebung für Programme schaffen, welche identisch mit der originalen Maschine ist. Zweitens zeigen mittels Hypervisor ausgeführte Programme nur geringe Geschwindigkeitseinbußen, da ein Großteil der Prozessorbefehle direkt vom echten Prozessor ausgeführt werden. Und drittens hat der Hypervisor volle Kontrolle über die Systemressourcen.
Goldberg [49] unterscheidet bereits 1973 in seiner Arbeit zwei Arten von Hypervisoren Typ-1- und Typ-2-Hypervisor. In aktuelleren Arbeiten, wie etwa von Baun [13] wird Typ-1-Hypervisor auch Paravirtualisierung und Typ-2 Vollständiger Virtualisierung genannt. In Abbildung 3.4 sind beide Arten dargestellt, welche nun kurz erklärt werden.

- **Typ-1-Hypervisor** (bare metal) läuft direkt auf der Hardware und kann dadurch direkt auf die physischen Ressourcen des Systems zugreifen. Um diesen Typ Hypervisor zu ermöglichen muss das Gerät die entsprechenden Treiber unterstützen [49].

- **Typ-2-Hypervisor** (hosted) läuft hingegen auf Betriebssystemebene. Typ-2 kann wie eine normale Software ausgeführt werden und wird somit von allen Systemen unterstützt. Der Typ-2-Hypervisor bietet eine komplette System-Umgebung inklusive BIOS. Er ist für die Zuweisung der Hardwareressourcen zuständig und kann diese auch teilweise emulieren (zum Beispiel Netzwerkkarte) [13].

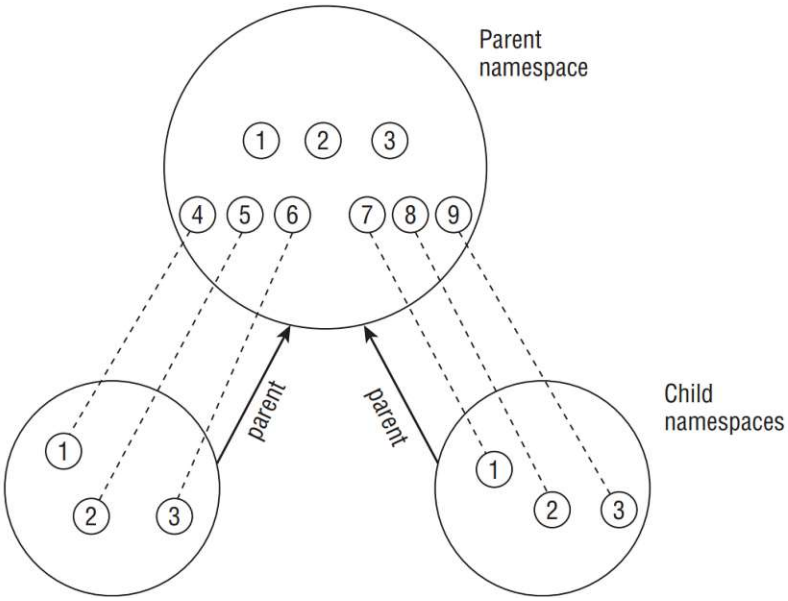

Abbildung 3.5: Hierarchische Gliederung von Namespaces [93]

3.3.3 Linux Namespaces

Unter Linux werden traditionell Ressourcen global verwaltet. Prozesse können zum Beispiel eindeutig durch eine Prozess-ID (PID) identifiziert werden. Der Linux Kernel erlaubt auch eine Art der Virtualisierung (Lightweight Process Virtualization), um systemweite Ressourcen zu isolieren. Dabei kann eine Gruppe von Prozessen unterschiedlichen Containern, sogenannten Namespaces, zugeordnet werden. Der Grad der Separation kann variieren. Namespaces können unter anderem komplett isoliert sein, sodass Prozesse eines Containers nicht von der Existenz anderer Container wissen. Jedoch können sie auch bestimmte Aspekte wie etwa Dateisysteme miteinander teilen [93].
Es gibt, so Rosen [112], sechs Arten von Namespaces wie etwa PID, User und Network. Zu Beginn erzeugt Linux von jeder Art einen Namespace. Diese werden auch Standard-Namespaces genannt und beinhalten alle Prozesse. Mittels System-Calls können weitere Container erzeugt und verwaltet werden (für weitere Informationen zu System-Calls siehe Abschnitt 4.1.2).
Namespaces können hierarchisch zueinander stehen, wobei die Eltern Elemente über die Prozesse ihrer Kinder Bescheid wissen, die Kinder jedoch nicht über das Vorkommen der Eltern oder der Geschwister. Im Falle von PID-Namespaces können PIDs mehrfach vergeben werden. So kann zum Beispiel jeder Container einen Init-Prozess mit der PID 1 haben. Außerhalb des Containers, im Eltern-Namespace, wird aber wie in Abbildung 3.5 ersichtlich, eine eigene, eindeutige PID verwendet [93].

3.4 Trusted Computing

Trusted Computing und Trusted Computing Base (TCB) sind Grundbegriffe in der Computersicherheit und sind bereits Teil der Orange Book Serie [5] von 1985. Die Orange Book Serie wurde vom amerikanischen Verteidigungsministerium veröffentlicht. Sie beinhaltet einen Leitfaden für Computer Sicherheit. Laut Trusted Computing Group (TCG) [73] ist TCB eine Sammlung von Hard- und Software-Komponenten, auf welche alle weiteren Sicherheitsmaßnahmen eines Computersystems aufbauen. Ein wichtiger Faktor von TCB ist, dass es sich selbst vor Kompromittierung von außerhalb der TCB schützt.
Die TCG ist ein Zusammenschluss von vielen Hardware- und Softwarefirmen, wie etwa IBM, Microsoft und Intel, um globale Standards für Trusted Computing-Plattformen zu entwickeln [1].

3.4.1 Sicheres Booten

TCG unterscheidet beim sicheren Booten eines Computers zwischen zwei Konzepten, Measured- und Secure Boot [127].

Measured Boot

Bei Measured Boot wird jede Komponente gemessen und der Hashwert gespeichert, bevor diese ausgeführt wird. Die Messwerte können später an eine dritte Stelle geschickt werden, um die Systemintegrität zu beweisen. Es wird nicht aktiv in den Bootvorgang eingegriffen. Das System kann somit auch in einen nicht vertrauenswürdigen Zustand booten, dieser wird jedoch gemessen und kann berichtet werden [127]. Wie Garrett [48] erklärt, kann eine maliziöse Komponente die Messwerte aller nachfolgenden Komponenten verfälschen, deshalb reicht es zum Beispiel nicht aus lediglich die Werte des Kernels zu evaluieren. Eine genaue Beschreibung des Mess- und Überprüfungsprozesses ist in den Abschnitten 5.2 und 5.5 zu finden. In manchen Fachartikeln, wie etwa bei Chen u. a. [22], wird für dieses Konzept auch der Begriff Trusted Boot oder Authenticated Boot verwendet. TCG bezieht sich mit Trusted Boot lediglich auf tboot, ein pre-Kernel Modul (ausführliche Beschreibung von tboot in Abschnitt 6.5) [4].

Secure Boot

Bei Secure Boot wird jede Komponente vor ihrer Ausführung überprüft. Sobald eine Komponente die Überprüfung nicht besteht, wird der Bootvorgang abgebrochen. Es kann entweder die Signatur der Komponenten überprüft werden, oder der Hashwert mit vertrauenswürdigen Werten verglichen werden [127]. Für das ausschließliche Booten von Komponenten mit einer gültigen Signatur wurde von Google der Begriff Verified Boot eingeführt [50]. In der Literatur wird er häufig als Synonym für Secure Boot verwendet, während Secure Boot häufig als Synonym für UEFI Secure Boot eingesetzt wird.

In dieser Arbeit werden die TCG-Bezeichnungen Measured Boot und Secure Boot als Überbegriffe verwendet. Trusted Boot wird nur in Bezug auf tboot verwendet und UEFI Secure Boot wird explizit als solches gekennzeichnet.

3.4.2 Root of Trust

Um einem Computersystem vertrauen zu können, muss es eine vertrauenswürdige Basis geben. Diese bildet das erste Glied einer Vertrauenskette (Chain of Trust). TCG bezeichnet diese Basis als Root of Trust (RoT). Roots of Trust sind Komponenten, auf deren Integrität Anwendende vertrauen müssen, da es nicht möglich ist zu überprüfen, ob diese sich richtig verhalten. Jedoch sind RoT-Komponenten mit dem Zertifikat der Herstellerfirma ausgestattet, welches versichern soll, dass die Komponenten auf eine Art implementiert wurden, welche sie vertrauenswürdig macht [73]. TCG spezifiziert drei RoT für eine vertrauenswürdige Plattform [73]:

- **Root of Trust for Storage (RTS)**: Der Root of Trust for Storage (RTS) ist verantwortlich für das sichere Speichern von Informationen, wie etwa von Messwerten und Schlüsseln. Das TPM kann als RTS fungieren. Es hat einen abgeschotteten Speicher, auf welchen nur das TPM zugreifen kann. Manche Informationen wie Messwerte werden vom TPM frei offengelegt, während etwa Schlüssel geheim gehalten werden.
- **Root of Trust for Measurement (RTM)**: Das RTM misst die Komponenten und schickt die Ergebnisse an das RTS. Normalerweise ist das RTM die CPU unter der Kontrolle von CRTM.
- **Root of Trust for Reporting (RTR)**: Das RTR generiert signierte Berichte aus einigen RTS Inhalten. Das TPM kann als RTR fungieren. Die Berichte können mit dem entsprechenden Befehl abgerufen werden. Diese Inhalte mit den entsprechenden Befehlen sind:
  - PCR-Werte: *TPM2_Quote()*
  - Überwachungs-Logfiles: *TPM2_GetCommandAuditDigest()*
  - Schlüsseleigenschaften: *TPM2_Certify()*

Static Root of Trust for Measurement (S-RTM)

Beim Static Root of Trust for Measurement (S-RTM) geht man, so Choinyambuu [24] von einer unveränderbaren vertrauenswürdigen Basis aus, die den Anfang der Chain of Trust bildet. Beim Starten des Systems beginnt der Bootprozess beim CRTM. Darunter ist ein statischer Code zu verstehen, der Systemzustände misst, indem er Hashwerte der Komponenten berechnet. Das CRTM beginnt, so Falk [33] seinen Messprozess beim BIOS und schreibt das Ergebnis in das TPM. Nachdem das BIOS erfolgreich gemessen wurde, misst dieses die nächste Komponente, den MBR. Dieser Vorgang wird wie in Abbildung

3.6 fortgesetzt und bildet somit eine Chain of Trust (für eine detaillierte Beschreibung des Bootprozesses siehe Abschnitt 3.2). Diese Art des Booten wird auch Measured Boot genannt.
Das S-CRTM kann auf verschiedenen Arten integriert werden. TCG stellen folgende Varianten vor:

- S-CRTM ist der BIOS Boot Block: das BIOS wird in zwei Teile unterteilt. Den BIOS Boot Block (S-CRTM) und das POST BIOS. Beide Komponenten können unabhängig voneinander upgedatet werden. Das POST BIOS muss dadurch jedoch auch gemessen werden [3].
- BIOS ist S-CRTM: Das gesamte BIOS bildet den CRTM und ist unter der Kontrolle der Herstellerfirma [3].
- Die CPU ist das CRTM. In diesem Fall wird es auch Hardware-CRTM genannt [73].

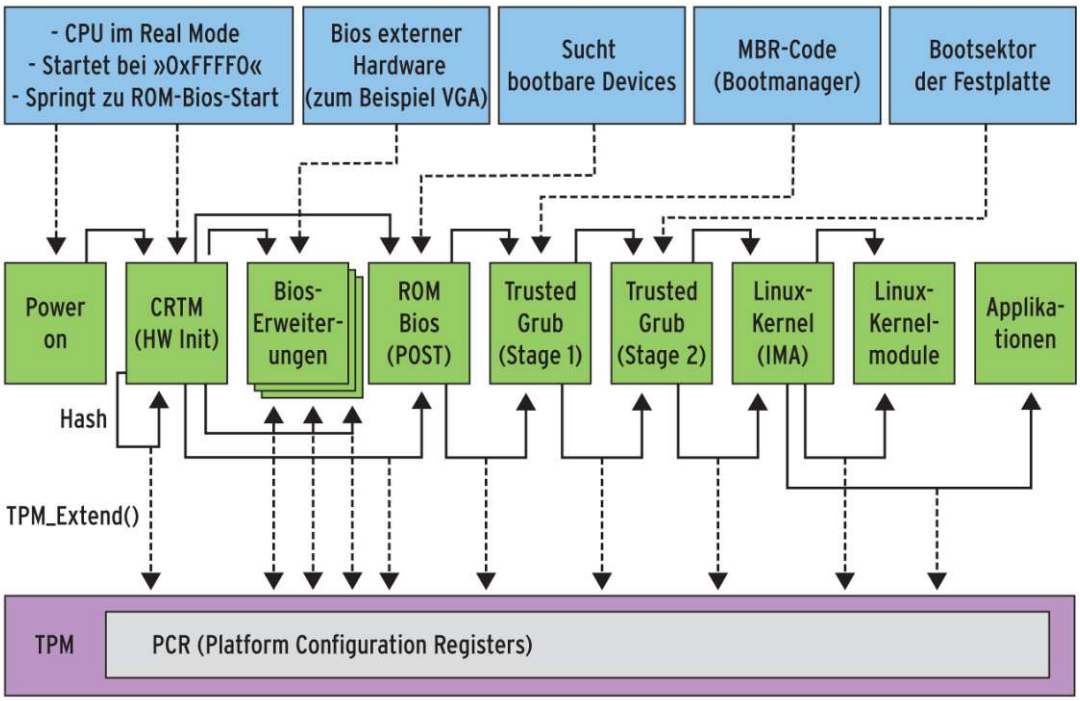

Abbildung 3.6: Measured Boot Prozess und Aufbau einer Chain of Trust [33]

Dynamic Root of Trust for Measurement (D-RTM)

Static Root of Trust weist, so Challener u. a. [21], einige Schwächen auf, unter anderem Erweiterbarkeit und Laufzeitgarantie. Damit das System vertrauensvoll ist, muss jede

ausführbare Komponente gemessen werden. Dies kann bei einer großen Anzahl schnell zu Problemen führen. Das größere Problem ist jedoch die Laufzeitgarantie. S-RTM garantiert lediglich, welcher Code geladen wird, aber nicht was zur Laufzeit ausgeführt wird. Angreifende können in der Zeit zwischen Messen und Ausführen das Programm kompromittieren. Daher definierte TCG Dynamic Root of Trust for Measurement (D-RTM). Der D-RTM kann im Gegensatz zum statischen Root of Trust jeder Zeit gestartet und so oft wie notwendig wiederholt werden. Die beiden Ansätze können auch kombiniert werden, wie zum Beispiel bei Intel TXT (siehe Abschnitt 6.5).

KAPITEL 4

Rootkits und ihre Angriffspunkte

In folgendem Abschnitt werden Angriffe auf den Bootprozess behandelt. Im Speziellen wird auf Rootkits und Bootkits eingegangen.

4.1 Rootkits

Rootkits sind eine Malware-Art, welche den Angreifenden einen langfristigen privilegierten Zugriff auf das Gerät erlaubt ohne, dass das Opfer es merkt. Dabei nutzen sie ihre ausgeweiteten Rechte um Systeminformationen, Dateien und Prozesse zu manipulieren und zu verbergen. Rootkits verbreiten sich nicht von selbst und sind meist Teil eines komplexeren Angriffes [82]. Pelaez [106] unterscheidet, wie in Abschnitt 3.3 beschrieben zwischen Rootkits auf Anwender-Ebene (Usermode) und auf Kernel-Ebene (Kernelmode). Um sich und eventuell andere Schadsoftware zu verbergen, manipulieren Rootkits Systembinarys, Librarys, System-Call-Tables, Prozesslisten etc.

4.1.1 initrd-Manipulation

Das Initial-RAMdisk (initrd)-Archiv wird dynamisch auf dem System generiert, da die verwendeten Kernel-Module abhängig von der Hardware, dem Dateisystem und anderen Systemumständen sind. Bei jedem Kernel-Update wird das Archiv neu generiert. Diese Faktoren machen es schwer, so Tappert [126], die Integrität der Datei zu schützen. In Ubuntu-Systemen wird die initrd-Datei zum Beispiel nicht mittels UEFI Secure Boot kontrolliert [138].
Ein proof-of-concept Rootkit, namens Horse Pill, wurde von Leibowitz [86] bei der Black Hat Konferenz 2016 vorgestellt. Das Rootkit verändert, so Tappert [126], das Programm *run-init*, um die Kontrolle über das System zu erlagen. Wie in Abschnitt 3.2.4 erklärt führt der Kernel anfangs ein kleines *init* Skript aus. Dies lädt Treiber sowie Kernel-Module und führt andere Programme, wie etwa *run-init*, aus. Dieses Programm startet den initialen

Usermode-Prozess des Systems. Das manipulierte Programm platziert das gesamte System in einem neu erstellten Container, genauer gesagt einer Reihe Linux Kernel-Namespaces. Weitere Informationen über Kernel-Namespaces sind in Abschnitt 3.3.3 zu finden. Des Weiteren wird eine Hintertür installiert, welche es Angreifenden erlaubt permanent auf das System zuzugreifen. Die Hintertür und andere Rootkit-Prozesse laufen außerhalb des isolierten Systems im Standard-Namespace. Dadurch sind die Prozesse innerhalb des normalen Systems nicht sichtbar.
Eine weitere Beispiel-Implementierung für Lehrzwecke ist Kitgen von Skarzynski [120]. Das Rootkit infiziert initrd-Dateien, speziell von Ubuntu Systemen. Die komprimierte Datei beinhaltet unter anderem Skripte zum Ver- und Entschlüsseln der Festplatte. Kitgen manipuliert dieses Skript, um das Passwort mitzuschneiden und schadhaften Code auf der Festplatte zu positionieren. Das Rootkit muss auf einem System mit derselben Version wie das Angriffssystem, vorbereitet werden. Es können verschiedene Skripte, wie etwa Keylogger zum Aufzeichnen des Passworts, miteinbezogen werden. Am Ende können Angreifende das finale Kitgen Skript auf einen bootbaren USB-Stick kopieren, von diesem Booten und das Skript ausführen.

4.1.2 Kernel-Manipulation

Laut Bunten [19] sind die meisten Rootkits kernelbasiert, da er die unterste Schicht des Betriebssystems bildet und jede Anwendung kontrolliert. Genauere Informationen zum Kernel sind in Abschnitt 3.2.3 zu finden. Kernel-Mode-Rootkits können diesen auf verschiedene Weisen manipulieren. Unter anderem mit:

- **Loadable Kernel-Module (LKM)** werden laut Shah u. a. [116] am häufigsten verwendet, um den Kernel zu infizieren. Loadable Kernel-Module (LKM) erlauben das dynamische Erweitern des Kernels, ohne diesen neu kompilieren zu müssen. Dabei können entweder neue Module erzeugt und geladen oder existierende kompromittiert werden. Bei Zweiterem ist die Wahrscheinlichkeit größer, dass das Rootkit einen Reboot des Systems überlebt (falls das Modul immer geladen wird).

- **Laufenden Kernel Patchen**: Durch das Korrumpieren des Speichers kann der laufende Kernel manipuliert werden. Das */dev/kmem* Device-File repräsentiert den virtuellen Speicher des laufenden Kernels. Dieses konnte verändert werden, um so System-Calls auszuführen [19]. In moderneren Kernel-Versionen ist es laut O'Neill [102] nicht mehr möglich den Kernel-Speicher mittels dieser Datei zu verändern.

Für Intrusion-Detection-Mechansimen wird es schwieriger einen Angriff zu erkennen, wenn kein zusätzlicher Code eingefügt wird, sondern existierender Code wiederverwendet wird. Hund u. a. [65] entwarfen und implementierten ein System, welches automatisiert eine Return-Oriented-Instruktionssequenz erzeugt. Am Ende konnten sie mit ihrem Return-Oriented-Rootkit bestehende Kernel-Schutz-Mechanismen umgehen.

System-Calls

Sobald eine Anwendung auf bestimmte Systemressourcen zugreifen will, muss sie System-Calls an den Kernel schicken. Dieser kann dann mit erhöhten Rechten (Ring 0) die Anfragen bearbeiten. Mehr Informationen zu CPU-Rechten befinden sich in Abschnitt 3.3.1. System-Calls sind die wichtigste Kommunikationsmethode zwischen Anwendungsprogrammen und dem Kernel [116]. Bei jedem Userland-API-Aufruf, zum Beispiel beim Lesen und Schreiben von Dateien, werden mehrere System-Calls ausgeführt. Wenn ein Rootkit alle System-Calls abfangen kann, hat es die komplette Kontrolle über das System. Wenn Security-Software zum Beispiel nach maliziösen Komponenten sucht, scannt es das gesamte Dateisystem. Dabei werden System-Calls abgesetzt, welche manipuliert werden können. Dadurch bleiben schadhafte Dateien unerkannt [91].
Eine weitere Komponente, welche laut Wang u. a. [142] häufig von Rootkits manipuliert wird, ist die Interrupt-Descriptor-Table (IDT). Die IDT wird verwendet um Interrupts der Hardware, wie zum Beispiel Tastatureingaben, oder andere Exceptions zu verarbeiten. Es handelt sich dabei um eine Tabelle, die jeden Interrupt oder Exception-Handler mit dem entsprechenden System-Call abbildet.
Rootkits haben mehrere Möglichkeiten System-Calls zu manipulieren oder zu ersetzen (hooken):

- **System-Call-Table hooken**: Es wird entweder, so Sd [114], die bestehende Tabelle manipuliert oder eine zusätzliche System-Call-Table erstellt. Bei Zweiterem wird der Pointer auf die neue Tabelle umgeleitet. Diese beinhaltet sowohl originale als auch veränderte Adressen. Da die alte Tabelle unverändert beibehalten wird, besteht diese die Integritätsprüfung und die Kompromittierung kann unbemerkt bleiben.
- **Sytem-Call-Handler hooken**: Dabei wird der Code des System-Call-Handlers manipuliert oder ein anderer Handler aufgerufen. Die System-Call-Table speichert für jeden erlaubten System-Call den entsprechenden Handler. Diese Adresse kann dahingehend verändert werden, sodass maliziöser Code ausgeführt wird [116].
- **IDT hooken**: Rootkits können eine neue IDT erstellen und manipulieren. Da die originale IDT noch existiert können manche Sicherheits-Scanner getäuscht werden [142].

Das Linux Rootkit SucKIT erstellt zum Beispiel eine Kopie der System-Call-Table. Es verändert zahlreiche Einträge, um schadhaften Code auszuführen. Es manipuliert den Interrupt-Handler, damit dieser auf die neue Tabelle verweist. Damit das Rootkit einen Reboot des Systems überlebt, verändert es die *init*-Datei, welche, wie in Abschnitt 3.2.4 genauer beschrieben, von UNIX-Systemen beim Start ausgeführt wird. Die originale *init*-Datei wird versteckt und verwendet, um valide Hashwerte für Intrusion-Detection-Systeme zu generieren [19]. Abbildung 4.1 zeigt, wie das Rootkit den Ausführungsverlauf verändert, um eigenen schadhaften Code auszuführen. Der Interrupt, verursacht durch den Aufruf des Befehls *open()*, beinhaltet den Index des Interrupt-Handlers (IH) in der

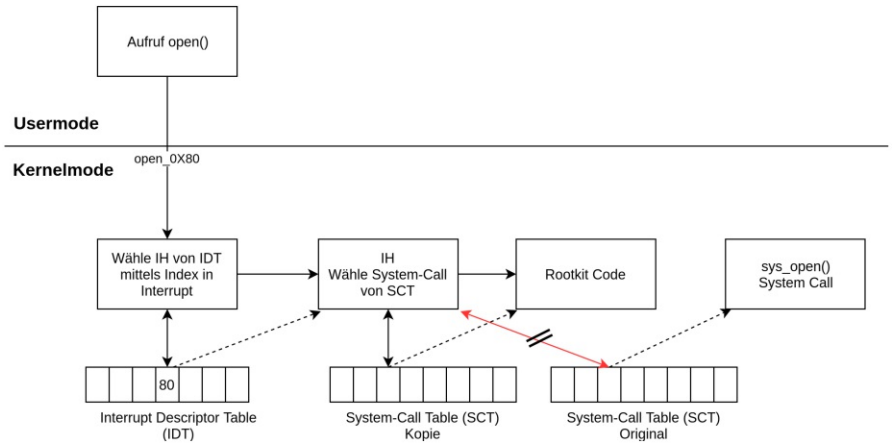

Abbildung 4.1: Angriffsablauf des Rootkit SucKIT (Basierend auf [19] und [6])

IDT. Der Interrupt-Handler sucht sich in der System-Call-Table den entsprechenden System-Call und führt diesen mit den gewählten Parametern aus. In der veränderten System-Call-Table weist dieser Eintrag auf den Rootkit-Code.

4.2 Bootkits

Eine weitere Form von Malware, welche wie Rootkits agieren, sind Bootkits. Doch anstatt Betriebssystem-Dateien wie etwa den Kernel anzugreifen, manipulieren Bootkits auf tieferer Ebene des Bootprozesses, das BIOS, den MBR oder den Bootloader [51] (genaue Beschreibung des Bootprozesses und der einzelnen Komponenten in Abschnitt 3.2). Bootkits haben gewisse Anforderungen [51]:

- Sie müssen mindestens einmal ausgeführt werden, bevor der Kernel die Kontrolle übernimmt.
- Sie müssen einen Neustart des Systems überstehen.
- Sie müssen effizient sein.
- Sie müssen im Real und Protected Mode der CPU funktionieren und einen Wechsel überleben.
- Sie dürfen den Betriebssystemstart nicht stark verzögern.

Bootkits gehören zu den beständigsten und fortgeschrittensten Technologien, die in moderner Malware eingesetzt werden [51]. Da sie vor dem Kernel ausgeführt werden, sind sie, so Gao u. a. [41], meist unabhängig von jeglichem Betriebssystem. Sie können durch eine Neuinstallation des Betriebssystems nicht entfernt werden.
Rootkits nutzen häufig sogenannte Dark Regions (DR), um ihren Code zu platzieren.

Dark Regions sind Bereiche einer physischen Festplatte, welche jedoch nicht Teil eines Dateisystems sind. Diese werden nur sehr selten verändert (zum Beispiel: durch große Betriebssystemupdates) und auch kaum von Sicherheitsmechanismen abgedeckt. Zudem sind sie für Anwendende während normalen Systemoperationen nicht sichtbar [51]. In Abbildung 4.2 sind die DR in Grau eingezeichnet und die Dateisysteme in Grün. Bestimmte Bereiche wie MBR, VBR und Bootloader werden bei Systemstart verwendet, während etwa Lücken zwischen den Bereichen und Partitionen nie verwendet werden.

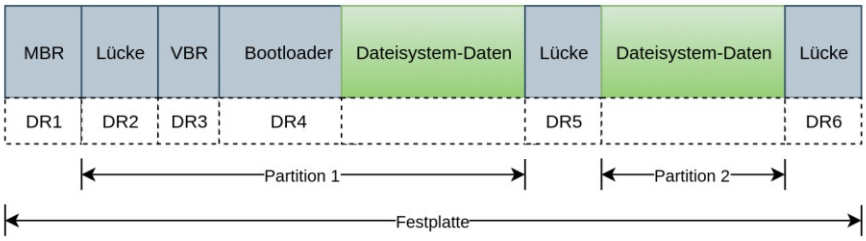

Abbildung 4.2: Typisches Dark Region (DR) Layout [51]

4.2.1 Bootloader-Manipulation

Der Bootloader GRand Unified Bootloader (GRUB)2 ist, so Babar [10], Teil zahlreiche Linux Distributionen. Ende Juli 2020 fanden Shkatov u. a. [119] von Eclypsium eine Schwachstelle (CVE-2020-10713) in GRUB2. Die Schwachstelle namens BootHole kann sogar mit aktiviertem UEFI Secure Boot ausgenutzt werden. GRUB2 ist anfällig für eine Buffer-Overflow-Schwachstelle beim Verarbeiten der Konfigurationsdatei (*grub.cfg*). Bei der GRUB2-Konfigurationsdatei handelt es sich um eine externe Textdatei in der EFI-System-Partition. Sie ist normalerweise nicht signiert und kann daher von Angreifenden verändert werden, ohne die Integrität von Shim oder GRUB2 zu beeinträchtigen (für weitere Informationen zu GRUB und Shim siehe Abschnitt 3.2.2 und 3.2.1). Die Datei kann dahingehend verändert werden, dass schadhafter Code in GRUB2 vor dem Start des Betriebssystems ausgeführt wird. Dies erlaubt einen persistente Infektion des Geräts. Da das UEFI-Execution-Environment weder Schutzmaßnahmen wie Address Space Layout Randomization (ASLR) noch Data Execution Prevention (DEP/NX) hat, ist es einfacher Exploits für diese Schwachstelle zu entwickeln. Der Code kann direkt im Heap ausgeführt werden ohne Return-Oriented-Programming (ROP)-Chains bilden zu müssen. Es sind alle GRUB2-Versionen, welche Befehle von der externen *grub.cfg* Datei laden, betroffen. Eine Lösung dieses Problems umfasst zahlreiche Entitäten. Zunächst muss ein Update für GRUB2 veröffentlicht werden und zusammen mit Shim in den Linux Distributionen erneuert werden. Das neue Shim muss von der Microsoft 3rd Party UEFI CA signiert und die UEFI-Revocation-List (dbx) muss aktualisiert werden [119].

4.2.2 MBR/VBR-Manipulation

Der Master Boot Record (MBR) ist, so Rodionov u. a. [110], der erste Sektor (512 Bytes) einer bootbaren Festplatte. Er besteht aus dem Boot-Code und der MBR-Partition-Table, welche das Partitionierungs-Schema der Festplatte beschreibt. Abhängig davon, welchen Bootsektor sie infizieren, unterscheidet man zwischen MBR und VBR-Bootkits. Beispiele für MBR-Bootkits sind TDL4 und Olmasco. Während Ersteres den Boot-Code manipuliert verändert Olmasco die Partition-Table. Beide haben jedoch das selbe Ziel: einen komprimierten Kernel-Mode-Treiber aus den Dark Regions zu laden und so die Microsoft Kernel-Mode Signatur-Policy zu umgehen. Aus Olmasco hat sich dann das VBR Bootkit Rovnix entwickelt, welche den Initial Program Loader (IPL) Code verändert.

4.2.3 BIOS/UEFI-Manipulation

Das BIOS bildet den Startpunkt und die Grundlage des gesamten Bootprozesses. Es hat den am tiefsten und direktesten Zugriff auf die Hardware und somit die Kontrolle über das System. Eine genaue Beschreibung des BIOS/UEFI-Bootprozesses sowie der dazugehörigen Komponenten befindet sich in Abschnitt 3.2. Im Allgemeinen erlangen BIOS-Bootkits die Kontrolle über den Bootprozess, indem sie ISA-Module ins BIOS laden. Anschließend werden mehrere Funktionen gehookt, um schadhaften Code im umgeleiteten Bootprozess auszuführen. Dabei kann auch jede nachfolgende Bootkomponente gehookt werden [41]. 2007 veröffentlichte jemand unter dem Deckname IceLord eine detaillierte Beschreibung eines BIOS-Bootkits [66]. Dabei wird das PE-Executable-File IceLord.exe und eine ISA-Module-Datei genutzt, um Schadcode in den BIOS Flash-ROM zu laden. Es werden einige Funktionen gehookt, um eine eigene Windowstreiber-Datei vor dem Betriebssystemstart auszuführen und so das System zu kompromittieren (für weitere Informationen siehe Li u. a. [87]).
UEFI bietet zahlreiche Vorteile gegenüber seinem Vorgänger, dem traditionellen BIOS, unter anderem UEFI Secure Boot. Eine detailliere Beschreibung ist in Abschnitt 6.3 zu finden. Wenn UEFI Secure Boot jedoch deaktiviert ist bietet UEFI, so Bashun u. a. [12] verschiedenste Angriffspunkte für Malware. Einige der Hauptangriffspunkte sind:

- **Unautorisierte Firmwareupdates** können leicht eingespielt werden, wenn UEFI Secure Boot nicht aktiv ist. In diesem Fall wird die Signatur der installierten Pakete nämlich nicht verifiziert. Es kann der UEFI-Runtime-Service verwenden werden, um Updates vom Betriebssystem aus zu starten [12].
- **Existierende DXE Treiber** können manipuliert oder neue, maliziöse Treiber hinzugefügt werden. Neue Treiber können entweder auf der Festplatte oder aber auch auf einem USB-Gerät gespeichert werden. Durch eine Änderung der Ladereihenfolge können schadhafte Treiber in einer frühen Phase des Bootprozesses geladen werden [12].
- **Bootloader-Datei** ersetzen. Normalerweise befindet sich die Bootloader-Datei an einem bekannten Ort der EFI-System-Partition. Diese kann mit administrativen

Rechten ersetzt werden. Es ist auch möglich den Fallback-Bootloader zu ersetzen oder mittels UEFI-Runtime-Service einen eigenen Bootloader (.efi Applikation) einzuspielen [12].

- **GUID Partition Table (GPT)** ist eine Partitionierungs-Tabelle, welche den MBR ersetzt. In UEFI-Plattformen kann sowohl GPT als auch MBR verwendet werden. Diese Partitionierungs-Tabelle können von Bootkits verändert werden [91].

Auch mit aktiviertem UEFI Secure Boot bietet das System Angriffspunkte, zum Beispiel Option-ROMs. Option-ROMs werden während der Plattform-Initialisierung ausgeführt und stellen in der Regel Firmware-Treiber bereit. Netzwerkadapter, Videokarten und Speicher Treiber benötigen zum Beispiel Option-ROMs. Diese können missbraucht werden, um Schadcode zu laden. Die EFI-Spezifikation sieht nicht vor, dass die Signatur von Treibern auf Option-ROMs überprüft werden [12].

Das erste UEFI Bootkit in der realen Welt, LoJax, wurde 2018 von ESET entdeckt [145]. LoJax basiert auf der Anti-Diebstahlsoftware LoJack. Diese hatte eine Schwachstelle, welche es erlaubte, LoJax Bootkit-Code auf das System zu laden. Auch die originale Software LoJack hatte Funktionen im BIOS/UEFI, damit sie nicht einfach entfernt werden konnte. LoJax nutzt eine Schwachstelle im SPI-Flash-Speicher aus, um diesen zu überschreiben und so ein maliziöses UEFI Modul zu integrieren. Das eigentliche Problem war, dass alle die Zugang zur Konfigurationsdatei von LoJack hatten, den URL zum Command und Control Server (C&C) verändern konnten. Dieser war lediglich mit einem One-Byte-XOR-Schlüssel verschlüsselt.

KAPITEL 5

Trusted Platform Module 2.0

Das folgende Kapitel gibt einen Einblick in die wichtigsten Funktionen und Konzepte von TPM 2.0. Zunächst wird auf die allgemeinen Aufgaben von TPM, die Architektur und die Unterschiede zur Vorgängerversion 1.2 eingegangen. Anschließend werden die wichtigsten Konzepte mithilfe von theoretischen Beispielen besser veranschaulicht. Am Ende wird auf die Software eingegangen, um das Hardwaremodul zu nutzen.

5.1 TPM 2.0 Allgemein und Architektur

Das Trusted Platform Module (TPM) ist ein kryptographischer Mikrocontroller, welcher in den meisten Computern bereits eingebaut ist oder einfach ergänzt werden kann [8]. Ein TPM ist nicht die Trusted Computing Base (TCB) (siehe Abschnitt 3.4.1) eines Systems, sondern dient der Überprüfung, ob die TCB kompromittiert wurde [73]. Zu seinen Hauptaufgaben zählen [8]:

- Identifikation von Geräten: Vor der Verwendung von TPMs wurden IP oder MAC Adressen zur Identifizierung von Geräten verwendet. Mittels TPM kann ein Gerät seine Identität eindeutig beweisen. Eine genauere Beschreibung ist in Abschnitt 5.5 zu finden.
- Zufallszahlengenerator (Random Number Generator RNG): TPM bietet einen Hardware-basierte RNG, welcher als Grundlage für die Generierung von sicheren Schlüsseln dient. Diese Funktionalität kann auch von anderen Applikationen verwendet werden, welche richtige Zufallszahlen benötigen.
- Sicheres Speichern von Schlüsseln, auch außerhalb des TPM. Wie in Abschnitt 5.4 genauer beschrieben können Schlüssel verschlüsselt außerhalb des Moduls gespeichert werden.

- Non-Volatile Random Access Memory (NVRAM): Dient als sicherer Speicher für zum Beispiel Rootkeys oder Root-Seeds, siehe Abschnitt 5.4. Daten im NVRAM bleiben auch nach dem Neustart des TPMs persistent.

- Geräteintegrität beglaubigen (Attestation): Ein Gerät kann seine Systemgesundheit gegenüber Dritten mittels Messwerten beglaubigen. Dies erlaubt es zum Beispiel Unternehmen die Korrektheit ihrer Systeme zu überprüfen und eventuell Zugriffe zu beschränken. Mehr dazu im Abschnitt 5.5.

2013 spezifizierte die TCG TPM 2.0 als Nachfolger von TPM 1.2 [122]. TPM 2.0 bietet viele Veränderungen zu 1.2, unter anderem [8]:

- Flexibilität bei der Wahl des kryptographischen Algorithmus. Es können bessere Verfahren verwendet werden. Anstatt SHA-1 kann zum Beispiel SHA-256 als Hash-Algorithmus oder Elliptische-Kurven-Kryptographie (ECC) anstatt RSA eingesetzt werden. Zudem gibt es die Möglichkeit symmetrische Verschlüsselung anzuwenden.

- Erweiterte Autorisierung (Enhanced Authorization): TPM 2.0 erlaubt flexiblere Autorisierungsmethoden mittels Policys, unter anderem Multifaktor- und Multiuser-Authentifizierung. Weitere Informationen zu Enhanced Authorization befinden sich in Abschnitt 5.3.

- Schnelleres Laden von Schlüsseln: durch den Umstieg beim Speichern des Schlüsselmaterials von asymmetrischer auf symmetrischer Verschlüsselung können diese nun schneller in das TPM geladen werden. Schlüssel werden, wie in Abschnitt 5.4 beschrieben, aus Platzgründen verschlüsselt außerhalb des TPM gespeichert.

Die Architektur von TPM 2.0 ist in Abbildung 5.1 zu finden und ist in der TCG Spezifikation [73] definiert. Das Modul beinhaltet Komponenten für die Schlüssel-Generierung, symmetrische- und asymmetrische Verschlüsselung beziehungsweise Signierung, Berechnung von Hashwerten, einen Random Number Generator (RNG), eine Komponente zur Erkennung der Leistung und Spannung, einen Puffer für Input/Output und jeweils eine Komponente für Management, Autorisierung und Ausführung. Der I/O Puffer ist für die Kommunikation zwischen System und TPM zuständig. Die Autorisierungs-Komponente wird aufgerufen, um die Berechtigung eines Befehls zu überprüfen und diesen zu autorisieren. Der Zugriff auf bestimmte Ressourcen bedarf keiner Authentifizierung während andere wiederum zum Beispiel eine Zwei-Faktor-Authentifizierung benötigen (siehe Abschnitt 5.3). Das TPM verfügt über zwei Arten von Speicher, den flüchtigen (volatile) und den nicht-flüchtigen (Non-Volatile Random Access Memory (NVRAM)) Speicher. Im Flüchtigen werden Sessions, PCR-Werte und Schlüssel, welche gerade in Verwendung sind, gespeichert. Daten im flüchtigen Speicher gehen verloren, sobald das TPM keinen Strom hat. Bestimmte Daten, wie zum Beispiel Schlüssel Seeds, müssen jedoch permanent gespeichert werden. Diese Informationen werden im NVRAM hinterlegt. Das TPM muss

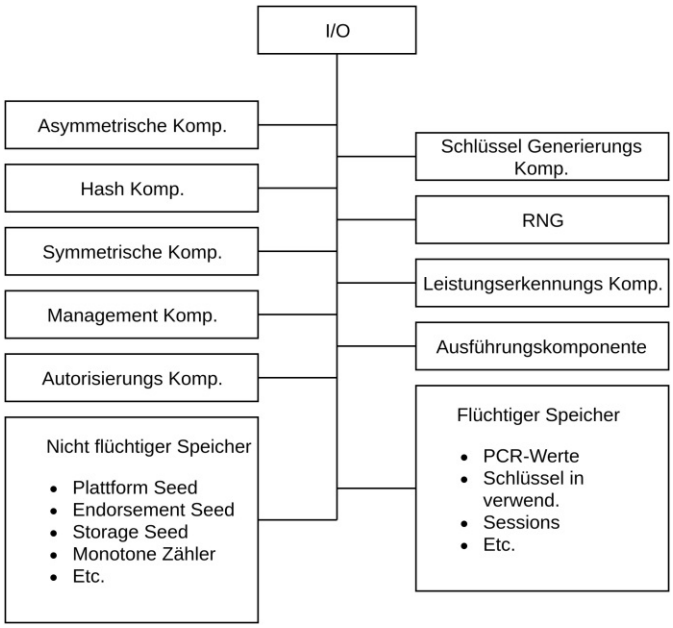

Abbildung 5.1: TPM 2.0 Architektur [122]

über jegliche Veränderung im Leistungszustand des Systems informiert werden. Dabei unterstützt das TPM lediglich die Zustände „Ein“ und „Aus“. Die Ausführungskomponente verarbeitet TPM Befehle und die Managementkomponente zum Verwalten des Moduls, wie zum Beispiel, zum „aktivieren“ [73].

5.2 Platform Configuration Registers (PCRs)

Platform Configuration Registers (PCRs) zählen zu den wichtigsten TPM-Komponenten. Sie erlauben das Speichern von Komponenten-Messwerten und spiegeln somit den Gerätestatus wider. Dabei werden Hashwerte bestimmter Komponenten, sowohl der ausführbaren Dateien als auch der Konfigurationsdateien des Bootprozesses, berechnet und in den Registern gespeichert. Beim Speichern eines neuen Wertes wird dieser mit dem alten Wert verknüpft und erneut gehasht (siehe Gleichung 5.1). Eine maliziöse Komponente kann alle nachfolgenden Messwerte verfälschen, jedoch nicht den Eigenen. Wenn eine Komponente verändert wurde, wird beim Booten ein anderer Hashwert in einem PCR gespeichert. Auch wenn die schadhafte Komponente die Kontrolle erhält, kann sie diesen Messwert nicht revidieren, indem sie das Register direkt mit einem gewünschten Wert überschreibt. Die Komponente kann lediglich versuchen weitere Hashwerte in dieses Register zu schreiben, um den originalen Wert zu erreichen [8]. Die Kollisionsresistenz von Hash-Algorithmen macht dies jedoch sehr unwahrscheinlich (für weitere Informationen zu Hash-Algorithmen siehe Damgård [28]).

$$PCR_{\text{Neu}}{=}Hash(PCR_{\text{Alt}}||berechneterWert) \tag{5.1}$$

Beim Booten eines Systems werden alle PCRs auf ihre Ausgangswerte gesetzt (meist alles Nullen). Der Startpunkt des Messprozesses ist das Root of Trust for Measurement (RTM) gesteuert durch das CRTM (siehe Abschnitt 3.2). Dadurch wird die Chain of Trust aus zwei unterschiedlichen Quellen gestartet. Das RTM wird von der CPU-Herstellerfirma und das CRTM wird von der Plattformherstellerfirma integriert. Normalerweise erweitert das CRTM ein PCR mit dem eigenen Hashwert [73]. CRTM misst die nächste auszuführende Software (z.B: BIOS) und schreibt den Hashwert in ein gerades PCR. Anschließend werden die Konfigurationsdaten gemessen und in ein ungerades Register geschrieben. Die nächste Komponente führt den Messprozess gleich weiter. Ein normaler PC hat 24 Register, andere Systeme können jedoch weit mehr haben [8]. Die ersten acht bis elf Register werden für Messungen des Bootprozesses verwendet. Eine Auflistung der möglichen PCR-Belegungen bei einem Measured Boot sind in Tabelle 5.1 zu finden [2]. Je nach System und verwendeter Komponenten können die PCR-Belegungen abweichen. Die Tabelle dient als Orientierung, wo die Messwerte bestimmter Komponenten gespeichert werden können.
Im ersten Register PCR 0 kann der Status von S-CRTM und bestimmten Teilen des BIOS abgelegt werden. Im darauffolgenden PCR 1 folgen dann die entsprechenden Konfigurationen. In PCR 2 werden der UEFI Applikations-Code und die UEFI-Treiber gespeichert. PCR 4 und 5 beinhalten den Hashwert des Master Boot Record (MBR) und der MBR-Konfigurationen. Register 6 ist Plattformherstellerfirma spezifisch. Die UEFI Secure Boot Policy wird in das Register 7 geschrieben. Linux IMA wird in PCR 10 gespeichert und Debug Informationen in PCR 16. Zum Aufbauen der Dynamic Chain of Trust und den D-RTM werden die Register 17 bis 22 verwendet (siehe tboot 6.5). Im PCR 23 werden Applikations-relevante Werte gespeichert [8].

PCR	Komponenten
0	S-CRTM, BIOS, Embedded Option ROMs
1	Host Platform Konfigurationen
2	UEFI-Treiber und Applikations-Code
3	UEFI-Treiber und Applikations Konfigurationen
4	UEFI Boot Manager Code (normalerweise Master Boot Record (MBR)) und Bootversuche
5	MBR-Konfiguartionen und GPT/Partition Table
6	Herstellerfirma spezifisch
7	UEFI Secure Boot Policy
10	Linux IMA
16	SRTM Debug
17	ACM, MLE, tboot Policy, Kernel-Messungen (DRTM)
18	Öffentlicher Schlüssel um ACM zu signieren, tboot Policy und Kontrollwerte (DRTM)
23	Applikationen

Tabelle 5.1: Mögliche PCR Belegungen (Basierend auf [8], [57] und [2])

Diese PCR-Werte werden entweder lokal mit vertrauenswürdigen Werten verglichen oder signiert an eine Überprüfungsstelle geschickt. Da der private Schlüssel zum Signieren nur dem TPM bekannt ist, kann gewährleistet werden, dass die Werte nicht manipuliert wurden [8].

5.3 Enhanced Authorization (EA)

Mittels TPM gespeicherte Objekte, wie Schlüssel, aber auch TPM-Befehle sind durch einen Autorisierung-Mechanismus geschützt. Unter TPM 2.0 unterscheidet man drei Mechanismen: Passwort-, HMAC- und Policy-Autorisierung. Die Zugriffskontrolle mittels Policys wird auch Enhanced Authorization genannt. Dadurch kann der Zugriff auf ein Objekt mithilfe von Policys beschränken werden. Bevor der Zugriff auf dieses Objekt erlaubt wird, überprüft das TPM, ob die definierte Policy erfüllt ist [8].
Policys sind sehr vielfältig und können schnell komplex werden. Eine Policy kann bestimmte Systembedingungen oder ausgeführte Aktionen voraussetzen, um den Zugriff auf das Objekt zu gestatten, zum Beispiel den Wert eines PCRs. Auch wenn eine Policy noch so komplex ist, wird sie mittels eines Hashwertes, dem sogenannten *authPolicy* (aP)-Wert, repräsentiert. Um Zugriff auf ein Objekt zu erlangen, muss eine Policy-Session erstellt werden. Anschließend werden eine Reihe von Policy-Befehlen zum Verifizieren an das TPM geschickt. Jede Session wird durch einen *PolicyDigest* (Hash) repräsentiert. Bei jeder erfolgreichen Überprüfung durch das TPM wird dieser aktualisiert. Die Berechnung funktioniert ähnlich wie bei den PCR-Werten (siehe Gleichung 5.2). Zu Beginn besteht

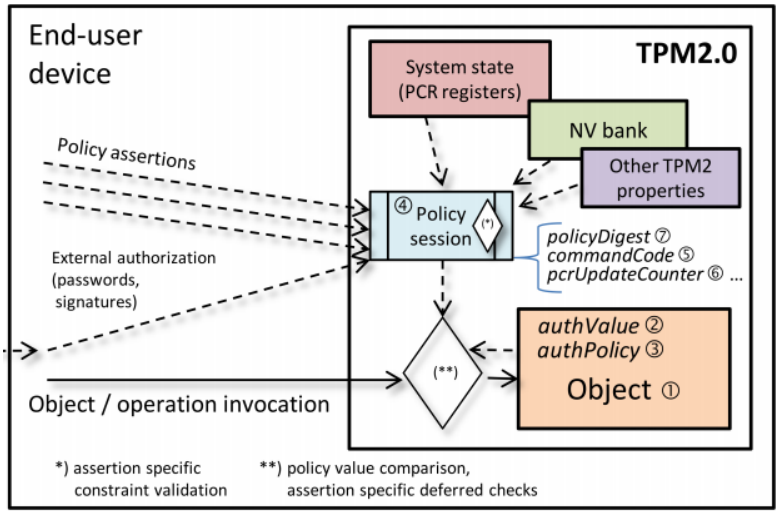

Abbildung 5.2: TPM 2.0 Enhanced Authorization Modell [101]

das *PolicyDigest* vollständig aus Nullen [8].
Ein Überblick des Enhanced Authorization Modells ist in Abbildung 5.2 zu finden. Der *commandCode* identifiziert, so Nyman u. a. [101] den Policy-Befehl und die *commandArgs* sind die verwendeten Argumente.

$$PolicyDigest_{\text{Neu}}{=}Hash(PolicyDigest_{\text{Alt}}||commandCode||commandArgs) \tag{5.2}$$

Ein einfaches Beispiel einer Policy von Shao u. a. [117]: Alice kann mit einem Schlüssel sk_A nur fünf-mal etwas signieren und nur Bob kann mit seinem sk_B die Signatur überprüfen. Zudem muss das System für die Überprüfung in einem bestimmten Zustand sein (PCR-Werte).

Zu Beginn muss ein Policy-Wert (aP) festlegt werden.

$$aP := HASH(0...0||'OR'||Z_1||Z_2) \tag{5.3}$$
$$Z_1 := HASH(Z_\text{N}||'Signed'||PK(sk_\text{A})||'Alice') \tag{5.4}$$
$$Z_2 := HASH(Z_\text{N}||'Signed'||PK(sk_\text{A})||'Alice') \tag{5.5}$$
$$Z_\text{N} := HASH(Z_\text{C1}||'NV'||NVC1||LT||5) \tag{5.6}$$
$$Z_\text{P} := HASH(Z_\text{C2}||'PCR'||PCR1||v) \tag{5.7}$$
$$Z_\text{C1} := HASH(0...0||'CmdCode'||CC\_Sign) \tag{5.8}$$
$$Z_\text{C2} := HASH(0...0||'CmdCode'||CC\_Certify) \tag{5.9}$$

Eine detaillierte Beschreibung vom Alice-Ablauf findet man in Tabelle 5.2. Die Schritte von Bob sind ähnlich.

$TPM2\_Create(aP) \rightarrow kB$ $TPM2\_Load(kB) \rightarrow hK$ $NVC1counter = 0$	Generierung eines Signierschlüssels im TPM. Setzt NVC1 Zähler auf 0 und legt fest, dass jede Verwendung von sk_A den Zähler erhöht

Alice möchte mit dem Schlüssel etwas signieren. Damit sie das kann müssen folgende Schritte erfolgreich ausgeführt werden:

$TPM2\_StartAuthSession() \rightarrow (hS, nt)$	Alice startet eine neue Policy Session mit dem PolicyDigest 0...0. nt ist die Session Nonce und hs ist das PolicyDigest.
$TPM2\_CommandCode(hS, CC\_Sign)$	Updated Sessions CommandCode zu CC_Sign. policyDigest wird auf Z_{C1} gesetzt (5.8)
$TPM2\_PolicyNV(hS, NVC1, LT, 5)$	Überprüfung ob der NVC1 kleiner als 5 ist. Wenn die Bedingung gültig ist setzt das TPM die PolicyDigest auf ZN (5.6).
$\sigma A = Sig(nt, `Alice`, sk_A)$	Alice erzeugt eine Signatur. Wobei die policy reference (polR) auf den Beispielwert „Alice“ gesetzt wird. Dies kann nur die autorisierenden Entität in diesem Fall Alice machen. Die polR ist normalerweise komplexer.
$TPM2\_PolicySigned(hS, \sigma A, `Alice`, PK(sk_A))$	Alice schickt die Signatur zum Validieren and das TPM. Wenn valide wird das PolicyDigest auf Z_1 gesetzt (5.4)
$TPM2\_PolicyOR(hS, < Z_1, Z_2 >)$	Überprüfen ob das aktuelle PolicyDigest auf der Liste <Z_1, Z_2>ist. Wenn ja, setzt das TPM das PolicyDigest auf aP (5.3)
$TPM2\_Sign(hK, hS, msg)$	Wenn das PolicyDigest der Session mit dem aP Wert des Schlüssels übereinstimmt und der commandCode der Session mit dem autorisierten commandCode CC_Sign übereinstimmt, wird der Befehl ausgeführt und die Nachricht signiert. Anschließend wird der NVC1 erhöht.

Tabelle 5.2: AuthPolicy Beschreibung (Basierend auf [117])

Das TPM kann Daten oder Schlüssel verschlüsselt außerhalb des Moduls speichern und eine Entschlüsselung erst dann wieder erlauben, wenn das System in einem bestimmten Zustand ist. Diesen Vorgang nennt man Sealing oder Sealed Binding. Durch die Bindung an PCR-Werten war ein BIOS-Update sehr aufwändig. Alle Daten referenzierend auf die entsprechenden PCR-Werte mussten ent- und dann wieder verschlüsselt werden [8]. Auch alle Policys, die PCR-Überprüfungen beinhalten, wurden unbrauchbar gemacht, da sie nicht direkt verändert werden können. Bei TPM 2.0 besteht dieses Problem immer noch, wenn die Policys komplett fixiert wurden. Jedoch kann unter TPM 2.0 eine flexible Policy mit dem Befehl *TPM2_PolicyAuthorize()* erstellt werden. Dabei bindet man die Policy nicht an einen fixen PCR-Wert, sondern daran, ob dieser von einer gültigen Instanz signiert wurde [73]. Eine Herstellerfirma genehmigt zum Beispiel mehrere BIOS-Versionen. Sie signiert die daraus resultierenden PCR-Werte und stellt diese zur Verfügung. Wenn die Policy nun ein *TPM2_PolicyAuthorize()* für die Herstellerfirma beinhaltet, können die Daten auch nach einem Update weiterhin entschlüsselt werden[8].

5.4 Schlüssel des TPM

Eine der Grundaufgaben des TPMs ist die Generierung und Speicherung von Schlüsseln. Wie man in Abbildung 5.1 erkennen kann, unterstützt TPM 2.0 sowohl symmetrische als auch asymmetrische Schlüssel. TPM-Schlüssel können eine Hierarchie bilden, wobei die höheren Schlüssel ihre Kinder verschlüsseln. Die obersten Schlüssel nennt man Primary Keys. Zur Generierung von Schlüsseln wird eine Key Derivation Function (KDF) verwendet. Diese kann, anhand eines geheimen Primary Seeds und eines öffentlichen Templates, Primary Keys erzeugen. Die Key Derivation Function (KDF) ist eine reproduzierbare Funktion. Für den gleichen Primary Seed und das gleiche Template wird immer derselbe Schlüssel erzeugt [8]. Ein Primary Seed ist ein großer Zufallswert, welcher von der Herstellerfirma im TPM erzeugt wird und das TPM nie verlässt. TPM 2.0 hat drei Primary Seeds für drei verschiedene Zwecke. Der Endorsement Primary Seed wird zum Generieren von Endorsement Key (EK), der Platform Primary Seed für Platform Key (PK) und der Storage Primary Seed für Storage Root Key (SRK) verwendet [73]. Es können unlimitiert Primary Keys, zum Beispiel mit verschiedenen Algorithmen, erstellt werden. Dank der KDF müssen diese nicht alle gespeichert werden, sondern können bei Gebrauch neu berechnet werden. Dadurch bilden die Primary Seeds die kryptographischen Wurzeln der TPM Schlüsselhierarchie [8].
SRK und die daraus abgeleiteten Kinder werden verwendet, um Daten verschlüsselt außerhalb des TPMs zu speichern (Sealing). PK werden vom BIOS und SMM verwendet und nicht von Personen. EKs werden zur Identifizierung des Systems verwendet und bilden somit die Basis für Root of Trust for Reporting (RTR). Die Architekturschaffenden des ersten TPM waren sehr auf Privacy bedacht. Es sollte möglich sein zu beweisen, dass ein Schlüssel von einem TPM erzeugt und geschützt wird, ohne zu wissen um welches TPM es sich handelt. Dafür wurde ein Direct Anonymous Attestation (DAA)-Protokoll entwickelt, welches Attestation Identity Keys (AIKs) unterstützt. Attestation Identity Keys (AIKs) sind Pseudo-Identitäts-Schlüssel, welche PCR-Berichte signieren. Es können

unendlich viele AIK erstellt und nach ihrer Verwendung vernichtet werden. Beim Erstellen eines AIK-Zertifikates weiß die Privacy Certificate Authority (PCA) zu welchem Endorsement Key (EK) der AIK gehört und kann das auch beglaubigen. Dadurch kann bewiesen werden, dass ein AIK von einem TPM stammt, ohne dessen genaue Identität preiszugeben [8]. Damit diese Signatur überprüft werden kann, braucht jeder EK ein zugehöriges beglaubigtes Zertifikat. Die Herstellerfirma generiert mittels Endorsement Primary Seed Schlüsselpaare der gängigsten kryptographischen Algorithmen. Anschließend wird ein Zertifikat für die öffentlichen Schlüssel erstellt, signiert und in das TPM gespeichert. Die Schlüsselpaare werden dann wieder gelöscht, um nicht wertvollen Speicher zu verschwenden [73].
DAA bietet zwei Signaturmodi an: Voll-Anonymisierten und Pseudo-Anonymisierten. Da dem TPM nur sehr begrenzte Ressourcen und Leistung zur Verfügung stehen, haben Yang u. a. [151] den DAA-Signierprozess optimiert. Durch ihr Konzept wird die Ausführungszeit des TPM reduziert, sodass DAA-Signaturen nicht mehr Ressourcen als eine normale Signatur benötigen.

5.5 Remote Attestation

Die PCRs spiegeln, so Müller [97], den aktuellen Zustand des Systems wider. Um die Vertrauenswürdigkeit des Systems zu beweisen, können die PCR-Werte als signierter Bericht ausgelesen werden. Diesen Vorgang nennt man Attestation oder Remote Attestation, wenn die Berichte an ein externes System gesendet werden. Das externe System kann anhand der Informationen Entscheidungen, wie zum Beispiel für ein Zugangskontrollsystem, treffen. Dafür schickt das externe System mittels *TPM2_Quote*-Befehl eine Challenge (Zufallszahl) und eine Liste der gewünschten PCRs an das TPM. Dieses generiert einen Bericht und signiert ihn mit dem AIK. Wenn keine PCR-Werte angefordert werden, wird lediglich die Challenge signiert, dies wird als Platform Authentication bezeichnet. Damit ein externer Service diesen Dienst beanspruchen kann, muss das System eine Netzwerkapplikation anbieten. Diese wird Attestation Client genannt und greift mittels TCG Software Stack (TSS) auf das TPM zu. Abbildung 5.3 beschreibt diesen Vorgang.

Das Problem bei PCR-Werten ist, dass verschiedene Komponenten in einem Register zusammengefasst werden können. Ändert sich eine dieser Komponenten ändert sich das Ergebnis im PCR und man weiß nicht, welche der Komponenten die Ungleichheit verursacht hat. Hierfür könnte wie in Abbildung 5.3 das Eventlog mitschicken [48]. Während des Systemstartes werden, so Yao [152], Firmwaremesswerte nicht nur in die PCRs eingefügt sondern auch in das Eventlog eingetragen. Allerdings darf nicht nur das Eventlog herangezogen werden, da sich dieses außerhalb des TPM befindet und nicht geschützt ist. Die Messwerte in den Eventlogs können jedoch von der anfragenden Stelle in derselben Reihenfolge verknüpft werden wie in Listing 5.1 beschrieben und das Ergebnis kann mit den effektiven PCR-Werten verglichen werden.

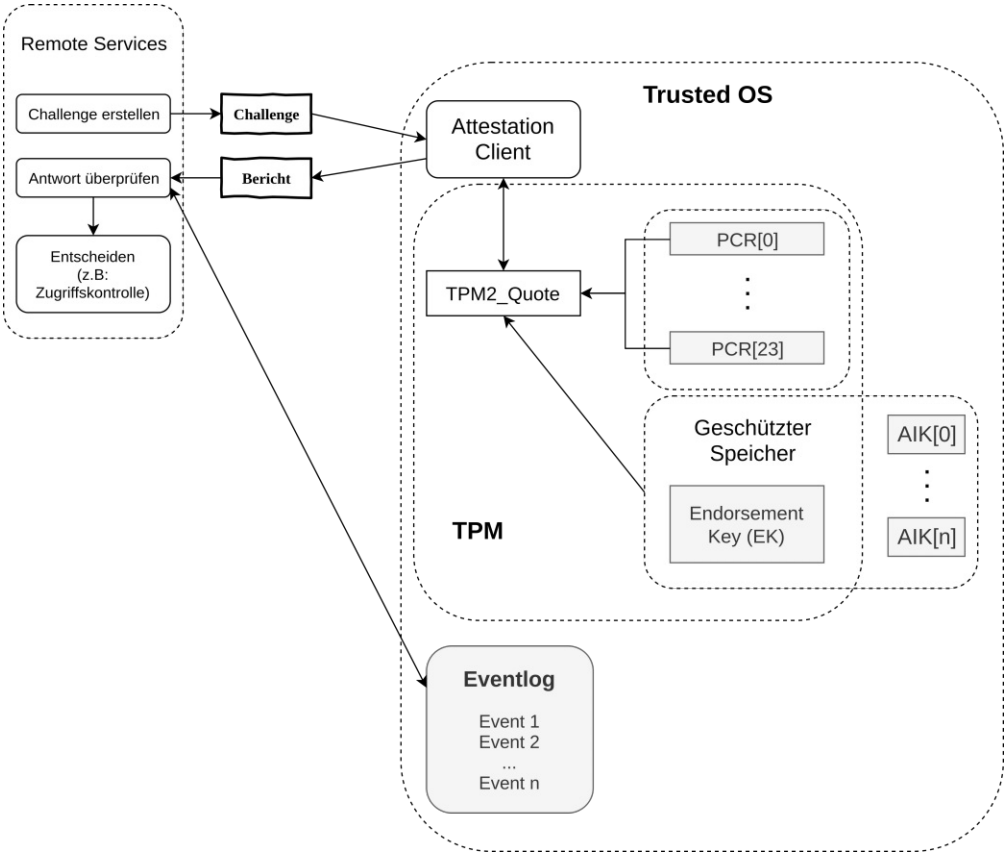

Abbildung 5.3: Remote Attestation (Basierend auf [97], [152])

5.6 Local Attestation

Eine lokale Attestation nach demselben Prinzip wie bei Remote Attestation, wo am Ende des Bootprozesses mittels einer Software bestimmte PCR-Werte gegen gültige Werte verglichen werden, macht laut Garrett [48] keinen Sinn. Denn wenn eine Komponente kompromittiert wurde, kann sie zwar die PCR-Werte nicht verfälschen, jedoch aber die Software manipulieren, welche zum Überprüfen verwendet wird. Zudem möchten Anwendende nicht ihre Zugangsdaten in ein System eingeben, welches möglicherweise kompromittiert wurde. Ohne einen Remote Server können Anwendende, wie in Abschnitt 5.3 beschrieben, bestimmte Aktionen an Bedingungen knüpfen und so kontrollieren, ob das System kompromittiert wurde.
Es kann zum Beispiel die Entschlüsselung der Festplatte an den Systemzustand geknüpft werden. Dafür wird eine Policy erstellt, welche für die Freigabe des benötigten Schlüssels gewisse PCR-Werte voraussetzt [8]. Dabei muss, so Garrett [43], darauf geachtet werden, welche PCR-Werte verwendet werden, denn PCR-Werte können sich auch ohne maliziöses Einwirken aus verschiedensten Gründen ändern. Zum Beispiel dann, wenn der Bootloader oder der Kernel upgedatet werden oder wenn sich die UEFI Datenbank, wie

in Abschnitt 6.3 beschrieben, ändert. Eine detaillierte Beschreibung des angewandten Überprüfungsmechanismus ist in Abschnitt 7.3 zu finden.

5.7 TCG Software Stack tpm2-tss

Um leichter mit dem TPM kommunizieren zu können, spezifizierte TCG den TCG Software Stack (TSS). Dieser Software-Standard erlaubt es Applikationen über Application Programming Interfaces (APIs), TPM-Funktionalitäten zu nutzen. Der TSS besteht aus mehreren Schichten, um sowohl für High-End-Systeme als auch Ressourceneingeschränkte Low-End-Systeme nutzbar zu sein. In Abbildung 5.4 findet man die Schichten nach TCG-Spezifikation. In den nachfolgenden Abschnitten werden alle Schichten genauer erklärt. Für diese Arbeit wird die tpm2-tss Implementierung [135] verwendet. Es ist ein von TCG unterstütztes Open-Source-Projekt, das zum Zeitpunkt dieser Arbeit noch weiterentwickelt wird.

5.7.1 TPM-Gerätetreiber

Der TPM-Treiber handhabt alle Lese- und Schreibzugriffe auf das TPM. Der betriebssystemspezifische Treiber nimmt eine Sequenz von Byte-Befehlen von höheren Schichten 5.4 entgegen und gibt diese an das TPM weiter. In den Plattformspezifikationen der Herstellerfirma ist definiert, welche Schnittstellen vom Treiber verwendet werden. Für PCs wird entweder die FIFO- (First In First Out) oder die Response Buffer (Command/Response Buffer (CRB))-Schnittstelle verwendet. Die CRB-Schnittstelle ist neu für TPM 2.0 und wird verwendet, wenn TPM-Instanzen einen Speicher teilen, um Befehle und Antworten auszutauschen [8]. Die CRB-Schnittstelle ist als linearer Speicherbereich definiert. Wobei aufrufende Software immer aufeinanderfolgende Adressen zum Füllen des Buffers verwenden muss [132].

5.7.2 TPM Access Broker (TAB) und Resource Manager (RM)

Der TPM Access Broker (TAB) verwaltet synchrone Zugriffe von mehreren Prozessen auf das TPM. Damit wird gewährleistet, dass ein Prozess einen Befehl beenden kann, ohne von einem anderen unterbrochen zu werden. Zudem erlaubt der TAB einem Prozess nur auf seine eigenen Objekte und Sessions zuzugreifen.
Der Resource Manager ist eine optionale Schicht, welche den TPM-Kontext verwaltet. Er arbeitet ähnlich wie ein Virtual Memory Manager in einem Betriebssystem. Er schiebt Objekte, Sessions und Sequences in und aus dem begrenzten TPM-Speicher, um Befehle auszuführen. Dabei fängt er die Befehle an das TPM ab, ermittelt wie viel Ressourcen dafür benötigt werden, macht im TPM-Speicher genügend Platz und lädt die entsprechenden Ressourcen. Wenn der RM nicht implementiert ist, müssen die höheren Schichten diese Aufgaben übernehmen 5.4. In den meisten Fällen wird der RM mit dem TPM Access Broker in einer Komponente kombiniert. Diese ist dann nur für ein TPM zuständig [8].

5.7.3 TPM Command Transmission Interface (TCTI)

Das TPM Command Transmission Interface (TCTI) bietet verschiedene Schnittstellen für höhere Schichten an 5.4. Wie in Abbildung 5.4 zu sehen ist, kann TCTI sich zu mehreren Arten von TPMs verbinden. Dabei benötigen Hardware-TPMs, Firmware-TPMs, virtuelle TPMs, remote TPMs und TPM-Simulatoren jeweils andere Schnittstellen. Zudem unterscheidet man bei Schnittstellen zwischen TIS (TPM Interface Specification)- und Command/Response Buffer (CRB)-Schnittstellen [128]. Die FIFO-Schnittstelle ist TIS-konform. TPM 2.0 erlaubt es Befehle zu stornieren, nachdem sie zum TPM geschickt wurden [8].

5.7.4 Marshaling/Unmarshaling API (MUAPI)

Marshaling/Unmarshaling API (MUAPI) wird sowohl von Enhanced System API (ESAPI) als auch System API (SAPI) verwendet und befindet sich daher in einem eigenen Bereich. MUAPI erzeugt TPM Byte-Streams (Marshalling) und zerlegt die Antwort-Streams (Unmarshalling) [8].

5.7.5 System API (SAPI)

Die System API (SAPI)-Schicht bietet eine Low-Level-Schnittstelle für Applikationen, wie zum Beispiel Firmware, Betriebssystem etc. [128]. SAPI, als unterste Schicht der drei User-Application Programming Interfaces (APIs) benötigt am wenigsten Platz und die geringsten Ressourcen. Das direkte Ansprechen der SAPI erfordert auf der einen Seite detailliertes Fachwissen der TPM 2.0 Befehle und Architektur, bietet aber auf der anderen Seite mehr Flexibilität bei der Gestaltung von Befehlen (für weitere Informationen siehe TCG-Spezifikation [129]). Man unterscheidet vier Gruppen von SAPI-Befehlen [8]:

- **Command Context Allocation**: mit Befehlen dieser Kategorie werden System-Kontexte *sysContext* neu aufgebaut, wiederverwendet oder wieder frei gegeben.
- **Command Preparation**: bestimmte TPM 2.0 Befehle müssen vorbereitet werden, um sie an das TPM zu schicken. Zum Beispiel bei HMAC-Berechnungen oder Aufrufe mit verschlüsselten Parametern müssen diese vorbereitet (siehe Marshalling 5.7.4) werden, bevor sie an das TPM geschickt werden.
- **Command Execution**: diese Kategorie führt die eigentlichen Befehle aus, sendet sie an das TPM und empfängt die Antworten.
- **Command Completion**: bieten das Gegenstück zu Command Preparation-Funktionen. Sie übernehmen das Post-Verarbeiten von Befehlen, wie etwa HMAC berechnen und Parameter entschlüsseln.

5.7.6 Enhanced System API (ESAPI)

Die Enhanced System API (ESAPI)-Schicht liegt direkt über dem SAPI. Sie vereinfacht Applikationen das Senden von individuellen TPM-Befehlen. ESAPI macht die Handhabung von Sessions einfacher, insbesondere von sicheren Sessions, welche Parameter ent- und verschlüsseln, HMAC-Operationen oder Policys verwenden. Trotz allem benötigt man ein detailliertes TPM-Verständnis, um ESAPI verwenden zu können. Daher wird es auch nur von „expert applications“ verwendet [128].

5.7.7 Feature API (FAPI)

Die oberste Schicht, das Feature API (FAPI), bietet eine abstrakte Schnittstelle für Software-Entwickelnde. Diese können TPM-Funktionalitäten, auch ohne genaue Kenntnisse der Low-Level-Prozesse, nutzen. Diese API wird von den meisten Programmen verwendet und soll die Anzahl der TPM-Aufrufe minimieren. Dabei wurden Standard-Konfigurations-Profile angelegt, sodass Anwendende, zum Beispiel beim Erstellen eines Schlüssels nicht explizit den Algorithmus, die Schlüssellänge und andere Parameter wählen müssen. Jedoch können auch explizit Konfigurationsdateien erstellt und ausgewählt werden [8].

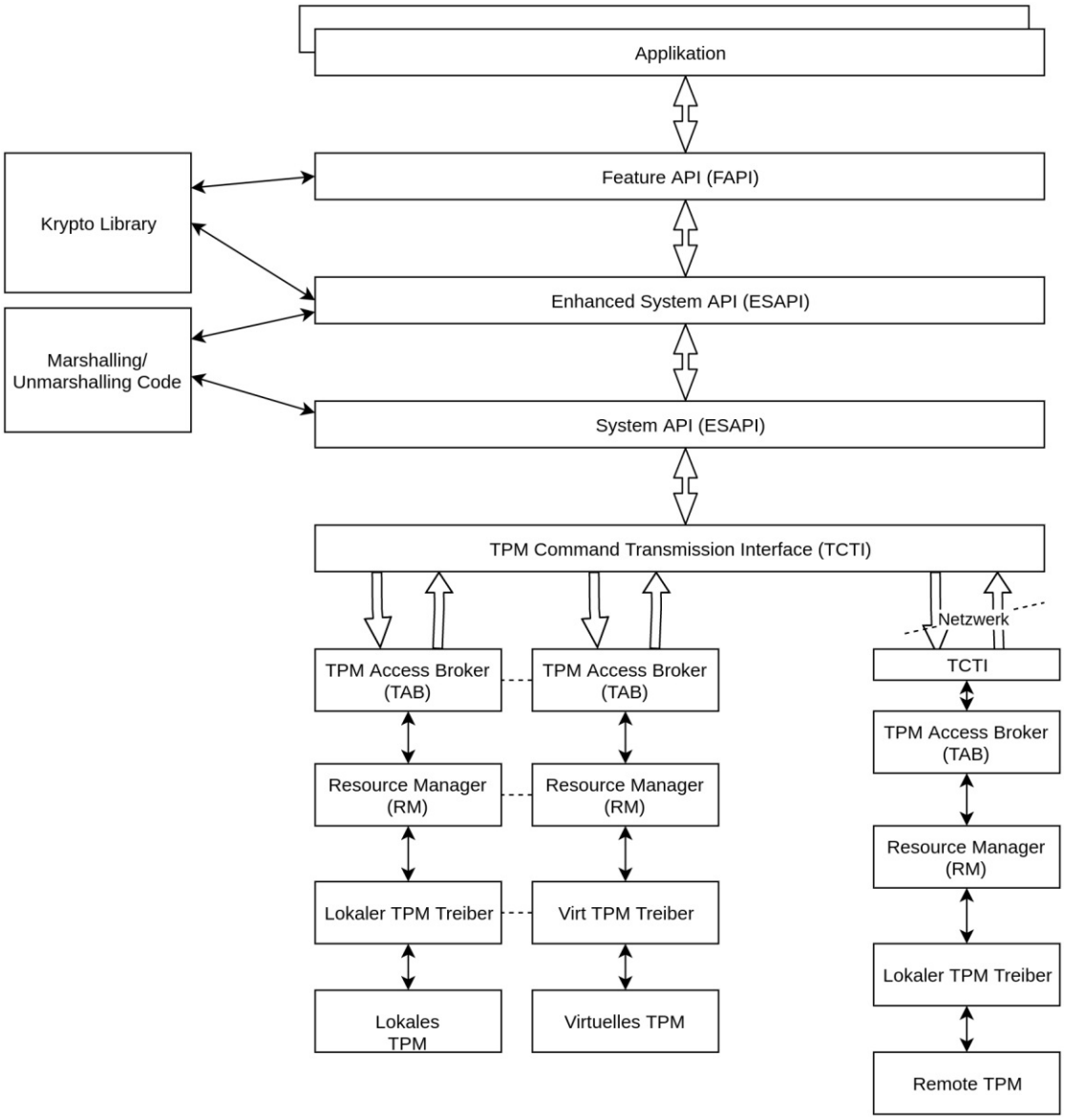

Abbildung 5.4: TCG Software Stack [128]

5.8 TPM 2.0 Tools (tpm2-tools)

Basierend auf der tpm2-tss Implementierung wurden zahlreiche Kommandozeilenbefehle entwickelt, um unter Linux mit TPM 2.0 kommunizieren zu können. Für Linux gibt es zum Beispiel das Projekt tpm2-tools [134], welches noch laufend weiterentwickelt wird. Daher können die Befehle je nach Version variieren. In diesem Kapitel werden einige Befehle und Szenarien erklärt, welche für ein besseres Verständnis des Konzeptes im Kapitel 7 hilfreich sind. Alle Informationen aus diesem Abschnitt stammen aus der Git-Dokumentation der tpm2-tools Version 4.3.0 [134].

5.8.1 PCR auslesen

Der Befehlt **tpm2_pcrread** gibt alle PCR-Werte in allen verfügbaren Hashalgorithmen aus, wenn keine Argumente übergeben werden. Es können jedoch auch explizit der Hashalgorithmus und die Register bestimmt werden zum Beispiel mit `tpm2_pcrread sha256:0,1,7` [134].

Ein bestehendes TPM-Eventlog kann mit dem Befehl **tpm2_eventlog** ausgegeben werden [134]. Die Messwerte der einzelnen Komponenten werden zum Beispiel in die Datei `/sys/kernel/security/tpm0/binary_bios_measurements` geschrieben. Diese werden durch den `tpm2_eventlog Dateiname` Befehl je nach Register gruppiert und pro Register in der entsprechenden Reihenfolge, wie in Listing 5.1, verknüpft[38]. Das Ergebnis sollte mit den aktuellen PCR-Werten übereinstimmen.

5.8.2 PCR-Policy

In Listing 5.1 wird anhand mehrere Befehle gezeigt, wie eine PCR-Policy erstellt wird und wie der Zugriff auf Daten mit dieser beschränkt wird. Der Befehl in Zeile 1 erzeugt ein Objekt unter der Hierarchie „Owner“ und schreibt den Kontext in die Datei `primary.ctx`. Anschließend wird eine Policy anhand der Register 0,1,2 und 3 erstellt. Das PolicyDigest wird in der Datei `pcr.policy` gespeichert. In Zeile 3 wird dann das Geheimnis (in diesem Fall nur das Wort „secret“) an die zuvor erstellte Policy gebunden. Mit `tpm2_unseal` kann das Geheimnis wieder aus dem TPM geholt werden. Dafür müssen jedoch die PCR dieselben Werte haben, wie beim Erstellen der Policy [134].

Listing 5.1: TPM 2.0 PCR-Policy [134]

```
1 tpm2_createprimary -C o -c primary.ctx
2 tpm2_policypcr -l sha256:0,1,2,3 -L pcr.policy
3 echo 'secret' | tpm2_create -C primary.ctx -L pcr.policy -i-\ -
    ↪ u seal.pub -r seal.priv -c seal.ctx -Q
4 tpm2_unseal -c seal.ctx -p pcr:sha256:0,1,2,3
```

5.8.3 Authorize-Policy

Wie in Abschnitt 5.3 erwähnt, können Policys auch daran gebunden werden, ob die PCR-Werte eine gültige Signatur haben. Zu Beginn wird, wie in Listing 5.2, mit OpenSSL ein Schlüsselpaar erstellt. Der private Schlüssel wird offline sicher verwahrt, während der öffentliche Schlüssel ungeschützt in das TPM geladen wird [134].

Listing 5.2: Schlüsselpaar erstellen und in TPM laden [134]

```
1 openssl genrsa -out signing_key_private.pem 2048
2 openssl rsa -in signing_key_private.pem -out signing_key_public
    ↪ .pem -pubout
3 tpm2_loadexternal -G rsa -C o -u signing_key_public.pem -c
    ↪ signing_key.ctx -n signing_key.name
```

Anschließend wird, wie in Listing 5.3, eine PCR-Policy erstellt und wie in Zeile 4 signiert. Für dieses Beispiel wurde nur PCR 0 verwendet. Die Session zum Erstellen dieser Policy wird nicht mehr benötigt und kann mit dem Befehl in Zeile 3 aus dem TPM-Speicher entfernt werden [134].

Listing 5.3: PCR-Policy erstellen und signieren [134]

```
1 tpm2_startauthsession -S session.ctx
2 tpm2_policypcr -S session.ctx -l sha256:0 -L pcr.policy
3 tpm2_flushcontext session.ctx
4 openssl dgst -sha256 -sign signing_key_private.pem -out pcr.
    ↪ signature pcr.policy
```

In Listing 5.4 wird die ausschlaggebende Authorize-Policy erstellt. Sie definiert, dass eine bestimmte Auswahl an PCR-Werten (definiert durch pcr.policy) mit einem bestimmten Schlüssel (signign_key.name) signiert sein müssen [134].

Listing 5.4: TPM 2.0 Authorize-Policy [134]

```
1 tpm2_startauthsession -S session.ctx
2 tpm2_policyauthorize -S session.ctx -L authorized.policy -i pcr
    ↪ .policy -n signing_key.name
3 tpm2_flushcontext session.ctx
```

Die erstellte Authorize-Policy wird nun verwendet, um das Geheimnis aus Zeile 1 des Listings 5.5 zu binden. Nach diesen Befehlen ist der erste Teil, das Erstellen und Binden der Policy, fertig [134].

Listing 5.5: Geheimnis mittels Authorize-Policy binden [134]

```
1 tpm2_createprimary -C o -g sha256 -G rsa -c prim.ctx
2 tpm2_create -g sha256 -u sealing_pubkey.pub -r sealing_prikey.
    ↪ pub -i- -C prim.ctx -L authorized.policy <<< "secret␣to␣
    ↪ seal"
```

Im zweite Teil, wie in Listing 5.6, erfolgt das Überprüfen der Policy und die Preisgabe des Geheimnisses. In diesen Befehlen wird die Signatur einer PCR-Policy überprüft. Die Signaturdatei (pcr.signature) wurde in Listing 5.3 Zeile 4 erstellt. Der öffentliche Schlüssel muss, falls nicht bereits gemacht, in das TPM geladen werden , Listing 5.2 Zeile 3. Die hier angegebene PCR-Policy (pcr.policy) kann auch neu erstellt werden. Die Überprüfung der Signatur liefert ein Ticket (`verification.tkt`), welches zusammen mit der neuen PCR-Policy und dem Namen des öffentlichen Schlüssels für die Authorize-Policy verwendet wird. Anschließend wird der private und öffentliche Teil des Sealing-Objekts in das TPM geladen und anhand des validen Session-Kontextes wird das Geheimnis wieder preisgegeben [134].

Listing 5.6: Signatur überprüfen und Geheimnis freigeben [134]

```
1 tpm2_verifysignature -c signing_key.ctx -g sha256 -m pcr.policy
    ↪  -s pcr.signature -t verification.tkt -f rsassa
2
3 tpm2_startauthsession \--policy-session -S session.ctx
4 tpm2_policypcr -S session.ctx -l sha256:0 -L pcr2.policy
5 tpm2_policyauthorize -S session.ctx -L authorized.policy -i
    ↪ pcr2.policy -n signing_key.name -t verification.tkt
6 tpm2_load -C prim.ctx -u sealing_pubkey.pub -r sealing_prikey.
    ↪ pub -c sealing_key.ctx
7 tpm2_unseal -p "session:session.ctx" -c sealing_key.ctx
8 tpm2_flushcontext session.ctx
```

5.9 Angriff auf TPM 2.0

In diesem Abschnitt werden zwei unterschiedliche Schwachstellen von TPM 2.0 und entsprechende Angriffsszenarien erklärt. Bei der ersten Schwachstelle werden die PCR-Werte manipuliert, sodass ein kompromittiertes System vertrauenswürdig erscheint. Bei zweiterem wird mittels Seitenkanalattacke der private Schlüssel ermittelt.

5.9.1 PCR-Manipulation

Die Sicherheit des Static Chain of Trust beruht auf der Unverfälschbarkeit der PCR-Werte. Han u. a. [57] fanden bei ihren Untersuchungen der TPM 2.0-Spezifikation eine Schwachstelle im S-RTM (siehe Abschnitt 3.4.2), die es erlaubt PCR-Werte zu manipulieren. Normalerweise können die PCR-Werte 0 bis 15 nur beim Rebooten oder Starten des Systems auf die Anfangswerte zurückgesetzt werden, andernfalls werden sie nur erweitert (siehe Gleichung 5.1). Han u. a. gehen bei ihrem Angriffsmodell von einem Measured-Bootprozess aus, dessen Ergebnisse von einer externen Stelle korrekt überprüft werden. Sie gehen davon aus, dass Angreifende bereits Root-Rechte haben und somit den Bootloader und den Kernel manipulieren können.
Die TCG-Spezifikationen definieren genau, wie der Systemzustand beibehalten werden

kann, wenn sich der Energiezustand des Systems ändert. Han u. a. verwenden für ihren Angriff den ACPI S3 Sleep Status. ACPI ist ein Standard für die Energieverwaltung von Plattformen. Er bestimmt welche Komponenten in welchem Energiezustand abschalten. In diesem Energiezustand schalten sich alle Komponenten (auch das TPM), bis auf den RAM, aus. Bevor das System in den Ruhezustand fährt, weist das Betriebssystem das TPM an, seinen aktuellen Zustand im Non-Volatile Random Access Memory (NVRAM) zu speichern (Schritt (1) in Abbildung 5.5). Wenn das System den Ruhezustand verlässt, weist das BIOS/UEFI das TPM an, den Zustand wiederherzustellen (5, Abb. 5.5) und das Betriebssystem zu starten (6, Abb. 5.5). Das Problem bei TPM 2.0 ist, dass wenn es keinen Wert zum Wiederherstellen gibt, das TPM zurückgesetzt wird. Dadurch werden alle PCRs auf vordefinierte Werte gesetzt. Abbildung 5.6 zeigt eine grobe Übersicht des Angriffsablaufs. Zu Beginn bootet das System in einem vertrauenswürdigen Zustand. Angreifende manipulieren dann den Bootloader und den Kernel. Bei einem Neustart des Systems werden dann die „guten“ Hashwerte des vorherigen Bootvorganges aus den BIOS/UEFI-Eventlogs gelesen und temporär im RAM gespeichert. Anschließend wird das System in den Ruhezustand geschickt, ohne den TPM-Zustand zu speichern (möglich aufgrund des manipulierten Kernels). Beim Beenden des Ruhezustandes können die zurückgesetzten PCR-Werte einfach mit den zwischengespeicherten Werten des RAM erweitert werden und so einen vertrauenswürdigen Zustand vortäuschen [57].
Diese Schwachstelle findet sich lediglich in der TPM 2.0-Spezifikation. In der 1.2 Version geht das TPM in einen Fehlermodus und kann bis zu einem Neustart nicht mehr verwendet werden. In den BIOS/UEFI-Einstellungen kann der S3-Ruhezustand deaktiviert werden. Eine bessere Lösung wäre jedoch eine Änderung der TPM 2.0-Spezifikation und ein Firmwareupdate der existierenden TPM 2.0-Chips [57]. Zum Zeitpunkt dieser Arbeit wurde die TPM 2.0-Spezifikation noch nicht angepasst [136].

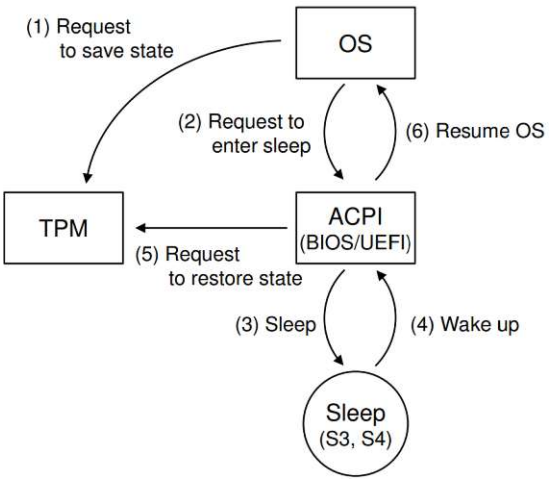

Abbildung 5.5: ACPI Sleep Prozess mit TPM 2.0 [57]

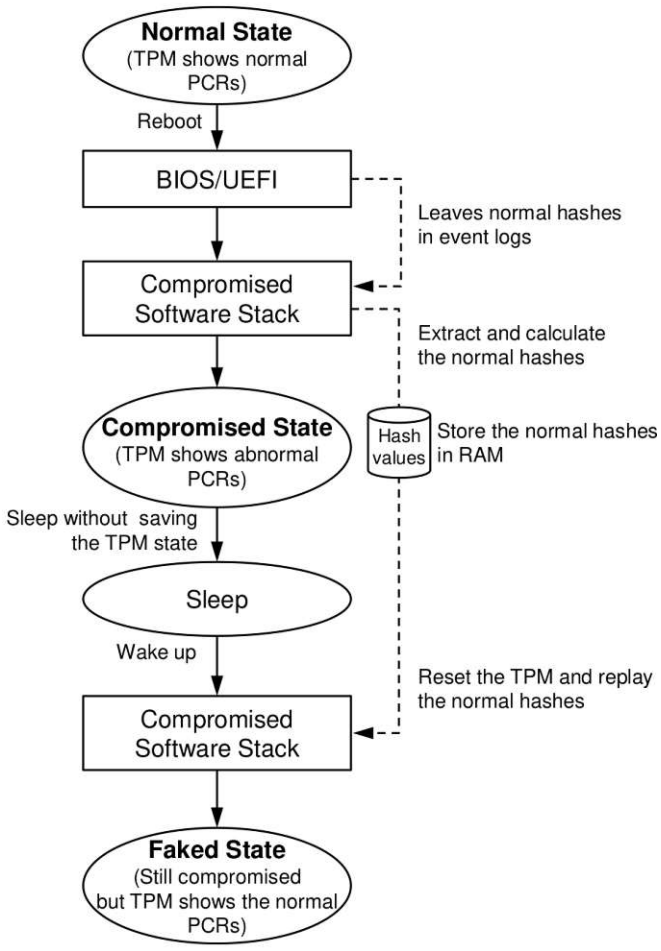

Abbildung 5.6: Angriffsszenario für die SRTM Schwachstelle [57]

5.9.2 Seitenkanalattacke

Seitenkanalattacke sind, so Gui u. a. [54], nicht invasive Angriffe auf kryptographische Geräte. Dabei wird das System bei kryptographischen Operationen beobachtet und das Verhalten für unterschiedliche Eingaben analysiert. Für die statistische Analyse kann die Laufzeit, der Energieverbrauch und andere Faktoren berücksichtigt werden, um sicherheitsrelevante Informationen wie etwa geheime Schlüssel, zu ermitteln.
In ihrer Arbeit analysieren Moghimi u. a. [96] das Laufzeitverhalten von sowohl Firmware- als auch Hardware TPMs der Version 2.0. Sie zeigen auf, dass das Laufzeitverhalten der Module bei kryptographischen Operationen wie zum Beispiel Signieren variiert. Abhängig vom verwendeten Schlüssel gibt es Diskrepanzen in der Ausführungszeit des Algorithmus.

Mittels einer Black-Box Zeitanalyse, bei Singnaturoperationen für elliptische Kurven,

konnten sie [96] für bestimmte TPM 2.0 Modelle den privaten Schlüssel ermitteln. Die Schwachstelle kann auch remote über das Netzwerk ausgenutzt werden. Dabei analysieren sie, wie in Abbildung 5.7 dargestellt, die Verarbeitungszeit eines VPN Servers. Der VPN Server verwendet eine Intel Firmware-TPM (fTPM) um den privaten Schlüssel zu speichern. Beim Angriff führt der maliziöse Client zuerst einen Handshake (INIT) aus, um die relevanten Sicherheitsparameter auszutauschen und einen Diffie-Hellman Schlüsselaustausch durchzuführen.
Der Diffie-Hellman Schlüsselaustausch ist, so Partala [103], eine der ältesten und meist verbreitetsten Pulbic-Key-Kryptoverfahren. Dies erlaubt es zwei Parteien einen gemeinsamen geheimen Schlüssel über einen öffentlichen Kanal zu ermitteln. Dafür einigen sich beide Parteien öffentlich auf eine Basis g und einen Modulus p. Beide wählen nun separat eine Geheimzahl (in der Graphik x und y). Anschließend senden sie an den Kommunikationspartner $g^{Geheimzahl}$. Der berechnete Schlüssel wird von beiden Seiten anhand von g^x und g^y wie in Listing 5.10 berechnet.

$$(g^x)^y \ mod \ p = (g^y)^x \ mod \ p \tag{5.10}$$

Dieser Schlüssel wird nun für die verschlüsselte Kommunikation des zweiten Handshakes verwendet. Im zweiten Handshake verifizieren beide Parteien die Integrität, indem sie die Zufallszahl (Nonce) des jeweils anderen signieren (n_{Client}, n_{Server}). Der Client misst dabei die Ausführungszeit des Servers und speichert diese zusammen mit den Signaturen. Anschließen wird die Session verworfen. Diese Schritte werden so oft wiederholt, bis der Client ausreichend Daten für eine statistische Analyse zur Ermittlung des privaten Server-Schlüssels hat [96].
Die Schwachstelle wurde 2019 Intel gemeldet (CVE-2019-11090) und behoben. In ihrer Arbeit wurde auch das Hadware-TPM Modell untersucht, welches im praktischen Teil dieser Arbeit verwendet wird. Für dieses konnten sie jedoch keine Seitenkanallachwachstellen in Bezug auf Ausführungszeit feststellen [96].

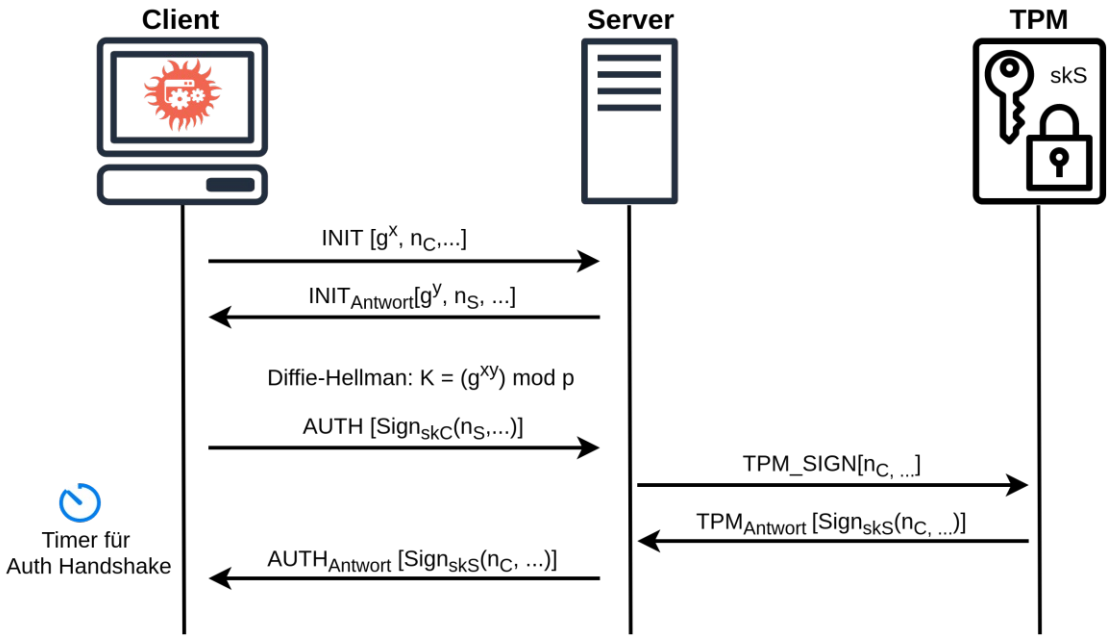

Abbildung 5.7: Seitenkanalattacke auf VPN Server [96]

KAPITEL 6

Bestehende Lösungsansätze zum sicheren Booten

Dieses Kapitel beschreibt einige bestehende Lösungen und konkrete Ansätze, um Trusted Computing umzusetzen. Dabei werden sowohl erste Lösungsansätze beschrieben, die teilweise die Grundlagen für heutige Technologien bilden, als auch weiterentwickelte Konzepte, die in modernen Systemen integriert werden.

6.1 Integrity Measurement Architecture (IMA)

Die Integrity Measurement Architecture (IMA) ist ein Sicherheitsmechanismus, welcher 2004 von Sailer u. a. [113] veröffentlicht wurde und die Prinzipien von Secure und Measured Boot auf Betriebssystem-Ebene erweitert, indem Applikationen und verwendete Dateien gemessen werden (mehr Informationen zu Secure und Measured Boot in Abschnitt 3.4.1). IMA ist eine Kernel-basierte Lösung, welche 2009 mit der Version 2.6.30 in den Linuxkernel integriert wurde. Sie erlaubt sowohl eine lokale Überprüfung der Messwerte als auch eine Remote Attestation. IMA bietet verschiedene Module zur Integritätsprüfung, unter anderem IMA Measurement und Appraisal (weitere Informationen im IMA Wiki [154]).

IMA Measurement weitet das Measured Boot-Prinzip auf Betriebssystem-Ebene aus. Dabei werden die Messwerte chronologisch in einer Liste im Kernel, dem sogenannten Stored Measurement Log (SML), gespeichert. Die Werte können aggregiert auch in einem TPM gespeichert werden. IMA verwendet dafür das PCR 10 [154]. Ein externer Server kann dann mittels Remote Attestation den Systemzustand abfragen. Das System schickt dafür die SML und den mittels TPM signierten PCR-Wert [113].

IMA Appraisal erweitert das Secure Boot-Konzept auf Betriebssystem-Ebene. Dabei werden die Komponenten gemessen und mit „guten“ Werten verglichen. Die guten Werte

werden als Attribute an die Dateien angehängt. Zusätzlich zu Hashwerten können auch Signaturen angehängt werden [154].

6.2 TrustedGRUB

TrustedGRUB erweitert den Standard-GRUB (beschrieben in Abschnitt 3.2.2), um TPM-Schnittstellen. Dabei unterstützt der Bootloader TPM-Funktionen, um die Integrität des Bootprozesses fortzusetzen. Die offizielle TrustedGRUB2-Implementierung [111] misst den GRUB2 Kernel mit Modulen und Konfigurationen, den Betriebssystem-Kernel inklusive Module und die dazugehörigen Parameter. Der TrustedGRUB MBR-Bootcode wird bereits vorher vom BIOS gemessen. Zum Zeitpunkt dieser Arbeit unterstützt das offizielle TrustedGRUB2 lediglich TPM der Version 1.2 mit SHA-1 Hash-Algorithmus.

6.3 UEFI Secure Boot

Es gibt verschiedene Ansätze, den Bootprozess abzusichern. Abhängig von der Bootprozessebene, in der die Funktion implementiert ist, unterscheidet Matrosov u. a. [91] zwischen OS Secure Boot (Ebene des OS-Loaders), UEFI Secure Boot (UEFI-Firmware-Ebene) und Platform Secure Boot (Hardwareebene). UEFI Secure Boot versucht Rootkits zu verhindern, indem es nur vertrauenswürdige Komponenten ausführt. Eine Komponente gilt als vertrauenswürdig, wenn sie entweder eine vertrauenswürdige Signatur hat oder in der internen Datenbank als vertrauenswürdig eingestuft wurde. Diese Datenbank und der UEFI-Bootprozess ist in Abbildung 6.1 dargestellt (für genaue Beschreibung des UEFI-Bootprozesses siehe Abschnitt 3.2).
Beim Starten eines Systems wird als erstes die Platform Initialization (PI)-Firmware ausgeführt. Diese konfiguriert den RAM und beginnt die Plattform zu initialisieren. Erst wenn die UEFI-Applikationen mit dem Initialisieren der Hardware beginnen, startet Secure Boot. Um Vertrauen während des Bootprozesses aufzubauen, wendet UEFI Secure Boot verschiedene Schlüssel, Signaturdatenbanken und Policys an. Damit die Signaturdatenbanken vor unautorisierten Veränderungen geschützt sind, werden sie mit einem Schlüssel signiert. Dieser Schlüssel wird Key Exchange Key (KEK) genannt und ist wiederum mit dem Platform Key (PK) signiert [91].
Ein System hat genau einen Platform Key (PK), kann jedoch mehrere KEKs haben. Der öffentliche Teil des PK wird standardmäßig von der Plattformherstellerfirma in das System integriert, während der private Schlüssel in der Kontrolle der Herstellerfirma bleibt. Die KEK werden von bestimmten Certificate Authoritys (CAs), wie etwa Microsoft oder Hardwareherstellerfirmen, ausgestellt. Der öffentliche Schlüssel wird wiederum im System hinterlegt [91]. Anwendende mit physischem Zugriff auf das Gerät können, so Paul u. a. [104], auch eigene Schlüssel in die UEFI-Firmware importieren, beziehungsweise Bestehende löschen. Um eine vollständige Kontrolle über UEFI Secure Boot zu erlangen, muss jedoch mindestens der PK, der KEK und gegebenenfalls die Signaturdatenbank *db* ausgetauscht werden.
Zum Überprüfen einer Komponente wird der Hashwert berechnet und die digitalen

Signaturen mit dem entsprechenden Zertifikat kontrolliert. Jedoch reicht es nicht aus, wenn eine Komponente eine gültige Signatur hat. Die Signatur muss zudem von einem autorisierten Schlüssel stammen. Das Zertifikat, inklusive öffentlichen Schlüssel befinden sich an einer bestimmten Stelle der ausführbaren Komponente. Nach erfolgreicher Signaturprüfung, wird kontrolliert, ob der Schlüssel in der *db* Datenbank vorhanden ist. Die *db* Datenbank beinhaltet eine Liste autorisierter öffentlicher Schlüssel und Hashwerte vertrauenswürdiger Komponenten. Da einige Komponenten aufgrund ihres Formats keine Signatur beinhalten, muss es eine weitere Möglichkeit geben diese zu überprüfen. Dabei werden die Hashwerte berechnet und mit den Werten in der *db* Datenbank verglichen. In Abbildung 6.1 ist eine weitere Datenbank, *dbx*, zu sehen. Diese beinhaltet verbotene oder abgelaufene Zertifikate und Hashwerte. Komponenten mit diesen Werten werden nicht ausgeführt [91].
Da die *db* und *dbx* Datenbanken, unter der Kontrolle des KEK stehen, können Anwendende diese nicht erweitern, sondern lediglich Bootkomponenten installieren, welche eine gültige Signatur und einen gültigen Hashwert in der Datenbank haben. Linux Bootloader, wie GRUB haben keine solche Signatur. Um UEFI Secure Boot unter Linux verwenden zu können ohne eigene Schlüssel generieren zu müssen, kann ein First-Stage Bootloader namens Shim verwendet werden. Dieser hat, wie in Abschnitt 3.2.1 beschrieben, eine gültige Microsoft Signatur und erlaubt das Laden von Linux Bootloadern und Kerneln [11].
Da der Root of Trust im PI-Flashspeicher liegt und nicht in einer Hardwarekomponente, ist UEFI Secure Boot anfällig für Firmware-Rootkits. Sobald die PI-Firmware kompromittiert wird, ist die komplette Chain of Trust gebrochen [91]. Es sind bereits einige Angriffsmöglichkeiten für UEFI Secure Boot bekannt. Bulygin u. a. [18] präsentierten folgenden Ansatz: der PK wird im NVRAM in einer EFI-Variable gespeichert. Wenn diese EFI-Variable korrumpiert wird, und dadurch nicht korrekt geladen werden kann, geht das System in den $SETUP\_MODE$ und die *SecureBoot* Variable wird auf $DISABLED$ gesetzt. Sobald Angreifende Schreibzugriff auf den PI-Flashspeicher haben, kann der Überprüfungsmechanismus manipuliert werden. Dabei wird der $DxeImageVerificationLib$ Code verändert, sodass für jede Datei EFI_SUCESS zurückgegeben wird. EFI_SUCESS bedeutet, dass die Authentifizierung erfolgreich war und die Datei ausgeführt werden kann.

6.4 Intel Boot Guard

Um sicherzustellen, dass die UEFI-Firmware nicht manipuliert wurde, muss die Chain of Trust nach unten auf Hardwareebene erweitert werden. Dies kann auf zwei Arten geschehen, Measured Boot oder Verified Boot. Ein System kann auch beide Arten implementieren. Bei Measured Boot werden die Komponenten gemessen und in den TPM PCRs gespeichert. Bei Verified Boot wird die digitale Signatur der Komponenten überprüft. Der Schlüssel zum Überprüfen der UEFI-Firmware befindet sich sicher in einem unveränderbaren Bereich der CPU [91].
Die Chip-Herstellerfirma kann diese Methoden einbauen. Eine Beispiellösung dafür ist

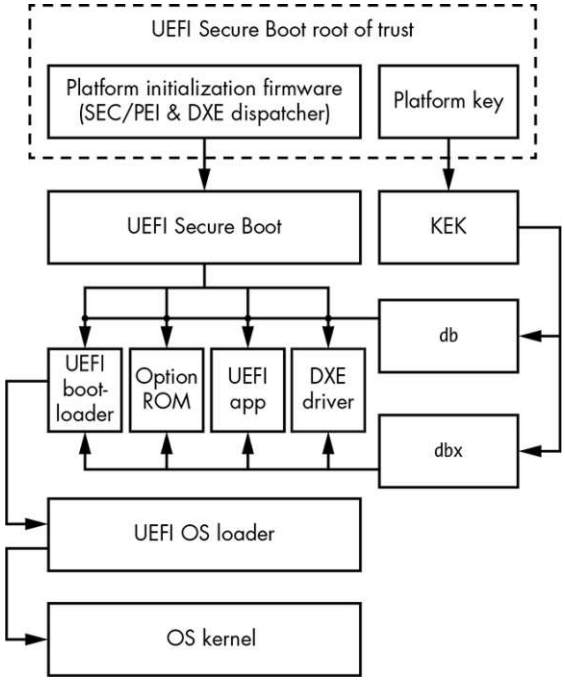

Abbildung 6.1: Normale UEFI Secure Boot-Verifizierungsprozess [91]

Intel Boot Guard. Intel Boot Guard ist seit der 4. Generation in Intel Chips integriert und bietet sowohl Measured als auch Verified Boot an [100]. Die PC-Herstellerfirma, so Hoffman [58], kann wählen, welche der beiden Methoden sie implementiert. Dies kann später nicht revidiert werden. Wenn die Firma Verified Boot wählt, schreibt sie den öffentlichen Schlüssel unveränderbar in die Hardware. Dadurch kann nur noch UEFI-Firmware installiert werden, die von der Herstellerfirma signiert wurde. Laut Garrett [45], müssen sich Herstellerfirmen zwischen Systemsicherheit und Systemfreiheit entscheiden. Wenn jedoch Measured Boot implementiert wird, kann eine beliebige Firmware installiert werden. Diese muss signiert werden und der dazugehörige Schlüssel muss im TPM vorhanden sein.
Intel Boot Guard ist laut Kuzminykh u. a. [82] sehr komplex und kann leicht falsch implementiert oder konfiguriert werden. Eine Studie von 2018 ergab, dass viele Herstellerfirmen ihren öffentlichen Schlüssel nicht in das Field-Programmable-Fuse (FPF) schreiben und Angreifende ihren eigenen Schlüssel dort platzieren können [91]. Das Problem sollte mit Intel BIOS Guard gelöst worden sein, da dieses jeglichen Schreibversuch vom Betriebssystem oder SMM auf den PI-Flashspeicher unterbinden. Jedoch kann auch dieser Mechanismus falsch konfiguriert oder deaktiviert werden, um die Bootzeit zu verkürzen oder BIOS Updates zu erleichtern [82].

6.5 Intel TXT und tboot

Intel Trusted Execution Technology (Intel TXT) ergänzt Intel-Prozessoren und -Chips, um das System sicherer zu machen. Intel TXT ist kein Ersatz für ein TPM. Im Gegenteil, es speichert seine Messungen im TPM. Intel TXT verwendet eine Kombination aus S-RTM und D-RTM aus Abschnitt 3.4.2, um ein vertrauenswürdiges System zu gewährleisten. Der statische Teil misst die Plattformkonfigurationen und der dynamische Teil die Software inklusive Konfigurationen und Policys [40]. Die Abbildungen 6.2 und 6.3 zeigen den Ablauf von Intel TXT und tboot. Währen Abbildung 6.2 sowohl die Static und Dynamic Chain of Trust darstellt, geht Abbildung 6.3 genauer auf tboot ein. Der Bootprozess startet mit einem Mikrocode (uCode), eingebettet in einen Intel-Prozessor. Dieser überprüft und startet das Authenticated Code Module (ACM). ACM ist ein von Intel signiertes Modul, welches in einem geschützten Bereich der CPU ausgeführt wird. Das ACM setzt die Chain of Trust fort, indem es Teile des BIOS misst und das Ergebnis in PCR0 schreibt. Der Flash-Speicher des BIOS wird zum Beispiel ausgelassen, da dieser sich auch in normalen Bootsituationen ändern kann [8]. Der weitere Verlauf des Static Chain of Trust und die Funktion des MBR ist wie in Abschnitt 3.4.2 erläutert.
Wenn das Betriebssystem in einen Vertrauensmodus booten möchte, startet der Dynamic Chain of Trust. Dafür führt es einen speziellen Prozessorbefehl ($GETSEC[SENTER]$) aus, welcher das SINIT ACM in derselben Weise, wie das BIOS ACM überprüft. Nach erfolgreicher Überprüfung misst das SINIT ACM den Trusted OS Code [40]. TCG bezeichnet als Trusted OS ein Betriebssystem, das einen sicheren Startvorgang hatte und daher mit mehr Rechten arbeiten kann [40]. Die Policy Engine in SINIT ACM überprüft, ob die Plattform die Launch Control Policy (LCP) erfüllt. Dabei wird der gemessene Trusted OS Code und die PCR-Werte mit einer Liste vertrauenswürdiger Werte verglichen. Bei einer Übereinstimmung wechselt das Betriebssystem in den sicheren Modus [8]. Da Vertrauen subjektiv ist, bezieht sich Intel mit Trusted OS auf das Measured Launch Environment (MLE) [40]. Bei einem falschen Wert kann, so Sharkey [118], entweder der Bootprozess abgebrochen oder mit einer Warnung fortgesetzt werden. Dieses Verhalten kann in der Policy festgelegt werden. Anfangs ist LCP mit der Herstellerfirmen-Policy ausgestattet, diese kann aber von Anwendenden mit einer eigenen Policy erweitert werden [72]. Dadurch kann zum Beispiel, ein beliebiger Kernel installiert werden, solange die LCP erweitert wird. Intel TXT unterstützt zwar TPM 2.0 auch unter Linux, ist jedoch nicht für jedes System anwendbar. Zudem gab es in den vergangenen Jahren einige Angriffe auf tboot und Intel TXT [118, 150, 147, 148, 57]. Wojtczuk u. a. [148] umgehen mit ihrem Ansatz Intel TXT und LCP zum Beispiel, indem sie eine Schwachstelle im SINIT-Modul ausnutzen. Es ist ihnen möglich, jeden Hypervisor zu laden und als vertrauenswürdig darzustellen. Intel veröffentlichte Ende 2011 einen Patch dafür [69].

Trusted Boot (tboot) [39] ist eine Open Source Pre-Kernel/VMM Modul, das auf Intel TXT aufbaut. Wie man in Abbildung 6.2 sieht, ist tboot in den Dynamic Chain of Trust integriert und wird vom Bootloader als Kernel ausgeführt. Um diesen zu starten wird der Prozessorbefehl $GETSEC[SENTER]$ ausgeführt, welcher anschließend die Kontrolle an tboot weitergibt, um einen gemessenen und verifizierten Kernel/VMM-Start auszuführen.

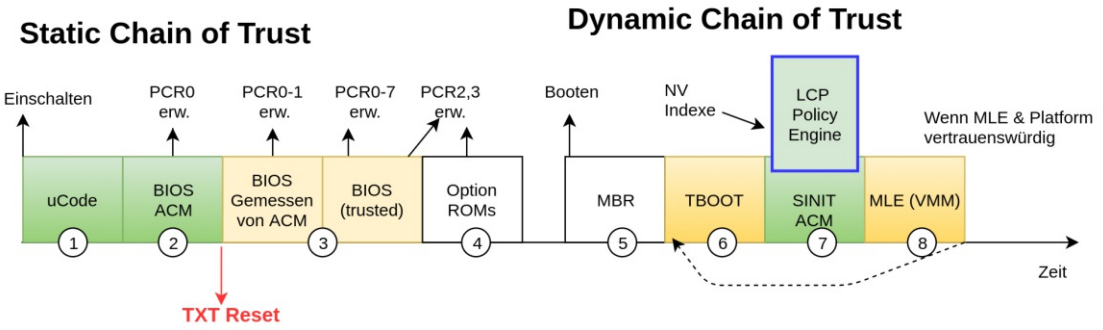

Abbildung 6.2: Intel TXT Bootprozess mit Static und Dynamic Chain of Trust (Basierend auf [8], [40])

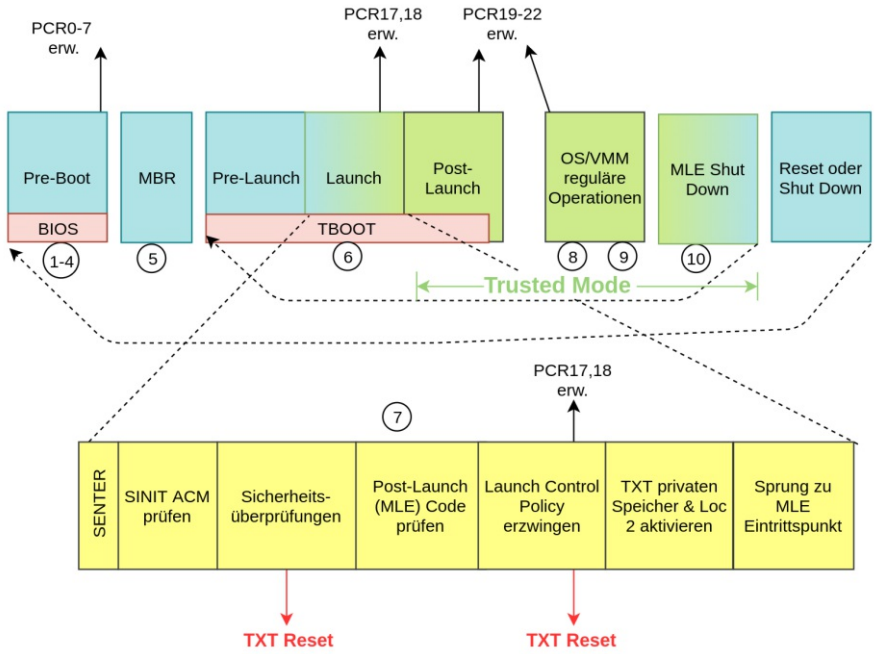

Abbildung 6.3: tboot, Ausschnitt aus Measured Boot-Prozess [8]

Auch tboot verwendet LCP, um die Integrität von Komponenten zu verifizieren. Die Policys sind komplett unter der Kontrolle von Anwendenden und können nach Wunsch angepasst werden. Es kann zum Beispiel eine eigene Launch Policy zur Verifizierung des Kernels und des initrd erstellt werden [24].
Da tboot als Bootloader-Modul verwendet wird, ist es anfällig für Rootkit-Angriffe [118]. Han u. a. [57] nutzen in ihrer Arbeit eine Schwachstelle in tboot aus, um einen kompromittierten Kernel zu laden. Der Angriff ähnelt dem in Abschnitt 5.9.

6.6 AMD Secure Virtual Machine und TrenchBoot

Eine ähnliches Konzept wie Intel TXT und tboot gibt es auch für AMD Prozessoren. Secure Viratual Machine (SVM) ist eine Dynamic Root of Trust for Measurement (D-RTM)-Implementierung für AMD Prozessoren. Das Synonym für tboot ist für AMD TrenchBoot. TrenchBoot ist ein Projekt für D-RTM für Linux, welches sowohl Intel TXT als auch AMD SVM unterstützt. Ähnlich wie Intel TXT wechselt SVM mittels CPU-Befehl *SKINIT* in den geschützten Modus. Dabei wird der AMD Secure Loader (SL) gemessen und in PCR 17 geschrieben. Nach erfolgreicher Kontrolle, prüft und initialisiert dieser wie in Abbildung 6.4 die TrenchBoot LandingZone. Diese übernimmt die Kontrolle und misst den vertrauenswürdigen Bootloader ebenfalls in PCR 17 [152]. Eine Beschreibung der restlichen Komponenten aus Abbildung 6.4 ist in Abschnitt 3.2 zu finden.

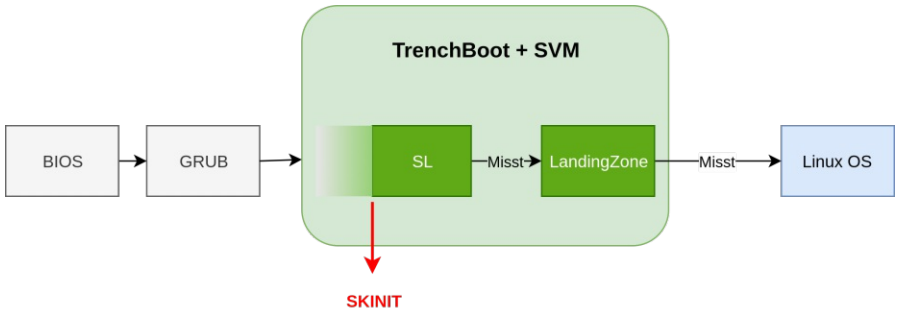

Abbildung 6.4: Measured Boot-Prozess AMD SVM and TrenchBoot [152]

KAPITEL 7

Sicherheitsmaßnahme TPM 2.0

In diesem Kapitel wird ein Konzept für einen sicheren Linux Bootprozess mit TPM 2.0 vorgestellt. Dabei wird darauf geachtet, dass Komponenten wie UEFI, Bootloader, Kernel aber auch das Betriebssystem aktualisiert werden können, ohne die lokale Attestation zu brechen. Des Weiteren wird versucht das Konzept so nutzungsfreundlich wie möglich zu gestalten.
Für die praktische Ausführung wird ein HP 250 G7 Notebook mit einem TPM 2.0 Chip von Infineon [1] verwendet.

7.1 TPM 2.0 kompatible Bootkomponenten

Wie in Abschnitt 3.2 beschrieben gibt es verschiedene BIOS-Firmwarearten. Moderne Firmware beruht meist auf der UEFI-Spezifikation und bieten damit zahlreiche Vorteile gegenüber dem herkömmlichen BIOS. Die Firmware wird bereits von der Herstellerfirma integriert. TCG spezifiziert für TPM 2.0 ausschließlich eine UEFI-Schnittstelle [2]. Dabei wird ein „native“ UEFI benötigt. Sowohl der Legacy- als auch der CSM-Modus wird nicht von TPM 2.0 unterstützt [7]. SeaBIOS hat die TCG Schnittstellen erweitert, um TPM 2.0 zu unterstützen. SeaBIOS ist eine Open Source BIOS-Implementierung, welche häufig für Emulatoren verwendet wird. Sie kann jedoch auch mittels Coreboot für Linux Desktop-Systeme eingesetzt werden (weitere Informationen in der Coreboot Dokumentation [27]). Obwohl Coreboot eine Open Source-Alternative für UEFI wäre, wird für das Konzept dieser Arbeit ein Standard-UEFI eingesetzt (UEFI ohne Legacy-Modus, siehe 3.2).
Die nächste Komponente im Bootprozess ist der Bootloader. Wie in Abschnitt 3.2.1 beschrieben, kann unter Linux, um UEFI Secure Boot nutzen zu können, der First-Stage Bootloader Shim eingesetzt werden. Wenn Anwendende die volle Kontrolle über UEFI Secure Boot haben möchten, können auch eigene UEFI-Schlüssel eingesetzt werden. In

[1] Infineons TPM Chip SLB 9670VQ2.0

diesem Fall wird Shim nicht benötigt. Anwendende müssen dann jedoch entweder jede Komponente, bei jedem Update neu signieren, oder Schlüssel von Dritten, wie etwa Fedora installieren (eine genaue Beschreibung dieses Vorgangs ist im Linux Journal [104] zu finden). Für diese Arbeit wird zugunsten der Nutzungsfreundlichkeit, in Variante 1 in Abschnitt 7.3.1, Shim eingesetzt. Für Second-Stage Bootloader gibt es eine große Auswahl und zahlreiche Abwandlungen in Git-Repositorys. Wie in Abschnitt 3.4.1 beschrieben ist TrustedGRUB2 BIOS basiert und unterstützt lediglich TPM 1.2. Daher wird es für diese Arbeit nicht in Betracht gezogen.
Die häufigsten Linux Distributionen verwenden GRUB2 als Bootloader bei der Standard-Installation. In Tabelle 7.1 sind die beliebtesten Linux Distributionen des Jahres 2020 laut DistroWatch [30] gelistet. Sechs davon verwenden GRUB2 in ihrer Standard-Installation. Nach ausführlichen Recherchen in diversen Git-Repositorys wurde festgestellt, dass TPM 2.0 relevanten Änderungen durch Garrett [53] in das Hauptverzeichnis von GRUB2 integriert wurden. Dadurch unterstützt die Version 2.04 von GRUB2 sowohl TPM 1.2 als auch 2.0. Es gilt zu überprüfen, ob diese offizielle Version alle TPM relevanten Änderungen beinhalten oder ob eine Alternative, zum Beispiel jene von Garrett [44] verwendet werden sollte.
TPM 2.0 wird ab dem Linux Kernel der Version 3.20 [83] unterstützt. Es könnten für diese Arbeit verschiedenste Linux Betriebssysteme wie etwa Debian, Ubuntu, openSUSE oder Fedora gleichermaßen eingesetzt werden. Abhängig davon, ob diese Distribution eine eigene Shim-Version zur Verfügung stellt, müssen Anwendende gegebenenfalls eigene Schlüssel (MOKs) generieren. Aufgrund der ausführlichen Dokumentation, der großen Beliebtheit und der guten Einbindung der TPM2-relevanten Pakete wird für diese Arbeit Fedora als Betriebssystem gewählt. Es wird die Fedora-Version 32 mit der Kernel-Version 5.8.13 eingesetzt. Fedora bietet in der Standard-Installation Shim mit einem Distributions-Zertifikat, welches zum Überprüfen des signierten Bootloaders GRUB2 und des Kernels verwendet wird. Dadurch müssen Anwendende keine eigenen Schlüssel generieren und die Komponenten signieren. Sie haben jedoch die Möglichkeit auch eigene MOKs hinzuzufügen [104].

7.2 Update-Prozess ausgewählter Linux Distribution

Dieser Abschnitt behandelt kurz die Update-Mechanismen der einzelnen Komponenten. Dabei wird speziell auf die in Abschnitt 7.1 ausgewählten Komponenten beziehungsweise die Hardware eingegangen. Des Weiteren wird in diesem Abschnitt aufgelistet, welche PCR-Werte sich verändern, wenn die jeweilige Komponente aktualisiert wird.

7.2.1 BIOS/UEFI-Update

Das BIOS/UEFI ist, wie in Abschnitt 3.2 beschrieben, die erste Komponente, die ausgeführt wird. Auf ihrer Integrität beruht die Sicherheit des gesamten Systems. Daher bedarf das Updaten des BIOS ein hohes Maß an Sicherheit [26]. Da das Updaten von UEFI und BIOS demselben Prozess folgen, wird in diesem Abschnitt der Überbegriff,

	Distribution	Standard-Bootmanager	Referenz
1	MX Linux	GRUB2	[88]
2	Manjaro	GRUB2	[89]
3	Mint	GRUB2	[95]
4	Pop!_OS	systemd-boot	[124]
5	Ubuntu	GRUB2	[137]
6	Debian	GRUB2	[29]
7	elementary OS	keine Info	
8	Fedora	GRUB2	[146]
9	EndeavourOS	keine Info	
10	Solus	clr-boot-manager	[123]

Tabelle 7.1: Beliebtesten Linux Distributionen 2020 [30]

BIOS verwendet.
BIOS-Updates werden von Plattformherstellerfirmen bereitgestellt und können in den meisten Fällen über einen speziellen, von der Herstellerfirma integrierten Mechanismus, von Endnutzenden installiert werden. Falls dieser Mechanismus nicht vorhanden ist, können Nutzende von einem externen Medium booten und so das Update installieren [26]. Für bestimmte Hardware können unter Linux Firmware-Updates über die Plattform Linux Vendor Firmware Service (LVFS) [62] installiert werden. Dafür muss die Herstellerfirma selbst die neuen Dateien und die dazugehörigen Informationen auf der Plattform hochladen. Das Verwalten der Firmware über das Betriebssystem wird, zum Beispiel für das verwendete HP-Produkt, lediglich für Windows-Systeme unterstützt. In anderen Systemen muss das Update mittels eines externen Mediums installiert werden. Es sollten ausschließlich BIOS-Versionen installiert werden, welche die Herstellerfirma für das entsprechende Gerät empfiehlt. Ansonsten kann dies dazu führen, dass das System nicht mehr booten kann (Genauere Informationen zum BIOS-Update für HP-Geräte ist unter [60] zu finden). Für den Fall eines falschen oder fehlgeschlagenen BIOS-Updates wird häufig ein Wiederherstellungs-Mechanismus angeboten, welcher es Anwendenden erlaubt entweder manuell oder automatisch die Firmware auf eine funktionierende, valide Version zurückzusetzen [26].
Um zu verhindert, dass maliziöse BIOS-Versionen installiert werden, soll laut NIST [26] sowohl ein authentifizierter als auch ein sicherer lokaler Update-Mechanismus integriert werden. Bei Ersterem soll die Signatur der Update-Datei überprüft werden. Der sichere lokale Update-Mechanismus soll lediglich verwendet werden, um das erste BIOS zu installieren oder als Wiederherstellungs-Mechanismus, wenn das authentifizierte Updaten nicht funktioniert. Sicherheit bietet dabei die physische Präsenz der Anwendenden und gegebenenfalls ein Passwort.

7.2.2 GRUB2-Update

Für die GRUB2-Installation unter Fedora sind, laut Fedora Wiki [146] drei Dinge erforderlich: die Erstellung einer EFI System Partition (ESP), die Installation von Bootloader-Dateien und die Anpassung von GRUB2-Konfigurationen. Mit dem `gdisk`-Kommando kann überprüft werden, ob bereits eine ESP existiert. Des Weiteren muss diese Partition auf den `/boot/efi` Mount-Point gemountet werden. (Die Befehle aus diesem Abschnitt stammen aus [146] und die Ergebnisse aus der praktischen Ausführung). Falls die Partition richtig eingerichtet und gemountet wurde sollten die Befehle in Listing 7.1 ein ähnliches Ergebnis liefern.

Listing 7.1: Partitionierung mit Mountpoints. Befehle aus [146]

```
1  $ sudo gdisk -l /dev/nvme0n1
2  Number  Start (sector)  Size       Code  Name
3  1           2048       600.0 MiB  EF00  EFI System Partition
4
5  $ lsblk
6  NAME           MAJ:MIN RM   SIZE RO TYPE  MOUNTPOINT
7  sr0             11:0    1  1024M  0 rom
8  zram0          252:0    0     4G  0 disk  [SWAP]
9  nvme0n1        259:0    0 476,9G  0 disk
10 |-nvme0n1p1    259:1    0   600M  0 part  /boot/efi
11 |-nvme0n1p2    259:2    0   1G    0 part  /boot
12 |-nvme0n1p3    259:3    0 475,4G  0 part
13   |-luks-xxxx  253:0    0 475,3G  0 crypt  /
```

Shim und GRUB2 Bootloader-Dateien können in Fedora mittels des Befehls in Listing 7.2 upgedatet werden. Die GRUB2-Konfigurationsdatei `/boot/efi/EFI/fedora/grub.cfg` kann, falls noch nicht vorhanden mit dem Befehl in Zeile 2 erstellt werden. Eine genauere Beschreibung des Vorgangs und Lösungen für eventuell auftretende Probleme sind unter [146] zu finden.

Listing 7.2: GRUB2 und Shim aktualisieren nach [146]

```
1  $ sudo dnf reinstall grub2-efi shim
2  $ sudo dnf grub2-mkconfig -o /boot/efi/EFI/fedora/grub.cfg
```

Ein Bootloader kann auch mittels Source Code installiert werden. Ein Beispiel „grub“ Projekt, ist in Garretts Git-Repository [44] zu finden. Um einen Bootloader mittels Source Code zu installieren wird der erste Befehl von Listing 7.2 mit folgender Prozedur ersetzt [44]:

- Das Projekt „grub“ clonen und in das Verzeichnis mit dem Source Code navigieren.
- Dort `./autogen.sh` ausführen.

- Mittels `./configure` kann das Paket für das System angepasst werden.
- Mit `make` wird das Paket kompiliert und es kann dann mittels `make install` installiert werden

Wenn ein Bootloader installiert wird, welcher nicht von Fedora signiert wurde, muss dieser mit einem eigenen Schlüssel signiert werden. Der verwendete Schlüssel muss, wie in Abschnitt 3.2.1 beschrieben, Teil der MOK-Liste sein.

7.2.3 Kernel-Update

Fedora Kernel haben ein RPM (Red Hat Package Manager)-Format. Pakete dieses Formats können entweder direkt mittel `rpm` Befehl oder mit Hilfe eines anderen Paketmanagers wie etwa PackageKit oder DNF verwaltet werden. DNF bietet eine bedienungsfreundliche Methode, um Pakete zu installieren und zu aktualisieren. Mit dem Befehl `dnf upgrade kernel` kann der Kernel aktualisiert werden. DNF installiert immer einen neuen Kernel unabhängig davon, ob man `install` oder `upgrade` verwendet (mehr Informationen zu DNF in der Fedora Dokumentation [36]).
Der Kernel kann auch manuell aktualisiert werden. Dafür sind folgende Schritte notwendig [36]:

- **Kernel-Pakete ermitteln**: neuere Kernel-Pakete können entweder über das Fedora Update System [2] oder mittels DNF Befehl `dnf check-update --enablerepo=updates-testing` ermittelt werden.
- **Update installieren**: die gewählte Version, zum Beispiel 5.8.16-300 [3] kann anschließend über `dnf upgrade --advisory=FEDORA-2020-1d0fc839f1` installiert werden. Die RPM Datei kann auch manuell heruntergeladen und mittels `rpm -ivh kernel-5.8.16-300.fc33.arch.rpm` installiert werden (wobei `arch` für die Systemarchitektur steht).
- **Initial RAM-based Filesystem (initramfs) überprüfen**: seit Fedora 32 wird bei der Installation eines Kernels automatisch eine initramfs-Datei erstellt (mehr Informationen zu initramfs in Abschnitt 3.2.4). Bei einer erfolgreichen Installation sollte im `/boot` Verzeichnis die Kernel-Version (vmlinuz-kernel_version) und die initramfs-Version (initramfs-kernel_version) wie in Listing 7.3 übereinstimmen. Falls diese nicht übereinstimmen reicht es den Befehl `dracut` auszuführen, welcher für die aktuellste Kernel-Version eine entsprechende initramfs-Datei erstellt. Des Weiteren muss die initramfs Datei in der GRUB2-Konfigurationsdatei gelistet sein.
- **Bootloader überprüfen**: obwohl `rpm` einen Eintrag in der GRUB2-Konfigurationsdatei anlegt, wird der neue Kernel jedoch nicht zum Standard-Kernel deklariert. Dies

[2] https://bodhi.fedoraproject.org/
[3] https://bodhi.fedoraproject.org/updates/FEDORA-2020-1d0fc839f1

muss manuell durch Anpassung der `linux` und `initrd` Einträge im grub.cfg erfolgen.

Listing 7.3: Kernel- und initramfs-Version müssen übereinstimmen. Befehl aus [36]

```
$ ls /boot
config-5.8.15-301.fc33.x86_64
efi
grub2
initramfs-5.8.15-301.fc33.x86_64.img
System.map-5.8.15-301.fc33.x86_64
vmlinuz-5.8.15-301.fc33.x86_64
```

7.2.4 Betriebssystem-Update

Auch für Betriebssystem-Updates kann unter Fedora der Paket-Manager DNF verwendet werden. Dafür muss wie in Listing 7.4 gezeigt ein bestimmtes Plugin installiert und verwendet werden. Damit die Integrität der neuen Pakete überprüft werden kann, werden die öffentlichen Schlüssel neuerer Fedora-Versionen an die alten geschickt, dadurch kann DNF die Signatur kontrollieren [36].

Listing 7.4: Fedora Betriebssystem Updaten. Befehle aus [36]

```
$ sudo dnf install dnf-plugin-system-upgrade
$ sudo dnf system-upgrade download - -refresh - -releasever=33
$ sudo dnf system-upgrade reboot
```

7.3 Geplantes Konzept für einen Bootprozess mit TPM 2.0

In diesem Abschnitt werden zwei Varianten aufgezeigt, wie TPM 2.0 in den Linux Bootprozess einbezogen werden kann. Variante eins 7.3.1 ist nutzungsfreundlicher, bietet jedoch zum Zeitpunkt dieser Arbeit keinen ausreichenden Schutz vor einer Bootloader-Schwachstelle, beschrieben in Abschnitt 4.2.1. Die zweite Variante 7.3.2 ist nicht anfällig für die Bootloader-Schwachstelle, ist jedoch komplexer und bedarf erhöhten Implementierungs- und Wartungsaufwand.

7.3.1 Bootprozess mit Shim und GRUB2

Den Anfang des geplanten Bootprozesses bildet ein natives UEFI, anschließend kommt der Bootloader aufgeteilt in First-Stage mit Shim und Second-Stage mit GRUB2. GRUB2 lädt den Fedora Kernel, welcher dann das Betriebssystem lädt. Das Konzept ist eine Hybrid-Lösung zwischen Secure- und Measured Boot.

Um die Ausführung von Rootkits zu verhindern wird UEFI Secure Boot aktiviert und die `/boot` Partition verschlüsselt. Diese Partition beinhaltet den Kernel und die initramfs-Datei. Die Entschlüsselung wird mittels TPM2-Policy genehmigt. Sollte die Policy nicht erfüllt sein, wird der Kernel nicht entschlüsselt und der Bootvorgang wird abgebrochen. Siehe TPM2-Policys in Abschnitt 5.3. Dieser Ansatz könnte auch in abgeschwächter Form zum Einsatz kommen, sodass Anwendende lediglich über eine mögliche Inkonsistenz informiert werden und mit der expliziten Eingabe eines Passwortes den Bootprozess fortsetzen können.
Zunächst wurde überprüft, welche PCR-Werte sich beim Updaten der unterschiedlichen Komponenten beziehungsweise durch das Abwandeln von Konfigurationen verändern. Tabelle 7.2 zeigt die Auswirkungen bestimmter Aktionen auf die PCR-Werte. Diese Daten wurden anhand eigener Messungen ermittelt und können je nach System und Version abweichen. Eine detaillierte Darstellung, welche Komponenten in welches Register gemessen werden sollten, ist in Tabelle 5.1 zu finden.
Die Register 0 bis 7 sind laut TCG-Spezifikation [2] für UEFI-Messungen vorgesehen. Die PCRs über 7 sind Betriebssystemabhängig.
Das Aktualisieren des BIOS hatte Auswirkungen auf die Register 0,1,2,3 und 5. Dies widerspricht nicht der Spezifikation, da bei einem Update nicht zwangsläufig alle Register von 0 bis 6 verändert werden müssen. Durch das Setzen eines BIOS- Administrator-Passwortes hat sich PCR 4 verändert. Diese Aktion würde jedoch laut TCG Spezifikation PCR 1 betreffen [2]. Sowohl das Verändern der UEFI Secure Boot-Schlüssel als auch das Deaktivieren beziehungsweise Aktivieren von UEFI Secure Boot (SB) haben Einfluss auf PCR 7 und entsprechen somit der Spezifikation. Das Laden von MOK verändert das Register 14. Ein Kernel-Update hat den Wert von Register 10 verändert. Es kann jedoch auch das Register 8, welche die `grub.cfg` beinhaltet und das Register 9 beeinflussen. Das Updaten des Betriebssystems hat auch den Kernel aktualisiert. Dadurch haben sich die PCR-Werte 8, 9 und 10 verändert. In Tabelle 10.1 sind die effektiven PCR-Werte beim ersten Booten und nach den ausgeführten Aktionen gelistet. Die Register 12,13 und 15-23 wurden bei keiner der Aktionen gesetzt. Diese können, wie in Abschnitt 6.5 genauer beschrieben, für die Dynamic Chain of Trust verwendet werden.

PCR	BIOS Passwort setzen	SB Schl. laden	SB akt.	MOK erstellen	Kernel Upd.	OS Upd. inkl. Kernel Upd.	BIOS Update
0	Nein	Nein	Nein	Nein	Nein	Nein	**Ja**
1	Nein	Nein	Nein	Nein	Nein	Nein	**Ja**
2	Nein	Nein	Nein	Nein	Nein	Nein	**Ja**
3	Nein	Nein	Nein	Nein	Nein	Nein	**Ja**
4	**Ja**	Nein	Nein	Nein	Nein	Nein	Nein
5	Nein	Nein	Nein	Nein	Nein	Nein	**Ja**
6	Nein	Nein	Nein	Nein	Nein	Nein	Nein
7	Nein	**Ja**	**Ja**	Nein	Nein	Nein	Nein
8	Nein	Nein	Nein	Nein	Nein	**Ja**	Nein
9	Nein	Nein	Nein	Nein	Nein	**Ja**	Nein
10	Nein	Nein	Nein	Nein	**Ja**	**Ja**	Nein
11	Nein	Nein	Nein	Nein	Nein	Nein	Nein
12	Nein	Nein	Nein	Nein	Nein	Nein	Nein
13	Nein	Nein	Nein	Nein	Nein	Nein	Nein
14	Nein	Nein	Nein	**Ja**	Nein	Nein	Nein

Tabelle 7.2: Veränderung der PCR-Werte durch bestimmte Aktionen

Laut Garrett [43] reicht es aus, UEFI Secure Boot zu aktivieren und die TPM-Policy an PCR 7 zu knüpfen, um den Bootprozess ausreichend abzusichern. PCR 7 beinhaltet den Wert der UEFI Secure Boot-Policy und die verwendeten Schlüssel. Die Idee dahinter ist, so Garrett, dass man dem Betriebssystemhersteller und den Komponenten, die von ihm signiert wurden, vertraut.
Durch das Binden an die UEFI Secure Boot-Konfigurationen und Schlüssel können Bootloader, Kernel und Betriebssystem aktualisiert werde, ohne das System als nicht vertrauenswürdig einzustufen.
UEFI Secure Boot bricht den Bootprozess ab, sobald Bootkomponenten keine gültige Signatur besitzen. Mittels TPM-Policy könnte des Weiteren sichergestellt werden, dass die Firmware nicht korrumpiert und UEFI Secure Boot nicht deaktiviert wurde. Der Bootloader wird von UEFI überprüft und geladen. Der Bootloader kann auf den Schlüssel zum Entschlüsseln der `/boot` Partition, inklusive Kernel und initramfs, mit dem Befehl *tpm2_unseal* lediglich bei einer erfüllten Policy zugreifen. Um sicherzustellen, dass die UEFI-Firmware nicht manipuliert wurde, sollten die UEFI-Firmware bezogenen Messwerte in PCR 0 in die Policy miteinbezogen werden. Damit die Policy nicht bei

einem BIOS-Update bricht, könnte eine „Authorize Policy“ eingesetzt werden. Diese bindet nicht direkt den PCR-Wert mit ein, sondern einen öffentlichen Schlüssel. Dieser Schlüssel wird dann verwendet, um die signierten PCR-Werte zu überprüfen (eine genaue Beschreibung des Ablaufes ist in Abschnitt 5.8 zu finden). Dieses Szenario ist nur dann sinnvoll einsetzbar, wenn die Herstellerfirmen signierte PCR-Werte für die BIOS-Versionen bereitstellen. Herstellerfirmen können, laut LVFS Dokumentation [63], beim Hochladen ihrer Firmware auf die Plattform LVFS in der `metainfo.xml`-Datei den Soll-Wert für PCR 0 angeben. Zum Zeitpunkt dieser Arbeit werden nur sechs HP-Geräte von der Plattform unterstützt. Zusätzlich werden nicht für alle Firmware-Versionen Soll-Wert für PCR 0 (Attestation Checksum) zur Verfügung gestellt [4]. Dadurch ist es effizienter und leichter in der Handhabung die PCR-Policy um das Register 0 zu erweitern anstatt eine Authorize Policy zu verwenden.
Bei der geplanten Implementierung haben Anwendende immer die Möglichkeit beim Fehlschlagen der Policy die `/boot` Partition manuell mit einem Passwort zu entschlüsseln. Ziel dieses Konzeptes ist es, Anwendende nicht zu entmündigen, sondern sie bei Unstimmigkeiten im Bootprozess aufmerksam zu machen. Im Falle eines gezielten BIOS-Updates oder einer expliziten Deaktivierung von UEFI Secure Boot, können Anwendende sich mit dem Passwort anmelden, die alte Policy löschen und mit den neuen Werten eine Neue erstellen. Da die neuen PCR-Werte erst nach einem Neustart des Systems bekannt sind, kann die aktualisierte Policy nicht im Voraus erstellt werden. Anwendende können sich auch entschließen, die Möglichkeit einer Passwortanmeldung zu entfernen und lediglich TPM-basierte Entschlüsselung zuzulassen. In diesem Fall müssen die PCR-Werte entsprechende gewählt werden, um sich nicht auszusperren.
Bei der praktischen Umsetzung des Konzeptes wird darauf geachtet, ob die Bootloader-Schwachstelle von Abschnitt 4.2.1 behoben wurde. Die Ausnutzung dieser Schwachstelle kann, laut National Security Agency [98], auf zwei Arten verhindert werden: entweder alle Bootkomponenten aktualisieren und die alten Versionen zur UEFI Revocation List *dbx* hinzufügen, oder eine eigene UEFI Secure Boot-Infrastruktur, wie im nächsten Abschnitt 7.3.2 beschrieben, zu implementieren.
Diese und weitere Schwachstellen wurde von Fedora für die Version 33 in der GRUB2-Version 2.04.27 bereits behoben [34]. Es gilt nun zu überprüfen, ob diese Updates durch Fedora ausreichen, um das Konzept mit Shim umzusetzen.

7.3.2 Bootprozess ohne Bootloader

Diese Variante verwendet denselben Ansatz wie in Abschnitt 7.3.1: UEFI Secure Boot aktivieren und die automatische Entschlüsselung einer Partition von einer TPM2-Policy abhängig machen.
Für diesen Ansatz werden jedoch eigene UEFI Secure Boot-Schlüssel generiert und

[4] Zum Beispiel wird für HP Inc. Z2 G4 Workstation https://fwupd.org/lvfs/devices/com.hp.workstation.Q50.firmware ein Hashwert für PCR 0 bereitgestellt für HP Inc. Z4 G4 Core-X Workstation https://fwupd.org/lvfs/devices/com.hp.workstation.P62.firmware jedoch nicht

eingesetzt. Dies erlaubt, so Paul u. a. [104], das direkte Booten eines selbst signierten Kernels ohne die Verwendung eines Bootloaders.
In diesem Bootprozess wird weder der von Microsoft signierte First-Stage Bootloader Shim noch GRUB2 eingesetzt. Dies eliminiert jegliche Schwachstellen in diesen Komponenten, unter anderem die BootHole Schwachstelle aus Abschnitt 4.2.1. Durch das Löschen der Standard-Plattform- und Microsoft-Zertifikate wird verhindert, dass mit physischem Zugriff ein anderes Betriebssystem über einen USB-Stick gebootet wird. UEFI Secure Boot überprüft, wie in Abschnitt 6.3 beschrieben, lediglich ob eine Komponente eine valide Signatur hat [104]. Wenn die Standard-UEFI-Schlüssel verwendet werden kann jedes Betriebssystem gebootet werden, welches mit diesen signiert wurde, zum Beispiel Windows [138]. Eigene UEFI Schlüssel bieten einen erhöhten Schutz gegen physische Angriffe und Malware [104].
Die UEFI-Firmware lädt anstatt eines Bootloaders ein sogenanntes „Unified Kernel Image“. Ein Unified Kernel Image ist ein Bündel aus Kernel, initramfs und Kernel-Bootparameter. Diese PE Datei wird in einem Verzeichnis in der `/boot/efi` Partition abgelegt, als Boot-Eintrag angeführt und von der Firmware ausgeführt [125]. Siehe UEFI Systeme und PE-Dateien in Abschnitt 6.3. Eine Beispielübersicht der Partitionen befindet sich in Listing 7.1. Da der Kernel aus dem Bündel und nicht direkt aus der `/boot` Partition geladen wird, wird bei dieser Variante die `/root` anstatt der `/boot` Partition verschlüsselt. Bei dieser Variante des Konzeptes wird nicht die Kernel-Datei alleine signiert, sondern das gesamte Bündel. Dadurch wird zusätzlich zum Kernel das initramfs vor Manipulationen, wie aus Abschnitt 4.1.1, geschützt. Bei jedem Update des Kernels oder des Betriebssystems muss das Bündel neu generiert und signiert werden. Dieser Aspekt macht diese Variante komplexer und Wartungsaufwändiger für Anwendende. Des Weiteren müssen sich Anwende um die sichere Speicherung der UEFI-Schlüssel kümmern.

7.4 Prototypische Umsetzung

Für die prototypische Umsetzung wurde ein Bootprozess mit Shim und GRUB2, wie in Abschnitt 7.3.1 beschreiben gestartet und dann auf das in Abschnitt 7.3.2 beschriebene Verfahren gewechselt, weil die Bootloader-Schwachstelle aus Abschnitt 4.2.1 nicht eliminiert werden konnte. Es wurde der HP-Laptop mit Fedora 32, der Kernel-Version 5.6.6, GRUB2 und Shim aufgesetzt. Die Verschlüsselung der `/root` Partition wurde bereits beim Installationsprozess aktiviert. Für die Verschlüsselung wurde Linux Unified Key Setup (LUKS)2, im weiteren Verlauf LUKS genannt, [35] mit einem Passwort verwendet. LUKS kann auch bei einem laufenden Betriebssystem nachträglich aktiviert werden, jedoch wird die entsprechende Partition formatiert und die vorhanden Daten gehen verloren.
Zu Beginn wurde überprüft, ob die aktuelle UEFI Revocation List (dbx) installiert werden kann.

7.4.1 UEFI dbx aktualisieren

Unter Fedora der Version 32 wurde mit den Befehlen in Zeile 8 des Listings 7.5 die aktuelle Firmware-Version ausgeben und auf Updates geprüft [64].

Listing 7.5: Firmware und UEFI dbx Version aktualisieren [64]

```
1  $ sudo fwupdmgr get-devices
2  |-UEFI dbx:
3  |     Device ID:          362301
   ↪ da643102b9f38477387e2193e57abaa590
4  |     Zusammenfassung:    UEFI Revocation Database
5  |     Current version:    77
6  |     Minimum Version:    77
7
8  $ sudo fwupdmgr update
9  System Firmware has no available firmware updates
10 UEFI dbx has no available firmware updates
11 WDC PC SN520 SDAPNUW-512G-1006 has no available firmware
   ↪ updates
```

Da auf der LVFS (Linux Vendor Firmware Service)-Plattform keine neuere Version vorhanden ist (Zeile 9-11 Listing 7.5) wurde eine zusätzliche Konfigurationsdatei für *fwupdmgr* erstellt. Mit dieser kann die neuste UEFI Revocation List [5] Version installiert werden, wenn diese keine Werte der aktuellen Komponenten beinhalten. Beim Versuch diese auszuführen ist die Fehlermeldung in Zeile 9 von Listing 7.6 erschienen und das Aktualisieren wurde abgebrochen [64].
Falls Anwendende den `fwupdmgr update` Befehl mit `--force` erzwungen haben und das System mit aktiviertem UEFI Secure Boot nicht mehr startet, können die UEFI-Schlüssel und die Datenbanken über das BIOS-Menü zurückgesetzt werden.

[5] https://uefi.org/revocationlistfile

Listing 7.6: Fehler beim Aktualisieren der UEFI dbx [64]

```
1 $ refresh --force
2 $ sudo fwupdmgr update
3
4 Upgrade available for UEFI dbx from 77 to 190
5 UEFI dbx and all connected devices may not be usable while
     ↪ updating. Continue with update? [Y|n]: Y
6 ...
7 UEFI dbx wird aktualisiert  ...-                                  ]
8 Überprüfung ...     [************************************]
9 Blocked executable in the ESP, ensure grub and shim are up to
     ↪ date: /boot/efi/EFI/BOOT/BOOTX64.EFI Authenticode
     ↪ checksum [0ce02100f67c7ef85f4eed368f02bf7092380a3c23ca91f
     ↪  d7f19430d94b00c19] is present in dbx
```

Obwohl die BootHole Schwachstelle von Abschnitt 4.2.1 nicht in den Shim Paketen ist, wurden diese jedoch als unsicher eingestuft und in die UEFI *dbx* aufgenommen [98]. Da die aktuellste Revocation List nicht verwendet werden kann, sollten Anwendende eigene UEFI Secure Boot Schlüssel einsetzen bis eine neue Shim-Version signiert wurde.

7.4.2 Eigene UEFI Secure Boot-Schlüssel

UEFI verwendet, wie in Abschnitt 6.3 beschrieben, verschiedene Schlüssel. In diesem Kapitel wird kurz beschrieben, wie die PK, KEK und DB Schlüssel erzeugt und geladen werden. Eine genauere Anleitung liefert Smith [121]. Zu Beginn wurde das Programm efitools [16], wie in Listing 7.7, heruntergeladen und installiert. Bit dem Befehl in Zeile 1 wurden die benötigten Abhängigkeiten installiert. Beim Ausführen des `make` Befehls wurden die benötigten Schlüssel und Zertifikate generiert. Alle Zertifikate (Dateien mit den Endungen .crt, .cer, .auth, .esl) wurden zusammen mit dem enthaltenen Keytool.efi auf einen FAT formatierten USB-Stick kopiert. Je nach UEFI werden andere Zertifikatsformate benötigt. Bei dem in dieser Arbeit verwendeten Laptop waren .auth Zertifikate erforderlich. Beim nächsten Bootvorgang wurde über das BIOS-Menü UEFI Secure Boot deaktiviert und alle Standard-Schlüssel gelöscht. Anschließend wurde über das Bootmenü „Boot from EFI-Image“ das Keytool.efi gestartet. Über dieses wurden die Zertifikate für den PK, den KEK und die DB in dieser Reihenfolge eingespielt. Das erfolgreiche Laden der Zertifikate wurde nach einem Neustart mit dem Befehl `efi-readvar` aus Listing 7.7 überprüft.

Listing 7.7: Installation der efitools. Befehele aus [16]

```
1 $ sudo dnf install openssl-devel sbsigntools gnu-efi-devel
    ↪ help2man perl perl-File-Slurp
2 $ git clone https://git.kernel.org/pub/scm/linux/kernel/git/
    ↪ jejb/efitools.git/
3 $ cd efitools
4 $ make
5 $ ./efi-readvar
```

Der erzeugte DB Schlüssel wurde verwendet um das Unified Kernel Image aus dem Konzept in Abschnitt 7.3.2 zu signieren. In Listing 7.8 befinden sich die Befehle, welche zum Erzeugen des Boot-Bündels, zum Signieren mit dem zuvor erstellten EFI DB Schlüssel und das Eintragen in den Bootmanager verwendet wurden. Um die Kernel-Bootparameter zu ermitteln wurde einmal mit Shim gebootet und dann der Befehl in Zeile 1 ausgeführt. Das Ergebnis [6] wurde in eine Textdatei geschrieben, welche zum Erzeugen des Boot-Bündels verwendet wurde. Mit dem Befehl, von Pronkitprasan [109] (Zeile 2 bis 7), wurde das Bündel erzeugt. Dieses wurde anschließend signiert und zum EFI-Bootmanager hinzugefügt. Die `--disk` und `--part` Parameter sind abhängig von der Ausgabe in Listing 7.1.

Listing 7.8: Boot-Bündel Erzeugen, Signieren und Booteintrag anlegen. Befehele aus [109, 121]

```
1 $ sudo cat /proc/cmdline > /boot/kernel-command-line.txt
2 $ sudo objcopy /
3 --add-section .osrel="/usr/lib/os-release" --change-section-vma
    ↪  .osrel=0x20000 /
4 --add-section .cmdline="/boot/kernel-command-line.txt" --change
    ↪ -section-vma .cmdline=0x30000 /
5 --add-section .linux="/boot/vmlinuz-5.9.10-200.fc33.x86_64" --
    ↪ change-section-vma .linux=0x40000 /
6 --add-section .initrd="/boot/initramfs-5.9.10-200.fc33.x86_64.
    ↪ img" --change-section-vma .initrd=0x3000000 /
7 "/usr/lib/systemd/boot/efi/linuxx64.efi.stub" "/boot/efi/EFI/
    ↪ Linux/Linux.efi"
8 $ sudo sbsign --key ~/efitools/DB.key --cert ~/efitools/DB.crt
    ↪ --output /boot/efi/EFI/Linux/Linux.efi /boot/efi/EFI/
    ↪ Linux/Linux.efi
9 $ sudo efibootmgr --create --disk /dev/nvme0n1 --part 1 --label
    ↪  "Signed␣Kernelbundle" --loader "\EFI\Linux\Linux.efi"
```

[6] BOOT_IMAGE=(hd0,gpt2)/vmlinuz-5.9.10-200.fc33.x86_64 root=UUID=548435c7-cb27-4afa-b2d1-ca37134b5e1e ro rootflags=subvol=root rd.luks.uuid=luks-12e487fb-b3e1-446d-8b08-7f4e1a67b8dd rhgb quie

7.4.3 LUKS und TPM2

Nach der erfolgreichen Betriebssysteminstallation musste das TPM aktiviert und zurückgesetzt werden. Dies wurde über das BIOS-Menü gemacht. Es kann auch über die Kommandozeile mit den Befehlen `tpm2_takeownership` und `tpm2_clear` gemacht werden. Um TPM2-Befehle ausführen zu können wurde der in Abschnitt 5.7 beschriebene TCG Software Stack (TSS) und die in Abschnitt 5.8 erklärten tpm2-tools der Version 4.3 installiert. Unter Fedora 32 sind die tpm2-tools bereits Teil der Paketverwaltung und konnten mit dem Befehl `dnf install tpm2-tools` installiert werden. Dadurch wurden auch automatisch alle Abhängigkeiten, wie etwa tpm2-tss, heruntergeladen.
Für Distributionen, welche die Pakete noch nicht inkludiert haben, können diese nach der Installationsanleitung von [134] mit den Befehlen in Listing 10.1 manuell installiert werden. Mit dem Befehl `tpm2_pcrread` wurden nach einer erfolgreichen Installation die PCR-Werte ausgelesen.
Dieser Befehl wurde nach jeder Aktionen aus Tabelle 7.2 ausgeführt, um Veränderungen der Registerwerte zu ermitteln. Das Setzen des BIOS-Administrator-Passwortes, das Laden der Hersteller Secure Boot-Schlüssel und das Aktivieren von UEFI Secure Boot wurde über das BIOS-Menü gemacht. Dafür mit der plattformspezifischen Tastenkombination, bei HP `Esc` und dann `F10`, in die *BIOS Setup Utility* booten. Im Abschnitt *Boot Options* können die Standard-UEFI Schlüssel geladen beziehungsweise gelöscht und UEFI Secure Boot aktiviert werden.
Nachdem festgestellt wurde, welche PCR-Werte sich bei einer regulären Anwendung verändern, kann die Entschlüsselung der Festplatte an die gewünschten PCR-Werte geknüpft werden. Für die automatische Entschlüsselung wurde das Programm Clevis [25] der Version 14 verwendet. Dieses erlaubt das automatische Entschlüsseln von LUKS-Partitionen mittels TPM 2.0 PCR-Policys. Das Programm wurde wie in Listing 7.9 installiert und mit einer LUKS-Partition (Zeile 2) verknüpft. Dieser Befehl hat einen Fehler ausgegeben, welche durch das Entfernen des Paketes clevis-pin-tpm2 gelöst wurde [37]. Der Befehl `cryptsetup` in Zeile 3 von Listing 7.9 hatte nach der Bindung mit Clevis zwei Schlüsseleinträge. Einen für die Passwortentschlüsselung und einen für die PCR-Policy. Die Ausgabe des Befehls in Zeile 3 ist im Anhang 10.2 zu finden. Anschließend wurde mit dem Befehl `dracut` die initramfs Datei neu erzeugt, sodass diese clevis, tpm2 und tss inkludiert wurden (Zeile 5 und 6). Nachdem ein neues initramfs generiert wurde, musste ein neues Boot-Bündel wie in Listing 7.8 erzeugt und signiert werden.

Listing 7.9: Clevis mit LUKS und TPM 2.0 [78]

```
1 $ sudo dnf install clevis clevis-luks clevis-dracut clevis-
    ↪ udisks2 clevis-systemd
2 $ sudo clevis luks bind -d /dev/nvme0n1p3 tpm2 '{"pcr_bank":"
    ↪ sha256", "pcr_ids":"0,7"}'
3 $ sudo cryptsetup luksDump /dev/nvme0n1p3
4 $ sudo dracut
5 $ lsinitrd /boot/initramfsXXX.img | grep -E 'clevis|tpm2|tss'
```

Clevis regelt die automatische Entschlüsselung der Partition. Ein Passwort muss nur dann eingegeben werden, wenn die Policy nicht erfüllt ist. Dadurch haben Anwendende einen indirekten Beweis, dass sich das System im gewünschten Zustand befindet und vertrauenswürdig ist. Doch aufgrund eines bekannten Bugs in Plymouth [17], einer Komponente für grafischen Animationen während des Bootprozesses, wird das Passwortfeld trotzdem für etwa 30 Sekunden angezeigt. Dieses verschwindet jedoch automatisch wieder und die Partition wird automatisch entschlüsselt, wenn die PCR-Policy erfüllt ist.
Beim Erstellen der PCR-Policy wird das LUKS-Passwort vom TPM geschützt gespeichert. Es wird erst wieder freigegeben, wenn die PCRs sich im definierten Zustand befinden. Eine detaillierte Beschreibung der Sicherung und Freigabe von Geheimnissen, wie etwa Passwörtern oder Schlüsseln, ist in Abschnitt 5.8.2 zu finden.

KAPITEL 8

Analyse und Resultate

Das Prinzip der verschlüsselten Partitionen liefert Anwendenden einen indirekten Beweis, dass das System in einem vertrauenswürdigen Zustand ist. Wenn das System automatisch bootet ist es vertrauenswürdig [48]. In Abbildung 8.1 sind beide Varianten 7.3.1 (grün), 7.3.2 (blau) des Bootprozesses dargestellt. UEFI Code und Konfigurationen werden in die PCRs 0 bis 7 gemessen. Wobei die Register 0 und 7 von großer Bedeutung sind. Diese beinhalten die Messwerte der UEFI-Firmware und der Secure Boot-Konfigurationen. Es könnten auch die Register 0 bis 7 für die Policy verwendet werden. Jedoch sind diese, wie in Tabelle 5.1 dargestellt, zum Teil Herstellerfirma spezifisch, beinhalten Konfigurationen oder Komponenten, welche sich im normalen Gebrauch verändern können.
Je nach Präferenzen kann, wie in Abbildung 8.1 dargestellt, entweder Shim mit GRUB2 oder ein selbst signiertes Kernel-Bündel für den Bootprozess verwendet werden.
In den folgenden Abschnitten werden beide Varianten kurz analysiert und jeweils auf die Vor- und Nachteile eingegangen. Im letzten Abschnitt dieses Kapitels werden die Konzepte auf die in Abschnitt 4 beschriebenen Angriffsvektoren untersucht.

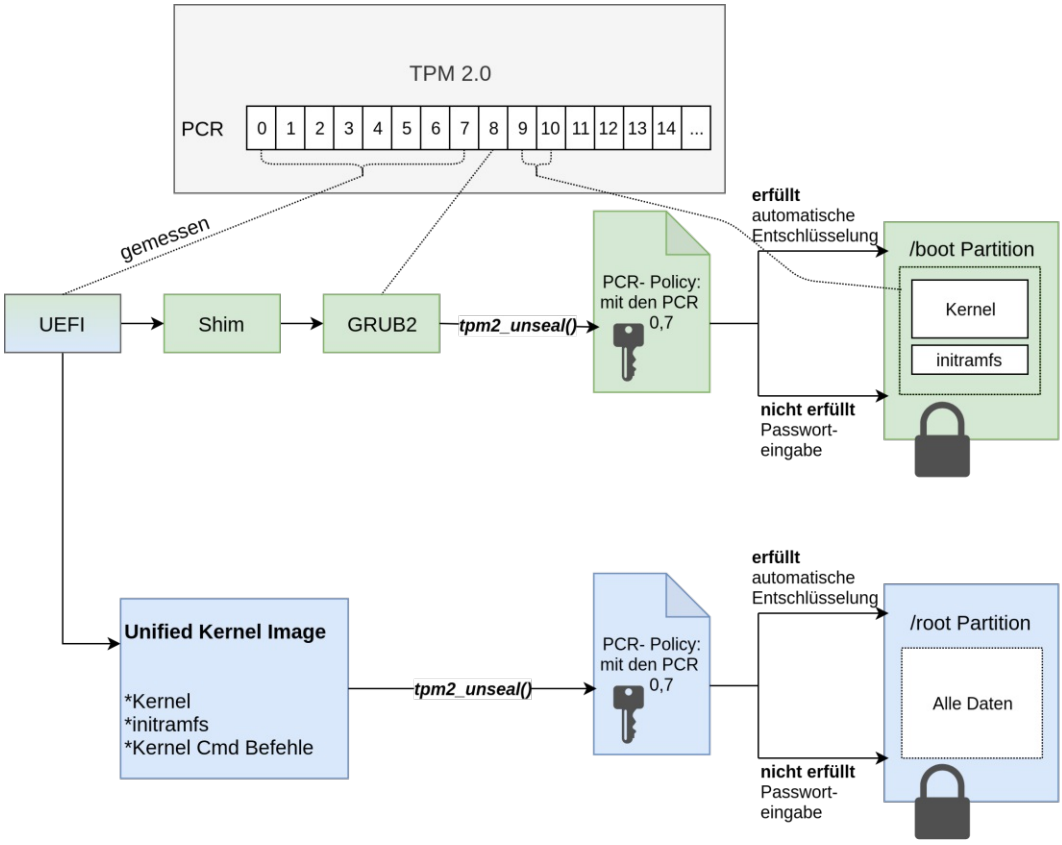

Abbildung 8.1: Absichern des Bootprozesses mittels PCR-Policy

8.1 Analyse Bootprozess mit Shim und GRUB2

Für das in Abschnitt 7.3.1 beschriebene Konzept wird die `/boot` Partition verschlüsselt. Dadurch wird der Bootprozess nicht automatisch fortgesetzt, wenn die PCR-Policy nicht erfüllt ist. Die Verschlüsselung dieser Partition verhindert die Manipulation des Kernels und des initramfs lediglich vor Systemlaufzeiten. Sobald das System vollständig hochgefahren ist, ist die Partition nicht mehr verschlüsselt und die Komponenten können verändert werden.
Im Abschnitt 7.4 wurde gezeigt, wie man die `/root` Partition mittels Clevis automatisch anhand einer PCR-Policy entschlüsseln kann. Jedoch unterstützt Clevis keine Entschlüsselung der `/boot` Partition, da die entsprechenden Befehle im initramfs ausgeführt werden und dieses sich auf besagter Partition befindet. Die Überprüfung der PCR-Policy wird dadurch erst ausgeführt, nachdem der Bootloader den Kernel und das initramfs geladen hat. Damit der Bootloader die `/boot` Partition entschlüsseln kann, müsste dieser den Befehl `tpm2_unseal()` ausführen. Doch laut einer Diskussion im Red Hat Forum Bugzilla [90], unterstützt GRUB2 lediglich die Erweiterung von PCR-Werten und keine anderen TPM2-Befehle.

Die in Abschnitt 7.3.1 beschriebene Variante bringt folgende Vor- und Nachteile.
Vorteile:

- Bedienungsfreundlicher durch die Verwendung von Distributionszertifikaten.
- Kein Signaturprozess bei Systemupdates notwendig.
- Dualboot möglich.

Nachteile:

- Bei physischem Zugriff auf das Gerät können Angreifende ein valides Fedora-System oder Windows booten, wenn kein BIOS-Passwort gesetzt wurde.
- Anfällig für Bootloader-Schwachstellen.
- Aktuelle UEFI dbx 6.3 kann nicht installiert werden. Bootloader-Schwachstelle aus 4.2.1 noch ausnutzbar.
- Kein Schutz des initramfs und der Kernel-Bootparameter.

8.2 Analyse Bootprozess ohne Bootloader

Durch die Verwendung von eigenen UEFI-Schlüsseln kann das Bündel, bestehend aus Kernel, initramfs und Kernel-Bootparameter, als Ganzes signiert werden. Dies verhindert eine Manipulation dieser Komponenten und wird daher von der Autorin als sicherer angesehen (siehe 4.1.2, 4.1.1). Sollte sich der Kernel, ein Kernelmodul oder das initramfs verändern muss ein neues Bündel erzeugt und signiert werden. Wenn Anwendende den privaten Schlüssel zum Signieren sicher verwahren, brechen maliziöse Veränderungen die Signatur. Dadurch werden die Komponenten von UEFI Secure Boot nicht geladen und die Ausführung von schadhaftem Code wird verhindert. Wenn UEFI direkt den Kernel bootet wird kein Bootloader benötigt. Dies schließt zwar eventuelle Schwachstellen in Shim und GRUB2 aus, verhindert jedoch auch, das System im Dual-Boot-Modus zu verwenden. Auch bei dieser Variante aus Abschnitt 7.3.2 wird die PCR-Policy erst nach dem Laden des Kernels überprüft.
Das Konzept ohne Bootloader bietet folgende Vor- und Nachteile:

Vorteile:

- Komplette Kontrolle über UEFI Secure Boot.
- Angreifende können bei physischem Zugriff auf das Gerät kein anderes System booten, ohne die Policy zu brechen.
- Kernel-Bootparameter und initramfs sind vor Manipulationen geschützt.

Nachteile:

- Kein Dualboot mit Windows möglich.
- Bei jedem Update oder neuem Kernelmodul muss ein neues Boot-Bündel erzeugt und signiert werden.
- UEFI-Schlüssel müssen von Anwendenden sicher verwahrt werden.
- Anwendende haben mehr Verantwortung und höheren Wartungsaufwand.

8.3 Analyse von Angriffsszenarien

Falls durch einen Angriff UEFI Secure Boot deaktiviert wird, kann die Ausführung von Rootkit-Code nicht verhindert werden. Es wird lediglich die unwissende Weiterverwendung des kompromittierten Systems unterbunden.
Angriffe mittels Options-ROMs, wie in Abschnitt 4.2.3, können mit beiden Varianten verhindert werden, da Embedded Option-ROMs laut Spezifikation in PCR 0 gemessen werden, welches in die Policy integriert wird (siehe 5.2). Angriffe auf das BIOS/UEFI, den Bootloader und den Kernel wie in den Kapiteln 4.2.3 und 4.2.1 können, in Kombination mit deaktiviertem Secure Boot, auch mit einer verschlüsselten Partition nicht verhindert werden, da mit diesem Konzept erst nach dem Laden des Bootloaders beziehungsweise des Kernels auf die PCR-Werte zugegriffen wird. Jedoch werden Kompromittierungen der Bootkomponenten erkannt, bevor das System vollständig hochgefahren wird.
Ein detailliertes Angriffsszenario auf die Variante 7.3.2 mittels physischem Zugriff könnte wie in Abbildung 8.2 aussehen. Der Angriff startet mit dem Deaktivieren von UEFI Secure Boot (SB) über das BIOS-Menü. Anschließend werden die UEFI-Schlüssel gelöscht. Beim anschließenden Bootvorgang werden die UEFI-Schlüssel wie in Abschnitt 7.4.2 erklärt überschrieben. Mit einem externen Medium zum Beispiel USB-Stick wird dann ein anderes Betriebssystem gebootet. Das neue Unified Kernel Image kann entweder jetzt generiert werden oder wurde bereits auf einem anderen System vorbereitet und signiert. Auf dem System wird die `/boot/efi` Partition gemountet, um darauf zugreifen zu können. Das ursprüngliche Boot-Bündel wird nun entweder mit dem Maliziösen komplett ersetzt oder es wird lediglich der Booteintrag überschrieben. Anschließend wird UEFI Secure Boot wieder aktiviert.
Beim nächsten Bootvorgang wird der maliziöse Kernel ausgeführt. Das Verändern der UEFI-Schlüssel hat jedoch den Wert in PCR 7 verändert, sodass das erneute Aktivieren von UEFI Secure Boot keinen Vorteil gebracht hat. Auch wenn der maliziöse Kernel einige System-Calls manipuliert und gegebenenfalls die Darstellung der Passworteingabe unterbindet, wird die Festplatte nicht automatisch entschlüsselt, da die PCR-Policy nicht erfüllt ist. Falls die Schritte 2 und 5 aus dem genannten Szenario nicht durchgeführt werden, ist die PCR-Policy zwar erfüllt das Boot-Bündel verfügt jedoch nicht über eine valide Signatur und wird dadurch nicht von UEFI geladen. Dieses Angriffsszenario ist für Variante 7.3.1 nicht anwendbar. Da in diesem Szenario die Partition mit dem Kernel

Abbildung 8.2: Mögliches Angriffsszenario mit physischem Zugriff

verschlüsselt ist, wodurch dieser nur nach einem erfolgreichem Bootvorgang verändert werden kann.

Eine Schwachstelle bleibt jedoch bestehend: Menschen. Wenn Anwendende aufgrund von Unwissenheit, Unachtsamkeit oder anderen Gründen das Passwort zur Festplattenentschlüsselung eingeben, könnte dies mitgeloggt werden. Angreifende könnten dank der erhöhten Ausführungsrechte des kompromittierten Kernels und der Kenntnis des Passwortes einen neuen Clevis-Eintrag anlegen. Dieser Eintrag könnte eine Policy mit den veränderten PCR-Werten sein oder auch andere Register verwenden, welche sich während des Boot-Prozesses nicht verändern zum Beispiel PCR 23. Clevis bietet keine Möglichkeit zu überprüfen, welche Policy für einen Eintrag verwendet wurde.
Die Schwachstelle Mensch könnte mittels Social Engineering-Angriffen auch durch einen remote-Angriff ausgenutzt werden, um das Sicherheitskonzept zu untergraben. Bei Social Engineering-Angriffen werden nicht direkt Schwachstellen in Computersystemen ausgenutzt, sondern menschliche Schwächen, wie Unsicherheit, Wut oder Gier. Das Ziel dieser Angriffe ist es Informationen zu sammeln oder Anwendende zu gewissen Aktionen zu verleiten [143]. Dabei könnten Anwendende ausgetrickst werden, mit den installierten UEFI-Schlüsseln ein manipuliertes Boot-Bündel zu Signieren. Angreifende ersetzen mit diesem das ursprüngliche Bündel. Wenn der Name und der Pfad gleich bleiben, muss der Booteintrag nicht ersetzt werden. Das System bootet wie gewohnt und die PCR-Policy wird nicht verletzt. Ein sicherer Umgang mit den UEFI-Schlüsseln ist daher essenziell. Des Weiteren sollte ein BIOS-Passwort festgelegt werden. Dies verhindert den unautorisierten Zugriff auf das BIOS-Menü und das Booten eines anderen Systems durch einen physischen Angriff, wenn das BIOS nicht zurückgesetzt wird. Das BIOS inklusive Passwort kann, so Halsey [56], bei einem physischen Angriff mittels CMOS Jumper, ein kleiner Schalter am Mainboard, zurückgesetzt werden. Dafür muss jedoch das Gerät geöffnet werden.

8.4 Analyse des Updateprozesses

Die Robustheit des Konzeptes dieser Arbeit, ist sehr von der Wahl der PCR-Werte abhängig. Wie in Tabelle 7.2 ersichtlich verändern sich die Register aus verschiedensten Gründen. Nicht jedes System agiert gleich, und verändert dieselben PCR-Werte auf dieselbe Weise. In ein Register können zudem mehrere Messwerte geladen werden. Der finale Hashwert ist damit abhängig vom Zustand der Komponenten und in welcher Reihenfolge diese gemessen werden. Beim Auslesen der PCR-Werte ist lediglich zu erkennen, dass sich der Wert gegebenenfalls geändert hat, jedoch nicht warum. Dies erschwert es Anwendenden zu erkennen, ob es sich um eine reguläre Veränderung oder um eine Kompromittierung handelt. Das Binden an PCR 7 verringert das Risiko der Fragilität des Konzeptes, jedoch verringert dies auch die Garantie unerwünschte Veränderungen zu erkennen. Anwendende müssen für sich selbst die Balance finden das System ausreichend abzusichern und den Wartungsaufwand akzeptabel zu halten oder gar sich selbst auszusperren.
Mit dem in dieser Arbeit präsentierten Konzept können Anwendende den Bootloader, den Kernel und das Betriebssystem Aktualisieren, ohne dabei die Policy zu brechen. Wenn, eigene UEFI-Schlüssel verwendet werden, müssen die Bootkomponenten nach dem Update entsprechend signiert werden. Ein BIOS-Update bricht zwar immer noch die Policy, doch da diese Updates von Anwendenden explizit und bewusst unter Linux

vorgenommen werden müssen, wird dies in Kauf genommen, um die Sicherheit zu erhöhen. Des Weiteren können Anwendende nach einem Update sich mittels Passwort Zugang verschaffen und die Policy ersetzen.
Die Policy kann auch vor einem BIOS-Update erneuert werden, falls Herstellerfirmen über die LVFS Plattform die Firmware inklusive Soll-Wert für PCR 0 bereitstellen. Anwendende sollten zudem ein Passwort für das BIOS-Menü festlegen, sodass Angreifende mit physischem Zugriff nicht UEFI Secure Boot deaktivieren beziehungsweise von einem USB-Stick ein anderes System booten können. Dies ist lediglich eine erweiterte Schutzmaßnahme, das Konzept ist jedoch, wie im Abschnitt 8.3 beschrieben nicht auf die Prävention von physischen oder Social Engineering- Angriffen ausgelegt.

KAPITEL 9

Zusammenfassung und Ausblick

Rootkits sind eine Malware-Art, die mit erhöhten Rechten das System kompromittieren. Ihr Ziel ist es unentdeckt zu bleiben und das System langfristig zu manipulieren. Rootkits sind meist nur ein Teil eines erweiterten Angriffes. Sie öffnen häufig Hintertüren, um weitere Schadsoftware zu laden. Des Weiteren versuchen sie jegliche Malwareaktionen und jeden maliziösen Prozess vor dem System zu verbergen. Rootkits können vom BIOS bis zum Kernel jede Bootkomponente befallen. Schadsoftware im BIOS oder im Bootloader wird auch Bootkit genannt. Je tiefer, also früher im Bootprozess, Rootkits agieren desto schwieriger ist es sie zu erkennen. Eine kompromittierte Komponente kann alle Nachfolgenden manipulieren und sich so verstecken. Daher reicht es nicht aus die Integrität der letzten Komponenten, zum Beispiel des Kernels, zu gewährleisten, um ein vertrauenswürdiges System zu haben. Es muss eine Vertrauenskette von einer vertrauenswürdigen Basis bis ins Betriebssystem aufgebaut werden. Wobei jedes Glied das Nächste vor seine Ausführung überprüft. Die erste Komponente der Vertrauenskette überprüft sich jedoch nur selbst. Dadurch ist es essenziell dieses vor Manipulationen zu schützen und gegebenenfalls in der Hardware zu verankern.
Das Trusted Platform Module (TPM) 2.0 ist eine kryptographisches Hardwaremodul, welches unter anderem die Messungen der Bootkomponenten sicher speichert und beglaubigen kann. Die Hashwerte der Komponenten und ihrer Konfigurationen werden in den Platform Configuration Registers (PCRs) abgelegt und spiegeln dadurch den Systemzustand wider. PCRs können eingesetzt werden um mittels TPM2-Policys den Zugriff auf Objekte, wie etwa Schlüssel oder andere Daten zu beschränken. Im Konzept dieser Arbeit wird der Linux Bootprozess pausiert, sobald die gewählten Register nicht die erwarteten Werte haben. Dafür wird eine Partition verschlüsselt und ihre Freigabe an eine PCR-Policy gebunden. Bei der Variante aus Abschnitt 7.3.1 ist es die `/boot` Partition, auf welcher sich der Kernel und das initramfs befinden. Bei der Variante aus Abschnitt 7.3.2 bootet das UEFI direkt den Kernel und es wird die `/root` Partition mit allen persönlichen Dokumenten verschlüsselt. Sollten die Messwerte mit den Er-

warteten Werten in der PCR-Policy übereinstimmen, wird die Partition automatisch entschlüsselt, andernfalls müssen Anwendende ein Passwort eingeben. Durch die Option der Passworteingabe sollen Anwendende über eine mögliche Kompromittierung informiert, jedoch nicht automatisch aus dem System ausgesperrt werden. Des Weiteren wird UEFI Secure Boot aktiviert, sodass nur Komponenten mit einer gültigen Signatur oder einem validen Hashwert ausgeführt werden. Bei der, bedienungsfreundlicheren Variante aus Abschnitt 7.3.1, werden die vorinstallierten UEFI-Schlüssel verwendet. Für die Variante aus Abschnitt 7.3.2 werden eigene Schlüssel eingesetzt, um erhöhte Kontrolle über UEFI Secure Boot zu erlangen. Dabei wird der Kernel direkt vom UEFI gebootet, wodurch der Bootloader und eventuelle Schwachstellen, wie in Abschnitt 4.2.1, umgangen werden können.
Da das TPM-gebundene Entschlüsseln der `/boot` Partition zum Zeitpunkt dieser Arbeit vom Bootloader GRUB2 nicht unterstützt wird, wird der zweite Ansatz demonstriert. Die, mit LUKS verschlüsselte, Partition wird vom Kernel mithilfe des Programms Clevis basierend auf den PCR-Werten der UEFI-Firmware und der Secure Boot-Konfiguration entschlüsselt. Das aktivierte UEFI Secure Boot erschwert in beiden Varianten die Ausführung von Kernel-basierten Rootkits. Falls jedoch im Zuge eines Angriffes UEFI Secure Boot deaktiviert oder die UEFI Firmware kompromittiert wird, kann die erste Ausführung von maliziösen Code nicht verhindert werden, da die Überprüfung durch das TPM erst nach dem Laden des Kernels erfolgt. Anwendende werden jedoch über Inkonsistenzen informiert und können entsprechende Maßnahmen, wie etwa eine Neuinstallation der Komponenten, vornehmen.

GRUB2 unterstützt zum Zeitpunkt dieser Arbeit lediglich das Erweitern von PCR-Werten. In zukünftigen Arbeiten, könnte GRUB2 um weitere TPM2-Befehle erweitert werden, sodass der Bootloader anhand von TPM-Policys in den Bootprozess eingreifen kann und so die Ausführung von Rootkit-Code im Kernel verhindern kann. Ein weiterer Ansatz für zukünftige Projekte wäre die Integration von Clevis in den Bootloader. Dadurch könnte die prototypische Umsetzung dieser Arbeit direkt von der `/root` Partition auf die `/boot` Partition umgesetzt werden.
Mit dem Konzept dieser Arbeit, wissen Anwendende lediglich über die Inkonsistenz der gewählten PCR-Werte Bescheid, weil sie ein Passwort für die Entschlüsselung eingeben müssen. Jedoch könnte in zukünftigen Projekten versucht werden, bei nicht Erfüllung der Policy, Informationen zur Verfügung zu stellen, welche Werte sich verändert haben. Das würde Anwendenden genauere Informationen liefern, welche Komponenten betroffen sind. Dadurch kann die Entscheidung das System trotzdem zu starten gezielter getroffen werden. Die betroffenen Komponenten könnten dadurch gegebenenfalls durch ein gezieltes Aktualisieren oder Zurücksetzen wieder in einen vertrauenswürdigen Zustand gebracht werden.
In dieser Arbeit wurde lediglich der Ansatz der TPM-abhängigen Entschlüsselung behandelt. Es gibt jedoch auch Ansätze, unter anderem von Garrett [48], Remote Attestation für private Systeme umzusetzen. Die Verfahrensweise ist ähnlich, wie in Abschnitt 5.5 erklärt, jedoch wird das Smartphone oder ein anderes Gerät als Attestation Server verwendet.

Bei seinem Ansatz wird ein TPM- und Zeit basiertes Einmalpasswort (TPMTOTP) auf dem Computer und dem anderen Gerät angezeigt. Wenn dieses übereinstimmt, können Anwendende damit rechnen, dass das System in einem vertrauenswürdigen Zustand ist. Es kann jedoch auch über Bluetooth die signierten PCR-Werte abgefragt und mit dem Eventlog gegengeprüft werden.

KAPITEL 10

Anhang

Listing 10.1: tpm2-tools und Abhängigkeiten installieren [134]

```
tpm2-tss
$  git clone https://github.com/tpm2-software/tpm2-tss.git
$  cd tpm2-tss
$  ./bootstrap
$  ./configure --prefix=/usr
$  make -j5
$  sudo make install

tpm2-abrmd
$  git clone https://github.com/tpm2-software/tpm2-abrmd.git
$  cd tpm2-abrmd
$  ./bootstrap
$  ./configure --with-dbuspolicydir=/etc/dbus-1/system.d (aus
   ↪ tevora) (muusste glib-devel noch installieren)
$  make -j5
$  sudo make install

tpm2-tools
$  git clone https://github.com/tpm2-software/tpm2-tools.git
$  cd tpm2-tools
$  ./bootstrap
$ ./configure --prefix=/usr
$  make -j5
$  sudo make install
```

PCR	5.6.6 Kernel, Fedora 32, SB aus, BIOS PW aus keine Kernel Module	5.8.18 Kernel, Fedora 33, SB ein , BIOS PW ein, MOK geladen, BIOS aktualisiert
0	0x5D8B9F33FE3374705A8A6928C9BA1502D5B53DE5DFC2EEF92745BA1C271EFEF9	0x6807296F8882091CCA11EBE302DBD7958FBD025928EDEA0F8FB428987722BC69
1	0x57160E6ED1D11A7DFBADA35A83A8B81F2B24EFE1F50964018EFE25E3A6E2C3E0	0x87C77E9C6E5ED3460D615C9FAF03FB78BAE606731BD3CE0E269DA5A1343A8585
2	0x7FC0DC82633D1D2E88F1B5D221CD3979FEEDEFD770234C303B0209EFCC3F01A4	0x8A78824A5ED538288BC70DF7AADDCA004F60AF679516B265507596B8191D6862
3	0x3D458CFE55CC03EA1F443F1562BEEC8DF51C75E14A9FCF9A7234A13F198E7969	0x3D458CFE55CC03EA1F443F1562BEEC8DF51C75E14A9FCF9A7234A13F198E7969
4	0x52CFB0C94DA043B271A557F7DACCBCD6973CF654508366B92A268F33F6F3BCB0	0x80D1CB1DC44838D93B208B0FE8C75024593BC93F4FC918A34C88BB6C90C0261B
5	0x22245B2FF1FF0344E961A7EE2857680295070C3619CB495E0B7EF99AA7E93014	0xBC9D4A9D4DD9BC5E5543591ED6BF80C040EC89DA93CC881D23746A8663AF7C05
6	0x3D458CFE55CC03EA1F443F1562BEEC8DF51C75E14A9FCF9A7234A13F198E7969	0x3D458CFE55CC03EA1F443F1562BEEC8DF51C75E14A9FCF9A7234A13F198E7969
7	0xF8FC7E18A97DF515FBCF0F939C5F1E02698729A7D7D64335C937EF83875F33F1	0x19EE7E27777996EB3690546CA1DA8D951B95CB2BFBF06F6D7973431C0CD734D1
8	0x0000000000000000000000000000000000000000000000000000000000000000	0x827F243B9BBF6509A644FDBD2F3A7C2CC8E675B391189C7EE8DAB09511D57AD9
9	0x0000000000000000000000000000000000000000000000000000000000000000	0xF56A01D7F45FB168346D558EC6B19D934A4DE7700B6A507A418ADA250C266D32
10	0xDDC601EBA47F85FFE6CB666DD8BD5E943C6A58F38AF8B13BA7182CC4A4C6AA71	0x9F6ECBF2EA8C6F5E161F48EE97C4AF37D0B1648B1BCDE01550B47182CF380060
11	0x0000000000000000000000000000000000000000000000000000000000000000	0x0000000000000000000000000000000000000000000000000000000000000000
12	0x0000000000000000000000000000000000000000000000000000000000000000	0x0000000000000000000000000000000000000000000000000000000000000000
13	0x0000000000000000000000000000000000000000000000000000000000000000	0x0000000000000000000000000000000000000000000000000000000000000000
14	0x0000000000000000000000000000000000000000000000000000000000000000	0x904C9DC4EB84B000840AA3E84B3998FBB354D51690455D74A8A7ED53EA76C697
15	0x0000000000000000000000000000000000000000000000000000000000000000	0x0000000000000000000000000000000000000000000000000000000000000000
16	0x0000000000000000000000000000000000000000000000000000000000000000	0x0000000000000000000000000000000000000000000000000000000000000000
17	0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF	0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
18	0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF	0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
19	0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF	0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
20	0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF	0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
21	0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF	0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
22	0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF	0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
23	0x0000000000000000000000000000000000000000000000000000000000000000	0x0000000000000000000000000000000000000000000000000000000000000000

Tabelle 10.1: PCR-Werte beim ersten Booten und nach den Aktionen von 7.2

Listing 10.2: LUKS Partition mit einem Zusätzlichen Eintrag für Clevis TPM2 Pin

```
LUKS header information
Version:        2
Epoch:          21
Metadata area:  16384 [bytes]
Keyslots area:  16744448 [bytes]
UUID:           12e487fb-b3e1-446d-8b08-7f4e1a67b8dd
Label:          (no label)
Subsystem:      (no subsystem)
Flags:          (no flags)

Data segments:
   0: crypt
   offset: 16777216 [bytes]
   length: (whole device)
   cipher: aes-xts-plain64
   sector: 512 [bytes]

Keyslots:
   0: luks2
      Key:        512 bits
      Priority:   normal
      Cipher:     aes-xts-plain64
      Cipher key: 512 bits
      PBKDF:      argon2i
      Time cost:  8
      Memory:     1048576
      Threads:    4
      Salt:       42 18 ab 18 df 50 93 ec a8 75 d9 86 01 33 ab
         ↪ 5a 0f 30 c8 53 83 92 b7 9d 3d 74 1c 91 1e 80 b3 e6
      AF stripes: 4000
      AF hash:    sha256
      Area offset:32768 [bytes]
      Area length:258048 [bytes]
      Digest ID:  0
   1: luks2
      Key:        512 bits
      Priority:   normal
      Cipher:     aes-xts-plain64
      Cipher key: 512 bits
      PBKDF:      argon2i
      Time cost:  8
      Memory:     1048576
```

```
      Threads:    4
      Salt:       5e a3 4c d8 94 bf 2f 8b 81 6d 8c 24 1d 05 bb
         ↪ 07 63 82 d2 6c b0 6f 2f c7 08 6d d8 1d af e7 e7 77
      AF stripes: 4000
      AF hash:    sha256
      Area offset:290816 [bytes]
      Area length:258048 [bytes]
      Digest ID:  0
Tokens:
   0: clevis
      Keyslot:  1
Digests:
   0: pbkdf2
      Hash:       sha256
      Iterations: 224823
      Salt:       45 d2 a3 f1 59 6a 4e 57 f5 0b 02 14 14 23 21
         ↪ b3 ff 82 fe 92 b5 2d bc e6 ee 8b d9 58 7d 6e 42 c3
      Digest:     23 20 bd 74 75 3d 08 61 2c 4a 97 07 bd 2a b7
         ↪ 3f e0 d5 de db 6a e5 08 f4 95 72 43 a6 fc 97 f3 5b
```

Abbildungsverzeichnis

Tabellenverzeichnis

Akronyme

ACM Authenticated Code Module 49

AIK Attestation Identity Key 36, 37

API Application Programming Interface 38

BIOS Basic Input/Output System 1, 6, 10, 11, 18, 21, 25, 26, 32, 37, 46, 48, 49, 59

CA Certificate Authority 11, 46

CMM Core Measurement Mechanism 5

CRB Command/Response Buffer 40

CRTM Core Root of Trust for Measurement 10, 17, 18, 32

CSM Compatibility Support Module 11, 51

D-RTM Dynamic Root of Trust for Measurement 6, 18, 32

DAA Direct Anonymous Attestation 36

DR Dark Regions 21, 22, 59

DXE Driver Execution Environment 11, 26

EK Endorsement Key 36

ESAPI Enhanced System API 38–40

FAPI Feature API 39

GPT GUID Partition Table 11, 26

GRUB GRand Unified Bootloader 12

IDT Interrupt-Descriptor-Table 23, 24

IMA Integrity Measurement Architecture 5, 32, 45

IMM Integrity Measurment Mechanism 5

initrd Initial-RAMdisk 14, 22, 49

Intel TXT Intel Trusted Execution Technology 2, 6, 18

KDF Key Derivation Function 36

KEK Key Exchange Key 46, 47

LCP Launch Control Policy 6, 49

LKM Loadable Kernel-Module 23

MBR Master Boot Record 10, 11, 18, 21, 25, 26, 32, 46, 49

MOK Machine Owner Key 12, 52

MUAPI Marshaling/Unmarshaling API 38, 40

NVRAM Non-Volatile Random Access Memory 30, 42, 47

PCA Privacy Certificate Authority 36

PCR Platform Configuration Register 2, 6, 18, 31, 33, 36, 37, 41, 45, 47, 49

PEI Pre-EFI Initialization 11

PI Platform Initialization 46–48

PID Prozess-ID 15, 16

PK Platform Key 36, 46, 47

RNG Random Number Generator 30

RoT Root of Trust 11, 17, 47

RTM Root of Trust for Measurement 17

RTR Root of Trust for Reporting 18

RTS Root of Trust for Storage 17, 18

S-CRTM Static Core Root of Trust for Measurement 6, 7, 18, 32

S-RTM Static Root of Trust for Measurement 6, 18, 41

SAPI System API 38–40

SML Stored Measurement Log 45

SMM System Management Mode 11, 14, 15, 37

SRK Storage Root Key 36

TAB TPM Access Broker 40

TCB Trusted Computing Base 16

TCG Trusted Computing Group 1, 2, 5, 16–18, 38, 51

TCTI TPM Command Transmission Interface 40

TPM Trusted Platform Module 1–3, 5, 17, 18, 29–34, 36–38, 40–43, 45–49, 51, 59

TSS TCG Software Stack 38

UEFI Unified Extensible Firmware Interface Spezification 1, 2, 10–12, 17, 25, 26, 46, 59

VBR Volume Boot Record 10, 21, 25

VMM Virtual Machine Monitor 6, 15, 49

Literatur

[1] The Trusted Computing Group (TCG). *About TCG*. URL: https://trustedcomputinggroup.org/about/ (besucht am 01. 04. 2021).

[2] The Trusted Computing Group (TCG). *TCG PC Client Platform Firmware Profile Specification*. Revision 1.04. 2019. URL: https://trustedcomputinggroup.org/wp-content/uploads/TCG_PCClientSpecPlat_TPM_2p0_1p04_pub.pdf (besucht am 01. 04. 2021).

[3] The Trusted Computing Group (TCG). *TCG PC Client Specific Implementation Specification for Conventional BIOS*. Revision 1.00. 2012. URL: https://www.trustedcomputinggroup.org/wp-content/uploads/TCG_PCClientImplementation_1-21_1_00.pdf (besucht am 01. 04. 2021).

[4] The Trusted Computing Group (TCG). *Trusted Boot*. 2009. URL: https://trustedcomputinggroup.org/resource/trusted-boot/ (besucht am 01. 04. 2021).

[5] Computer Security Center (US). *Computer Security Requirements: Guidance for Applying the Department of Defense Trusted Computer System Evaluation Criteria in Specific Environments*. 1985. DOI: 10.1007/978-1-349-12020-8_5.

[6] André Jorge Marques de Almeida. „Rootkits-Detection and prevention“. In: (2008).

[7] archlinux. *Trusted Platform Module*. URL: https://wiki.archlinux.org/index.php/Trusted_Platform_Module (besucht am 01. 04. 2021).

[8] Will Arthur und David Challener. *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*. 2015. DOI: 10.1007/978-1-4302-6584-9.

[9] Naveeda Ashraf u. a. „Analytical study of hardware-rooted security standards and their implementation techniques in mobile“. In: *Telecommunication Systems* (2020). DOI: 10.1007/s11235-020-00656-y.

[10] Yogesh Babar. 2020. DOI: 10.1007/978-1-4842-5890-3.

[11] Yogesh Babar. *Hands-On Booting: Learn the Boot Process of Linux, Windows, and Unix*. 2020. ISBN: 978-1484258897.

[12] Vladimir Bashun u. a. „Too young to be secure: Analysis of UEFI threats and vulnerabilities“. In: *14th Conference of Open Innovation Association FRUCT*. 2013. DOI: 10.1109/FRUCT.2013.6737940.

[13] Christian Baun. „Virtualization/Virtualisierung“. In: *Operating Systems/Betriebssysteme*. 2020. DOI: 10.1007/978-3-658-29785-5_10.

[14] *Booten und Konfigurieren eines Linux-Systems*. URL: https://www.pks.mpg.de/~mueller/docs/suse10.3/opensuse-manual_de/manual/cha.boot.html (besucht am 01. 04. 2021).

[15] David Both. *An introduction to GRUB2 configuration for your Linux machine*. 2017. URL: https://opensource.com/article/17/3/introduction-grub2-configuration-linux (besucht am 01. 04. 2021).

[16] James Bottomley. *efitools*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/jejb/efitools.git/ (besucht am 01. 04. 2021).

[17] Plymouth Bug. *plymouth splash is not dismissed when LUKS device is unlocked non-interactively*. 4. Sep. 2020. URL: https://gitlab.freedesktop.org/plymouth/plymouth/-/issues/126 (besucht am 01. 04. 2021).

[18] Yuriy Bulygin u. a. „Summary of attacks against BIOS and secure boot“. In: *Defcon-22* (2014).

[19] Andreas Bunten. „Unix and linux based rootkits techniques and countermeasures“. In: *16th Annual First Conference on Computer Security Incident Handling, Budapest*. 2004. DOI: 10.17877/DE290R-2026.

[20] Dhiman Chakraborty, Lucjan Hanzlik und Sven Bugiel. „simTPM: User-centric TPM for Mobile Devices“. In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019. ISBN: 978-1-939133-06-9.

[21] David Challener u. a. *A practical guide to trusted computing*. 2007. ISBN: 978-0132398428.

[22] Liqun Chen, Chris J Mitchell und Andrew Martin. *Trusted Computing: Second International Conference, Trust 2009 Oxford, UK, April 6-8, 2009, Proceedings*. 2009. DOI: 10.1007/978-3-642-00587-9.

[23] Ronny Chevalier u. a. „BootKeeper: Validating Software Integrity Properties on Boot Firmware Images“. In: *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*. 2019. DOI: 10.1145/3292006.3300026.

[24] Sansar Choinyambuu. *A Root of Trust for Measurement*. 2011.

[25] *Clevis*. URL: https://github.com/latchset/clevis (besucht am 01. 04. 2021).

[26] David Cooper u. a. „BIOS protection guidelines“. In: *NIST Special Publication 800-147* (2011). DOI: 10.6028/NIST.SP.800-147.

[27] *coreboot Documentation*. URL: https://doc.coreboot.org/index.html (besucht am 01. 04. 2021).

[28] Ivan Bjerre Damgård. „A design principle for hash functions“. In: *Conference on the Theory and Application of Cryptology*. 1989. DOI: 10.1007/0-387-34805-0_39.

[29] Debian. *Grub*. URL: https://wiki.debian.org/Grub (besucht am 01. 04. 2021).

[30] DistroWatch. *DistroWatch Page Hit Ranking*. URL: https://distrowatch.com/dwres.php?resource=popularity (besucht am 01. 04. 2021).

[31] Christopher Domas. „The memory sinkhole“. In: *BlackHat USA* (2015).

[32] Loc Duflot, Daniel Etiemble und Olivier Grumelard. „Using CPU System Management Mode to Circumvent Operating System Security Functions“. In: *CanSecWest/core06* (2001).

[33] Nedwal Falk. *Hardware absichern mit IMA und Trusted Boot*. URL: https://www.linux-magazin.de/ausgaben/2011/11/trusted-boot/2/ (besucht am 01. 04. 2021).

[34] Fedora. *"Minor"bug fixes*. 1. Aug. 2020. URL: https://src.fedoraproject.org/rpms/grub2/c/47cf63735cd44c6a1b9de8797933f730c5e4969e?branch=f33 (besucht am 01. 04. 2021).

[35] Fedora. *Encrypting drives using LUKS*. URL: https://docs.fedoraproject.org/en-US/quick-docs/encrypting-drives-using-LUKS/ (besucht am 01. 04. 2021).

[36] Fedora. *User Documentation*. URL: https://docs.fedoraproject.org/en-US/docs/ (besucht am 01. 04. 2021).

[37] *Fedora 33 - NVMe Drive - clevis luks bind tpm2 issue*. URL: https://github.com/latchset/clevis/issues/260 (besucht am 01. 04. 2021).

[38] flihp. *Parsing the TPM2 Event Log from Userspace*. URL: https://twobit.org/2020/01/27/parsing-the-tpm2-event-log-from-userspace/ (besucht am 01. 04. 2021).

[39] Source Forge. *Trusted Boot*. 2020. URL: https://sourceforge.net/projects/tboot/ (besucht am 01. 04. 2021).

[40] William Futral und James Greene. *Intel Trusted Execution Technology for Server Platforms*. 2013. DOI: 10.1007/978-1-4302-6149-0.

[41] Hongbo Gao u. a. „Research on the working mechanism of Bootkit“. In: *2012 8th International Conference on Information Science and Digital Content Technology (ICIDT2012)*. 2012. ISBN: 978-1467312882.

[42] Matthew Garrett. *A detailed technical description of Shim*. 30. Okt. 2012. URL: https://mjg59.dreamwidth.org/19448.html (besucht am 01. 04. 2021).

[43] Matthew Garrett. *Avoiding TPM PCR fragility using Secure Boot*. 17. Juli 2017. URL: https://mjg59.dreamwidth.org/48897.html (besucht am 01. 04. 2021).

[44] Matthew Garrett. *grub*. 2016. URL: https://github.com/mjg59/grub (besucht am 01. 04. 2021).

[45] Matthew Garrett. *Intel Boot Guard, Coreboot and user freedom*. 2015. URL: http://mjg59.dreamwidth.org/33981.html (besucht am 01. 04. 2021).

[46] Matthew Garrett. *Shim*. URL: https://github.com/mjg59/shim (besucht am 01. 04. 2021).

[47] Matthew Garrett. *Towards Measured Boot Out of the Box*. 2016. URL: http://events17.linuxfoundation.org/sites/events/files/slides/security_summit_2016_tpms.pdf (besucht am 01. 04. 2021).

[48] Matthew Garrett. *TPM based attestation - how can we use it for good?* (LCA2020), 17. Jän. 2020. URL: https://archive.org/details/lca2020-TPM_based_attestation_how_can_we_use_it_for_good (besucht am 01. 04. 2021).

[49] Robert P Goldberg. „Architecture of virtual machines“. In: *Proceedings of the workshop on virtual computer systems*. 1973. DOI: 10.1145/800122.803950.

[50] Google. *Verified Boot*. 2020. URL: https://source.android.com/security/verifiedboot (besucht am 01. 04. 2021).

[51] Bernhard Grill u. a. „“Nice Boots!”-A Large-Scale Analysis of Bootkits and New Ways to Stop Them“. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 2015. DOI: 10.1007/978-3-319-20550-2_2.

[52] Jonathan Grimm u. a. „Automatic Mitigation of Kernel Rootkits in Cloud Environments“. In: *International Workshop on Information Security Applications*. 2018. DOI: 10.1007/978-3-319-93563-8_12.

[53] *grub.git*. 2018. URL: http://git.savannah.gnu.org/cgit/grub.git/commit/?h=grub-2.04&id=d6ca0a90cac78f40ea4a04ac6894784c5313cbe9 (besucht am 01. 04. 2021).

[54] Yutian Gui u. a. „Key Update Countermeasure for Correlation-Based Side-Channel Attacks“. In: *Journal of Hardware and Systems Security* (2020). DOI: 10.1007/s41635-020-00094-x.

[55] Jakob Hagl, Oliver Mann und Martin Pirker. „Securing the Linux Boot Process: From Start to Finish“. In: *Proceedings of the 7th International Conference on Information Systems Security and Privacy - Volume 1: ICISSP,* 2021. DOI: 10.5220/0010313906040610.

[56] Mike Halsey. „Understanding PC Hardware“. In: *Windows 10 Troubleshooting*. 2016. DOI: 10.1007/978-1-4842-0925-7_6.

[57] Seunghun Han u. a. „A bad dream: Subverting trusted platform module while you are sleeping“. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018. ISBN: 978-1931971461.

[58] Chris Hoffman. *How Intel and PC makers prevent you from modifying your laptop's firmware*. 2015. URL: https://www.pcworld.com/article/2883903/how-intel-and-pc-makers-prevent-you-from-modifying-your-pcs-firmware.html (besucht am 01. 04. 2021).

[59] Greg Hoglund und James Butler. *Rootkits: subverting the Windows kernel*. 2006. ISBN: 978-1593277161.

[60] HP. *HP Verbraucher-Notebook - Aktualisierung des BIOS (Basic Input/Output System)*. URL: https://support.hp.com/at-de/document/c00132083 (besucht am 01. 04. 2021).

[61] *HP Sure Start*. HP. 2019. URL: http://h10032.www1.hp.com/ctg/Manual/c06216928 (besucht am 01. 04. 2021).

[62] Richard Hughes. *Linux Vendor Firmware Service*. URL: https://fwupd.org/users (besucht am 01. 04. 2021).

[63] Richard Hughes. *Linux Vendor Firmware Service*. URL: https://lvfs.readthedocs.io/en/latest/claims.html#device-checksums (besucht am 01. 04. 2021).

[64] Richard Hughes. *Updating Secure Boot dbx with fwupd and the LVFS*. 17. Aug. 2020. URL: https://blogs.gnome.org/hughsie/2020/08/17/updating-secure-boot-dbx-with-fwupd-and-the-lvfs/ (besucht am 01. 04. 2021).

[65] Ralf Hund, Thorsten Holz und Felix C Freiling. „Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms". In: *Proceedings of the 18th USENIX Security Symposium*. 2009.

[66] IceLord. *BIOS RootKit: Welcome Home, My Lord*. 2007. URL: https://blog.csdn.net/icelord/article/details/1604884 (besucht am 01. 04. 2021).

[67] AV-TEST Institute. *AV-TEST Sicherheitsreport*. URL: https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Sicherheitsreport_2018-2019.pdf (besucht am 01. 04. 2021).

[68] AV-TEST Institute. *Malware Statistics*. URL: https://www.av-test.org/en/statistics/malware/ (besucht am 01. 04. 2021).

[69] Intel. *SINIT Buffer Overflow Vulnerability*. 6. Dez. 2011. URL: https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00030.html (besucht am 01. 04. 2021).

[70] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel. 2020. URL: https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html (besucht am 01. 04. 2021).

[71] *Intel® Platform Innovation Framework for UEFI Compatibility Support Module Specification*. Intel. 2013. URL: https://www.intel.com/content/dam/www/public/us/en/documents/reference-guides/efi-compatibility-support-module-specification-v098.pdf (besucht am 01. 04. 2021).

[72] *Intel® Trusted Execution Technology (Intel® TXT) Software Development Guide*. Intel. 2020. URL: http://www.intel.com/content/www/us/en/software-developers/intel-txt-software-development-guide.html?wapkw=measured+launched+environment+developer%E2%80%99s+guide (besucht am 01. 04. 2021).

[73] *Trusted Platform Module Library, Part 1: Architecture*. Techn. Ber. 2015.

[74] Trent Jaeger, Reiner Sailer und Umesh Shankar. „PRIMA: Policy-Reduced Integrity Measurement Architecture“. In: *Proceedings of the eleventh ACM symposium on Access control models and technologies*. 2006. DOI: 10.1145/1133058.1133063.

[75] Xiaoqi Jia u. a. „Performing Trusted Computing Actively Using Isolated Security Processor“. In: *Proceedings of the 1st Workshop on Security-Oriented Designs of Computer Architectures and Processors*. 2018. DOI: 10.1145/3267494.3267498.

[76] Yongjin Kim und Evan Kim. „HTPM: Hybrid Implementation of Trusted Platform Module“. In: 2019. DOI: 10.1145/3338511.3357348.

[77] Michael Kofler. *Linux: Installation, Konfiguration, Anwendung*. 2008. ISBN: 978-3827327529.

[78] Kowalski7cc. *Automatic LUKS 2 disk decryption with TPM 2 and Clevis on Fedora 31*. URL: https://kowalski7cc.xyz/blog/luks2-tpm2-clevis-fedora31 (besucht am 01. 04. 2021).

[79] Piotr Król. „TPM 2.0 Linux sysfs interface“. LPC 2019: System Boot and Security MC. 2019. URL: https://linuxplumbersconf.org/event/4/contributions/516/attachments/396/639/TPM_2.0_Linux_sysfs_interface.pdf (besucht am 01. 04. 2021).

[80] Christopher Kruegel, William Robertson und Giovanni Vigna. „Detecting kernel-level rootkits through binary analysis“. In: *20th Annual Computer Security Applications Conference*. 2004. DOI: 10.1109/CSAC.2004.19.

[81] Eva-Katharina Kunst und Jürgen Quade. *Linux control over Secure Boot*. 2018. URL: https://www.linux-magazine.com/Issues/2018/206/Linux-Secure-Boot-with-Shim (besucht am 01. 04. 2021).

[82] Ievgeniia Kuzminykh und Maryna Yevdokymenko. „Analysis of Security of Rootkit Detection Methods“. In: *2019 IEEE International Conference on Advanced Trends in Information Theory (ATIT)*. 2019. DOI: 10.1109/ATIT49449.2019.9030428.

[83] Michael Larabel. *TPM 2.0 Support Sent In For The Linux 3.20 Kernel*. 15. Feb. 2015. URL: https://www.phoronix.com/scan.php?page=news_item&px=Linux-3.20-TPM-2.0-Security (besucht am 01. 04. 2021).

[84] Hojoon Lee, Chihyun Song und Brent Byunghoon Kang. „Lord of the X86 Rings: A Portable User Mode Privilege Separation Architecture on X86“. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018. DOI: 10.1145/3243734.3243748.

[85] Suhho Lee und Seehwan Yoo. „Tux: Trust Update on Linux Booting“. In: *International Workshop on Security and Trust Management*. 2018. DOI: 10.1007/978-3-030-01141-3_7.

[86] Michael Leibowitz. *HORSE PILL, A New Kind of Linux Rootkit*. (Blackhat USA 2016), 30. Juli 2026. URL: https://www.blackhat.com/docs/us-16/materials/us-16-Leibowitz-Horse-Pill-A-New-Type-Of-Linux-Rootkit.pdf (besucht am 01. 04. 2021).

[87] X. Li u. a. „An Overview of Bootkit Attacking Approaches“. In: *2011 Seventh International Conference on Mobile Ad-hoc and Sensor Networks*. 2011. DOI: 10.1109/MSN.2011.19.

[88] MX Linux. *HELP: MX Boot options*. URL: https://mxlinux.org/wiki/help-files/help-mx-boot-options/ (besucht am 01. 04. 2021).

[89] Manjaro. *GRUB/Restore the GRUB Bootloader*. URL: https://wiki.manjaro.org/index.php/GRUB/Restore_the_GRUB_Bootloader (besucht am 01. 04. 2021).

[90] Javier Martinez Canillas. *Auto unlock support for boot partition in Grub2*. 16. Juli 2020. URL: https://gitlab.freedesktop.org/plymouth/plymouth/-/issues/126 (besucht am 01. 04. 2021).

[91] Alex Matrosov, Eugene Rodionov und Sergey Bratus. *Rootkits and bootkits: reversing modern malware and next generation threats*. 2019. ISBN: 978-1593277161.

[92] Gordon Matzigkeit und Yoshinori K Okuji. *the GNU GRUB manual. The GRand Unified Bootloader, version 2.04*. 2019. URL: https://www.gnu.org/software/grub/manual/grub/grub.pdf.

[93] Wolfgang Mauerer. *Professional Linux kernel architecture*. 2010. ISBN: 978-0470343432.

[94] Microsoft. *Minimum hardware requirements*. 2017. URL: https://docs.microsoft.com/en-us/windows-hardware/design/minimum/minimum-hardware-requirements-overview (besucht am 01. 04. 2021).

[95] Linux Mint. *Grub Boot Menu*. URL: https://linuxmint-user-guide.readthedocs.io/en/latest/grub.html (besucht am 01. 04. 2021).

[96] Daniel Moghimi u. a. „TPM-FAIL: TPM meets Timing and Lattice Attacks“. In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020. ISBN: 978-1-939133-17-5.

[97] Thomas Müller. *Trusted-computing-systeme*. 2008. DOI: 10.1007/978-3-540-76410-6.

[98] Cybersecurity Advisory National Security Agency. *Mitigate the GRUB2 BootHole Vulnerability*. 1. Juli 2020. URL: https://www.nsa.gov/Portals/70/documents/resources/cybersecurity-professionals/CSA_Mitigate_the_GRUB2_BootHole_Vulnerability_20200730_nsa_gov%20-%20Copy.pdf?ver=2020-07-30-170540-600 (besucht am 01. 04. 2021).

[99] Christopher Negus. *Linux Bible 2009 Edition: Boot Up Ubuntu, Fedora, KNOPPIX, Debian, OpenSUSE, and More*. 2009. ISBN: 978-0470373675.

[100] *New Microarchitecture for 4th Gen Intel Core Processor Plaforms*. Intel. 2013. URL: https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/4th-gen-core-family-mobile-brief.pdf (besucht am 01. 04. 2021).

[101] Thomas Nyman, Jan-Erik Ekberg und N Asokan. „Citizen electronic identities using TPM 2.0“. In: *Proceedings of the 4th International Workshop on Trustworthy Embedded Devices*. 2014. DOI: 10.1145/2666141.2666146.

[102] Ryan Elfmaster O'Neill. *Learning Linux Binary Analysis*. 2016. ISBN: 978-1782167105.

[103] Juha Partala. „Algebraic generalization of Diffie–Hellman key exchange“. In: *Journal of Mathematical Cryptology* (2018). DOI: 10.1515/jmc-2017-0015.

[104] Greig Paul und James Irvine. „Take control of your PC with UEFI secure boot“. In: *Linux Journal* (2015). DOI: 10.5555/2846055.2846056.

[105] Vojtěch Pavlík. *SUSE and Secure Boot: The Details*. 9. Sep. 2012. URL: https://www.suse.com/c/uefi-secure-boot-details/ (besucht am 01. 04. 2021).

[106] Raul Siles Pelaez. „Linux kernel rootkits: protecting the system's ring-zero“. In: *White paper of SANS Institute* (2004).

[107] Sandro Pinto und Nuno Santos. „Demystifying Arm TrustZone: A Comprehensive Survey“. In: *ACM Comput. Surv.* (2019). DOI: 10.1145/3291047.

[108] Gerald J Popek und Robert P Goldberg. „Formal requirements for virtualizable third generation architectures“. In: *Communications of the ACM* (1974). DOI: 10.1145/957195.808061.

[109] Pawit Pronkitprasan. *Full Disk Encryption on Arch Linux backed by TPM 2.0*. URL: https://medium.com/@pawitp/full-disk-encryption-on-arch-linux-backed-by-tpm-2-0-c0892cab9704 (besucht am 01. 04. 2021).

[110] DH Eugene Rodionov, Aleksandr Matrosov und David Harley. „Bootkits: Past, present and future“. In: *VB Conference*. 2014.

[111] Rohde-schwarz-cybersecurity. *TrustedGRUB2*. URL: https://github.com/Rohde-Schwarz/TrustedGRUB2 (besucht am 01. 04. 2021).

[112] Rami Rosen. „Linux containers and the future cloud“. In: *Linux Journal* (2014). DOI: 10.5555/2618216.2618219.

[113] Reiner Sailer u. a. „Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13“. In: *USENIX Security symposium*. 2004. DOI: 10.5555/1251375.1251391.

[114] Devik Sd. „Linux on-the-fly kernel patching without LKM“. In: *Phrack Magazine* (2001).

[115] Arvind Seshadri u. a. „SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes“. In: *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 2007. DOI: 10.1145/1294261.1294294.

[116] Alkesh Shah und Jonathon Giffin. „Analysis of rootkits: Attack approaches and detection mechanisms“. In: *Technical report, Georgia Institute of Technology, Tech. Rep.* (2008). DOI: 10.1007/978-3-642-24037-9_36.

[117] Jianxiong Shao u. a. „Formal Analysis of Enhanced Authorization in the TPM 2.0“. In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. 2015. DOI: 10.1145/2714576.2714610.

[118] Joseph Sharkey. „Breaking hardware-enforced security with hypervisors“. In: *Black Hat USA* (2016).

[119] Mickey Shkatov und Jesse Michael. *There's a hole in the boot*. 2020. URL: https://eclypsium.com/2020/07/29/theres-a-hole-in-the-boot/ (besucht am 01. 04. 2021).

[120] Pawel Skarzynski. *Kitgen*. 2015. URL: https://github.com/chesteroni/kitgen/ (besucht am 01. 04. 2021).

[121] Rod Smith. *Managing EFI Boot Loaders for Linux: Controlling Secure Boot*. 7. Juli 2018. URL: http://www.rodsbooks.com/efi-bootloaders/controlling-sb.html (besucht am 01. 04. 2021).

[122] *TPM 2.0 Library Specification*. Techn. Ber. 2013.

[123] Solus. *Boot Management*. URL: https://getsol.us/articles/troubleshooting/boot-management/en/ (besucht am 01. 04. 2021).

[124] system76. *Repair the Bootloader*. URL: https://support.system76.com/articles/bootloader/ (besucht am 01. 04. 2021).

[125] systemd. *The Boot Loader Specification*. URL: https://systemd.io/BOOT_LOADER_SPECIFICATION/#type-2-efi-unified-kernel-images (besucht am 01. 04. 2021).

[126] Andrew Tappert. *The Horse Pill Rootkit vs. Forcepoint Threat Protection for Linux*. 29. Nov. 2016. URL: https://www.forcepoint.com/blog/x-labs/horse-pill-rootkit-vs-forcepoint-threat-protection-linux (besucht am 01. 04. 2021).

[127] *TCG Guidance for Securing Network Equipment*. Version 1.0 Revision 01.29. The Trusted Computing Group (TCG). 2018. URL: https://trustedcomputinggroup.org/wp-content/uploads/TCG_Guidance_for_Securing_NetEq_1_0r29.pdf (besucht am 01. 04. 2021).

[128] *TCG TSS 2.0 Overview and Common Structures Specification*. Techn. Ber. Version 0.90 Revision 03. 2019. URL: https://trustedcomputinggroup.org/wp-content/uploads/TCG_TSS_Overview_Common_Structures_v0.9_r03_published.pdf (besucht am 01. 04. 2021).

[129] *TCG TSS 2.0 System Level API (SAPI) Specification*. Version 1.1 Revision 29. The Trusted Computing Group (TCG). 2019. URL: https://trustedcomputinggroup.org/wp-content/uploads/TSS_SAPI_v1p1_r29_pub_20190806.pdf (besucht am 01. 04. 2021).

[130] Alexander Tereshkin und Rafal Wojtczuk. „Introducing ring-3 rootkits“. In: *Black Hat USA* (2009).

[131] Donghai Tian u. a. „A Kernel Rootkit Detection Approach Based on Virtualization and Machine Learning“. In: *IEEE Access* (2019). DOI: 10.1109/ACCESS.2019.2928060.

[132] *TPM 2.0 Mobile Command Response Buffer Interface*. Level 00 Revision 12. The Trusted Computing Group (TCG). 2014. URL: https://trustedcomputinggroup.org/wp-content/uploads/Mobile-Command-Response-Buffer-Interface-v2-r12-Specification_FINAL2.pdf (besucht am 01. 04. 2021).

[133] *TPM 2.0 Mobile Reference Architecture*. Version 1.1 Revision 29. The Trusted Computing Group (TCG). 2014. URL: https://trustedcomputinggroup.org/wp-content/uploads/TPM-2-0-Mobile-Reference-Architecture-v2-r142-Specification_FINAL2.pdf (besucht am 01. 04. 2021).

[134] *tpm2-tools*. 2020. URL: https://github.com/tpm2-software/tpm2-tools (besucht am 01. 04. 2021).

[135] *tpm2-tss*. 2020. URL: https://github.com/tpm2-software/tpm2-tss (besucht am 01. 04. 2021).

[136] *Trusted Platform Module Library, Part 1: Architecture*. Revision 01.59. The Trusted Computing Group (TCG). 2019. URL: https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part1_Architecture_pub.pdf (besucht am 01. 04. 2021).

[137] Ubuntu. *Grub2*. URL: https://help.ubuntu.com/community/Grub2 (besucht am 01. 04. 2021).

[138] Ubuntu. *SecureBoot*. URL: https://wiki.ubuntu.com/UEFI/SecureBoot (besucht am 01. 04. 2021).

[139] Arun Wadkar Harshad S. and Mishra. „Secure Web Browsing Using Trusted Platform Module (TPM)“. In: *Information and Communication Technology for Intelligent Systems*. 2021. DOI: 10.1007/978-981-15-7062-9_28.

[140] Paul Georg Wagner, Pascal Birnstill und Jürgen Beyerer. „Establishing Secure Communication Channels Using Remote Attestation with TPM 2.0“. In: *International Workshop on Security and Trust Management*. 2020. DOI: 10.1007/978-3-030-59817-4_5.

[141] Yong Wang u. a. „Kernel Integrity Measurement Architecture Based on TPM 2.0“. In: (2018). DOI: 10.3969/j.issn.1000-3428.2018.03.028.

[142] Yong Wang u. a. „Virus analysis on idt hooks of rootkits trojan“. In: *2009 International Symposium on Information Engineering and Electronic Commerce*. 2009. DOI: 10.1109/IEEC.2009.52.

[143] Zuoguang Wang, Limin Sun und Hongsong Zhu. „Defining Social Engineering in Cybersecurity“. In: *IEEE Access* (2020). DOI: 10.1109/ACCESS.2020.2992807.

[144] Chenglong Wei u. a. „Expanding an Operating System's Working Space with a New Mode to Support Trust Measurement“. In: *International Conference on Information Security Practice and Experience*. 2015. DOI: 10.1007/978-3-319-17533-1_2.

[145] ESET Research Whitepapers. *LOJAX First UEFI rootkit found in the wild, courtesy of the Sednit group*. 2018. URL: https://www.welivesecurity.com/wp-content/uploads/2018/09/ESET-LoJax.pdf (besucht am 01. 03. 2021).

[146] Fedora Wiki. *GRUB 2*. URL: https://fedoraproject.org/wiki/GRUB_2 (besucht am 01. 04. 2021).

[147] Rafal Wojtczuk und Joanna Rutkowska. „Attacking intel trusted execution technology“. In: *Black Hat DC* (2009).

[148] Rafal Wojtczuk und Joanna Rutkowska. „Attacking Intel TXT via SINIT code execution hijacking“. In: *Invisible Things Lab* (2011).

[149] Rafal Wojtczuk und Joanna Rutkowska. „Attacking SMM Memory via Intel ® CPU Cache Poisoning“. In: *Invisible Things Lab* (2009).

[150] Rafal Wojtczuk, Joanna Rutkowska und Alexander Tereshkin. „Another way to circumvent Intel trusted execution technology“. In: *Invisible Things Lab* (2009).

[151] Kang Yang u. a. „Direct Anonymous Attestation With Optimal TPM Signing Efficiency“. In: *IEEE Transactions on Information Forensics and Security* (2021). DOI: 10.1109/TIFS.2021.3051801.

[152] Vincent Yao Jiewen and Zimmer. „Building Secure Firmware: Armoring the Foundation of the Platform“. In: 2020. DOI: 10.1007/978-1-4842-6106-4_7.

[153] Vincent Zimmer, Philip Oswald und Gary Lin. *UEFI Secure Boot in Linux*. 2013. URL: https://software.intel.com/content/www/us/en/develop/download/uefi-secure-boot-in-linux.html (besucht am 01. 04. 2021).

[154] Mimi Zohar und Dmitry Kasatkin. *Avoiding TPM PCR fragility using Secure Boot*. 5. Feb. 2018. URL: https://sourceforge.net/p/linux-ima/wiki/Home/ (besucht am 01. 04. 2021).