HAW HAMBURG

FORSCHUNGSWERKSTATT 1
Julian Ahrens

Secure Boot zur Absicherung von Trusted Execution Environments im Internet of Things

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

Betreuung durch: Prof. Dr. Thomas Schmidt
Eingereicht am: 31. März 2025

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Inhaltsverzeichnis

1 Einleitung **1**

2 Internet of Things **2**

3 Trusted Execution Environments **3**

3.1 Überblick 3

3.2 Trusted Execution Environments im Internet of Things 3

4 Secure Boot **5**

4.1 Überblick 5

4.2 Secure Boot für eingebettete Geräte 8

5 Zusammenfassung **11**

6 Ausblick **11**

Literaturverzeichnis **13**

Abstract: Diese Ausarbeitung befasst sich mit Trusted Execution Environments und speziell mit der Secure Boot Technologie, die diese ermöglicht. Es werden Trusted Execution Environments allgemein beschrieben und einige Implementierungen dieser für Geräte im Internet of Things vorgestellt. Dann werden Konzepte von und Techniken für Secure Boot erläutert. Auch hier werden verschiedene Umsetzungen der Technik vorgestellt. Anhand des Konzepts der Fernattestierung wird beschrieben, warum Secure Boot für die Implementierung von Trusted Execution Environments notwendig ist.

Keywords: Internet of Things, Trusted Execution Environment, Secure Boot

1 Einleitung

Milliarden eingebettete Systeme weltweit sind über das Internet of Things miteinander und mit dem übergeordneten Internet verbunden. Sie sammeln und verarbeiten Daten und steuern komplexe Prozesse, teils auch in sicherheitskritischen Bereichen. Angriffe auf Geräte im Internet of Things mit dem Ziel Daten zu entwenden oder Geräte zu übernehmen sind für böswillige Akteure lohnenswert. Die Absicherung solcher Geräte ist dagegen aufgrund der spezifischen Eigenschaften eingebetteter Geräte mit Herausforderungen verbunden.

Ein Ansatz, die Sicherheit von Geräten im Internet of Things zu erhöhen, ist die Verwendung von Trusted Execution Environments. Hierbei wird eine von anderen Aufgaben des Gerätes isolierte und daher als vertrauenswürdig deklarierte Ausführungsumgebung bereitgestellt, auf der besonders sicherheitsrelevante Aufgaben ausgeführt werden können. Eine solche Architektur bietet zwar bereits an sich eine erhöhte Sicherheit. Wird allerdings, beispielsweise durch Ausnutzung eines Softwarefehlers oder durch physischen Zugang, der Code des Trusted Execution Environments überschrieben, könnte ein Angreifer Zugriff auf eigentlich geschützte Ressourcen erlangen. Dies ließe sich durch Verwendung eines durch Secure Boot abgesicherten Startvorgangs unterbinden.

Ziel dieser Arbeit ist es, einen Überblick über Secure Boot im Bereich des Internet of Things zu geben und zu erläutern, wie ein sicherer Starvorgang dazu genutzt werden

kann, die Integrität eines Trusted Execution Environments sicherzustellen und nach außen nachweisbar zu machen. Dazu werden zunächst in Abschnitt 2 Hintergründe erläutert und in Abschnitt 3 ein kurzer Überblick über Trusted Execution Environments im Internet of Things gegeben. In Abschnitt 4 wird Secure Boot konzeptionell dargestellt, um dann den aktuellen Stand der Forschung und Entwicklung im Feld von Secure Boot im Bereich des Internet of Things wiederzugeben. In Abschnitt 6 wird ein Ausblick auf mögliche zukünftige Forschungsfragen gegeben.

2 Internet of Things

Maschinen und Objekte des alltäglichen Lebens werden im Zuge der fortschreitenden Digitalisierung unserer Welt zunehmend mit Mikrocontrollern versehen, um ihre Steuerung durch Programmlogik zu ermöglichen. Solche eingebetteten Geräte können mittels Sensoren und Aktuatoren mit der Außenwelt interagieren und so etwa Prozesse steuern oder Sensordaten sammeln [1]. Im Internet of Things (im Folgenden IoT) werden diese eingebetteten Geräte in einem Netzwerk verbunden, um gemeinsam komplexere Aufgaben lösen zu können [2]. IoT-Geräte werden in diversen Einsatzgebieten genutzt [3] und finden auch in sicherheitskritischen Bereichen wie z.B. der Gesundheitsversorgung [4] oder als Teil von Frühwarnsystemen in Bergwerken [5] Verwendung.

Aus den besonderen Eigenschaften von IoT-Geräten und -Netzwerken ergeben sich spezielle Überlegungen bezüglich IT-Sicherheit. So besitzen in IoT-Geräten verwendete Mikrocontroller oft eingeschränkte Hardwareressourcen, was vor allem auf kryptographischen Verfahren basierende Sicherheitsmaßnahmen erschwert. Zudem befinden sich IoT-Geräte oft in exponierten Lagen, was physische Angriffe schwer zu verhindern macht [6]. Eine bei IoT-Geräten besonders verbreitete Angriffstaktik ist es, diese mit dem Ziel zu übernehmen, sie zu Botnetzen zu verbinden. Vergangene Verfehlungen von IoT-Herstellern und Nutzern im Bereich der IT-Sicherheit haben zu mehreren spezifisch aus IoT-Geräten zusammengesetzten Botnetzen geführt [7].

RIOT-OS [8] ist ein Echtzeit-Betriebssystem für das IoT, welches Hardware verschiedener Hersteller unterstützt und insbesondere für die Verwendung auf IoT-Geräten mit geringen Hardwareressourcen optimiert ist.

3 Trusted Execution Environments

3.1 Überblick

Eine Möglichkeit, mobile Systeme im Allgemeinen und IOT-Geräte im Speziellen gegen häufige Angriffsvektoren abzusichern, ist die Verwendung von Trusted Execution Environments (im Folgenden TEEs). Um spezielle Operationen von einem möglicherweise nicht vertrauenswürdigen Betriebssystem und darauf laufenden Programmen zu isolieren, wird eine separate Ausführungsumgebung vom Rest des Systems abgespalten. Diese als TEE bezeichnete Ausführungsumgebung kann nun für besonders sicherheitsrelevante Operationen wie beispielsweise kryptographische Berechnungen oder das Abspeichern der dafür verwendeten Keys genutzt werden. Ein Zugriff von außerhalb des TEEs, also aus der als Rich Execution Environment (im Folgenden REE) bezeichneten potenziell nicht vertrauenswürdigen Ausführungsumgebung auf Ressourcen und Daten der TEE ist dabei nicht direkt möglich, wobei unterschiedliche Implementierungen unterschiedliche Möglichkeiten zur Kommunikation zwischen den Ausführungsumgebungen liefern [9].

TEEs werden beispielweise auf der Intel Plattform über die Intel Software Guard Extensions [10] umgesetzt. Eine verbreitete Implementierung für TEEs im mobilen und IOT-Bereich ist ARM TrustZone für die ARM-Prozessorarchitektur [11]. Für die offene und erweiterbare RISC-V Prozessorarchitektur [12] wird die Entwicklung von Trusted Execution Environments unter anderem durch die Alibaba Group vorangetrieben [13].

3.2 Trusted Execution Environments im Internet of Things

Trusted Execution Environments für IoT-Geräte sind ein Gegenstand der aktuellen Forschung. Ein Ansatz ist hierbei, Softwarelösungen auf Basis der durch ARM entwickelten TrustZone-M Hardwarearchitektur [11] zu entwickeln [14], [15]. Als Referenzimplementierung der TrustZone-M Technologie wird durch ARM die Trusted Firmware-M (im Folgenden TF-M) [16] herausgegeben. Auch für auf der RISC-V Architektur basierende IoT-Geräte wurden verschiedene Wege für die Implementierung von TEEs erkundet. Hier ist ein Ansatz, TEEs auf Basis von Privilegienstufen eines RISC-V Prozessors [17] umzusetzen, indem das REE im User Mode ausgeführt wird [18]. Andererseits erlaubt die offene Architektur von RISC-V auch, eigene Hardwareerweiterungen zu entwerfen, um auf diesem Wege Softwareisolierung zu ermöglichen [19].

Für RIOT-OS befinden sich aktuell zwei Implementierungen für Firmwares mit Unterstützung von TEEs in Entwicklung, wobei bedingt durch die typische Ressourcenbeschränktheit von IoT-Systemen ein möglichst geringer Performance-Overhead, sowie eine möglichst geringe Größe der Softwareartefakte angestrebt wird. Eine Lösung verfolgt für RISC-V Prozessoren den Weg, Softwareisolierung auf Basis von Privilegienstufen zu implementieren [20]. Nach Versuchen, RIOT-OS für die Verwendung auf ARM-Prozessoren mit der TF-M zu integrieren [21], die einen hohen Performance-Overhead gezeigt haben, wird nun an einer eigenen Firmware basierend auf der TrustZone-M Technologie gearbeitet.

Im Zuge der Implementierung haben sich beide Autor*innen mit dem Problem eines starken Anwachsens der Größe der Softwareartefakte durch Bibliotheken für kryptographische Operationen, die sowohl im TEE als auch im REE verwendet werden, beschäftigt. Die TF-M bietet hierfür die Lösung eines in der TEE ausgeführten Crypto Services für kryptographische Operationen [22], der aus TEE sowie REE über die ebenfalls durch ARM entwickelte PSA Crypto API [23] ansprechbar ist und als dessen Implementierung die Mbed TLS Softwarebibliothek verwendet [24]. RIOT-OS nutzt ebenfalls die PSA Crypto API, um kryptographische Operationen und Schlüsselverwaltung für das Betriebssystem und darauf ausgeführte Software anzubieten, verwendet allerdings anstelle von Mbed TLS eine eigens entwickelte Implementierung [25]. Für die TEE-Implementierung basierend auf der TF-M konnten Aufrufe der PSA Crypto API in RIOT-OS an den in der TEE ausgeführten Crypto Service geleitet werden. Auch in der Implementierung für RISC-V wurde ein ähnlicher Ansatz verfolgt, wobei hier die RIOT-eigene PSA Crypto Implementierung in das TEE ausgelagert wurde.

Ein häufiger Anwendungsfall für TEEs im IoT ist die Fernattestierung. Hierbei wird durch ein entferntes System eine Anfrage an ein IoT-Gerät gestellt, die dieses nur dann beantworten kann, wenn sein interner Zustand dem erwarteten Zustand entspricht, das Gerät also nicht kompromittiert wurde [26]. Für eine korrekte Antwort wird eine Messung der auf dem Gerät ausgeführten Software benötigt sowie zum Zweck der Authentifizierung des Gerätes ein Private-Key, mit dem diese Messung signiert wird. Dieser als Hardware Unique Key bezeichnete Private-Key wird typischerweise durch den Prozessorhersteller während der Fertigung in einem sicheren nicht-volatilen Speicherbereich abgelegt. Er ist unveränderlich und identifiziert das Gerät eindeutig [27]. Er ist zudem als vertraulich einzustufen und unbefugter Zugriff, insbesondere auch aus dem REE, ist zu unterbinden [19].

Um der Messung der ausgeführten Software tatsächlich Vertrauen schenken zu können und um die Isolation des zur Attestierung verwendeten Private-Keys sicherzustellen, muss aber bereits der Attestierungssoftware vertraut werden. Ließe sich die Firmware des Gerätes, beispielsweise durch einen phyischen Angriff, überspielen und so unter Umgehung des TEEs auf den Private-Key zugreifen, so könnte sich ein System mit einer kompromittierten Attestierungssoftware erfolgreich als vertrauenswürdiges Gerät ausgeben. Um dies zu verhindern ist ein Startvorgang notwendig, der sicherstellt, dass auf einem Gerät nur durch den Eigentümer vertraute Software ausgeführt werden kann. Dieses als Secure Boot bezeichnete Merkmal eines Softwaresystems wird typischerweise als notwendig für die Bereitstellung einer TEE angesehen [28].

4 Secure Boot

4.1 Überblick

Das digitale Signieren von Software ist eine seit langem etablierte und verbreitete Technik zum Zweck der Validierung der Integrität der Software und der Identität des Softwareherstellers. Ein digital signiertes Software-Artefakt wird zusammen mit einer Signatur ausgeliefert, die einen kryptographischen Hash über das jeweilige Artefakt enthält, signiert durch einen Private-Key des Softwareherstellers. Soll die Software nun installiert werden, kann anhand des Public-Keys des Softwareherstellers aus dieser Signatur der Hash des Softwareartefakts zum Zeitpunkt der Auslieferung durch den Softwarehersteller berechnet werden. Stimmt dieser mit dem Hash des tatsächlich vorliegenden Softwareartefakts überein, so ist die Integrität des Artefakts belegt [29]. Vertrauen in die verwendeten Public-Keys und damit in die Authentizität der Softwarehersteller kann über unterschiedliche Wege hergestellt werden. So können Plattform-Hersteller Schlüssel vertrauenswürdiger Softwarehersteller mit ihren Plattformen mitliefern. Im Falle von Anwendungssoftware sind es meist externe Zertifizierungsstellen, die die Identität von Softwareherstellern prüfen und eine Verifizierung der Authentizität ermöglichen [30].

Entscheidend für die korrekte Funktionalität dieses Verfahrens ist dabei sicherzustellen, dass Private-Keys nicht kompromittiert werden, wie beispielsweise mit einem Key des Herstellers Realtek im Jahr 2010 geschehen ist. Durch Zugriff auf diesen Key konnte eine wichtige Komponente des Stuxnet-Wurms mit einer scheinbar validen Signatur von Realtek ausgeliefert werden und wurde so trotz bestehender Signaturüberprüfung durch

das Betriebssystem des angegriffenen Gerätes als vertrauenswürdig gewertet und ausgeführt [31]. Das Entwenden von Private-Keys ist ein historisch verbreiteter Weg, Malware an Schutzmechanismen vorbei auf Geräte der Opfer zu bringen [32]. Dies macht ein Vorgehen zum Entzug der Zertifizierung der jeweils mit diesen verknüpften Public-Keys notwendig. Für Anwendersoftware wird dies typischerweise durch die Zertifizierungsstelle durchgeführt [33].

Der UEFI Secure Boot Prozess, beschrieben in [34] setzt die Prüfung von Softwaresignaturen im Bootprozess um. UEFI-Images, die zum Beispiel einen Bootloader enthalten können, werden mit aktiviertem Secure Boot nur dann gestartet, wenn das mitgelieferte Zertifikat in der UEFI-Variable DB gespeichert, also als vertrauenswürdig hinterlegt, ist und sich außerdem nicht in der UEFI-Variable DBX, in der nicht mehr vertrauenswürdige Zertifikate gespeichert sind, wiederfindet. Wird ein UEFI-Image ohne Zertifikat ausgeliefert, so kann es unter Umständen trotzdem ausgeführt werden, nämlich dann, wenn sich seine Signatur in der Variable DB befindet und nicht in der Variable DBX. Zwei weitere UEFI-Variablen werden durch den Hardware-Hersteller während der Fertigung in der Firmware hinterlegt. Key Exchange Keys (im Folgenden KEK) werden durch Betriebssystem-Hersteller verwaltet und ermöglichen es diesen, Änderungen an den DB- und DBX-Variablen durchzuführen. KEKs ermöglichen dadurch auch den Entzug der Vertrauenswürdigkeit von mit kompromittierten Private-Keys verbundenen Public-Keys. Der Platform-Key (im Folgenden PK) steht unter Kontrolle des Hardwareherstellers und ermöglicht wiederrum die Verwaltung der gespeicherten KEKs sowie Änderungen am PK selbst [35].

Problematisch wird dieser Prozess, wenn ein Open-Source Betriebssystem wie zum Beispiel Linux auf einem Gerät mit aktiviertem UEFI Secure Boot installiert werden soll. Herausgeber von Linux-Distributionen müssten als Betriebssystem-Hersteller eigentlich ihre eigenen KEKs verwalten und diese müssten durch Gerätehersteller mit neuen Geräten ausgeliefert werden. Da dies aber aufgrund der Natur von Open-Source Projekten unpraktikabel ist, wurde der Shim-Bootloader entwickelt. Dieser ist durch Microsoft signiert, deren KEKs in der großen Mehrzahl der Implementierungen von UEFI Secure Boot mitgeliefert werden. Shim verwaltet die Variablen MOKList sowie MOKListX, die in ihrer Funktion äquivalent zu den Werten DB bzw. DBX aus UEFI Secure Boot sind. Soll ein neues Software-Artefakt zur Ausführung durch Shim freigeschaltet werden, so wird zunächst ein Machine Owner Key (im Folgenden MOK) generiert, mit dem dieses signiert wird. Beim nächsten Neustart kann der generierte MOK nun in die MOKList geschrieben werden. Ein*e physisch anwesende*r Nutzer*in muss dies dabei durch In-

teraktion mit Shim bestätigen. Der Boot-Prozess sieht nun so aus, dass UEFI Secure Boot den Shim-Bootloader validiert und Shim in einem nächsten Schritt den folgenden Bootloader oder Kernel. Unterstützt auch dieser die Verwendung der MOKList und MOKListX Variablen, so können diese zur Validierung der wiederum darauffolgenden Bootloader, Kernel oder Gerätetreiber verwendet werden [36]

Dieses Modell, in dem einzelne Stufen des Bootprozesses durch die jeweils vorherige Stufe validiert werden, wird als Chain of Trust bezeichnet. Alle Glieder der Chain of Trust mit Ausnahme des ersten Gliedes, dem Root of Trust (im Folgenden RoT), können auf beschreibbarem Speicher liegen und auch, beispielsweise bei einem Update, verändert werden. Ließe sich auch der RoT beliebig ändern, so könnte auf einem Gerät trotz Secure Boot beliebige Software ausgeführt werden. Der RoT muss daher in einem Festwertspeicher (Read-Only Memory, im Folgenden ROM) oder anderweitig in der Hardware auf unveränderbare Weise implementiert werden. Dadurch ist für den gesamten Lebenszyklus eines Gerätes festgelegt, dass in der auf den RoT folgenden Stufe der Chain of Trust - und im Falle einer strengen Chain of Trust auch in allen weiteren Stufen – nur durch den Bereitsteller des RoT autorisierte Software ausgeführt wird. Ein erfolgreicher Startvorgang des letzten Gliedes der Chain of Trust kann daher als Integritätsbeweis für den Zustand der Systemsoftware eines Gerätes gesehen werden. Wäre ein Glied der Chain of Trust kompromittiert worden, so wäre es durch das vorherige Glied nicht erfolgreich validiert und daher auch nicht ausgeführt worden, rückverfolgbar bis zum unveränderbaren und daher nur schwer kompromittierbaren RoT [37]. Ein RoT kann beispielsweise implementiert werden durch einen im ROM gespeicherten Bootloader in Kombination mit einem ebenfalls im ROM gespeicherten Public-Key, anhand dem die folgenden Glieder der Chain of Trust validiert werden können [38].

Soll das Konzept eines sicheren Starvorgangs nun mit dem eines Trusted Execution Environment kombiniert werden, so lässt sich die Chain of Trust durch das TEE weiter bis zur Software des REE führen. Dazu wird im Bootprozess zunächst die Software des TEEs gestartet, die dann die Integrität der im REE auszuführende Software vor dessen Start verifizieren muss. Alle für den Startvorgang des Gerätes an sich sowie der TEE nötigen Operationen finden dadurch vor möglicher Manipulation aus dem REE geschützt innerhalb des TEEs statt [39].

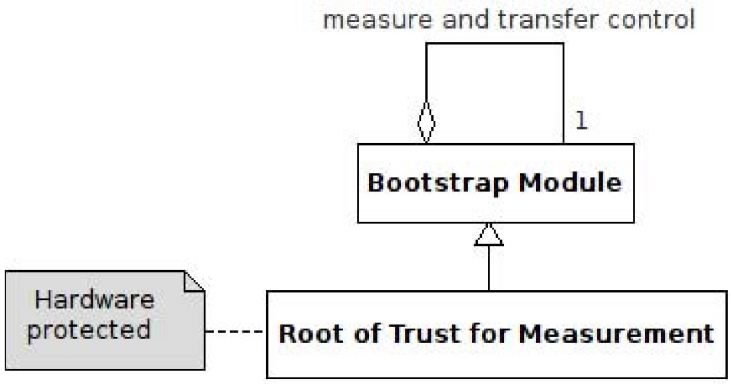

Abbildung 1: Schematische Darstellung einer Chain of Trust (Auszug aus [37])

4.2 Secure Boot für eingebettete Geräte

Implementierungen von Secure Boot für IoT-Geräte verfolgen typischerweise die Strategie, strikte Chains of Trust aufzubauen und bieten keine Mechanismen zum Austausch von Keys wie UEFI Secure Boot. Profentzas et al. [40] untersuchen zwei verschiedene Implementierungen von Secure Boot für IoT-Geräte, wobei eine Implementierung als softwarebasiert und eine Implementierung als hardwarebasiert beschrieben wird. Die softwarebasierte Implementierung verfolgt dabei die klassische Variante, einen initialen Bootloader sowie einen Public-Key zur Verifizierung der folgenden Bootstufen im ROM abzulegen und von diesem zu starten. Die Autor*innen untersuchen die durch diese Implementierung ausgelöste Verzögerung im Bootprozess bei der Ausführung des Linux-Kernels auf ARM-basierten Mikrochips mit mittleren Hardwareressourcen. Sie messen dabei für verschiedene verwendete Hashfunktionen und Hardwareplattformen eine Verzögerung zwischen 1,4 und 4%. Auch eine Variante unter Verwendung eines externen Hardwarebausteins wird durch die Autor*innen untersucht. Dieser, einem Trusted Platform Module nachempfundene Baustein, übernimmt die Verifizierung der einzelnen Bootstufen. Auch hierbei wird ein unveränderlicher Bootloader im ROM als RoT verwendet, wobei dieser den Kontrollfluss für die Verifizierung an den Hardwarebaustein weitergibt. Die hardwarebasierte Implementierung sorgt im Versuchsaufbau der Autor*innen für eine deutlich größere Verzögerung des Bootprozesses als dies bei der softwarebasierten Implementierung der Fall war, um zwischen 5,6 und 15%. Die Autor*innen weisen weiter darauf hin, dass die von ihnen für die hardwarebasierte Implementierung getätigten Modifizierungen der Hardware nicht für jedes beliebige IoT-Gerät umsetzbar ist. Zudem geben sie zu bedenken, dass IoT-Geräte mit geringeren Hardwareressouren womöglich eine zu ge-

ringe Rechenleistung für die den softwarebasierten Secure Boot Prozess ermöglichenden kryptografischen Berechnungen haben könnten.

Eine weitere hardwarebasierte Implementierung bieten Cano-Quiveu et al. [41]. Der durch sie entwickelte als „Instant Retrieval Information System“ (IRIS) bezeichnete Hardwarebaustein besteht aus zwei Komponenten, dem Boot- sowie dem E-LUKS-Modul. E-LUKS beschreibt dabei eine Abwandelung des LUKS-Verfahren für Festplattenverschlüsselung, mit dem ein an das IoT-Gerät angeschlossener Massenspeicher verschlüsselt ist. Im Unterscheid zu LUKS ist E-LUKS als Hardwaremodul umgesetzt. Im Bootprozess wird zunächst das Boot-Modul aufgerufen, das aus einem angeschlossenen Speicher ein Passwort ausliest und dieses zusammen mit Informationen über die Position des zu startenden Bootloaders im angeschlossenen Massenspeicher an das E-LUKS-Modul sendet. Dieses nutzt das Passwort, um die angefragten Daten zu entschlüsseln und sendet diese dann zurück an das Boot-Modul, welches den geladenen Bootloader in den RAM legt und den Kontrollfluss zur Ausführung wieder an den Prozessor übergibt. In dieser Implementierung agiert das an das Boot-Modul übergebene Passwort also als RoT und die erfolgreiche Entschlüsselung des Bootloaders aus dem Massenspeicher als erfolgreiche Validierung des Bootloaders. Die Autor*innen beschreiben in ihrem Papier nur den Entwurf des Hardwaremoduls an sich, die unveränderliche Speicherung des genutzten Passworts ist also nicht Teil ihrer Ausarbeitung. In einem Versuch messen die Autor*innen einen um zwischen 13 und 80% verzögerten Startvorgang durch Secure Boot beim Start eines Linux-Kernels, abhängig von den Funktionen von IRIS, die verwendet werden.

Ling et al. entwickeln in [42] eine System mit einem TEE, wobei das TEE mit Secure Boot gestartet wird und die Validität des REE attestieren kann. Die Autor*innen benutzen Werte auf zwei verschiedenen Speichermedien gemeinsam als RoT. Zum einen wird ein auf dem ROM liegender initialer Bootloader als erstes Glied der Chain of Trust verwendet. Zudem wird bei Bespielung des Mikrocontrollers ein Public-Key auf einem eFuse, einer Art von einmalig programmierbarem Speicher, abgelegt, die der Bootloader zur Verifizierung der nächsten Bootstufe verwendet. Ein durch den Secure Boot Prozess abgesichertes TEE kann nun bei Start des REE dieses messen und die Korrektheit der gestarteten Umgebung im Attestierungsprozess bestätigen. Bei einem Test, bei dem auf einem Mikrochip mit mittleren Hardwareressourcen eine Anwendung gestartet wird, führt die implementierte Secure Boot Lösung des TEE zu einer geringen Verzögerung im Bootvorgang von nur 0,5%. Die Messung des REE zum Zweck der Attestierung führt dagegen zu einem deutlichen zeitlichen Overhead. Hier kommt es zu einer Verzögerung im

Start des Betriebssystems von 44%. Die Autor*innen erklären diese starke Verzögerung mit der Größe des Betriebssystem-Images.

Auch die TF-M verwendet einen im ROM oder anderem schreibgeschützten Speicherbereich abgelegten Bootloader, sowie einen ebenfalls schreibgeschützt abgelegten Public-Key als RoT, um Secure Boot zu ermöglichen [43]. Zusätzlich unterstützt TF-M eine Rollback-Protection. Jedes Software-Image enthält dabei einen sogenannten Security Counter. Wird versucht, eine Softwareversion zu installieren mit einem niedrigeren Security Counter als eine Version, die bereits vorher auf dem Gerät ausgeführt wurde, so wird diese durch den Bootloader von TF-M nicht gestartet [44].

Einige ARM-Prozessoren besitzen als Unterstützung für die Verwaltung kryptographischer Schlüssel ein als ARM Lifecycle Manager [45] bezeichnetes Hardwaremodul, welches zur gemeinsamen Speicherung von diversen durch den Prozessor- sowie den Gerätehersteller gesetzten Schlüsseln verwendet wird. Dieses kann eine Reihe von Zuständen einnehmen, die während der Herstellung eines Gerätes durchlaufen werden. Der Prozessorhersteller setzt initial den Zustand auf „Chip Manufacturing“. In diesem Zustand können durch den Prozessorhersteller verwaltete Schlüssel wie beispielsweise der Hardware Unique Key eingeladen werden. Vor Auslieferung des Prozessors setzt der Prozessorhersteller den Zustand auf „Device Manufacturing“, was es dem Gerätehersteller nun ermöglicht, die durch ihn verwalteten Schlüssel, wie beispielsweise den Public Key zur Verifizierung von Softwareartefakten während des Secure Boot Prozesses, einzuladen. Abschließend setzt der Gerätehersteller den Zustand zur Auslieferung auf „Secure Enabled“. Schlüssel können dabei nur dann gesetzt werden, wenn sich der Lifecycle Manager in dem dafür vorgesehenen Zustand befindet. Ein Zurückdrehen auf einen früheren Gerätezustand ist nicht möglich. Schlüssel werden in ROM gespeichert und sind gegen physisches Auslesen gesichert.

Während sich für klassische persönliche Computersysteme UEFI Secure Boot als Standardlimplementierung durchgesetzt hat, gibt es einen solchen Standard im Bereicht der IoT-Geräte also noch nicht. Eigene Hardwarebausteine werden genauso verwendet wie Softwarelösungen mit verschiedenen Roots of Trust. Trotz der geringen Rechenleistung eingebetteter Geräte zeigen die hier gesichteten Implementierungen die Möglichkeit, performante Softwarelösungen für Secure Boot auf diesen umzusetzen ohne die Notwendigkeit zusätzlicher Hardware außer einem ROM zur Speicherung des RoTs. Abseits des Problems der eingeschränkten Kompatibilität maßgeschneiderter Hardwaremodifikatio-

nen zeigen diese unter den gesichteten Implementierungen außerdem eine höhere Verzögerung des Startvorgangs im Vergleich zu Softwarelösungen.

5 Zusammenfassung

Netzwerke aus IoT-Geräten bieten naturgemäß Herausforderungen im Gebiet der IT-Sicherheit. Wichtig ist daher, dass Vertrauen in die Integrität einzelner IoT-Geräte sowie in die auf diesen ausgeführte Software aufgebaut werden kann. In dieser Ausarbeitung wurde ein Weg, dieses Vertrauen herzustellen, allgemein beschrieben. Dabei wird zunächst im Bootprozess eine Chain of Trust aufgebaut, in der jede Bootstufe durch die jeweils vorherige Bootstufe vor der Ausführung validiert wird. Durch einen unveränderlichen Root of Trust wird dabei sichergestellt, dass auf dem Gerät tatsächlich nur durch den Geräteentwickler vertraute Software ausgeführt werden kann. Auf diese Weise wird eine als TEE bezeichnete vertraute Ausführungsumgebung gestartet, die mit dem Gerät vernetzten anderen Geräten durch den Attestierungsprozess ihre Vertrauenswürdigkeit beweisen sowie vertrauenswürdige Aussagen über das separate REE treffen kann.

Zudem wurden verschiedene Implementierungen von sicheren Bootprozessen sowohl für Mikroprozessoren als auch für klassische persönliche Computersysteme vorgestellt. Dabei wurden sowohl software- als auch hardwarebasierte Verfahren betrachtet sowie verschiedene Möglichkeiten, einen RoT auf IoT-Geräten zu implementieren. Auch für die Implementierung von TEEs auf eigebetteten Geräten wurden verschiedene Implementierungen betrachtet. Auf diese Weise wurde ein Überblick über die Themen Secure Boot sowie Trusted Execution Environments gegeben sowie der aktuelle Stand der Entwicklung dieser Techniken im Kontext des Internet of Things beschrieben.

6 Ausblick

Auch die folgenden Projektarbeiten sollen sich mit den Themen Secure Boot sowie der Entwicklung vertrauenswürdiger und attestierbarer IoT-Systeme beschäftigen. Eine Möglichkeit dafür wäre, bereits in Arbeit befindliche Implementierungen für Trusted Execution Environments für RIOT-OS um die Unterstützung von Secure Boot zu erweitern.

Zu entscheiden ist dabei zunächst, ob ein software- oder ein hardwarebasiertes Verfahren gewählt werden soll. Da RIOT-OS aber eine hohe Kompatibilität mit verschiedensten Mikrocontrollern anstrebt, wird die Wahl hier vermutlich auf eine softwarebasierte Lösung fallen. Zudem ist eine Strategie für die sichere Bereitstellung eines Hardware Unique Keys sowie eines Public-Keys zur Verifizierung von Software im Secure Boot Prozess zu finden. Hierbei sollte die klassische Variante eines Boot-ROMs bestehend aus Public Key und Bootloader sowie weitere Varianten wie ein ARM Lifecycle Manager betrachtet werden. Gerade für den zur Softwareverifizierung verwendeten Schlüssel ist zudem zu evaluieren, ob eine Möglichkeit zur Schlüsselrotierung, wie in UEFI Secure Boot implementiert, angeboten werden muss und wie eine solche umsetzbar wäre.

Zu untersuchendes Leistungsmerkmal einer Implementierung von Secure Boot sollte einerseits die Verzögerung im Startvorgang sein und andererseits die Größe der entstehenden Softwareartefakte. Kryptographische Operationen sollten unter Verwendung der PSA Crypto API und der Implementierung in RIOT-OS durchgeführt werden. PSA Crypto definiert dabei bereits Methoden zur Verifizierung von Signaturen, die für den Secure Boot Prozess verwendet werden können. In vorherigen Arbeiten wurde bereits eine Deduplizierung von Softwarebibliotheken durchgeführt, sodass Bibliotheken nur einmalig anstatt separat für TEE und REE ausgeliefert werden müssen. Bei Implementierung eines Secure Boot Prozesses wächst die Zahl unabhängiger Komponenten, die kryptographische Operationen unterstützen müssen, um einen bzw. mehrere Bootloader. Eine Deduplizierung von Code zwischen Bootloader und TEE wurde bereits durch Ban für die TF-M umgesetzt [46]. Hieran kann sich bei der Implementierung im Grundprojekt orientiert werden.

Literaturverzeichnis

[1] J. A. Stankovic, „Real-time and embedded systems," *ACM Computing Surveys*, Jg. 28, Nr. 1, S. 205–208, März 1996, ISSN: 1557-7341. DOI: 10.1145/234313.234400.

[2] L. Atzori, A. Iera und G. Morabito, „The Internet of Things: A survey," *Computer Networks*, Jg. 54, Nr. 15, S. 2787–2805, Okt. 2010, ISSN: 1389-1286. DOI: 10.1016/j.comnet.2010.05.010.

[3] F. Samie, L. Bauer und J. Henkel, „IoT technologies for embedded computing: a survey," in *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, Ser. ESWEEK'16, ACM, Okt. 2016. DOI: 10.1145/2968456.2974004.

[4] S. M. Riazul Islam, D. Kwak, M. Humaun Kabir, M. Hossain und K.-S. Kwak, „The Internet of Things for Health Care: A Comprehensive Survey," *IEEE Access*, Jg. 3, S. 678–708, 2015, ISSN: 2169-3536. DOI: 10.1109/access.2015.2437951.

[5] W. Qiuping, Z. Shunbing und D. Chunquan, „Study On Key Technologies Of Internet Of Things Perceiving Mine," *Procedia Engineering*, Jg. 26, S. 2326–2333, 2011, ISSN: 1877-7058. DOI: 10.1016/j.proeng.2011.11.2442.

[6] I. Makhdoom, M. Abolhasan, J. Lipman, R. P. Liu und W. Ni, „Anatomy of Threats to the Internet of Things," *IEEE Communications Surveys & Tutorials*, Jg. 21, Nr. 2, S. 1636–1675, 2019, ISSN: 2373-745X. DOI: 10.1109/comst.2018.2874978.

[7] M. Antonakakis et al., „Understanding the Mirai Botnet," in *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC: USENIX Association, Aug. 2017, S. 1093–1110, ISBN: 978-1-931971-40-9. Adresse: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis.

[8] E. Baccelli et al., „RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT," *IEEE Internet of Things Journal*, Jg. 5, Nr. 6, S. 4428–4440, Dez. 2018, ISSN: 2372-2541. DOI: 10.1109/jiot.2018.2815038.

[9] O. Hosam und F. BinYuan, „A Comprehensive Analysis of Trusted Execution Environments," in *2022 8th International Conference on Information Technology Trends (ITT)*, IEEE, Mai 2022, S. 61–66. DOI: 10.1109/itt56123.2022.9863962.

[10] Intel. „Intel® Software Guard Extensions (Intel® SGX),“ besucht am 20. März 2025. Adresse: https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/software-guard-extensions.html.

[11] ARM Ltd. „TrustZone for Cortex-M,“ besucht am 20. März 2025. Adresse: https://www.arm.com/technologies/trustzone-for-cortex-m.

[12] RISC-V. „RISC-V,“ besucht am 4. Mai 2025. Adresse: https://riscv.org/.

[13] S. C. Yuehai Chen Huarun Chen, „TEE SoC Based on RISC-V,“ Risc-V, Techn. Ber., 2023. Adresse: https://riscv.org/blog/2023/02/tee-soc-based-on-risc-v/.

[14] U. Lee und C. Park, „SofTEE: Software-Based Trusted Execution Environment for User Applications,“ *IEEE Access*, Jg. 8, S. 121 874–121 888, 2020, ISSN: 2169-3536. DOI: 10.1109/access.2020.3006703.

[15] L. Guan et al., „TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone,“ 2017. DOI: 10.48550/ARXIV.1704.05600.

[16] ARM Ltd. „Trusted Firmware-M,“ besucht am 25. März 2025. Adresse: https://developer.arm.com/Tools%20and%20Software/Trusted%20Firmware-M.

[17] K. A. Andrew Waterman und J. Hauser. „The RISC-V Instruction Set Manual Volume II: Privileged Architecture, Document Version 20211203,“ besucht am 4. Mai 2025. Adresse: https://github.com/riscv/riscv-isa-manual/releases/tag/Priv-v1.12.

[18] L. Pfau. „Measuring the Performance Overhead of RIOT Running in RISC-V User-Mode. “Adresse: https://www.inet.haw-hamburg.de/teaching/ws-2023-24/project-class/pr1_lars_pfau.pdf.

[19] V. Costan, I. Lebedev und S. Devadas, „Sanctum: Minimal Hardware Extensionsfor Strong Software Isolation,“ in *25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX: USENIX Association, Aug. 2016, S. 857–874, ISBN: 978-1-931971-32-4. Adresse: https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_costan.pdf.

[20] L. Pfau. „Integrating a RISC-V Secure Firmware into RIOT. “Adresse: https://www.inet.haw-hamburg.de/teaching/ss-2024/project-class/pr2_lars_pfau.pdf.

[21] L. Boeckmann. „Evaluation of a Secure Processing Environment in RIOT OS. “Adresse: https://www.inet.haw-hamburg.de/teaching/ws-2023-24/project-class/pr2_lena_boeckmann.pdf.

[22] A. de Angelis. „TF-M Crypto Service design,“ besucht am 9. Mai 2025. Adresse: https://trustedfirmware-m.readthedocs.io/en/latest/design_docs/services/tfm_crypto_design.html#tf-m-crypto-as-a-particular-configuration-of-mbed-tls.

[23] ARM Ltd. „PSA Certified Crypto API 1.1,“ besucht am 8. Mai 2025. Adresse: https://arm-software.github.io/psa-api/crypto/1.1/.

[24] Linaro Ltd. „Mbed TLS,“ besucht am 9. Mai 2025. Adresse: https://www.trustedfirmware.org/projects/mbed-tls/.

[25] L. Boeckmann, P. Kietzmann, L. Lanzieri, T. Schmidt und M. Wählisch, „Usable Security for an IoT OS: Integrating the Zoo of Embedded Crypto Components Below a Common API,“ 2022. DOI: 10.48550/ARXIV.2208.09281.

[26] R. V. Steiner und E. Lupu, „Attestation in Wireless Sensor Networks: A Survey,“ *ACM Computing Surveys*, Jg. 49, Nr. 3, S. 1–31, Sep. 2016, ISSN: 1557-7341. DOI: 10.1145/2988546.

[27] I. Lebedev, K. Hogan und S. Devadas, „Invited Paper: Secure Boot and Remote Attestation in the Sanctum Processor,“ in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, IEEE, Juli 2018. DOI: 10.1109/csf.2018.00011.

[28] M. Sabt, M. Achemlal und A. Bouabdallah, „Trusted Execution Environment: What It is, and What It is Not,“ in *2015 IEEE Trustcom/BigDataSE/ISPA*, IEEE, Aug. 2015, S. 57–64. DOI: 10.1109/trustcom.2015.357.

[29] Microsoft. „Introduction to Code Signing,“ besucht am 23. März 2025. Adresse: https://learn.microsoft.com/de-de/previous-versions/windows/internet-explorer/ie-developer/platform-apis/ms537361(v=vs.85).

[30] D. Cooper et al., „Security Considerations for Code Signing,“ NIST, Techn. Ber., 2018. Adresse: https://nvlpubs.nist.gov/nistpubs/CSWP/NIST.CSWP.01262018.pdf.

[31] N. Falliere, L. O. Murchu und E. Chien, „W32.Stuxnet Dossier,“ Symantec, Techn. Ber., 2011. Adresse: https://docs.broadcom.com/doc/security-response-w32-stuxnet-dossier-11-en.

[32] D. Kim, B. J. Kwon und T. Dumitraş, „Certified Malware: Measuring Breaches of Trust in the Windows Code-Signing PKI," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, Ser. CCS '17, ACM, Okt. 2017, S. 1435–1448. DOI: 10.1145/3133956.3133958.

[33] D. Kim, B. J. Kwon, K. Kozák, C. Gates und T. Dumitras, „The Broken Shield: Measuring Revocation Effectiveness in the Windows Code-Signing PKI," in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD: USENIX Association, Aug. 2018, S. 851–868, ISBN: 978-1-939133-04-5. Adresse: https://www.usenix.org/conference/usenixsecurity18/presentation/kim.

[34] M. Nystrom, M. Nicoles und V. Zimmer, „UEFI Networking and Pre-OS Security," *Intel Technology Journal*, Jg. 15, S. 80–1, Okt. 2011. Adresse: https://www.researchgate.net/publication/235258577_UEFI_Networking_and_Pre-OS_Security.

[35] R. Wilkins und B. Richardson, „UEFI Secure Boot in modern computer security solutions," UEFI, Techn. Ber., 2013. Adresse: https://uefi.org/sites/default/files/resources/UEFI_Secure_Boot_in_Modern_Computer_Security_Solutions_2013.pdf.

[36] R. Haas und M. Pirker, „The State of Boot Integrity on Linux - a Brief Review," in *Proceedings of the 19th International Conference on Availability, Reliability and Security*, Ser. ARES 2024, ACM, Juli 2024, S. 1–6. DOI: 10.1145/3664476.3670910.

[37] H. Löhr, A.-R. Sadeghi und M. Winandy, „Patterns for Secure Boot and Secure Storage in Computer Systems," in *2010 International Conference on Availability, Reliability and Security*, IEEE, Feb. 2010, S. 569–573. DOI: 10.1109/ares.2010.110.

[38] N. Redini et al., „BootStomp: On the Security of Bootloaders in Mobile Devices," in *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC: USENIX Association, Aug. 2017, S. 781–798, ISBN: 978-1-931971-40-9. Adresse: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/redini.

[39] GlobalPlatform. „TEE System Architecture v1.3," besucht am 28. März 2025. Adresse: https://globalplatform.org/wp-content/uploads/2022/

05/GPD_SPE_009-GPD_TEE_SystemArchitecture_v1.3_PublicRelease_signed.pdf.

[40] C. Profentzas, M. Gunes, Y. Nikolakopoulos, O. Landsiedel und M. Almgren, „Performance of Secure Boot in Embedded Systems," in *2019 15th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, IEEE, Mai 2019, S. 198–204. DOI: 10.1109/dcoss.2019.00054.

[41] G. Cano-Quiveu, P. Ruiz-de-Clavijo-Vazquez, M. Bellido, J. Juan-Chico und J. Viejo-Cortes, „IRIS: An embedded secure boot for IoT devices," *Internet of Things*, Jg. 23, S. 100 874, Okt. 2023, ISSN: 2542-6605. DOI: 10.1016/j.iot.2023.100874.

[42] Z. Ling et al., „Secure boot, trusted boot and remote attestation for ARM TrustZone-based IoT Nodes," *Journal of Systems Architecture*, Jg. 119, S. 102 240, Okt. 2021, ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2021.102240.

[43] ARM Ltd. „Trusted Firmware-M Secure Boot," besucht am 30. März 2025. Adresse: https://tf-m-user-guide.trustedfirmware.org/design_docs/booting/tfm_secure_boot.html.

[44] T. Ban. „Rollback protection in TF-M secure boot," besucht am 30. März 2025. Adresse: https://tf-m-user-guide.trustedfirmware.org/design_docs/booting/secure_boot_rollback_protection.html.

[45] ARM Ltd. „Arm® Lifecycle Manager Specification Revision: 1.0," besucht am 9. Mai 2025. Adresse: https://developer.arm.com/documentation/107616/0000.

[46] T. Ban. „Code sharing between independently linked XIP binaries," besucht am 9. Mai 2025. Adresse: https://trustedfirmware-m.readthedocs.io/en/latest/design_docs/software/code_sharing.html.